Using Probabilistic Reasoning to Automate Software Tuning

A thesis presented

by

David Gerard Sullivan

to

The Division of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

September, 2003

Copyright © 2003 David Gerard Sullivan

All rights reserved.

Advisor: Margo I. Seltzer

Using Probabilistic Reasoning to Automate Software Tuning

David Gerard Sullivan

Abstract

Complex software systems typically include a set of parameters that can be adjusted to improve the system's performance. System designers expose these parameters, which are often referred to as knobs, because they realize that no single configuration of the system can adequately handle every possible workload. Therefore, users are allowed to tune the system, reconfiguring it to perform well on a specific workload. However, manually tuning a software system can be an extremely difficult task, and it is often necessary to dynamically retune the system as workload characteristics change over time. As a result, many systems are run using either the default knob settings or non-optimal alternate settings, and potential performance improvements go unrealized. Ideally, the process of software tuning should be automated, allowing software systems to determine their own optimal knob settings and to reconfigure themselves as needed in response to changing conditions.

This thesis demonstrates that probabilistic reasoning and decision-making techniques can be used as the foundation of an effective, automated approach to software tuning. In particular, the thesis presents a methodology for automated software tuning that employs the influence diagram formalism and related learning and inference algorithms, and it uses this methodology to tune four knobs from the Berkeley DB embedded database system. Empirical results show that an influence diagram can effectively generalize from training data for this domain, producing considerable performance improvements on a varied set of workloads and outperforming an alternative approach based on linear regression.

The thesis provides a detailed roadmap for applying the proposed software-tuning methodology to an arbitrary software system. It also proposes novel methods of addressing three challenges associated with using an influence diagram for software tuning: modeling the performance of the software system in a tractable manner, determining effective discretizations of continuous variables in the model, and estimating parameters of the model for which no training data is available. In addition, the thesis discusses the design of workload generators that can produce the necessary training data, and it proposes a technique for ensuring that a workload generator captures the steady-state performance of the system being tuned.

Contents

List of Figures

List of Tables

Acknowledgments

I owe a tremendous debt of gratitude to my advisor, Margo Seltzer. She took me on when I was an incoming master's student with a limited background in computer science, and she taught me how to do research. Margo has been an advisor in the fullest sense of the word—offering wisdom, care, and support about not only academic and professional matters, but personal ones as well. She taught by example the importance of balancing work and family, and she supported me as I worked to maintain my own balance in this regard, especially after the arrival of my son. Last fall, when the completion of this thesis still seemed out of reach, Margo helped me to see that the end was in sight, and she used praise and encouragement, rather than pressure, to coax me to the finish line.

I am also extremely grateful to Barbara Grosz, who introduced me to artificial intelligence and encouraged me to bring together my dual interests in systems and AI. Through her advice and encouragement, Barbara helped me to become a more confident researcher and a better writer and presenter. When I contemplated leaving the Ph.D. program several years ago, both Barbara and Margo offered an amazing degree of understanding and support, giving me the freedom to leave if I needed to, while helping me to realize that it was worthwhile to stay. Without the two of them, I might never have finished.

I also had the good fortune to have Avi Pfeffer on my thesis committee. The work presented in this dissertation began as a project for one of his courses, and his expertise in probabilistic reasoning was essential to its completion. I am grateful for his willingness to answer my questions and to ask good questions of his own—questions that forced me to clarify my thinking and to flesh out the methodology presented in this thesis.

During my six years as a graduate student, I benefited from the company and assistance of fellow students from two research groups: the SYRAH group (formerly known as VINO), and the AI research group. Chris Small, who was finishing up when I was just starting, generously and patiently answered my questions about a wide range of topics, from UNIX tips and kernel hacking to career plans and home ownership. Keith Smith, Yaz Endo, David Krinsky, and Robert Haas also provided valuable assistance as I got started. Dan Ellard and Lex Stein were my office-mates for the bulk of my time in grad school, and I enjoyed my time and conversations with both of them, including talks about parenthood with Dan and about politics with Lex (especially during the Florida election debacle of 2000). Dan graciously allowed me to use for my experiments some of the machines that he maintained, which greatly facilitated my work. I shared many enjoyable walks and lunches with Kobi Gal, whose friendship and advice helped me to make it through the second half of grad school. He understood both what I was working on and what I was working through—the insecurities, doubts, and questions—and he helped me to maintain a positive outlook. I am grateful to all of these people, and to the others with whom I shared my time in ESL and Maxwell Dworkin.

I couldn't have made it to this point without the love and support of my friends and family members, who allowed me to remain sane, happy, and focused on the important things in life. Many thanks to each of them. In particular, John Hamilton and Scott MacDonald helped me to get through some rough patches that I encountered along the way, and generous baby-sitting help from my sisters, Karen and Kristen, my mother, Diane, and my mother-in-law, Rita Nuscher, gave me extra time for both work and relaxation. Were it not for the extra workdays that my mother and mother-in-law made possible during my final months of work on this thesis, I would still be running experiments.

I feel blessed to have been raised in such a wonderful family, with three great siblings and two amazing parents, who valued education and helped me to grow and

develop as an individual. The self-giving of my parents continues to this day, and my own experiences as a parent have only increased my love and respect for them.

This thesis is dedicated to the two loves of my life—my partner, Dave, and my son, Perry. Dave's love has been the foundation of all that I have done as a grad student. He encouraged me to pursue my dream of getting a Ph.D., and when I found myself questioning my original career plans, he enabled me to envision a future in which I would be doing work that makes me happy and fulfilled. Dave's attentiveness to my needs, his willingness to listen when things were difficult, and his good-hearted companionship through everything were essential to my completion of this thesis, and I am so grateful to him for his unwavering love and support. Along the way, we managed to build a family and a life together—a life that brings me tremendous happiness. Perry's arrival two years ago was the fulfillment of our dream of being parents, and he has transformed our lives for the better with his affection, enthusiasm, and curiosity. I look forward to watching him develop his own interests and pursue his own dreams in the months and years ahead.

To Dave and Perry

Chapter 1

Introduction

It is...black magic...how databases are tuned. It is tough to ship a tuning guru with every database.

– Surajit Chaudhuri, database researcher [Kan02]

As the above remarks indicate, manually tuning the performance of a database system is an extremely challenging task. The same can be said of other complex software systems—operating systems, Web servers, and the like. Because the optimal configuration of such a system is typically workload-dependent, system designers often include a set of parameters called knobs that can be used to reconfigure the system for a particular workload. But the process of tuning these knobs—finding the optimal knob settings for a given workload—is complicated enough that it typically requires a "tuning guru" with extensive knowledge and experience. Moreover, because the optimal settings tend to change over time as workload characteristics change, the guru needs to be available to continually retune the system. As a result, manufacturers of these systems would need to "ship a tuning guru" with every copy of the software to ensure good performance.

In light of the challenges involved in manual software tuning, many have called for the development of software systems that are *self-tuning* or *knob-free* [Bar93, Ber98, Bro93, Cha00, *inter alia*]. Such systems would be able to determine their own optimal knob settings for a given workload and to reconfigure themselves as needed in response to changing conditions. The idea of self-tuning software systems is not new [Bad75, Ble76], but the need for them is increasing as a result of several trends. First, the complexity of software systems continues to grow, which makes manual tuning more and more difficult. Second, complex software systems increasingly are used by organizations and individuals who do not possess the necessary tuning expertise and who cannot afford to hire someone who does. Even large organizations suffer from both a shortage of qualified people and the growing human costs associated with computer systems. Alan Ganek, a vice-president at IBM, projected in April of 2002 that these human costs—which were then approximately equal to equipment costs—would be twice as large as equipment costs in five or six years [Kan02]. Third, software systems are increasingly being embedded in a wide variety of devices, and systems running in such contexts must, by necessity, be self-tuning [Ber98, Sel99a]. Given all of these trends, there is clearly a need for an automated approach to software tuning.

A number of prior research efforts have addressed the problem of automating the software-tuning process; Chapter 2 includes an overview of this research. In addition, a number of commercial software developers have added a limited degree of self-tuning to their products [Cha99, Sch99, Spi98, Zil01]. However, much remains to be done before software systems are fully self-tuning. For one thing, most of the prior work on automated tuning has attempted to adjust individual knobs in isolation, whereas a comprehensive approach to automated tuning will need to be capable of tuning multiple knobs simultaneously—taking into account interactions between the knobs and ways in which they jointly affect the performance of the system. In addition, most previous efforts have focused on a specific algorithm or module, without attempting to provide general guidance about how to automate the software-tuning process. To facilitate the construction of self-tuning systems, what is needed is a *methodology* for automated software tuning—a systematic approach that can be applied to an arbitrary software system.

This thesis proposes just such a methodology, one based on the use of probabilistic reasoning and decision-making techniques that have been developed by researchers in artificial intelligence, operations research, and other related fields. In particular, the methodology proposed here employs the influence diagram formalism

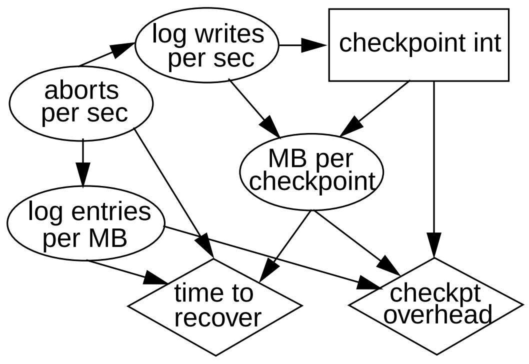


Figure 1.1. An example of an influence diagram.

[How84] and related learning and inference algorithms [Hec95, Sha86] to predict the system's performance under various combinations of workload characteristics and knob settings, and to thereby determine the optimal knob settings for a given workload. Influence diagrams are graphical, probabilistic models that allow decision-makers to reason about the variables that influence the outcome of their decisions. In particular, influence diagrams model probabilistic dependencies among the relevant variables, and they exploit conditional independencies to simplify both the specification of the model and the process of inferring the expected outcomes of a particular set of decisions.

In applying influence diagrams to software tuning, the decisions are the knob settings, and the relevant variables include workload characteristics (e.g., degree of concurrency), underlying performance statistics (e.g., number of cache misses), and overall performance measures (e.g., throughput or, conversely, measures of events such as page faults that reduce performance). For example, Figure 1.1 shows a possible influence diagram for tuning the frequency with which checkpoints are taken in a database system. The rectangular *checkpoint int* node is an example of what is known as a *decision node*; it represents the knob being tuned. The diamond-shaped *time to recover* and *checkpt overhead* nodes are *value nodes*; they represent the values that the tuning process seeks to optimize. Finally, the oval-shaped nodes are *chance nodes*; they represent random variables that are relevant to the tuning process. Arcs are used to connect these nodes in ways that provide information about how the nodes are related; if there is an arc from node *A* to node *B*, *A* is referred to as a *parent* of *B*. More information about the meaning of these nodes and arcs is provided in Chapter 3.

An influence diagram associates a set of parameters with each node. These parameters include a set of expected values for each value node—one parameter for each assignment of values to the node's parents—and one or more probability distributions for each chance node. For a chance node without parents, there is a single distribution over the possible values of the variable; for a chance node with parents, there is a conditional probability distribution for each assignment of values to its parents. By applying well-known techniques for learning these parameters from training data [Hec95], we can effectively learn a mapping from combinations of workload characteristics and knob settings to expected performance, and thereby determine the optimal knob settings for a given workload.

A tuning methodology based on influence diagrams takes advantage of the knowledge of domain experts about how the relevant variables are related—but this expertise is only needed once, when the structure of the influence diagram is designed by the software developer. Once the appropriate model structure is determined, it can be used as the foundation of a tuning module that runs without human intervention. This module can learn the initial parameters of the model using statistics gathered from actual or simulated workloads, even if the model includes variables that cannot be directly measured or estimated. Over time, newly gathered statistics can be used to modify these parameters, allowing the tuning module to refine its performance estimates and to adapt to changes in the environment in which the software system runs. And as the workload imposed on the system varies over time, the tuning module can use the influence diagram to determine the optimal knob settings for each workload—including previously unseen combinations of workload characteristics—and adjust the knobs accordingly. In short, the methodology presented in this thesis provides an effective, automated approach to software tuning.

1.1 Overview of the Thesis

The next three chapters provide the necessary background material for the thesis. Chapter 2 presents a more complete overview of the problem of automated software tuning. It begins by positioning the software module that determines the optimal knob settings within a high-level architecture for automated software tuning and discusses the criteria that this module should meet. It then offers a taxonomy of three possible approaches to automating the software-tuning process and discusses representative examples of prior work using each of these approaches. The third of these approaches, model-based tuning, is the one that I adopt in this thesis, with an influence diagram serving as the model.

Influence diagrams are essentially a tool for decision-making under uncertainty. Chapter 3 provides an overview of the concepts from probability and decision theory that are needed to understand the semantics of influence diagrams and how they are used to infer optimal decisions. The chapter then explains the structure and parameters of an influence diagram, focusing in particular on the conditional independence relationships that are encoded in the model's structure. Finally, it reviews the standard algorithm for evaluating an influence diagram to determine the optimal decisions.

Chapter 4 introduces the Berkeley DB embedded database system [Ols99, Sle01], the software system that I use to assess the effectiveness of an influence-diagram-based approach to software tuning. The chapter discusses the essential features of this software system, including the four knobs that I tune and the characteristics of the workloads that I consider. It also introduces the structure of the influence diagram that I have developed to tune the system.

Chapter 5 presents the actual software-tuning methodology—a step-by-step approach for using an influence diagram for software tuning. It includes guidelines for designing the structure of the model, as well as explanations of how to learn the initial parameters of the model from training data and how to update the values of these parameters over time. This chapter also addresses three challenges associated with using an influence diagram for tuning: incorporating performance measures in the model, dealing with continuous variables in the model, and estimating parameters for which no training data is available.

In developing an influence diagram for software tuning, one or more performance measures must be included as part of the model; the influence diagram is used to determine the knob settings that optimize these performance measures for a given workload. Ideally, we would like to be able to use a single performance measure—e.g., the throughput of the system—so that the influence diagram can simply optimize the value of that measure. However, this approach is often impractical. Because the number of parameters associated with a node in an influence diagram is exponential in the number of parents of the node, it is important to limit the number of parents associated with any one node. A single, overarching performance measure will typically have a large number of parents, and thus the processes of learning its parameters and performing inference using the model can become intractable. As a result, it is often necessary to use multiple performance-related nodes—each with a smaller number of parents—and to optimize the sum of the expected values of these nodes. Chapter 5 explains how regression can be used to learn appropriate weights for these separate nodes so that optimizing their sum is equivalent to optimizing the overall performance measure.

The challenge posed by continuous variables in an influence diagram stems from the fact that current algorithms and tools for working with influence diagrams are limited in their ability to handle such variables. As a result, each continuous variable in the model must be *discretized* by dividing its range of possible values into subranges, each of which is treated as a single value. Determining an effective discretization for a single variable is a non-trivial problem, and the need to discretize multiple variables simultaneously makes the problem that much more difficult. In order for an influence-diagram-based approach to software tuning to be truly automated, we need a general, systematic method for learning the appropriate discretizations from training data. Otherwise, we will simply replace one tuning problem (tuning the knobs) with another (tuning the discretizations). Fortunately, Friedman and Goldschmidt [Fri96] have addressed the problem of learning discretizations in the context of another type of graphical, probabilistic model, and Chapter 5 presents a discretization method that builds on this prior work.

As discussed in the previous section, if a node in an influence diagram has one or more parents, it will also have parameters associated with each assignment of values to those parents. When a given assignment of values to a node's parents does not appear in the training data, it becomes more difficult to learn accurate estimates of the associated parameters. To address this problem, Chapter 5 presents an algorithm for estimating these parameters that exploits the nature of the relationships that typically exist between a node and its chance-node parents in an influence diagram developed for software tuning. Because these relationships are typically monotonic—i.e., increasing the value of a node's parent either never decreases or never increases the value of the node itself—it is often possible to use the values of parameters associated with parent-value assignments that are seen in the training data (*seen parameters*) to devise constraints on the values of parameters associated with parent-value assignments that are not seen (*unseen parameters*). The algorithm uses these constraints to augment a nearest-neighbor approach to parameter estimation, in which an unseen parameter is estimated by averaging the values of the seen parameters whose parent-value assignments are determined to be "nearest" to the unseen parent-value assignment.

Chapter 6 addresses another challenge involved in using an influence diagram—or any type of model, for that matter—for software tuning. Although the data needed to train the model can be collected online as the system being tuned handles actual workloads, doing so can degrade the system's performance as non-optimal knob settings are tried. Therefore, it can be preferable to collect the data offline—for example, by running the same system on a separate but similar hardware platform. To perform this type of offline training, a workload generator may be needed to create synthetic workloads that simulate the workloads for which the system is being tuned. Chapter 6 discusses the design of workload generators for software tuning, including a technique for ensuring that a generator captures the steady-state performance of the system for the workloads being simulated. It also presents *db_perf*, a workload generator for use with Berkeley DB.

Chapter 7 presents experimental results that demonstrate the ability of the methodology presented in the thesis to determine optimal or near-optimal knob settings for Berkeley DB for a wide range of workloads—including previously unseen combinations of workload characteristics. The data gathered for these experiments is also used to explore additional issues, including the efficacy of the method proposed in Chapter 5 for estimating unseen parameters.

Finally, Chapter 8 concludes the thesis by assessing the methodology, discussing possible future directions, and summarizing the lessons learned.

1.2 Contributions

This dissertation makes the following contributions:

- It presents a methodology for automated software tuning that employs a graphical, probabilistic model known as an influence diagram and related learning and inference algorithms. The methodology offers step-by-step guidance on how to use these probabilistic reasoning and decision-making techniques to tune an actual software system.

- It provides an example of using this methodology to simultaneously tune four knobs from the Berkeley DB embedded database system. The influence diagram developed for this system recommends knob settings that offer considerable performance improvements over the default knob settings on a varied set of workloads, and it outperforms an alternative approach that uses regression models to determine the optimal knob settings. In addition, the influence diagram is able to effectively generalize from experience, predicting optimal or near-optimal knob settings for previously unseen combinations of workload characteristics.

- It explains how a workload generator can be used to facilitate the software-tuning process. It presents an example workload generator for Berkeley DB and addresses issues involved in designing and using such a generator. It also proposes a technique for ensuring that the workload generator captures the steady-state performance of the system.

- It addresses several challenges involved with using an influence diagram for software tuning and proposes novel methods for meeting these challenges. These methods include: a regression-based approach for incorporating performance measures in an influence diagram in a tractable manner; an algorithm (based on the work of Friedman and Goldschmidt [Fri96]) for learning effective discretizations of continuous variables in an influence diagram; and an algorithm for estimating influence-diagram parameters for which no training data is available.

In summary, this thesis demonstrates that probabilistic reasoning and decision-making techniques can be used as the basis of an effective, automated approach to the tuning of complex software systems, and it provides practical advice on how to apply these techniques to tune an actual system.

Chapter 2

Automated Software Tuning

To automate the process of tuning a complex software system, we need to construct a software module that is capable of determining the optimal knob settings for a given state of the system. This chapter begins by positioning this module within a high-level architecture for automated software tuning and delineating the criteria that it should meet. The chapter continues with an overview of three possible approaches to automated software tuning and a discussion of the role that a workload generator can play in two of these approaches. It then presents a justification of the approach that I have selected for the work in this thesis, and it concludes with a survey of related work.

2.1 High-Level Architecture

There can be a variety of methods for incorporating self-tuning capabilities into a software system. I will assume for the sake of argument that there is a distinct module known as *the tuner* that is responsible for determining the appropriate knob settings for the system. The tuner takes as input a description of the relevant aspects of the current state of the system, and, in light of that description, makes the necessary adjustments to each of the knobs being tuned.

As the system runs, a separate module known as *the monitor* periodically gathers statistics about the state of the system. Depending on the design of the tuner, these statistics may be used to update the tuner's internals over time (e.g., by improving the models that it uses to determine the optimal knob settings). When the monitor detects a significant change in the state of the system, it feeds a description of the new state to the tuner, which determines if the system should be retuned and alters the knob settings accordingly. The tuner may also request additional samples of the statistics from the monitor as part of the process of determining what knob adjustments are needed.

There are a number of issues that need to be addressed in order to implement the monitor, such as determining an appropriate interval between samples of the statistics and distinguishing substantial changes in the state of the system from natural statistical fluctuations. For the most part, these issues have already been successfully addressed by others (e.g., by Brown [Bro95]). In this thesis, I focus exclusively on the design of the tuner.

In the sections that follow, I refer frequently to the *environment* in which the tuner runs. This term encompasses all of the factors that can affect the accuracy of the tuner's decisions. Examples of such factors include aspects of the software system that are not under the control of the tuner and features of the hardware platform on which the system runs. One crucial aspect of the environment is the types of workloads experienced by the system, or, more specifically, the distributions of possible values for the variables used to characterize the workloads. A tuner is typically calibrated to work in a particular environment, and it may need to be recalibrated as aspects of the environment change over time.

2.2 Criteria

There are a number of criteria that an effective tuner should meet. They include accuracy and reasonable time costs, as well as the ability to handle previously unseen workloads, tune multiple knobs simultaneously, respond to changes in the environment, and run with little or no human intervention. This section describes each of these criteria in turn.

An accurate tuner should recommend knob settings that optimize the system's performance most, if not all, of the time. Although perfect accuracy may not be possible, any deviations from the optimal level of performance should be small.

Table 2.1. Assessing a tuner's accuracy. To assess the accuracy of a tuner's recommended knob settings for a given workload, the workload is run once using each possible combination of knob settings. The results of one such set of runs are shown in the table below for eight knob settings labelled S1 through S8. The slowdown of a combination of knob settings is the percent difference between the performance obtained using those settings and the best performance seen during the set of runs. For example, the slowdown associated with the S6 settings is (21.22 - 19.71)/21.22 = 7.12 %. A tuner's recommended knob settings are considered accurate if their slowdown is less than five percent.

settings	**throughput** (txns/s)	**slowdown**
S1	19.66	7.35 %
S2	18.71	11.83 %
S3	21.13	0.42 %
S4	21.22	0
S5	21.21	0.05 %
S6	19.71	7.12 %
S7	19.40	8.58 %
S8	19.97	5.89 %

Accuracy can be assessed by trying all possible combinations of knob settings for a given workload and seeing whether the tuner's recommended settings actually produce the best performance. Ideally, multiple measurements would be gathered for each knob setting, so that it would be possible to determine whether the observed differences in performance are statistically significant. However, it is often impractical to obtain more than one measurement per setting. As a result, the following methodology will be used to assess the accuracy of a tuner's recommended knob settings for a given workload. The workload will be run once using each possible combination of knob settings to produce a single performance value for each setting. The *slowdown* of a given combination of knob settings will be computed by taking the percent difference between the performance achieved using those settings and the best performance seen during the entire set of runs for the workload, as shown in Table 2.1. Knob settings will be considered *near-optimal* if their slowdown is greater than zero but less than five percent, and a tuner's recommendations for a given workload will be considered *accurate* if they are either optimal or near-optimal.

In addition to being accurate, a tuner should have reasonable time costs. Depending on the nature of the tuner, there can be as many as two different time costs to consider: time spent training the tuner before it is deployed, and time spent determining the optimal knob settings for a given workload once the tuner is deployed. Generally, the former time costs are less problematic because they are amortized over the life of the tuner.

In order for a tuner to be both accurate and timely, it needs to be able to make accurate tuning recommendations for previously unseen workloads—i.e., workloads for which it has no prior performance measurements. Otherwise, the tuner would need to try a large number of different knob settings for each new workload, and doing so would result in unreasonable time costs. There may be environments in which the number of possible workloads is small enough that a tuner does not need to handle previously unseen workloads, but a general methodology for automated tuning needs to construct tuners that are capable of doing so.

An effective tuner should also be able to tune multiple knobs simultaneously. Much of the prior work on automated software tuning has focused on individual knobs in isolation, but because the optimal setting for a given knob can depend on the settings of other knobs in the system, it is important to be able to tune groups of knobs together. There is almost certainly a limit to the number of knobs that a single tuner can reasonably handle. In the context of database tuning, for example, Chaudhuri and Weikum [Cha00] argue that it is impossible for a single tuner to handle all of the knobs present in today's database management systems, and they propose a new paradigm in which such systems are composed of simple components that are easier to model and tune. However, even simple components may have several knobs, and high-level tuners will also be needed to tune the interactions among the components. Therefore, effective software tuners will still need to be capable of adjusting multiple knob at the same time.

A tuner should also be responsive to changes in the environment, revising its recommended knob settings as necessary in light of these changes. For example, if a tuner uses one or more models to determine the optimal knob settings for a given workload, these models should be updated over time to ensure that the tuning recommendations remain accurate. Even if no significant changes occur in the environment, the tuner should still revise its recommendations as needed to reflect the additional experience that it acquires over time.

Lastly, the tuner should be truly automated, meaning that it should require little or no effort on the part of users of the software system. The input of domain experts may be needed during the initial development of the tuner, but it should be possible to ship the tuner with the software system and have it work “out of the box.” This criterion does not rule out the possibility of human intervention. For example, it may make sense to allow a user to provide input that improves the tuner’s performance (e.g., by instructing it to gather extra training data after a hardware upgrade). However, the tuner should function reasonably without this input.

2.3 Possible Approaches

There are a number of approaches that can be used to automate the tuning of a software system. This section describes three classes of techniques: ones based on empirical comparisons, feedback mechanisms, and analytical models, respectively.

2.3.1 Empirical Comparisons

One way of determining the optimal knob settings for a given workload is to empirically compare the performance of some or all of the possible knob settings. In other words, the tuner can vary the knob settings according to some search technique, measure the performance of the system for each of the settings visited by the search, and compare the resulting performance measurements to determine the settings that are optimal for that workload. The measurements needed for the comparisons can be made on the software system itself, or they can be obtained through the use of simulations.

In its simplest form, a tuning approach based on empirical comparisons involves trying all possible knob settings to see which ones work best. However, for tuning problems with a large number of possible settings, an exhaustive comparison may require an unreasonable amount of time, and some method of pruning the space of possible settings is typically employed. Various search techniques can be used for this purpose. Section 2.6.1 provides examples of some of them.

If the workloads faced by the system are easily characterized and known in advance, it may be possible to carry out the necessary empirical comparisons before the tuner is installed, allowing it to predetermine the optimal settings for each workload faced by the system. If the workloads are not known in advance or are likely to change over time, or if it is not possible to easily reproduce the workloads ahead of time, the tuner will need to perform empirical comparisons whenever a new workload arises. It may also be necessary to periodically conduct new comparisons for previously seen workloads so that the tuner can adapt to changes in the environment.

A key limitation of tuning approaches based on empirical comparisons is that it can be difficult to find accurate knob settings in a reasonable amount of time. This is especially problematic if the empirical comparisons must be performed dynamically as new workloads appear. In addition, an approach based on empirical comparisons is unable to generalize from experience, because the results of a given round of comparisons are only applicable to one particular workload.

2.3.2 Feedback Mechanisms

In a feedback-driven approach to software tuning, the tuner iteratively adjusts the knobs based on the values of one or more performance metrics; typically, what matters is how the current value of each metric compares to some critical value. For example, if a tuner were attempting to maintain a particular response time for a database system, its knob adjustments would be based on how the system's current response time compares to some response-time goal. After making a given set of knob adjustments, the tuner observes the new values of the performance metrics and uses them to determine whether any additional adjustments are needed. This repeated cycle of observations and adjustments is often referred to as a *feedback loop*, because the effects of one set of adjustments on the performance metrics are fed back into the tuner and used to determine the next set of adjustments. The mathematical methods of classical control theory [Fra01] can be used to determine the magnitudes of a given set of knob adjustments, but simpler approaches—including ones based on heuristics and estimation techniques—have typically been used in applications of feedback to software tuning.

Like tuning methods based on empirical comparisons, feedback-based tuning methods also involve measuring the performance obtained using various knob settings. However, empirical-comparison-based tuning methods *compare* the performance of various knob settings *on a particular workload* and use these comparisons to determine the optimal knob settings for that workload. Feedback-based methods, on the other hand, simply map the current values of the guiding performance metrics to a set of knob adjustments and repeat this process until no further adjustments are needed. Such methods never explicitly compare the performance of the possible knob settings or consider the workload for which the system is being tuned.

Although feedback mechanisms have been used effectively to create self-tuning systems, they have a number of disadvantages. First, it is often difficult to use feedback to tune multiple knobs simultaneously. With a single knob, the tuner only needs to decide whether to increase or decrease the current knob setting, whereas adjustments to multiple knobs require an understanding of how the knobs interact and how they jointly affect the performance of the system. Second, a large number of knob adjustments may be needed before the optimal knob settings are reached, and this may lead to unreasonable runtime costs for the tuner.[1] Third, it is difficult to apply feedback mechanisms to tuning problems whose performance metrics do not have obvious critical values. For example, if we are attempting to maximize the throughput of a database system, the critical (i.e., maximal) throughput value is not known ahead of time, and the impact of the knobs on throughput may be complicated

1. In addition, some knobs may be expensive enough to adjust that it would be impractical to iteratively adjust them in a search for the optimal settings, even if only a small number of adjustments are needed. For example, in my work with Berkeley DB, I consider adjustments to knobs that involve reconfiguring the layout of a database (Sections 4.2.1 and 4.2.2). Such reconfigurations are extremely costly, so it is important to minimize the number of adjustments to these knobs.

enough that the feedback mechanism would have no way of knowing when the optimal settings had been reached.

Provided that a tuning problem is amenable to the use of feedback and that the optimal knob settings can be reached in a reasonable amount of time, feedback-based tuning methods have the advantage of being able to adapt easily to changes in the environment. In addition, such methods are able to perform well without any training (and thus can handle previously unseen workloads), although experimentation may be needed to determine the critical values of the performance metrics or to devise the heuristics used to guide the knob adjustments. Feedback mechanisms are also well-suited to tuning problems that involve rapidly fluctuating workloads. In such cases, we can eliminate the monitor process entirely and simply allow the feedback-based tuner to continuously make knob adjustments based on the changing values of its guiding performance metrics.

2.3.3 Analytical Models

Analytical models form the basis of a third approach to automated software tuning. In particular, we can construct a tuner using models that predict the system's performance for any combination of workload characteristics and knob settings. When confronted with a particular workload, the tuner can use these models to determine the knob settings that maximize the expected performance of the system for that workload and tune the knobs accordingly. Like tuners based on empirical comparisons, model-based tuners compare the performance of various knob settings on a given workload, but the comparisons are based on the models' predictions, not on measurements of the actual or simulated performance of the system. And like any tuner based on comparisons, a model-based tuner may need to employ special search techniques when the space of possible knob settings is large enough that exhaustive comparisons are impractical.

There are a number of types of models that can be used to construct this type of tuner, including statistical models based on regression and the graphical, probabilistic models that I employ in this thesis. Each type of model has an associated set of parameters, and the values of these parameters are typically learned from a collection of training examples—each of which consists of statistics capturing the workload characteristics, knob settings, and performance of the system over some interval of time. As the system runs, the model's parameters can be updated to reflect newly collected statistics, allowing the tuner to make better predictions and to adapt to changes in the environment.

Provided that its analytical models reflect the current enviroment, a model-based tuner can determine the optimal knob settings for a given workload after a single set of computations, avoiding the series of iterative knob adjustments that a feedback-based tuner often requires. In addition, a model-based tuner is able to generalize from experience, using its models to predict the performance of previously unseen workloads and to thereby determine the optimal knob settings for those workloads. A tuner based on empirical comparisons, by contrast, must actually test a series of knob settings when confronted with a previously unseen workload—either by performing a series of simulations or by actually configuring the system to use each of the knob settings being tested. Therefore, a model-based tuner should typically have lower runtime costs than either of the other types of tuner.

However, there are potential drawbacks of model-based approaches to tuning. First, they typically require an initial expenditure of time to train the model; in such cases, the model's predictions are not accurate until sufficient training data has been collected. Moreover, the process of collecting training examples from an already deployed software system will often degrade the system's performance, because the need to see many different combinations of workload characteristics and knob settings means that some of the knob settings chosen during training will necessarily be non-optimal, and they may lead to significantly poorer performance than the system's default settings. Second, it can be difficult to devise an accurate performance model for a complex software system. As the number of variables relevant to the tuning problem increases—and, in particular, as we attempt to tune an increasing number of interacting knobs—the challenge of developing an accurate model also increases. Although these potential drawbacks of model-based approaches are significant, the next two sections explain how they can be overcome to a certain degree.

2.4 Using a Workload Generator for Tuning

Tuners based on both empirical comparisons and analytical models need to see various combinations of knob settings and workload characteristics before they can determine accurate knob settings for a given workload. A tuner that performs empirical comparisons needs to measure the performance of a single workload using different knob settings, and a model-based tuner needs to be trained with statistics gathered from examples involving various combinations of knob settings and workload characteristics. Both the comparison and training processes can be conducted online as a deployed system handles workloads, but doing so can have a negative impact on the system's performance as non-optimal settings are tried. Therefore, it may be preferable to conduct the comparisons or training *offline*—either on the system itself before it is actually deployed, on a version of the system running on a separate but similar hardware platform, or on the deployed system during periods of idleness.

To perform an offline evaluation of the effectiveness of one or more combinations of knob settings on a given workload, the workload must be reproduced in some way. For some software systems, it may be possible to replicate the workload more or less exactly using traces or logs obtained from the running system. For other systems, the process of reproducing the workload may be more difficult, either because sufficiently detailed traces are not available or because traces alone are insufficient for generating the workload in question. For example, when tuning a database system or a Web server, it may be necessary to reproduce the behavior of multiple, concurrent clients of the system, which cannot be accomplished by simply rerunning the trace. Therefore, some sort of *workload generator* may be needed to create synthetic workloads that simulate the workloads for which the system is being tuned. Chapter 6 provides more detail about the use of workload generators for tuning.

A workload generator can be used in conjunction with tuners that use either empirical comparisons or models, but a model-based tuner is often a better fit. Using a model-based approach means that the generated workloads do not need to precisely match the actual workloads run on the system. The training examples obtained using the workload generator may be seen as providing reasonable starting estimates of the model's parameters, and the parameters can be refined incrementally over time as the system uses the model for tuning. Moreover, because a model-based tuner can generalize from its training examples, the workload generator can simply sample from a wide range of possible workloads, and thereby avoid the need for exact characterizations of the workloads that run on the system. With a tuner that performs empirical comparisons, on the other hand, the workloads produced by the generator must correspond to actual workloads that the system will face.

2.5 Choosing an Approach to Automated Tuning

Given the three possible approaches to automated software tuning presented in Section 2.3, the choice of which approach to take will depend in part on the nature of the system being tuned and the goal of the tuning process. For example, if a system has a small number of regularly recurring workloads, an approach based on empirical comparisons may make the most sense. If the goal of tuning is to enforce some type of performance guarantee in the face of rapidly changing workloads, an approach based on feedback may work best. But if the goal of tuning is to optimize a given measure of the system's performance, and if the workload or environment can vary substantially over time in unpredictable ways, a model-based tuning approach has a number of advantages. These include the ability to determine the optimal knob settings without performing a series of iterative adjustments (unlike a feedback-based approach) and the ability to generalize from experience to previously unseen workloads (unlike an approach based on empirical comparisons).

As discussed in Section 2.3.3, there are two potential drawbacks of a model-based approach. The first is the possibility of performance degradations during the training of the model (a limitation shared by methods that employ empirical comparisons). However, as discussed in Section 2.4, it may be possible to avoid this problem by training the tuner offline, perhaps through the use of a workload generator.

The second potential drawback of a model-based approach is the difficulty of devising an accurate model. A feedback-based approach—which requires the determination of critical values and of heuristics for adjusting the knobs—shares this limitation, at least to a certain degree. Although this drawback is a significant one, choosing the right type of model can reduce the difficulties involved in building a good model. In particular, the influence diagram formalism presented in Chapter 3 has features that facilitate the modeling of complex decision problems like the ones involved in software tuning. In particular, the ability of influence diagrams to explicitly model the interactions between the relevant variables in ways that take advantage of an expert's knowledge of the underlying system should allow these models to handle more complex tuning problems than models that do not explicitly encode such interactions. Similar graphical, probabilistic models have already been used to model systems of significant complexity [e.g., Pra94], and there is active research into further increasing the expressive power of these models [Pfe99, Pfe00].

Given the advantages of a model-based approach to software tuning and the potential for workload generators and graphical, probabilistic models to overcome this approach's limitations, I have adopted a model-based approach using an influence diagram for the work in this thesis. Chapter 3 provides more detail about influence diagrams and the probability theory and decision theory on which they are based, and Chapter 5 presents a methodology for using an influence diagram for software tuning.

2.6 Related Work

A considerable amount of research has been devoted to automating the tuning of software systems. This section presents some representative examples of this prior work, concentrating primarily on research in database systems and operating systems. The examples are grouped according to the type of tuning approach that they employ.

2.6.1 Work Based on Empirical Comparisons

Reiner and Pinkerton [Rei81] present an approach that uses empirical comparisons to dynamically adapt the knob settings in an operating system. The comparisons are performed online during a special experimental phase. Each time the state of the system changes significantly, one of the candidate knob settings is chosen randomly without replacement and used to configure the system, and the resulting performance of the system is measured. If a given system state persists long enough, multiple settings may be tried. Over time, the system acquires enough data to determine the optimal settings for each possible state of the system, and it concludes the experimental phase. Thereafter, the knobs are dynamically adjusted according to the optimal settings for each system state, and additional experiments are occasionally conducted to allow the system to adapt to changes in the environment. The authors obtained a small performance improvement in experiments that used this methodology to tune two scheduling parameters on a time-sharing system. They did not attempt to assess the degree to which the the experimental phase degraded the system's performance.

Seltzer and Small [Sel97] propose a methodology for constructing an operating system kernel that monitors its own performance and adapts to changing workloads. Their approach includes the use of comparisons to determine the optimal policies (e.g., the optimal buffer-cache replacement policy) for handling a particular workload. Because their methodology was developed for an extensible operating system that allows many of its modules to be replaced on a per-process basis [Sel96], the comparisons can be performed by downloading a special simulation module into the kernel for each of the possible policies and using these modules to process actual workload traces. Simulation modules maintain their own separate state and do not affect the global state of the system, and thus the kernel can continue to use the current default policies while the simulations explore possible alternatives. This significantly reduces the potential negative effects of tuning on the performance of the system. However, most operating systems do not provide the extensibility needed to conduct this type of simulation.

Feitelson and Naaman [Fei99] propose a methodology based on empirical comparisons for constructing self-tuning operating systems. Instead of exhaustively comparing all possible knob settings, their methodology uses genetic algorithms to guide the search for good settings. In each round of comparisons, several candidate settings are tested using simulations that are conducted when the system is idle. Settings that perform well in a given round of simulations are more likely to persist into the next round either in whole or in part, as various transformations combine and modify the current group of candidate settings according to their relative performance. By favoring the knob settings that perform best on the simulations, the genetic algorithms should eventually converge on knob settings that improve the performance of the system. The authors' only validation of this approach involves tuning a parameterized scheduling algorithm that can be evaluated in isolation from the rest of the operating system, and thus they are able conduct the comparisons offline. They acknowledge that it may be difficult to conduct good simulations of other aspects of an operating system, and that the overhead imposed by running simulations on the system as it is tuned would need to be assessed.

Vuduc et al. [Vud01] discuss methods for determining the best implementation of a library subroutine for a particular platform and set of input parameters. They note that such methods typically use empirical comparisons to find the optimal implementation, and that application-specific heuristics are often used to reduce the number of settings that are compared. To complement the use of such heuristics, the authors propose using performance statistics gathered during the search for the optimal settings to decide when the search should be stopped. Specifically, they employ statistical methods to estimate the probability that the performance of the best implementation seen thus far differs from the performance of the optimal implementation by more than some user-specified value. When that probability falls below a second user-specified value, the search is halted.

The tuning approaches discussed in this section—like all approaches that base their tuning recommendations on empirical comparisons—are unable to generalize from experience. As a result, additional training is needed when new workloads arise—something that is not necessary in the model-based approach that I propose. Seltzer and Small's approach mitigates the impact of this additional training on the system's performance by using special simulation modules, but these modules are not available on most systems. As a result, approaches based on empirical comparisons can degrade the performance of the system when new workloads arise, unless they defer the additional training to periods when the system is idle or run the training offline. In any case, the need for additional training means that these systems are unable to make timely tuning recommendations for previously unseen workloads. The model-based approach presented in this thesis, on the other hand, can recommend optimal or near-optimal tunings for new workloads in a reasonable amount of time by generalizing from previously seen workloads.

2.6.2 Work Based on Feedback Mechanisms

Weikum et al. [Wei94] use a feedback-driven approach to tune a database system's multiprogramming level (MPL), a knob that limits the number of concurrent accesses to the database. They base the adjustments to this knob on a measure of lock contention in the system: when this metric exceeds a critical value, the MPL is reduced, and when it drops below the critical value, the MPL is increased. The authors determined the critical value experimentally, and they claim that it does not need to be fine-tuned; rather, they present the results of experiments demonstrating that there is a range of critical values that perform well on a wide range of workloads. The authors also present results showing that their approach allows the system to provide acceptable response times under extremely high loads.

Kurt Brown et al. [Bro94, Bro95, Bro96] use mechanisms based on feedback to tune knobs related to memory management and load control in a database system. The objective of the tuning is to meet the response-time goals of individual workload classes in a multiclass database workload, and the knobs are adjusted until either these goals are met or until the tuner determines that they cannot be met. The workload classes are tuned separately, and heuristics are used to address interdependencies between classes. Depending on the nature of a given class's memory usage, either one or two knobs are adjusted, and estimates and heuristics are used to guide the adjustments. Simulations used to validate the authors' proposed mechanisms show that both the one-knob and two-knob tuners are able to meet the goals of a variety of workloads, although the authors acknowledge that it can take a long time to achieve the response-time goals of certain types of workloads. The model-based approach presented in this thesis, on the other hand, is able to avoid the potentially lengthy series of iterative adjustments that a feedback-based approach may require.

Microsoft's SQL Server employs feedback-based tuning to adjust the size of its cache of database pages [Cha99]. The adjustments are based on the amount of free physical memory in the system: when the number of free memory pages drops below one threshold, the size of the database cache is reduced; when the number of free pages exceed a second threshold, the cache size is increased. The authors of the paper that mentions this use of feedback-based tuning do not explain how the threshold values are chosen. The authors also outline plans to use feedback to adjust the number of pages read into the cache when the system performs read-ahead (i.e., when it proactively reads in pages that it anticipates will be accessed soon).

In the SEDA framework for highly concurrent Internet applications [Wel01], applications consist of a series of components called stages that are connected by queues of events, and feedback-driven tuners called resource controllers are used to dynamically adjust each stage's resource usage. For example, a stage's thread pool controller tunes the number of threads associated with the stage, adding a thread when the length of the stage's queue rises above some threshold and removing a thread when it sits idle for longer than a second threshold. SEDA's resource controllers operate at the application level, without needing to be aware of the resource management policies of the underlying operating system. It is unclear how sensitive these controllers are to the thresholds used to guide the knob adjustments, but the authors present results that demonstrate the ability of the resource controllers to effectively adapt to increasing load.

Feedback mechanisms have also been widely applied to resource management problems in operating systems, including CPU scheduling [Cor62, Mas90, Ste99] and network congestion and flow control [Jac88, Kes91]. To facilitate the use of feedback-based tuners in this domain, Goel et al. [Goe99] have developed a toolkit of simple, modular feedback components that can be combined and reused. However, all of their example tuners adjust a single knob, and it is unclear whether their components can effectively handle software-tuning problems that involve the simultaneous adjustment of multiple knobs.

More generally, almost all of the examples presented in this section involve tuning individual knobs in isolation. The one exception is the work of Brown et al. [Bro94, Bro95, Bro96], who tune two knobs simultaneously for one class of workloads. However, Brown himself explicitly mentions the difficulty of developing a feedback-based tuner that controls more than one knob [Bro95, Section 5.2], and he is forced to conduct extensive experimentation to devise the heuristics that his tuner uses to adjust two knobs in concert. The model-based approach presented in this thesis is capable of tuning multiple knobs simultaneously, as demonstrated by the experiments in Chapter 7, which involve four knobs.

Although feedback-based methods can, in theory, avoid the need for training and model building required by model-based methods, the examples presented above demonstrate that experimentation and the development of heuristics are often required to construct an effective feedback-based tuner. Even the examples that do not mention the need for this type of preliminary work would need some means of determining the threshold values that guide the tuner's knob adjustments.

2.6.3 Work Based on Analytical Models

Brewer [Bre94, Bre95] uses regression models to tune library subroutines. He employs linear regression but allows the independent variables to be nonlinear (e.g., an independent variable can represent the product of two or more of the parameters of the subroutine being optimized). Although this approach works well for tuning subroutines, it is unclear whether it would be possible to produce accurate regression-based performance models of large-scale software systems. The influence diagram models employed in this thesis, on the other hand, are able to explicitly model interactions between the relevant variables, and they should thus be able to provide more accurate models of complex systems. Section 7.4.3 compares the tuning recommendations of an influence diagram and with those of a set of regression models for four knobs from the Berkeley DB embedded database system. This comparison shows that an influence diagram outperforms regression models in this domain.

Matthews et al. [Mat97] use a model-based approach to tune a modified version of the log-structured file system (LFS). For example, they enable LFS to dynamically choose the better of two methods for performing garbage collection on the log that the system maintains on disk. Their models consist of simple formulas for estimating the cost or cost-benefit ratio of the possible knob settings; they are based on an understanding of the operations performed by the system and their associated costs. The models' only parameters are measurements of the costs of various operations on a particular disk. The authors assess the effectiveness of their approach through simulations of both LFS and the disk on which it resides. It is unclear how well such simple models would work on an actual system, or whether it would even be possible to predict the performance of more complex systems using such models. Here again, influence diagrams should be able to handle complex, multi-knob systems more easily.

In a relational database system, indices and materialized views are supplemental data structures that can be created in an attempt to speed up frequently occurring database queries. The AutoAdmin project has developed model-based techniques for automating the selection of which indices and materialized views to create [Cha97, Agr00]. The authors use the cost estimates of the database system's query optimizer as the model, and they develop novel methods for selecting which indices and materialized views to consider and for efficiently searching through the space of possible combinations of indices and materialized views. The internal models of the query optimizer are not discussed, and thus it would be difficult to transfer their approach to an arbitrary software system.

In the Odyssey platform for remote computing [Nob97], applications adapt to changes in resource availability and user goals by varying the *fidelity* with which they operate (e.g., the frame rate used by a streaming video application). Narayanan et al. [Nar00] augment Odyssey with a system that uses models to predict an application's resource usage as a function of the relevant input parameters and fidelity metrics, and to thereby recommend appropriate fidelity levels for a given operation. To avoid annoying the user, the initial training data is collected during a special offline mode in which a given operation is repeatedly run using randomly chosen fidelities and inputs, and the parameters of the models are refined as the system runs. These features of Odyssey are also found in the methodology proposed in this thesis, which advocates offline training (perhaps using a workload generator) and which includes a process for refining the parameters of the influence diagram over time. For their initial prototype, Narayanan et al. employ linear regression to derive the models, and they use linear gradient descent [Mit97] to update the models' coefficients over time. To determine the appropriate fidelities for a given set of inputs, the tuning system employs a gradient-descent solver. The potential limitations of regression-based models mentioned above also apply here.

Menascé et al. [Men01] use a tuner based on queueing network models [Laz84] to optimize the quality of service (QoS) of an e-commerce site. When their system detects that a QoS guarantee has been violated, it employs a hill-climbing search guided by the models' predictions to find the knob settings that yield the locally maximal QoS. The authors present results showing that their tuner, which adjusts four knobs, is able to maintain reasonable QoS values in the face of increasing load. However, although queueing network models work well in this domain—in which the knobs being tuned are directly connected to queues of requests waiting to be processed by a Web server and an application server—it is unclear whether they could form the basis of a general software tuning methodology. Influence diagrams, on the other hand, do not require that the software system be modeled as a series of queues.

Vuduc et al. [Vud01], after discussing how empirical comparisons can be used to determine the optimal implementation of a library subroutine for a given platform (see Section 2.6.1), note that the best implementation may depend on the input parameters. Therefore, they propose taking a set of several "good" implementations (possibly found using empirical comparisons) and using models derived from training data to determine which of these implementations is best for a given set of inputs. They experimentally compare the ability of three types of models—including the regression models proposed by Brewer and a statistical classification algorithm known as the *support vector method* [Vap95]—to choose between three candidate algorithms for matrix multiplication. The latter method has the best performance of the three, but it is unclear how well it would scale to tuning problems with more than one knob or, more generally, to problems with larger numbers of possible knob settings.

2.7 Conclusions

Section 2.3 outlined three possible approaches to automated software tuning. The choice of which approach to take will depend in part on the nature of the system being tuned and the goal of the tuning process. However, as argued in Section 2.5, a model-based approach that uses an influence diagram for the model seems like a good choice for situations in which the tuner needs to optimize a measure of the system's performance in the face of unpredictable workload variations and environmental changes.

The remaining chapters of this thesis present a methodology for automated software tuning that employs a tuner based on influence diagrams. Depending on the nature of the system being tuned, the methodology may also use a workload generator to obtain the data needed to train the influence diagram. Chapter 5 describes how influence diagrams can be used to construct an effective model-based tuner, Chapter 6 provides more detail about the use of a workload generator to produce the necessary training data, and Chapter 7 provides an example of using a workload generator and an influence diagram to tune an actual software system.

Chapter 3

Probabilistic Reasoning and Decision-Making

This chapter provides an overview of the probabilistic reasoning and decision-making techniques that are at the heart of the automated approach to software tuning presented in this thesis. The chapter begins with a review of concepts from probability theory and decision theory that are needed to understand the rest of the thesis. It continues with an introduction to influence diagrams, including a description of the structure and parameters of these probabilistic, graphical models and an explanation of how they can be used to determine optimal decisions. Chapter 5 explains in detail how to construct, train, and use a model of this type for software tuning.

3.1 Basics from Probability Theory and Decision Theory

Software tuning may be viewed as an example of decision-making under uncertainty. We are trying to decide which knob settings to use to maximize the performance of a software system for a given workload, and we need to make this decision in light of potentially uncertain knowledge about the state of the system and how the system will be affected by different settings of the knobs.[1] By employing probability theory and decision theory, we can represent our uncertainty about the relevant variables and determine the knob settings that maximize the expected performance of the system, given a particular set of workload characteristics.

1. Although this chapter refers to human decision-makers (“we”), the type of probabilitic reasoning and decision-making that is described can also be performed by an automated tuner. Chapter 5 explains how to construct such a tuner.

Consider, for example, a scenario in which we need to tune the frequency with which checkpoints are taken in a database system—a parameter known as the *checkpoint interval*. To determine the optimal checkpoint interval for a given workload, we need to reason about aspects of the workload and the system that are relevant to this decision. In addition to the checkpoint interval itself, the relevant variables might include the following:

- *checkpoint overhead:* some measure of the overhead of taking checkpoints (e.g., the average number of seconds that the system spends taking checkpoints in a given one-hour period).
- *time to recover*: the average number of seconds needed to recover from a failure such as a crash of the server machine on which the database system runs. Longer checkpoint intervals increase the time needed to recover.
- *log writes per sec:* the average number of times per second that the system writes a collection of log entries to the on-disk log file. The more log entries that are written to the log between checkpoints, the more time it will take to perform a checkpoint or to recover from failure.
- *log entries per MB:* the average number of log entries contained in one megabyte of the log file.
- *MB per checkpoint*: the average number of megabytes written to the log file between successive checkpoints.
- *aborts per sec*: the average number of database transactions that must be aborted every second. Aborts tend to increase the number of log writes, and they also affect the nature of the operations that must be performed during recovery.

When selecting a checkpoint interval for a given workload, we are uncertain about how the possible values for this knob will affect the performance of the system—as measured by some combination of the variables *checkpoint overhead* and *time to recover*. In addition, we may also be uncertain about the values of variables like *aborts per sec* that characterize the workload. Probability theory and decision theory allow us to quantify our uncertainty and to determine the best possible decisions given that uncertainty.

The following sections present an overview of the material from probability theory and decision theory that is needed to understand the thesis. A more comprehensive introduction to these topics can be found in a number of introductory texts [Dra67, Ros01, *inter alia*].

3.1.1 Joint and Marginal Probability Distributions

To represent and reason about a software-tuning problem, we first need to define the relevant variables. As part of this process, we need to specify, for each variable V, a corresponding set of possible values $\Omega(V)$. A variable is either *discrete*, meaning that its possible values can be counted, or *continuous*, meaning that it can take on any real number from some interval (e.g., any real number between 0 and 100). The remainder of this discussion will focus on discrete variables. A continuous variable like the ones in our example can be converted to a discrete variable by dividing its range of possible values into subranges, each of which is treated as a single value. This process is known as *discretization*. For example, if *log writes per sec* can take on any value from the interval $[0, \infty)$, we can divide this interval into the subintervals $[0, 10)$, $[10, 100)$, and $[100, \infty)$, and give all values in $[0, 10)$ a value of 0, all values in $[10, 100)$ a value of 1, and all values in $[100, \infty)$ a value of 2 in the discretized version of the variable. Section 5.3 discusses various approaches that can be taken to discretize continuous variables.

Once we have determined the relevant variables $V_1, \ldots, V_n$, we can use probability theory to represent the likelihood of seeing various assignments of values to these variables; I will refer to a particular assignment of values $V_1 = v_1, \ldots, V_n = v_n$ as an *instantiation* of the variables. The *probability* of the instantiation $V_1 = v_1, \ldots, V_n = v_n$, written $P(V_1 = v_1, \ldots , V_n = v_n)$, is a real number from the interval $[0, 1]$ that represents the relative likelihood of seeing that instantiation. For the sake of conciseness, the notation $P(\boldsymbol{V} = \boldsymbol{v})$ will typically be used to refer to the same probability, where $\boldsymbol{V}$ is the set of variables (written as a vector) and $\boldsymbol{v}$ is the vector of values assigned to those variables. Similarly, $\Omega(\boldsymbol{V})$ will be used to refer to all possible vectors of values that can be assigned to the variables in set $\boldsymbol{V}$. Probability values can be based on experimental observations, subjective assessments, or a combination of

Table 3.1. A sample joint probability distribution for the variables A and B

	A = 0	**A = 1**
B = 0	0.23	0.11
B = 1	0.08	0.14
B = 2	0.04	0.25
B = 3	0.13	0.02

the two. Section 5.4 explains how the probabilities needed for software tuning can be learned from data.

A collection of probabilities for all possible instantiations of a set of variables is known as a *joint probability distribution* for the variables; the individual probabilities in such a joint distribution must sum to 1. Table 3.1 shows a sample joint probability distribution for two variables, *A* and *B*. For example, we can determine that *P*(A = 1, B = 2) is 0.25 by finding the value at the intersection of the column in which A = 1 and the row in which B = 2.

From the joint probability distribution for a set of variables, we can derive a probability distribution for a subset of those variables using a process known as *marginalization*. If $\boldsymbol{V}$ is the full set of variables, $\boldsymbol{X}$ is the subset of $\boldsymbol{V}$ whose values we are interested in, and $\boldsymbol{Y}$ is the subset containing the remaining variables (i.e., $\boldsymbol{Y} = \boldsymbol{V} - \boldsymbol{X}$), then for any $\boldsymbol{x} \in \Omega(\boldsymbol{X})$:

$$P(\boldsymbol{X} = \boldsymbol{x}) = \sum_{\boldsymbol{y} \in \Omega(\boldsymbol{Y})} P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y}) \qquad \text{(EQ 1)}$$

For example, given the joint probability distribution in Table 3.1, we can the compute the probability that $A = 0$ as follows:

$$P(A = 0) = P(A = 0, B = 0) + P(A = 0, B = 1) + P(A = 0, B = 2) + P(A = 0, B = 3)$$
$$= 0.23 + 0.08 + 0.04 + 0.13 = 0.48$$

Marginalization effectively eliminates variables from a probability distribution by summing over all possible instantiations of those variables. The resulting distribution is known as a *marginal distribution*.

3.1.2 Conditional Probabilities

Learning the value of one or more of the variables in which we are interested can lead us to change our probability assessments for the remaining variables. In our tuning problem, for example, if we know that the value of *log writes per sec* is large, it becomes more likely that the value of *time to recover* is also large. Probabilities that are assessed in the light of information concerning one or more of the variables are known as *conditional* or *posterior* probabilities, and we say that the resulting probabilities are conditioned on the available information. The conditioning information itself is often referred to as *evidence*.

I will use the notation $P(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Y} = \boldsymbol{y})$ to refer to the conditional probability that $\boldsymbol{X} = \boldsymbol{x}$ given that $\boldsymbol{Y} = \boldsymbol{y}$ and that there is no information about the values of the variables that are not in $\boldsymbol{Y}$. Probabilities that are assessed without any knowledge of the values of the variables are sometimes called *unconditional* or *prior* probabilities to distinguish them from conditional probabilities.

We can compute conditional probabilities from unconditional probabilities as follows:

$$P(\boldsymbol{X} = \boldsymbol{x} | \boldsymbol{Y} = \boldsymbol{y}) = \frac{P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y})}{P(\boldsymbol{Y} = \boldsymbol{y})} \qquad \text{(EQ 2)}$$

This equation holds provided that $P(\boldsymbol{Y} = \boldsymbol{y}) > 0$. When $P(\boldsymbol{Y} = \boldsymbol{y}) = 0$, conditional probabilities that are conditioned on the instantiation $\boldsymbol{Y} = \boldsymbol{y}$ are undefined. The collection of conditional probabilities $P(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Y} = \boldsymbol{y})$ for all possible instantiations of the variables in $\boldsymbol{X}$, given the instantiation $\boldsymbol{Y} = \boldsymbol{y}$ of the variables in $\boldsymbol{Y}$, is known as the *conditional probability distribution* for the variables in $\boldsymbol{X}$, given $\boldsymbol{Y} = \boldsymbol{y}$.

From equation 2, we can derive what is often referred to as the Product Rule:

$$P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y}) = P(\boldsymbol{Y} = \boldsymbol{y})P(\boldsymbol{X} = \boldsymbol{x} | \boldsymbol{Y} = \boldsymbol{y}) \qquad \text{(EQ 3)}$$

This equation can be used to compute elements of a joint probability distribution, given elements of the corresponding marginal and conditional distributions. More generally, given the evidence $\boldsymbol{Z} = \boldsymbol{z}$:

$$P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y} | \boldsymbol{Z} = \boldsymbol{z}) = P(\boldsymbol{Y} = \boldsymbol{y} | \boldsymbol{Z} = \boldsymbol{z})P(\boldsymbol{X} = \boldsymbol{x} | \boldsymbol{Y} = \boldsymbol{y}, \boldsymbol{Z} = \boldsymbol{z}) \qquad \text{(EQ 4)}$$

And repeated applications of the Product Rule can be used to derive another useful equation known as the Chain Rule, which holds for any instantiation of the variables $\{X_1, ..., X_n\}$:

$$P(X_1 = x_1, \ldots, X_n = x_n) = \prod_{i=1}^{n} P(X_i = x_i | X_{i-1} = x_{i-1}, \ldots, X_1 = x_1) \quad \text{(EQ 5)}$$

Finally, we can also perform two variants of marginalization using conditional probabilities. As with marginalization using unconditional probabilities (equation 1), both of these variants effectively eliminate one or more variables from a probability distribution by summing over all instantiations of those variables. If we have conditional probability distributions for the variables in $\boldsymbol{V}$ given the variables in $\boldsymbol{W}$ (i.e., probabilities of the form $P(\boldsymbol{V} = \boldsymbol{v} \mid \boldsymbol{W} = \boldsymbol{w})$), the first variant performs marginalization over a subset of the variables in $\boldsymbol{V}$, and the second performs marginalization over a subset of the variables in $\boldsymbol{W}$.

First variant. If $\boldsymbol{V} = \boldsymbol{X} \cup \boldsymbol{Y}$, then $P(\boldsymbol{V} = \boldsymbol{v} \mid \boldsymbol{W} = \boldsymbol{w}) = P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{W} = \boldsymbol{w})$ and we can eliminate the variables in $\boldsymbol{Y}$ using a straightforward extension of equation 1 in which the probabilities are conditioned on the evidence $\boldsymbol{W} = \boldsymbol{w}$:

$$P(\boldsymbol{X} = \boldsymbol{x} | \boldsymbol{W} = \boldsymbol{w}) = \sum_{\boldsymbol{y} \in \Omega(\boldsymbol{Y})} P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y} | \boldsymbol{W} = \boldsymbol{w}) \quad \text{(EQ 6)}$$

Second variant. If $\boldsymbol{W} = \boldsymbol{T} \cup \boldsymbol{Z}$, then $P(\boldsymbol{V} = \boldsymbol{v} \mid \boldsymbol{W} = \boldsymbol{w}) = P(\boldsymbol{V} = \boldsymbol{v} \mid \boldsymbol{T} = \boldsymbol{t}, \boldsymbol{Z} = \boldsymbol{z})$ and we can eliminate the variables in $\boldsymbol{T}$ from the distribution as follows. First, we note that if we had probabilities of the form $P(\boldsymbol{V} = \boldsymbol{v}, \boldsymbol{T} = \boldsymbol{t} \mid \boldsymbol{Z} = \boldsymbol{z})$, we could use equation 6 to obtain the desired probabilities:

$$P(\boldsymbol{V} = \boldsymbol{v} | \boldsymbol{Z} = \boldsymbol{z}) = \sum_{\boldsymbol{t} \in \Omega(\boldsymbol{T})} P(\boldsymbol{V} = \boldsymbol{v}, \boldsymbol{T} = \boldsymbol{t} | \boldsymbol{Z} = \boldsymbol{z})$$

To use the available probabilities, we apply the Product Rule to transform the right-hand side of the above equation, which gives us the second variant:

$$P(\boldsymbol{V} = \boldsymbol{v} | \boldsymbol{Z} = \boldsymbol{z}) = \sum_{t \in \Omega(T)} P(\boldsymbol{V} = \boldsymbol{v} | \boldsymbol{T} = \boldsymbol{t}, \boldsymbol{Z} = \boldsymbol{z}) P(\boldsymbol{T} = \boldsymbol{t} | \boldsymbol{Z} = \boldsymbol{z}) \quad \text{(EQ 7)}$$

3.1.3 Bayes' Law

If we have conditional probabilities for one set of variables, ***X***, given instantiations of a second set of variables, ***Y***, it is often useful to be able to determine conditional probabilities for the variables in ***Y*** given instantiations of the variables in ***X***. To do so, we first use the Product Rule (equation 3) to show that

$$P(\boldsymbol{X} = \boldsymbol{x})P(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x}) = P(\boldsymbol{Y} = \boldsymbol{y})P(\boldsymbol{X} = \boldsymbol{x}|\boldsymbol{Y} = \boldsymbol{y}.$$

Then, by solving this equation for $P(\boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{X} = \boldsymbol{x})$, we obtain an equation known as Bayes' Law that allows us to compute the conditional probabilities in which we are interested:

$$P(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x}) = \frac{P(\boldsymbol{Y} = \boldsymbol{y})P(\boldsymbol{X} = \boldsymbol{x}|\boldsymbol{Y} = \boldsymbol{y}}{P(\boldsymbol{X} = \boldsymbol{x})} \tag{EQ 8}$$

To use Bayes' Law, we need to know both the conditional probability $P(\boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{X} = \boldsymbol{x})$ and the marginal probability $P(\boldsymbol{Y} = \boldsymbol{y})$, but we can use marginalization to compute the marginal probability $P(\boldsymbol{X} = \boldsymbol{x})$ that appears in the denominator of equation 8. This yields an equivalent version of Bayes' Law:

$$P(\boldsymbol{Y} = \boldsymbol{y}|\boldsymbol{X} = \boldsymbol{x}) = \frac{P(\boldsymbol{Y} = \boldsymbol{y})P(\boldsymbol{X} = \boldsymbol{x}|\boldsymbol{Y} = \boldsymbol{y})}{\sum_{\boldsymbol{y} \in \Omega(\boldsymbol{Y})} P(\boldsymbol{X} = \boldsymbol{x}|\boldsymbol{Y} = \boldsymbol{y})P(\boldsymbol{Y} = \boldsymbol{y}} \tag{EQ 9}$$

3.1.4 Independence and Conditional Independence

Knowing the value of one or more of the variables of interest will often affect our probability assessments for the remaining variables, but there can be combinations of variables for which this is not the case. In particular, we say that two sets of variables, ***X*** and ***Y***, are *independent* if, for all $\boldsymbol{x} \in \Omega(\boldsymbol{X})$ and all $\boldsymbol{y} \in \Omega(\boldsymbol{Y})$,

$$P(\boldsymbol{X} = x, \boldsymbol{Y} = y) = P(\boldsymbol{X} = x)P(\boldsymbol{Y} = y).$$

We can use this definition and equation 3 to show that, if ***X*** and ***Y*** are independent, then for all $\boldsymbol{x} \in \Omega(\boldsymbol{X})$ and all $\boldsymbol{y} \in \Omega(\boldsymbol{Y})$,

$$P(\boldsymbol{X} = x \mid \boldsymbol{Y} = y) = P(\boldsymbol{X} = x) \text{ and}$$

$$P(\boldsymbol{Y} = y \mid \boldsymbol{X} = x) = P(\boldsymbol{Y} = y).$$

In other words, if two sets of variables are independent, knowing the values of the variables from one set does not affect our probability assessments for the variables in the other set.

If two sets of variables are not independent in general, it may still be the case that they are independent given the values of a third set of variables. We say that $\boldsymbol{X}$ and $\boldsymbol{Y}$ are *conditionally independent* given $\boldsymbol{Z}$ if, for all $\boldsymbol{x} \in \Omega(\boldsymbol{X})$, all $\boldsymbol{y} \in \Omega(\boldsymbol{Y})$, and all $\boldsymbol{z} \in \Omega(\boldsymbol{Z})$,

$$P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{Z} = \boldsymbol{z}) \quad = \quad P(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Z} = \boldsymbol{z}) P(\boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{Z} = \boldsymbol{z}).$$

We can use equation 3 to show that, if $\boldsymbol{X}$ and $\boldsymbol{Y}$ are conditionally independent given $\boldsymbol{Z}$, then for all $\boldsymbol{x} \in \Omega(\boldsymbol{X})$, all $\boldsymbol{y} \in \Omega(\boldsymbol{Y})$, and all $\boldsymbol{z} \in \Omega(\boldsymbol{Z})$,

$$P(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Y} = \boldsymbol{y}, \boldsymbol{Z} = \boldsymbol{z}) \quad = \quad P(\boldsymbol{X} = \boldsymbol{x} \mid \boldsymbol{Z} = \boldsymbol{z}) \text{ and}$$

$$P(\boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Z} = \boldsymbol{z}) \quad = \quad P(\boldsymbol{Y} = \boldsymbol{y} \mid \boldsymbol{Z} = \boldsymbol{z}).$$

In other words, provided that we know the values of the variables in $\boldsymbol{Z}$, knowing the values of the variables in $\boldsymbol{Y}$ will not affect our probability assessments for the variables in $\boldsymbol{X}$, and vice versa.

In our simple tuning example, *MB per checkpoint* captures the impact of both *log writes per sec* and *checkpoint interval* on *time to recover*. Therefore, it seems reasonable to conclude that once we know the value of *MB per checkpoint*, knowing the value of either *log writes per sec* or *checkpoint interval* will not affect our probability assessments for *time to recover*. We can thus say that *time to recover* is conditionally independent of *log writes per sec* and *checkpoint interval*, given *MB per checkpoint*. Section 3.2.3 explains how influence diagrams take advantage of conditional independence relationships to reduce the number of parameters needed to model a decision problem.

3.1.5 Expected Values

When dealing with a variable, X, we often summarize the distribution of its possible values using a quantity known as the *expected value* of the variable, written $E(X)$:

$$E(X) = \sum_{x \in \Omega(X)} xP(X = x) \qquad \textbf{(EQ 10)}$$

The expected value is essentially a weighted average of the possible values of the variable, with the values' probabilities serving as the weights. If the variable in question is itself the sum of two or more variables, the overall expected value is simply the sum of the expected values of the individual terms:

$$E(X_1 + X_2 + \ldots + X_n) = E(X_1) + E(X_2) + \ldots + E(X_n \qquad \textbf{(EQ 11)}$$

We can also compute the *conditional expected value* of a variable X given some evidence $\boldsymbol{Y} = \boldsymbol{y}$. This quantity, which is written $E(X \mid \boldsymbol{Y} = \boldsymbol{y})$, is computed as follows:

$$E(X|\boldsymbol{Y} = \boldsymbol{y}) = \sum_{x \in \Omega(X)} x \cdot P(X = x|\boldsymbol{Y} = \boldsymbol{y}) \qquad \textbf{(EQ 12)}$$

And if we have a table of conditional expected values of the form $E(X \mid \boldsymbol{Y} = \boldsymbol{y}, \boldsymbol{Z} = \boldsymbol{z})$ for all instantiations of the variables in the sets $\boldsymbol{Y}$ and $\boldsymbol{Z}$, we can use marginalization over the variables in $\boldsymbol{Z}$ to compute a corresponding table of expected values conditioned on instantiations of only the variables in $\boldsymbol{Y}$:

$$E(X|\boldsymbol{Y} = \boldsymbol{y}) = \sum_{\boldsymbol{z} \in \Omega(\boldsymbol{Z})} E(X|\boldsymbol{Y} = \boldsymbol{y}, \boldsymbol{Z} = \boldsymbol{z}) P(\boldsymbol{Z} = \boldsymbol{z}|\boldsymbol{Y} = \boldsymbol{y}) \qquad \textbf{(EQ 13)}$$

3.1.6 Making Optimal Decisions Under Uncertainty

Decision theory is based on the idea that individuals should attempt to maximize the *utility* of the outcomes of their decisions, where utility is a measure of the relative desirability of an outcome to an individual. When tuning a software system, utility is typically based on one or more performance measures, such as throughput or response time.

If the outcome of a particular decision is uncertain, decision theory stipulates that the optimal decision is the one that maximizes the conditional expected value of the utility, given the decision made and any other available evidence. More formally, if the decision is represented by the variable X and the utility is represented by the variable U, and if evidence is available for a set of variables $\boldsymbol{Y}$, then the optimal decision given the evidence $\boldsymbol{Y} = \boldsymbol{y}$ is written $d^*(X \mid \boldsymbol{Y} = \boldsymbol{y})$ and is determined as follows:

$$d^*(X|\boldsymbol{Y} = \boldsymbol{y}) = \underset{x \in \Omega_X}{\arg\max} (E(U \mid X = x, \boldsymbol{Y} = \boldsymbol{y})) \qquad \textbf{(EQ 14)}$$

The collection of optimal decisions for all possible instantiations of the evidence variables is known as the *optimal policy* for the decision.

3.1.7 Conclusions

When tuning a software system, probability theory and decision theory allow us to represent our uncertainty about the relevant variables and to choose the knob settings that maximize the expected performance of the system. However, if the number of variables involved is large—as will typically be the case for a complex system—it can be difficult to specify and manipulate the full joint probability distribution. Influence diagrams are graphical, probabilistic models that provide one method of dealing with this problem. The next section explains what these models are and how they are used.

3.2 Influence Diagrams

Influence diagrams were developed by decision analysts to provide an intuitive yet powerful tool for representing and reasoning about difficult decision problems [How84]. The following paragraphs provide an overview of the structure and parameters of these graphical models and how they can be used to infer optimal decisions. For more information, readers are encouraged to consult related work from decision analysis and artificial intelligence [How84, Jen01, Sha86].

3.2.1 Structure of the Model

In structure, an influence diagram is a *graph*, a data structure that consists of a set of *nodes*, $\boldsymbol{N}$, and a set of *arcs* of the form (n_i, n_j), where n_i and n_j are elements of $\boldsymbol{N}$. It is a *directed* graph because each arc has an associated direction: arc (n_i, n_j) goes from node n_i (the *initial node* of the arc) to node n_j (the *terminating node* of the arc), and it is typically depicted as an arrow with its tail at node n_i and its head at node n_j. Figure 3.1 displays a simple directed graph.

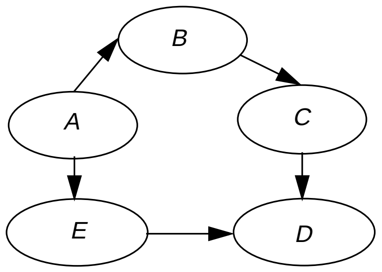
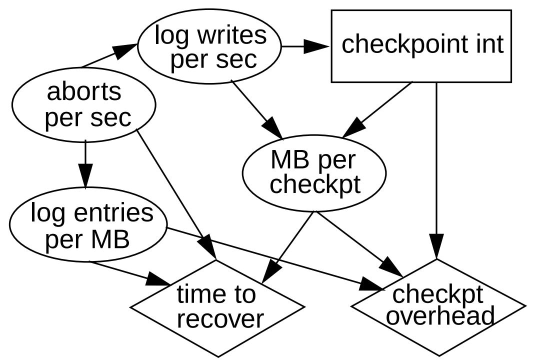


Figure 3.1. An example of a directed acyclic graph.

A *directed path* is a sequence of arcs of the form $\{(n_1, n_2), (n_2, n_3), (n_3, n_4), ...\}$ in which the terminating node of a given arc is the initial node of the next arc in the sequence. If there is an arc from node n_i to node n_j, n_i is referred to as a *parent* of n_j and n_j is referred to as a *child* of n_i. In addition, n_i is said to *inherit from* n_j. More generally, if there is a directed path involving one or more arcs from n_i to n_j, n_i is considered an ancestor of n_j and n_j is considered a *descendant* of n_i. If there is no directed path from n_i to n_j, n_j is referred to as a *nondescendant* of n_i. A *cycle* is a path that begins and ends at the same node; adding the arc (C, A) to the graph in Figure 3.1 would create a cycle. Influence diagrams are not allowed to have any cycles, and thus they are referred to as *acyclic* graphs.

Influence diagrams contain three types of nodes: rectangular *decision nodes* that represent the choices being made, diamond-shaped *value nodes* that represent components of the decision-maker's utility, and oval *chance nodes* that represent the other variables relevant to the decision problem. Value nodes cannot have any descendants. Figure 3.2 shows an influence diagram that captures the relationships between the variables in the simple tuning example from Section 3.1. This influence diagram could be used to determine the optimal checkpoint interval based on a utility function that is a weighted sum of the time needed to recover and the overhead of taking checkpoints.

The arcs in an influence diagram convey two different types of information. Arcs into chance or value nodes represent a possible probabilistic dependence relationship. If there is an arc from A to B, knowing the value of A may affect our probability assessments for B, and vice versa. These arcs are known as *conditional*



P(MB per checkpt | 50 log writes/s, checkpt int)

	checkpoint int			
MB per checkpt	1 min	10 min	30 min	...
0-5	0.60	0.10	0.05	...
5-10	0.15	0.17	0.15	...
10-50	0.10	0.65	0.33	...
...	...	...	...	...

E(time to rec | log entr/MB, aborts/s MB/checkpt)

log entr/MB	aborts/s	MB/checkpt	time to rec
1000-5000	0-10	0-5	1.0 min
1000-5000	0-10	5-10	2.5 min
1000-5000	0-10	10-50	8.3 min
...	...	...	...

Figure 3.2. An example of an influence diagram and its associated parameters. The structure of the model is shown on the left-hand side of the figure. Also shown is an example of a portion of a conditional probability table for one of the chance nodes (*upper right*) and a portion of the table of expected values for a value node (*lower right*).

arcs, and they are typically added on the basis of intuitive notions of causality. In Figure 3.2, for example, increasing the checkpoint interval typically leads to an increase in the amount of data logged between checkpoints, and thus there is a conditional arc from *checkpoint int* to *MB per checkpt*.

Arcs into decision nodes represent known information. If there is an arc from C to D, the value of C is known when decision D is made. In Figure 3.2, for example, the arc from *log writes per sec* to *checkpoint int* indicates that the value of the former variable is available when the checkpoint interval is chosen. These arcs are referred to as *informational arcs*.

If an influence diagram has multiple decision nodes, it must have at least one directed path that includes all of these nodes. This restriction effectively requires that the decisions be made sequentially, in the order in which they are encountered on one of these directed paths.[2] However, as noted by Nielsen and Jensen [Nie99], there are cases in which a full temporal ordering of the decision nodes is not strictly necessary. In particular, if C_i is the set of chance nodes whose values are known when decision D_i is made, two decision nodes D_1 and D_2 can be commuted whenever $C_1 = C_2$. In such cases, either ordering of the decision nodes will produce the same optimal policies.

2. Note that because an influence diagram cannot have any cycles, any directed path containing all of the decision nodes must encounter them in the same order.

The algorithm for evaluating an influence diagram (Section 3.2.4) makes the reasonable assumption that all information known when a decision is made is available for all subsequent decisions. As a result, if the value of a node, N, is known in advance of one or more of the decisions, we only need to include an informational arc from N to the first of these decisions. The algorithm will add so-called *no-forgetting arcs* from N to any subsequent decisions.

3.2.2 Parameters of the Model

Each node in an influence diagram has an associated set of parameters. For a decision node, there is the set of possible alternatives for the corresponding decision. For a chance node, there is the set of possible values of the node and the parameters needed to specify one or more probability distributions. A chance node without parents (a *root chance node*) has a marginal distribution over its values. A chance node with parents (an *intermediate chance node*) has a collection of conditional distributions, one for each instantiation of its parents (Figure 3.2, *upper right*). Similarly, a value node has one expected value for each instantiation of its parents (Figure 3.2, *bottom right*). In Section 5.4, I discuss how to learn these parameters from data.

3.2.3 Conditional Independencies in Influence Diagrams

A key feature of influence diagrams and related graphical models is the conditional independence relationships that they encode. In particular, given the values of its parents, any chance or value node is conditionally independent of its nondescendants.[3]

As a result of these conditional independencies, fewer parameters are needed to specify the model. To see that this is the case, consider an influence diagram that contains only chance nodes[4], and let the labels of the nodes, $\{V_1, ..., V_n\}$, be assigned

3. More generally, conditional independencies can be deduced from the structure of an influence diagram using a criterion known as *d-separation* [Pea88].

4. I limit the influence diagram to chance nodes so that the parameters of the model will encode a single joint probability distribution for the nodes in the model. This simplifies the illustration of how conditional independencies can reduce the number of parameters in the model. Similar reductions are also seen in the general case.

in such a way that every node's descendants have indices that are larger than the index of the node itself. Given this labelling, it follows that, for any index *i*, the nodes $V_{i-1}, ..., V_1$ are nondescendants of the node V_i. Therefore, because each node is conditionally independent of its nondescendants given its parents, we can take the factorization of the joint probability distribution provided by the Chain Rule (equation 5), i.e.,

$$P(V_1 = v_1, \ldots, V_n = v_n) = \prod_{i=1}^{n} P(V_i = v_i | V_{i-1} = v_{i-1}, \ldots, V_1 = v_1)$$

and rewrite it as

$$P(V_1 = v_1, \ldots, V_n = v_n) = \prod_{i=1}^{n} P(V_i = v_i | Parents(V_i))$$

where $Parents(V_i)$ represents the assignment of values to V_i's parents in the instantiation $V_1 = v_1, ..., V_n = v_n$. In other words, we can represent a node V_i's contribution to the full joint distribution by specifying one probability distribution for each instantiation of its parents, rather than one distribution for each instantiation of all nodes with indices less than *i*, as would be required in the standard factorization. This can significantly reduce the number of parameters in the model. Similarly, when specifying the parameters for a value node, we need one expected value for each combination of the values of its parents, rather than one value for each instantiation of all the chance and decision nodes in the model. And because fewer parameters are needed to specify the model, it becomes easier to learn these parameters and to use the influence diagram to determine optimal decisions.

3.2.4 Evaluating an Influence Diagram

Evaluating an influence diagram involves determining the decisions that maximize the total expected value of the value nodes (which, by equation 11, is simply the sum of the expected values of the individual value nodes) given the probability distributions over the chance variables and any observed values of these variables. If the value nodes represent costs, we can multiply the expected costs by -1 and thereby minimize the total expected cost using the same algorithm. For instance, in our simple tuning example, we would observe the value of *log writes per sec* for a given workload (because the informational arc from *log writes per sec* to *checkpoint int* indicates that the *log writes* value is known at the time of the decision), and solve the influence diagram to determine the value of *checkpoint int* that leads to the smallest expected cost for that workload.

The standard algorithm for evaluating an influence diagram [Sha86] repeatedly transforms the diagram by removing a node or reversing the direction of an arc. At the conclusion of the algorithm, we are left with the optimal policy (Section 3.1.6) for each decision node *D*, i.e., a mapping from instantiations of *D*'s parents to the alternative or alternatives that maximize the total expected value of the value nodes, given the instantiation of *D*'s parents.[5] We can also apply the algorithm in light of evidence concerning the chance nodes, in which case we can skip computations of parameters that are inconsistent with the evidence. For example, if we know the value of *log writes per sec*, we need never revise its table of parameters or compute parameters that are conditioned on other values of this variable. In the discussion that follows, I will present the version of the algorithm that computes the full optimal policy. For more details, readers are encouraged to consult the original paper by Shachter. Other evaluation algorithms can also be used [e.g., Jen94]. All exact algorithms for evaluating influence diagrams have a worst-case complexity that is exponential in the number of nodes in the model, but the runtime costs are typically reasonable in practice.

There are four types of transformations that the standard evaluation algorithm can apply. Three of them involve removing a node from the diagram, in

5. In his classic paper on evaluating influence diagrams [Sha86], Shachter assumes that there is a single value node. The algorithm that I present here is a straightforward extension that can handle influence diagrams with multiple value nodes, where the objective is to maximize the expected value of the sum of the values of these nodes.

```
id_eval(I) {
   add no-forgetting arcs as needed (see Section 3.2.1);
   remove all barren nodes;
   while (one or more value nodes still have parents) {
      if (∃ a chance node C that can be removed) {
            remove chance node C;
      } else if (∃ a decision node D that can be removed) {
            remove decision node D;
            remove barren nodes;
      } else {
            find a chance node X that is a parent of a value node
               and is not the parent of any decision node;
            while (X has children) {
                  find a child Y of X such that there is no other
                     directed path from X to Y besides arc (X,Y);
                  reverse arc (X,Y);
            }
            remove chance node X;
      }
   }
}
```

Figure 3.3. Pseudocode for evaluating an influence diagram. The algorithm takes an influence diagram, *I*, and transforms it to determine the optimal policy for each decision node, *D*, in the model. The optimal policy for a decision, *D*, is constructed during the process of removing *D* from the model. See Section 3.2.4 for the criteria for removing a chance or decision node and for more details about reversing arcs and removing barren, chance, and decision nodes.

which case all arcs that begin at that node are also removed. The four transformations are as follows:

- *barren node removal*: if a chance or decision node has no children, it can simply be removed from the influence diagram because it cannot affect the value nodes. If a decision node is barren, any of the alternatives associated with it can be added to the optimal policy for that decision.

- *chance node removal*: if chance node *C* is a parent of one or more value nodes and it is not a parent of a chance or decision node, it can be removed from the diagram. The value nodes of which *C* is a parent inherit all of *C*'s parents, and the expected-value tables for these value nodes are revised using marginalization over *C*.

- *decision node removal*: if a decision node *D* is a parent of one or more of the value nodes, and if all other parents of those value nodes are nodes whose values are known when decision *D* is made (i.e., the nodes in question are parents of *D*), *D* can be removed from the diagram. For each possible instantiation of *D*'s parents, the alternative or alternatives that maximize the total expected value of the value nodes (given the values of *D*'s parents) are added to the optimal policy. In addition, the expected-value tables of the value nodes in question are modified to reflect the optimal policy.

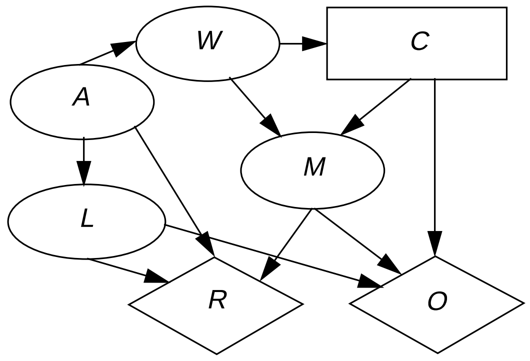


Figure 3.4. The checkpoint-interval influence diagram. This is a version of the influence diagram from Figure 3.2 in which the variable names have been abbreviated for the sake of conciseness. The remaining figures in this chapter will demonstrate how the standard algorithm for evaluating an influence diagram transforms this diagram to determine the optimal policy for C.

- *arc reversal*: if chance nodes C_1 and C_2 are connected by the arc (C_1, C_2) and there is no other directed path from C_1 to C_2, arc (C_1, C_2) can be replaced by arc (C_2, C_1). The two nodes inherit each other's parents, and two sets of computations are performed to determine the nodes' new parameters. Marginalization over C_1 is used to revise the conditional probabilities for C_2, and Bayes' Law is used to revise the conditional probabilities for C_1.

Pseudocode for the algorithm is presented in Figure 3.3. To give a sense of how it works, I will sketch out its application to the influence diagram in Figure 3.2. For the sake of conciseness, I will use abbreviations for the variables in the diagram, as shown in Figure 3.4.

The algorithm would begin by removing the two chance nodes L and M, both of which have the value nodes as their only children. The order in which these nodes are removed is arbitrary. Beginning with L yields the diagram shown on the left-hand side of Figure 3.5. Note that R already had L's parent, A, as a parent, so it simply loses L as a parent, whereas O loses L and inherits A. The conditional expected values of the value nodes are revised using marginalization over L (equation 13):

$$E(R|A=a, M=m) = \sum_{l \in \Omega_L} E(R|A=a, L=l, M=m)P(L=l|A=a)$$

$$\begin{aligned} E(O|A=a, C=c, M=m) &= \sum_{l \in \Omega_L} E(O|A=a, C=c, L=l, M=m)P(L=l|A=a) \\ &= \sum_{l \in \Omega_L} E(O|C=c, L=l, M=m)P(L=l|A=a) \end{aligned}$$

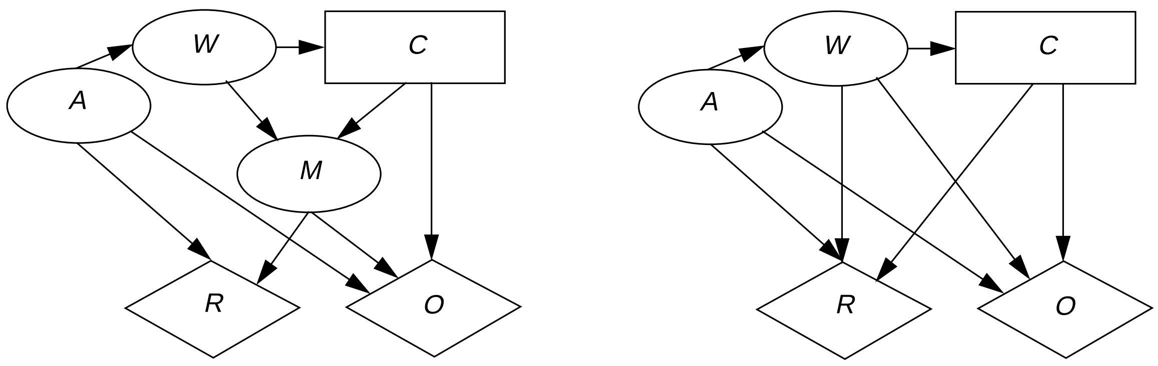


Figure 3.5. Evaluating the checkpoint-interval influence diagram, part I. Shown are the results of the first two steps of evaluating the checkpoint-interval influence diagram: the removal of chance node L (*left*), and the removal of chance node M (*right*).

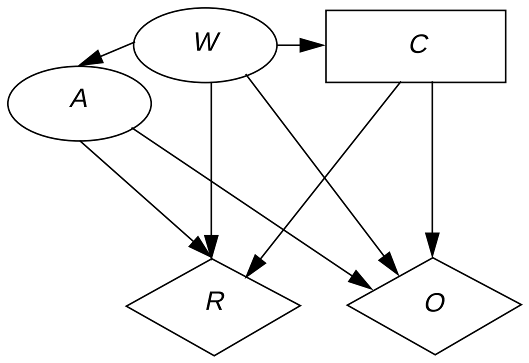
Note that all of the probabilities and expected values needed by these computations—and, more generally, by all of the transformations—can be found in the tables of parameters associated with the relevant nodes at the time of the transformation. It is also worth noting that the computations that revise *O*'s expected values take advantage of the conditional independence of *O* and *A*, given the values of *C*, *L*, and *M*, to substitute $E(O \mid C = c, L = l, M = m)$ for $E(O \mid A = a, C = c, L = l, M = m)$. Similar substitutions will be performed implicitly in the remainder of this section.

In the next step, *M* is removed. The value nodes inherit its parents (Figure 3.5, *right*), and their conditional expected values are updated using marginalization over *M*:

$$E(R|A = a, C = c, W = w) = \sum_{m \in \Omega_M} E(R|A = a, M = m)P(M = m|C = c, W = w)$$

$$E(O|A = a, C = c, W = w) = \sum_{m \in \Omega_M} E(O|A = a, C = c, M = m)P(M = m|C = c, W = w)$$

At this point in the evaluation of the influence diagram, no nodes can be removed directly. Chance nodes *A* and *W* are parents of the values nodes, but they are also parents of other, non-value nodes, so they cannot be removed. Decision node *C* is a parent of the value nodes, but the value of *A* is not known when this decision is made, so *C* cannot be removed. Therefore, the algorithm is forced to perform an arc reversal. Specifically, it selects some chance node, *N*, whose children include a value



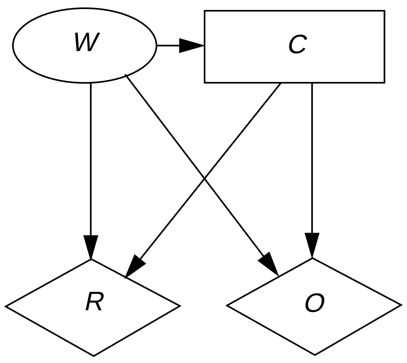


Figure 3.6. Evaluating the checkpoint-interval influence diagram, part II. Shown are the results of the third and fourth steps of evaluating the checkpoint-interval influence diagram: the reversal of arc (*A*, *W*) (*left*), and the removal of chance node *A* (*right*).

node and one or more other chance nodes but whose value is not known prior to the decisions, and it reverses all arcs from *N* to its chance-node children. In our example, the algorithm selects node *A* and reverses the arc (*A,W)* (Figure 3.6, *left*). Marginalization (equation 7) is used to convert the conditional probabilities for *W* to marginal probabilities, and Bayes' Law (equation 8) is used to convert the marginal probabilities for *A* into conditional probabilities:

$$P(W = w) = \sum_{a \in \Omega_A} P(W = w | A = a) P(A = a)$$

$$P(A = a | W = w) = \frac{P(A = a) P(W = w | A = a)}{P(W = w)}$$

Note that the Bayes' Law computation makes use of both the previous parameters associated with the two nodes and the just computed marginal probabilities for *W*.

Upon completion of the arc reversal, *A* becomes eligible for removal; doing so yields the diagram on the right-hand side of Figure 3.6, and the conditional expected values of the value nodes are revised using marginalization over *A*:

$$E(R | C = c, W = w) = \sum_{a \in \Omega_A} E(R | A = a, C = c, W = w) P(A = a | W = w)$$

$$E(O | C = c, W = w) = \sum_{a \in \Omega_A} E(O | A = a, C = c, W = w) P(A = a | W = w)$$

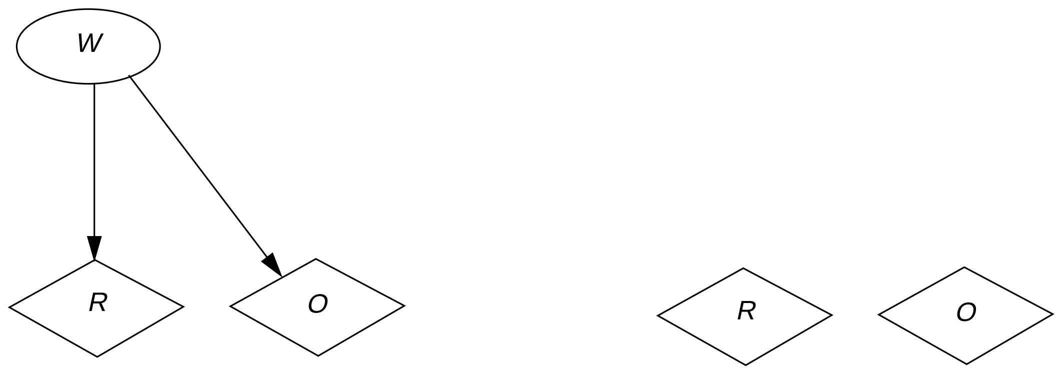


Figure 3.7. Evaluating the checkpoint-interval influence diagram, part III. Shown are the results of the final two steps of evaluating the checkpoint-interval influence diagram: the removal of decision node *C* (*left*), and the removal of chance node *W* (*right*).

The algorithm still cannot remove chance node *W*, because it is the parent of decision node *C*. However, the algorithm can remove *C*, because the only other parent of the value nodes is *W*, and its value is known when the value of *C* is selected. Removing *C* produces the diagram on the left-hand side of Figure 3.7. For each possible value of *W*, the optimal alternative for *C* given that value of *w* is determined using equation 11 and equation 14:

$$d^*(C|W=w) = \underset{c \in \Omega_C}{\arg\max}(E(R|C=c, W=w) + E(O|C=c, W=w))$$

In addition, the conditional expected values of the value nodes are revised to reflect the optimal policy:

$$E(R|W=w) = E(R|C=d^*(C|W=w), W=w)$$

$$E(O|W=w) = E(O|C=d^*(C|W=w), W=w)$$

Finally, the algorithm concludes by removing chance node *W*, which leaves only the value nodes (Figure 3.7, *right*), each of which has a single parameter corresponding to the expected value of the node in the absence of any evidence:

$$E(R) = \sum_{w \in \Omega_W} E(R|W=w)P(W=w)$$

$$E(O) = \sum_{w \in \Omega_W} E(O|W=w)P(W=w)$$

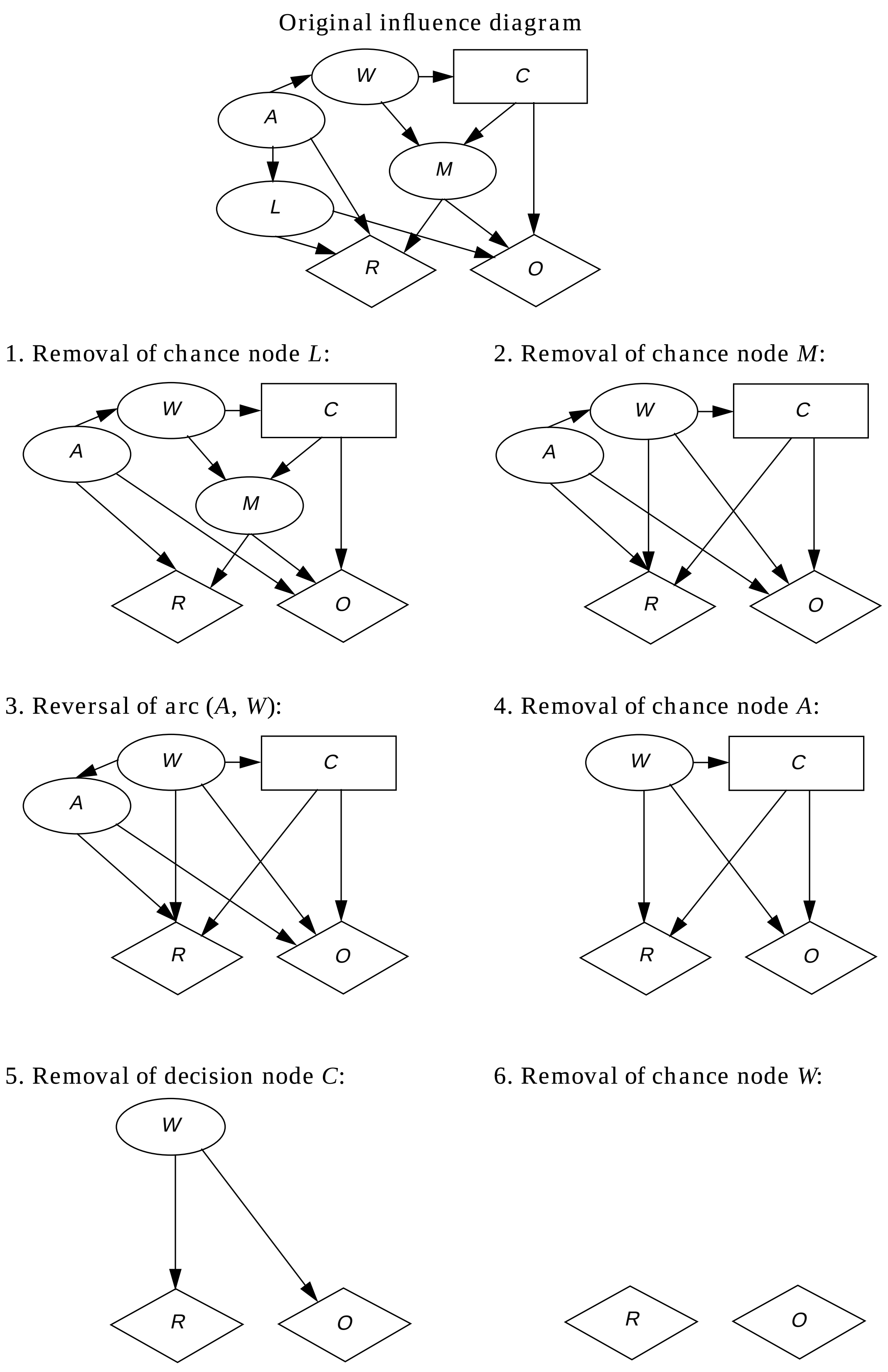

Original influence diagram
W
C
A
M
L
R
O
1. Removal of chance node *L*:
W
C
A
M
R
O
2. Removal of chance node *M*:
W
C
A
R
O
3. Reversal of arc (*A*, *W*):
W
C
A
R
O
4. Removal of chance node *A*:
W
C
R
O
5. Removal of decision node *C*:
W
R
O
6. Removal of chance node *W*:
R
O

Figure 3.8. Evaluating the checkpoint-interval influence diagram, a summary.

To allow the process of evaluating this influence diagram to be visualized in its entirety, Figure 3.8 consolidates the steps previously shown in Figures 3.5 through 3.7.

Once an influence diagram is evaluated, we can use the optimal policies for the decisions in the model to determine the optimal sequence of decisions given a particular set of observations. In our tuning example, we can observe the value of *log writes per sec* (W) and use the optimal policy to determine the optimal checkpoint interval, $d^*(C \mid W = w)$.

There are a number of software toolkits that can be used to evaluate influence diagrams. I have used the Netica inference engine [Nor03], but others are also available [Coz01, Hug01, *inter alia*].

3.2.5 Conclusions

An influence diagram is a probabilistic, graphical model that can be used as the foundation of an effective, model-based software tuner. By taking advantage of conditional independence relationships, influence diagrams allow us to simplify the specification of both the joint probability distribution for the relevant variables and the expected performance of the system under various combinations of workload characteristics and knob settings. They thus make it easier to learn the relevant probabilities and expected values from training data, and they speed up the process of determining the optimal knob settings for a given workload. Chapter 4 presents an influence diagram for tuning the Berkeley DB embedded database system. Chapter 5 explains how to construct, train, and use a model of this type for software tuning.

Chapter 4

Test System: Berkeley DB

To illustrate and validate the approach to automated software tuning presented in this thesis, I have used it to tune the Berkeley DB embedded database system. This chapter provides the necessary background material on this software system. Although the validation itself is not presented until Chapter 7, references to Berkeley DB occur throughout the intervening chapters, and thus it helps to cover this material at this point in the thesis. The chapter begins with an overview of Berkeley DB, and it continues with a discussion of the knobs that I tune—including the underlying performance issues associated with each knob—and of the variables used to characterize the workloads that run on the system. It concludes by presenting the influence diagram model developed for Berkeley DB.

4.1 Overview

Berkeley DB [Ols99, Sle01] is a programmatic database toolkit that applications can use to manage data that is stored as collections of *(key, value)* pairs; I will refer to these pairs as *data items*. The *key* serves as an identifier that can be used to retrieve one or more data items, and the *value* is the other information associated with a particular data item. For example, in a student registration database, the *key* might be the student's identification number, and the *value* a string containing all of the other information stored in the database for that student.

Berkeley DB is referred to as an *embedded* database system because it is linked directly into the address space of an application that uses it. This is in contrast to traditional database engines, which typically run as a stand-alone server application that other applications must connect to using some form of inter-process communication. Berkeley DB provides full support for features that are ordinarily found only in high-end database systems, including concurrent access to databases by multiple threads of control[1]; transactions, which allow a set of operations to be grouped together and treated as a unit; and recovery from hardware or software failures. In addition, its small memory footprint means that it can be used for applications running on embedded systems such as those found in telecommunications switches and information appliances—applications for which manual tuning is often impractical and which could thus benefit from automated tuning.

Berkeley DB supports several methods for storing and accessing data, including the Btree access method that I have used for the experiments in this thesis. In addition to these access methods, Berkeley DB consists of four major subsystems: a shared memory cache that stores recently accessed portions of a database, and subsystems that support locking, transactions, and logging. In the sections below, I briefly touch on each of these components of the system and define the relevant terminology. More information is available in the Berkeley DB documentation [Sle01].

4.1.1 Btree Access Method

Berkeley DB's Btree access method stores and accesses data in a tree data structure that is sorted and balanced. The tree is composed of *nodes*, each of which contains some data and zero or more pointers to other nodes that are referred to as its *children*. If a node, *N*, is a child of another node, *P*, then *P* is known as *N*'s *parent*. Nodes with children are known as *internal nodes*; nodes without children are termed *leaf nodes*. One node in the tree (the *root node* or *root*) has no parents; it can be thought of as the "top" of the tree. The *depth* of a node is equal to the number of pointers that

1. The term *thread of control* refers either to a process with its own address space or to a lightweight thread running within the address space of a process. I will use the word *thread* to represent either a thread or a process.

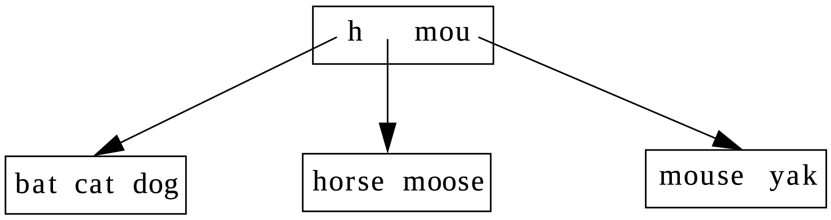


Figure 4.1. An example of a B+link tree.

must be followed to reach the node, starting from the root node. The Btree access method implements a variant of the standard Btree known as a *B+link tree* [Com79], which differs from a standard Btree in two ways. First, the data items are stored only in the leaf nodes and the internal nodes contain key prefixes with enough characters to determine which pointer to follow when searching for a particular key. For example, in Figure 4.1, the second key in the root node, *mou*, contains just enough characters to distinguish between the node containing the string *moose* and the node containing the string *mouse*. Second, the leaf nodes are linked together to facilitate sequential accesses that span multiple leaf nodes.

The keys or key prefixes on a given page are sorted according to some comparison function; the default ordering is lexicographic. A search for a particular key k starts at the root node and recursively descends the tree by comparing k with the key prefixes in the current node and following the appropriate pointer to one of the node's children. More specifically, the pointer followed when searching for key k is the one that precedes key prefix k_i, where k_i is the first prefix that is greater than or equal to k; if all prefixes are less than k, the rightmost pointer is followed. This process continues until the search reaches a leaf node. If the key exists in the database, it will be found on that leaf node.

Like all Btree data structures, a B+link tree is maintained in such a way that the tree is *balanced*, which means that all leaf nodes have the same depth; the depth of the leaf nodes is known as the *height* of the tree. Because the tree is balanced, the number of nodes that must be visited to determine if a key is in the database is $O(\log_b N)$, where b is the average number of keys per page and N is the total number of keys in the database. Btree databases can also be accessed sequentially using a database *cursor*, which essentially maintains a pointer to one of the data items in the database. Once the cursor is positioned on a given item, it can be used to update that item, or it can be advanced to the next or previous item in the database.

4.1.2 Memory Pool Subsystem

The Berkeley DB *memory pool* is a shared memory cache that can be accessed by multiple threads or processes. Berkeley DB organizes its databases in units called *pages*, and each internal node or leaf node in a Btree database corresponds to a single page. When Berkeley DB needs a particular page, it searches for it in the memory pool; a successful search is referred to as a *hit*, and an unsuccessful search is referred to as a *miss*. When a miss occurs, Berkeley DB reads the page into the memory pool from the on-disk database file. Ideally, the entire database will fit in the memory pool so that accesses can be satisfied without the overhead of going to disk. If the database is larger than the memory pool, pages are evicted as needed using an approximation of the least-recently-used (LRU) algorithm [Tan92]. Pages that have been modified (*dirty pages*) are written to the backing file before they are evicted.

In addition to the memory pool, the operating system maintains its own *buffer cache* of recently accessed file blocks. (To avoid confusion between these two caches, I will limit my use of the term “cache” to refer to the operating system’s buffer cache and use the term “memory pool” for Berkeley DB’s own cache.) When a miss occurs and Berkeley DB asks the operating system to read a page from the database file, the operating system first checks in the buffer cache for the corresponding block or blocks from the file. If the necessary blocks are not found, it reads them into the buffer cache and returns the contents of the requested page to Berkeley DB. On many operating systems—including the one that I have used for my experiments—the buffer cache is integrated with the virtual memory system. As a result, the failure to find a file block in the buffer cache is recorded as a *page fault* [Tan92], and the page-fault statistics maintained by the operating system can be used to determine the number of times that a Berkeley DB application performs a disk read.

The buffer cache provides some of the same functionality as the memory pool—storing file blocks in memory in an attempt to reduce the amount of disk I/O generated by file accesses—but it does not provide the degree of control that Berkeley DB needs to provide the transaction-related guarantees discussed in Section 4.1.4. Therefore, Berkeley DB must provide its own cache, and this can lead to the unfortunate side-effect of *double buffering*, in which database pages are stored in both the memory pool and the buffer cache. This effectively reduces the number of database pages that can fit in a given amount of physical memory.

4.1.3 Locking Subsystem

Berkeley DB's locking subsystem synchronizes concurrent accesses to a database, allowing multiple threads of control to read and modify a database at the same time without interfering with each other. Locking is performed automatically by the access methods. For example, when a thread modifies a data item by invoking the appropriate Berkeley DB function, that function obtains the locks needed to read and write the requisite pages of the database. Locks are acquired on a per-page basis. A *read lock* is acquired before a page is read, and a *write lock* is acquired before a page is modified. If a thread holds a read lock for a page and it then acquires a write lock for the same page, it is said to perform a *lock upgrade*.

Berkeley DB enforces *multiple-reader, one-writer* semantics: multiple threads performing read-only operations can access the same pages in the database simultaneously, but a thread that wishes to update the database must obtain exclusive access to the pages that it wishes to modify. To ensure these semantics, a thread is forced to wait when it: (1) attempts to acquire a read lock for a page, and another thread already holds a write lock for that page; or (2) attempts to acquire a write lock for a page, and another thread already holds a read or write lock for that page.

In Btree databases, a technique known as *lock coupling* is used to improve concurrency: as the tree is descended, the lock on the current page is released once the next page to be searched has been locked. When modifying a data item, a write lock is only needed for the leaf page that contains the item; the internal pages are locked using read locks.[2]

When locking is used, it is possible for two or more threads of control to *deadlock* if they each hold a lock that prevents one of the other threads from making forward progress. For example, consider a situation in which thread *A* and thread *B* each hold a read lock for page *P*. Before either thread can upgrade its read lock by acquiring a write lock for *P*, the other thread must relinquish its read lock for *P*. Therefore, if both *A* and *B* try to perform a lock upgrade for *P*, neither will be able to make forward progress. Berkeley DB can detect when two or more operations are deadlocked, and it forces one of the offending operations to surrender its locks and return an error; this continues until one of the operations is able to proceed.

4.1.4 Transaction Subsystem

The transaction subsystem allows a set of operations to be grouped together and treated as a unit known as a *transaction*. For example, an application may want to apply a set of changes to a database in such a way that they all occur at once. In Berkeley DB, this is achieved by surrounding the relevant operations with function calls signaling the begin and end of a transaction, and by instructing the relevant operations to operate as part of the transaction.

Berkeley DB guarantees the four ACID properties of transactions: atomicity, consistency, isolation, and durability [Gra93]. Atomicity means that the changes made in the context of a transaction are applied to the database all at once or not at all. If the transaction is *committed*, all of the changes are applied together. If the transaction is *aborted*, none of the changes are applied. Consistency involves the integrity of the database. For example, it may be the case that one entry of a database must always store the sum of several other entries. Provided that an application performs the operations needed to maintain such an invariant within the context of a

2. The one exception is if the modification results in the leaf page being split or merged, in which case one or more internal pages will also be write-locked because they, too, will be modified as part of the split or merge.

transaction, the other three transaction properties will ensure that users always see a consistent picture of the data. Isolation means that a transaction is free from the interference of other threads of control. For example, if a thread of control accesses a particular page as part of a transaction, no other thread can modify that page for the duration of the transaction. Durability means once a transaction is committed, the changes made during the course of the transaction will survive any subsequent application or system failure.

One technique that Berkeley DB uses to ensure the four ACID properties is called *rigorous two-phase locking*. This means that locks acquired during the course of a transaction are held until the end of the transaction, at which point they are all released. The one exception to this rule stems from the practice of lock coupling (Section 4.1.3), which leads locks on internal pages to be released as a Btree is descended. As a result of this practice, it is typically the case that only leaf pages remain locked for the duration of a transaction.[3] This fact—coupled with the fact that almost all write locks are acquired for leaf pages (Section 4.1.3)—means that leaf pages are the primary source of lock contention.

When a set of operations is performed in the context of a transaction, all locks acquired as part of those operations are acquired on behalf of the transaction. If one of the operations returns an error because of deadlock, the application must abort the transaction so the deadlock can be resolved.

4.1.5 Logging Subsystem

The logging subsystem maintains a log file that records all changes to the database and all commits of transactions that made changes to the database. Berkeley DB employs a policy called *write-ahead logging*, which requires that before a change is written to the database file (e.g., before a dirty page is evicted from the memory pool), the log entry describing that change must be written to stable storage. This policy allows the system to restore the database file to a consistent state after a system or

3. If a Btree node is split or merged—which involves modifying one or more internal nodes—then internal pages can also remain locked until the transaction is committed or aborted.

application failure. To recover from a failure, Berkeley DB reads the log file and ensures that any changes that were part of a committed transaction are reflected in the database file, and that any changes that were part of an aborted or incomplete transaction are undone.

Log information is stored in an in-memory log buffer until either a transaction commits or the log buffer overflows, and then it is written to stable storage. Recent versions of Berkeley DB also support *group commit*, which allows multiple transaction commits to be logged using a single filesystem write. The version of Berkeley DB used in the experiments—version 4.0.14—included this feature.

4.2 The Knobs

Berkeley DB provides a number of knobs that can be adjusted in an attempt to improve performance. The experiments in Chapter 7 focus on four of them, and the sections below consider each knob in turn, discussing the ways in which it can affect the system's performance.

4.2.1 Page Size

The first knob, *page_size*, governs the size of pages used for a database; it can take on any power of two between 512 bytes and 64 kilobytes (KB). This knob affects the physical layout of the database, and thus it can affect both the amount of I/O and the amount of lock contention that a given workload will produce.

The impact of the *page_size* knob on I/O performance stems from at least two different mechanisms. First, items that are too large to fit on a regular page are forced into special *overflow pages*. Overflow pages exist outside of the ordinary Btree structure, and thus they tend to be more costly to access. In particular, because an overflow page contains data for only one item, it is less likely to be found in the memory pool than a leaf or internal page that contains data for multiple items (unless the item stored on the overflow page is accessed more frequently than all of the leaf or internal page's items combined). Moreover, when overflow pages are brought into the memory pool, they tend to reduce its overall efficiency by reducing the total number of items stored in the memory pool. The smaller the page size, the more likely it is that overflow pages will be created and I/O performance will suffer.

The *page_size* knob can also reduce I/O performance if the size chosen is smaller or larger than the block size of the filesystem on which the database resides. If the page size is smaller than the block size, then the eviction of a dirty page from the memory pool can lead to an extra disk read. More specifically, if the block containing the dirty page is not in the buffer cache, the operating system will need to read it in before modifying it, so that the unmodified portions of the block (the ones that do not contain the page) can be maintained. These extra reads can also make the buffer cache less efficient, since they tend to consume more memory than would otherwise be needed. On the other hand, if the page size is larger than the block size, the system may end up reading more data than is necessary. In particular, some operating systems are configured to recognize when an application is accessing a file sequentially and to respond by prefetching some number of the blocks that follow the ones requested by the application. Choosing a page size that is large enough to trigger prefetching can lead to unnecessary I/O. Because of these potential effects, Berkeley DB's default policy is to use a page size equal to the filesystem's block size, but other considerations may outweigh the benefits of using pages of this size.

The *page_size* knob can also affect the amount of lock contention in the system and thus the degree of concurrency that the system can sustain. Because Berkeley DB uses page-level locking, smaller page sizes tend to reduce lock contention by decreasing the probability that more than one thread of control will attempt to access the same page at the same time. However, smaller page sizes can also increase lock contention if they lead to more overflow pages. Modifying an item in an overflow page involves freeing the page containing the old version of the item and allocating a page for the new version; to do this, a special metadata page must be locked. If there are enough updates of overflow items, this metadata page can become a significant source of contention.

4.2.2 Minimum Keys Per Page

The second knob that I consider is the minimum keys per page setting (*min_keys*), which indicates the smallest number of keys that can exist on a single page in a Btree database. In combination with the *page_size* knob, this knob governs the maximum size of an entry (either key or value) that can be stored in a regular database page (*max_onpage*), according to the following formula:

$$max\_onpage = page\_size/(2 \cdot min\_keys) \qquad \text{(EQ 15)}$$

All entries larger than this maximum size are forced into overflow pages. Although overflow pages are ordinarily undesirable for the reasons discussed in the previous section, there are situations in which it makes sense to decrease *max_onpage* by increasing *min_keys*. In particular, if a database has infrequently accessed large items, increasing *min_keys* in order to force these items into overflow pages can allow more of the smaller, frequently accessed items to fit in the memory pool.

4.2.3 DB_RMW

As discussed in Section 4.1.3, threads may end up waiting—and may even become involved in a deadlock—when they attempt to perform a lock upgrade for pages that are already locked by other threads. To avoid waits and deadlocks that result from attempted lock upgrades, Berkeley DB allows a thread to acquire a write lock when reading an item, which eliminates the need to perform a lock upgrade if the thread subsequently decides to modify the item. A thread specifies this option by using the DB_RMW flag when performing a read (where RMW stands for read-modify-write). In the experiments reported in Chapter 7, I consider a binary *db_rmw* knob that indicates whether threads should use the DB_RMW flag when reading items that they will consider modifying.

Although using the DB_RMW flag may reduce the number of aborts due to attempted lock upgrades, it can also lead to increased lock contention and to more aborts due to lock requests that do not involve upgrades. The potential negative effects of this flag stem from the fact that it causes threads to hold write locks for longer than they otherwise would. In fact, if a thread needs to read an item before it can decide whether to modify it, using the DB_RMW flag can lead it to acquire write locks unnecessarily. In such cases, the probability that an item will be subsequently updated can be a key factor in determining whether the DB_RMW flag should be used.

4.2.4 Deadlock-Resolution Policy

When Berkeley DB detects that a deadlock has occurred, it rejects one of the offending lock requests in an attempt to allow the remaining requests to proceed. A number of different policies can be used to determine which request to deny. They include:

- *maxlocks*: reject the lock request made by the locker holding the most locks
- *minlocks*: reject the lock request made by the locker holding the fewest locks
- *minwrite*: reject the lock request made by the locker holding the fewest write locks
- *oldest*: reject the lock request made by the locker with the oldest locker ID
- *random*: reject a random lock request
- *youngest*: reject the lock request made by the locker with the newest locker ID

I will refer to the knob that governs the choice of deadlock-resolution policy as the *deadlock_policy* knob. DB uses the random policy by default.

When operations are performed in the context of a transaction, the transaction is the locker and the ID of the transaction serves as the locker ID. When a transaction is chosen to resolve a deadlock, the application typically aborts the transaction and retries it. All updates that were performed as part of the original transaction must be undone during the abort and then reapplied; read-only operations are simply redone. In general, update operations are more costly to redo, because: (1) the original, provisional changes must be undone when the transaction is aborted; and (2) reapplying the changes involves both acquiring new write locks (which tend to be more difficult to acquire than read locks) and producing additional log entries.

Because several of the above policies are based on the number and type of locks held by a transaction, and because the locks that a transaction holds typically reflect the types of operations that it has performed, the choice of deadlock-resolution policy can affect the number and types of operations that must be redone when aborted transactions are retried. For example, transactions selected to resolve a deadlock under the minwrite policy tend to hold fewer write locks on average than transactions chosen using one of the other policies, and thus applications may end up redoing fewer update operations under this policy. However, the number of redone read-only operations also matters, and thus the minwrite policy will not always be optimal.

4.3 Workload Characteristics

Tuning the knobs of a software system like Berkeley DB involves determining the optimal knob settings for a given workload. This section describes the variables that I use to characterize workloads that run on Berkeley DB.

4.3.1 Transaction Characteristics

The transactions run on the system are characterized by five workload characteristics: the percentage of transactions that use a cursor to perform sequential accesses (*pct_cursors*), the average numbers of items accessed by transactions that use a cursor (*items_curs*) and those that do not (*items_non_curs*), and two variables (*pct_updates* and *pct_writes/upd*) that specify the degree to which transactions modify items in the database. I define an *update transaction* to be any transaction that *considers* modifying a set of data items. It reads each of the items in the set, but it may not modify all of them, perhaps because it needs to see the contents of an item before it can determine whether it should be modified. The *pct_updates* variable represents the percentage of transactions that are update transactions, and the *pct_writes/upd* variable represents the average percentage of the items accessed by an update transaction that are actually modified.

The values of these five variables can be measured by instrumenting either the application performing the transactions or Berkeley DB itself. All that is required is to maintain a series of counts of the relevant events—e.g., accessing items with a cursor—and to periodically compute the values of the workload characteristics from these counts.

4.3.2 Access Locality

Most database access patterns exhibit some degree of *locality*, accessing certain items or pages in the database more often than others. The degree of access locality that a workload displays has at least two potential impacts on performance:

- It affects the likelihood of finding a given page in the memory pool or buffer cache, and thus can affect the rates at which misses and page faults occur (see Section 4.1.2).
- It affects the degree of contention over locks, and thus can affect the number of lock waits and deadlocks per transaction (see Section 4.1.3).

In tuning Berkeley DB, I have focused on two aspects of locality: locality based on the location of the accessed items in the database (*page locality*) and locality based on the size of the accessed items (*size locality*).

To capture page locality, I make use of an abstraction known as the *locality set*, which I define as the collection of leaf pages that receive “most” of the accesses. I do not include internal pages in the locality set because the number of such pages is so much smaller than the number of leaf and overflow pages, and the internal pages tend to fit in the memory pool, regardless of the settings chosen for the *page_size* and *min_keys* knobs. I also omit overflow pages from consideration because they are unlikely to be accessed frequently enough to be included in the set of most frequently accessed pages. The reason for this stems from the fact that accessing an overflow-page item involves first accessing the leaf page containing the key for that item, and thus the associated leaf pages—which each contain keys for multiple items—are almost always accessed much more frequently than the overflow pages themselves, which each contain only a single item.

Leaf pages can be chosen for inclusion in the locality set based on observations of accesses to the database. For example, if a database application uses the date field of its records as the key, and if an application-maintained trace of accesses to the database shows that 95% of them go to records from the past year, leaf pages containing those dates could be characterized as the locality set. More generally, the locality set could be estimated in an application-independent manner from information about page accesses in the Berkeley DB log files. To take into account read-only accesses (which are not logged by write-ahead logging), Berkeley DB could be modified to maintain a per-page access count that is incremented whenever the page is accessed, and to log the value of a page's access count before evicting it from the cache. (Note that evictions of unmodified pages would still not require any synchronous writes to disk, because log records containing the counts could be buffered until the next log write was needed.) At any point in time, these logged counts—along with the access counts of pages currently in the memory pool—could be used to determine the current contents of the locality set.

Given a particular locality set, I define page locality (*page_loc*) as the percentage of leaf pages that are included in the locality set, and *leaf_loc_rate* as the percentage of leaf-page accesses that go to pages in the locality set. For a given value of *leaf_loc_rate*, smaller *page_loc* values correspond to higher degrees of locality. When *page_loc* equals *leaf_loc_rate*, the leaf pages are accessed uniformly.

Both *page_loc* and *leaf_loc_rate* may depend on the knob settings chosen to configure the database. For example, if the locality-set pages contain most of the large items in the database, the value of *page_loc* will decrease when these large items are forced into overflow pages, because fewer leaf pages will be needed to hold the smaller items. Therefore, it is necessary to specify the associated workload characteristics with respect to one of the possible combinations of knob settings. I define *def_page_loc* as the percentage of leaf pages in the locality set when the database is configured using the default knob settings for *page_size* and *min_keys*, and *def_leaf_loc* as the percentage of leaf-page accesses that go to the locality set under those same settings. When the database is configured with non-default *page_size* or *min_keys* settings, it may not be possible to determine the current values of these workload characteristics. However, measurements of *page_loc*, *leaf_loc_rate*, and other aspects of the current state of the system can be used to determine the probabilities associated with the possible values of these variables. Section 5.5.1 describes how probabilistic-reasoning techniques can be used to handle situations in which a workload characteristic is unobservable.

The second aspect of access locality that I consider is size locality, which measures the degree to which accesses are based on the size of the data items. Size locality can affect performance because large items may be forced into overflow pages under one or more of the knob combinations, and accessing these items could thus lead to additional misses and disk reads (Section 4.2.1). Therefore, I define *size_loc* as the percentage of accesses made to items that are large enough to be forced into overflow pages in one or more of the database configurations being considered. This value of this variable could be estimated from information in the log files about the lengths of the items that are modified, but instrumenting the application would probably be an easier and more thorough way to measure this workload characteristic.

4.3.3 Concurrency

The final workload characteristic that I consider is *concurrency*—the number of transactions that simultaneously access the database. Higher values of concurrency tend to lead to more lock contention—and thus to more lock waits and more deadlocks. The value of this variable could be measured by instrumenting either the application or Berkeley DB itself. Berkeley DB already keeps track of the maximum number of concurrent lockers (entities, including transactions, that hold locks), and it could be modified to maintain some measure of the average of the number of concurrent lockers in the recent past—perhaps using exponential smoothing [Mak98, Chapter 4].

4.4 An Influence Diagram for Berkeley DB

Figure 4.2 shows the influence diagram that I have designed to tune Berkeley DB, with the goal of maximizing the throughput of the system (i.e., the average number of transactions committed per second). To increase readability, I have omitted the informational arcs that indicate that all of the workload characteristics are observed before the decisions are made.

The influence diagram was created using information from the Berkeley DB documentation [Sle01] and input from one of the system's designers. Brief descriptions of the nodes in the influence diagram are given in Table 4.1, and additional detail about both the nodes and arcs is provided below. Chapter 5 explains how to construct a model of this type for an arbitrary software system.

4.4.1 Decision Nodes

The knobs being tuned—*page_size*, *min_keys*, *db_rmw*, and *deadlock_policy*, as described in Section 4.2—are represented by the four rectangular decision nodes in Figure 4.2. As mentioned in Section 3.2.1, it is necessary to use informational arcs to impose an ordering on the decisions in an influence diagram. I selected the following ordering: *page_size*, *min_keys*, *db_rmw*, *deadlock_policy*. As a result, informational arcs are shown from *page_size* to *min_keys*, from *min_keys* to *db_rmw*, and from *db_rmw* to *deadlock_policy*. Section 5.1.2 discusses the impact of a given knob ordering on the accuracy and efficiency of influence diagrams developed for software tuning.

4.4.2 Value Nodes

Although I am attempting to maximize throughput, a single value node corresponding to throughput would have a large number of parents, and it would require too much training data to learn an expected value for each combination of the value of its parents. As a result, I have chosen to use the two diamond-shaped value nodes shown

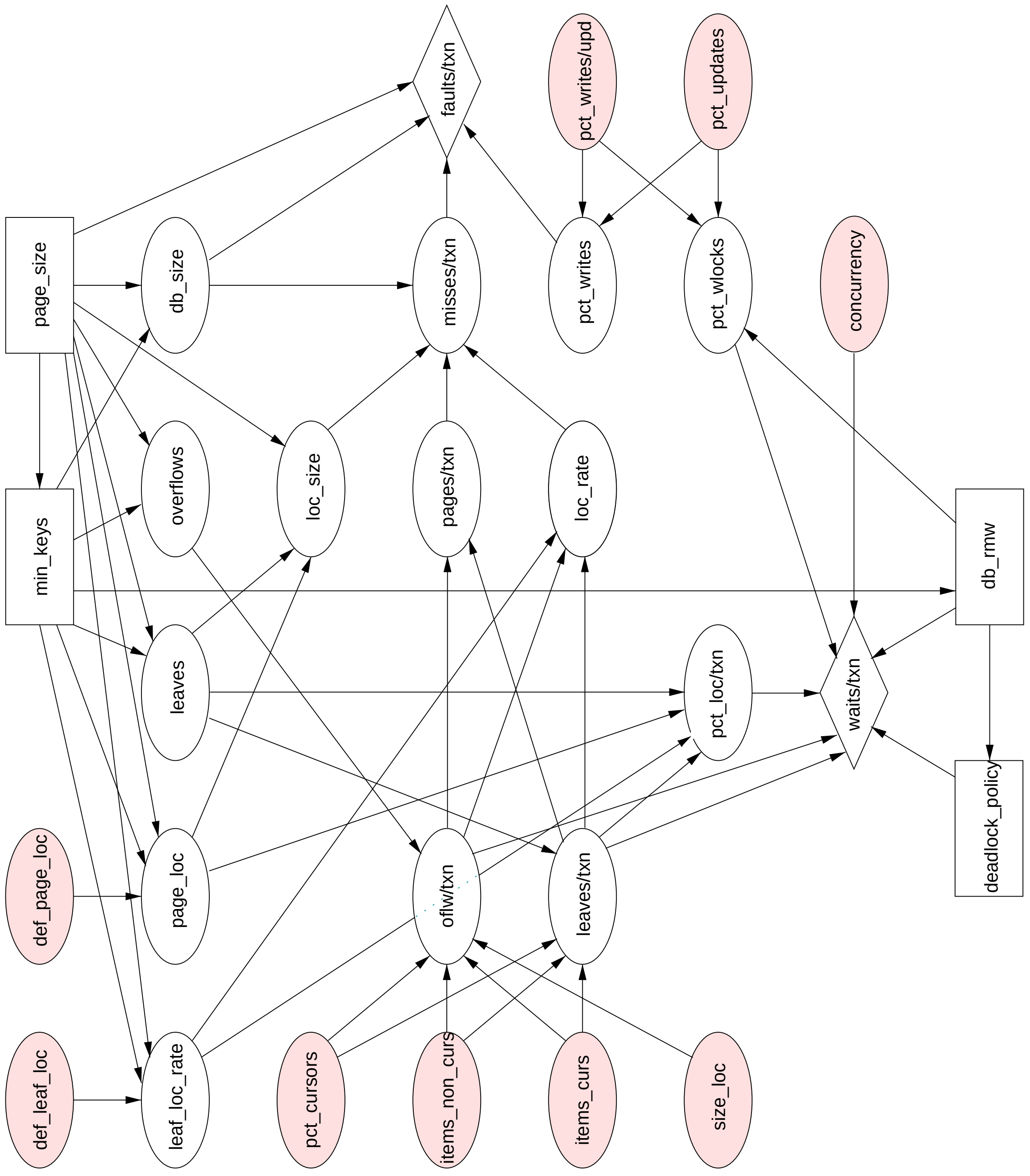


Figure 4.2. An influence diagram for tuning the Berkeley DB database system. The shaded nodes are the root chance nodes, which represent characteristics of the workloads faced by the system. Most if not all of these characteristics would be observed before a tuning decision is made, but the informational arcs that would make this fact explicit have been omitted for the sake of readability. Brief descriptions of the nodes are given in Table 4.1, and additional detail about both the nodes and arcs is provided in Section 4.4.

Table 4.1. Overview of the nodes in the influence diagram in Figure 4.2. See Sections 4.4.1, 4.4.2, and 4.4.3 for more detail. Update transactions are defined in Section 4.3.1, and the locality set is defined in Section 4.3.2. Per-transaction statistics are obtained by dividing by the number of *committed* transactions.

Node name	Description
concurrency	number of concurrent threads performing transactions
deadlock_policy	the policy used to select the transaction that will be aborted to resolve a deadlock
def_page_loc	percent of leaf pages in the locality set under the default *page_size* and *min_keys* settings
def_leaf_loc	percent of leaf-page accesses that go to the locality set under the default *page_size* and *min_keys* settings
db_rmw	indicates whether the DB_RMW flag is used to acquire write locks when reading an item in the database
db_size	size of the database
faults/txn	mean number of page faults in the OS buffer cache per transaction
items_curs	mean number of items accessed by transactions that use a cursor
items_non_curs	mean number of items accessed by transactions that do not use a cursor
leaf_loc_rate	percent of leaf-page accesses that go to the locality set
leaves	total number of leaf pages
leaves/txn	mean number of leaf pages accessed per transaction
loc_rate	percent of the leaf- and overflow-page accesses that go to the locality set
loc_size	size of the locality set
min_keys	minimum number of keys per page
misses/txn	mean number of misses in the memory pool per transaction
overflows	total number of overflow pages
oflw/txn	mean number of overflow pages accessed per transaction
page_loc	percent of leaf pages in the locality set
page_size	size of each page in the database
pages/txn	mean number of leaf and overflow pages accessed per transaction
pct_cursors	percent of transactions that use a cursor
pct_loc/txn	mean percentage of the locality set accessed per transaction
pct_wlocks	percent of accesses that involve acquiring a write lock
pct_writes	percent of accesses that involve modifying an item
pct_writes/upd	percent of items accessed as part of update transactions that are actually modified
pct_updates	percent of transactions that are update transactions
size_loc	percent of accesses made to items large enough to be forced into overflow pages under one or more knob settings
waits/txn	mean number of lock waits per transaction

in Figure 4.2. They correspond to per-transaction averages[4] of the number of page faults (*faults/txn*) and the number of times that a thread waits to acquire a lock (*waits/txn*). These two quantities reflect the types of performance losses that can be affected by the knobs I am tuning. The *faults/txn* node captures the impact of the knobs on I/O performance because of the connection between disk reads and page faults discussed in Section 4.1.2. The *waits/txn* node captures the impact of the knobs on lock contention. The values of these nodes will be weighted so that minimizing their sum is equivalent to maximizing throughput. Section 7.2.1 explains the process used to learn the weights from training data.

For databases that fit in the operating system's buffer cache, page faults may seldom or never occur. In this case, it would probably be necessary to replace the *faults/txn* value node with a value node representing the number of misses in the memory pool, converting the existing *misses/txn* chance node to a value node. In my experiments, the databases are too large to fit in memory, and therefore page faults are a better predictor of performance because, as discussed in Section 4.2.1, the number of disk reads (and thus page faults) per memory-pool miss depends on the *page_size* value used to configure the database.

4.4.3 Chance Nodes and the Structure of the Model

The oval chance nodes in the diagram are of two types. The root chance nodes (the ones without parents) represent the characteristics of the workloads, and the intermediate chance nodes represent random variables that mediate the impact of the knob settings and workload characteristics on the value nodes. Information about the workload characteristics was already presented in Section 4.3. The following sections describe the intermediate chance nodes and explain the arcs into both the intermediate chance nodes and the value nodes.

4. More specifically, I compute these averages by dividing the total number of times that an event occurs by the total number of *committed* transactions. Computed in this way, the averages reflect the amount of time wasted during the course of an average transaction, and larger averages thus tend to mean smaller throughputs.

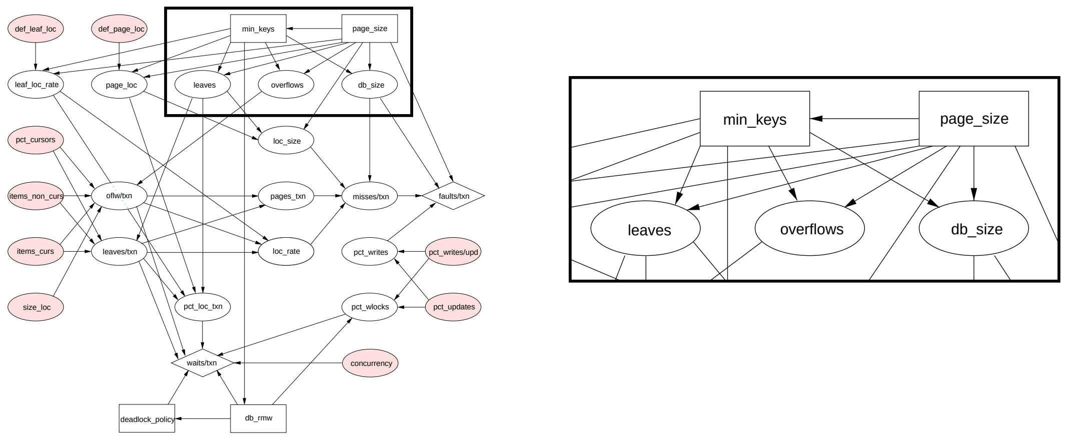


Figure 4.3. The *leaves, overflows,* and *db_size* nodes and their parents. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. The three chance nodes shown all inherit from the *min_keys* and *page_size* decision nodes, as discussed in Section 4.4.3.1.

4.4.3.1 Intermediate Chance Nodes and Their Parents

The settings selected for the *page_size* and *min_keys* knobs govern the number of leaf pages (*leaves*) and the number of overflow pages (*overflows*) in the database; see Sections 4.2.1 and 4.2.2 for more details. As a result, these knob settings also determine the size of the database (*db_size*). The arcs connecting the *page_size* and *min_keys* nodes to the above-mentioned chance nodes (Figure 4.3) reflect the causal relationship between those knobs and the layout of the database.

There are three intermediate chance nodes that represent the rates at which different types of Btree pages are accessed; the portion of the influence diagram that includes these nodes is highlighted in Figure 4.4. The *leaves/txn* node represents the expected number of leaf pages accessed per committed transaction, which can affect both the I/O performance and the level of lock contention in the system. The value of this node depends on the numbers of items accessed with and without a cursor (*items_curs* and *items_non_curs*, respectively), the percentage of transactions that use a cursor (*pct_cursors*), and the mean number of data items per leaf, the last of which is fully captured for a particular collection of data items by the number of leaves (*leaves*). Therefore, *leaves/txn* inherits from these four nodes. Similarly, *oflw/txn*—the average number of overflow pages accessed per committed transaction—

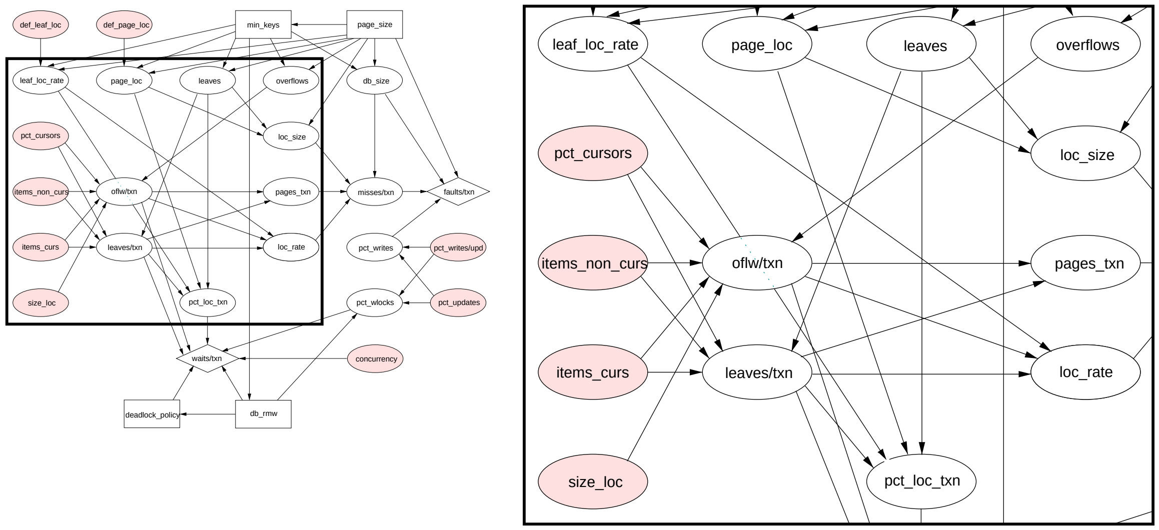


Figure 4.4. The *leaves/txn, oflw/txn,* and *pages/txn* nodes and their parents. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. An explanation of the arcs into the *leaves/txn*, *oflw/txn*, and *pages/txn* nodes is provided in Section 4.4.3.1.

inherits from *items_curs*, *items_non_curs*, and *pct_cursors*, and it also inherits from *size_loc*, which indicates the percentage of accesses going to items that could be in overflow pages, and from *overflows*, which indicates how many overflow pages are actually present in a given configuration. I also include a variable called *pages/txn* that represents the mean number of leaf *and* overflow pages accessed per committed transaction—a variable that is relevant to the performance of the memory pool; *pages/txn* naturally inherits from *leaves/txn* and *oflw/txn*.

Several of the intermediate chance nodes are related to the concept of page locality discussed in Section 4.3.2. The portion of the influence diagram that includes these nodes is highlighted in Figure 4.5. The value of the *page_loc* variable is based on the page locality under the default *page_size* and *min_keys* settings (*def_page_loc*) and on the current settings for those knobs, and thus it inherits from *def_page_loc*, *page_size*, and *min_keys*. Similarly, *leaf_loc_rate* inherits from *def_leaf_loc, page_size*, and *min_keys*. The size of the locality set (*loc_size*) is computed from the values of *page_loc* and *leaves*—which together determine the number of leaves in the locality set—and from the setting for *page_size*—which determines the size of each leaf—according to the following equation:

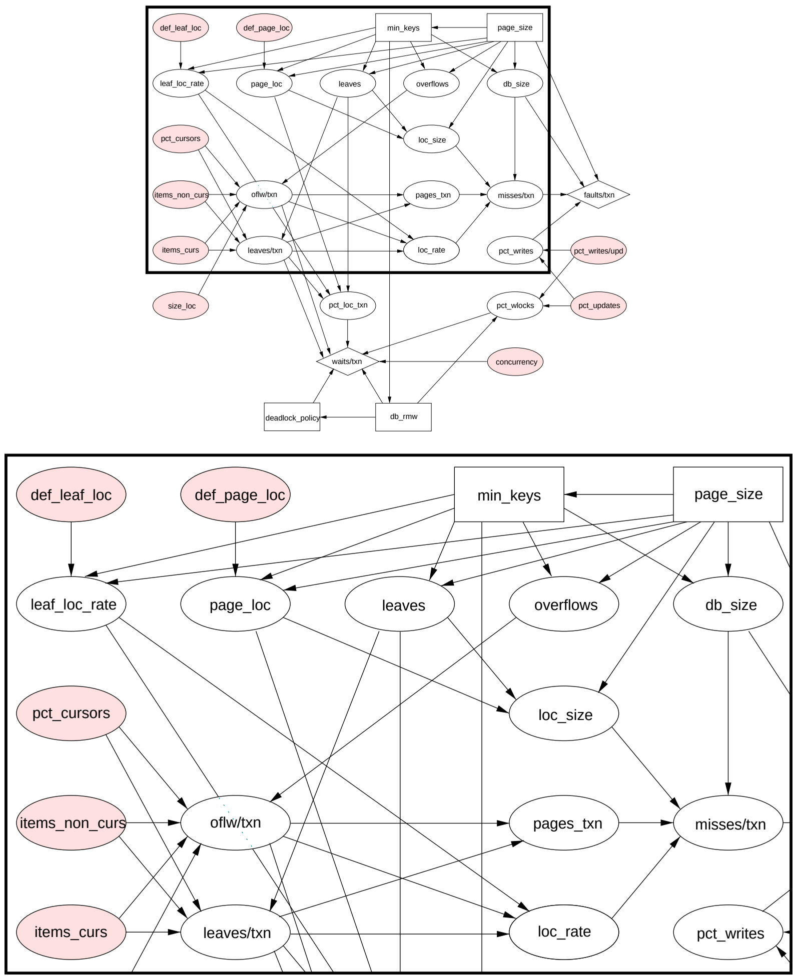


Figure 4.5. The *page_loc, leaf_loc_rate, loc_rate,* and *loc_size* nodes and their parents. The figure at the bottom focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure at the top. An explanation of the arcs into the *page_loc*, *leaf_loc_rate*, *loc_rate*, and *loc_size* nodes is provided in Section 4.4.3.1.

$$loc\_size = \left(\frac{page\_loc}{100}\right) \cdot leaves \cdot page\_size \qquad \text{(EQ 16)}$$

It thus inherits from *page_loc*, *leaves*, and *page_size*. .

In assessing the likelihood of a miss in the memory pool, one relevant factor is the percentage of page accesses that involve pages in the locality set, because such pages are more likely to reside in the cache. The *leaf_loc_rate* variable represents the percentage of *leaf-page* accesses that go to the locality set. However, when estimating memory-pool misses, overflow-page accesses also matter. To take this into account, I define the *loc_rate* variable, which represents the percentage of the accesses to either leaf or overflow pages that go to the locality set. (I ignore internal-page accesses under the assumption that they almost never miss.) If the settings for *page_size* and *min_keys* produce no overflow pages, *loc_rate* will equal *leaf_loc_rate*. If overflow pages are produced, the value of *loc_rate* will decrease as the number of overflow-page accesses increases, because overflow pages are never in the locality set. More specifically, the value of *loc_rate* can be computed as follows:

$$loc\_rate = leaf\_loc\_rate \cdot \frac{leaves/txn}{leaves/txn + oflw/txn} \qquad \textbf{(EQ 17)}$$

Thus, *loc_rate* inherits from *leaf_loc_rate*, *leaves/txn*, and *oflw/txn* (Figure 4.5).

One factor that influences the level of lock contention in the system is the degree to which accesses are concentrated on a small number of leaf pages. I capture this aspect of a workload using the variable *pct_loc/txn*, which represents the percentage of the locality-set leaves that are accessed—and thus locked—per committed transaction. This variable depends on the number of leaves per transaction (*leaves/txn*), the number of leaves in the locality set (which is simply the product of *page_loc* and *leaves*), and *leaf_loc_rate*, which measures the percentage of leaf-page accesses that go the locality set; *pct_loc/txn* thus inherits from these four nodes (Figure 4.6).

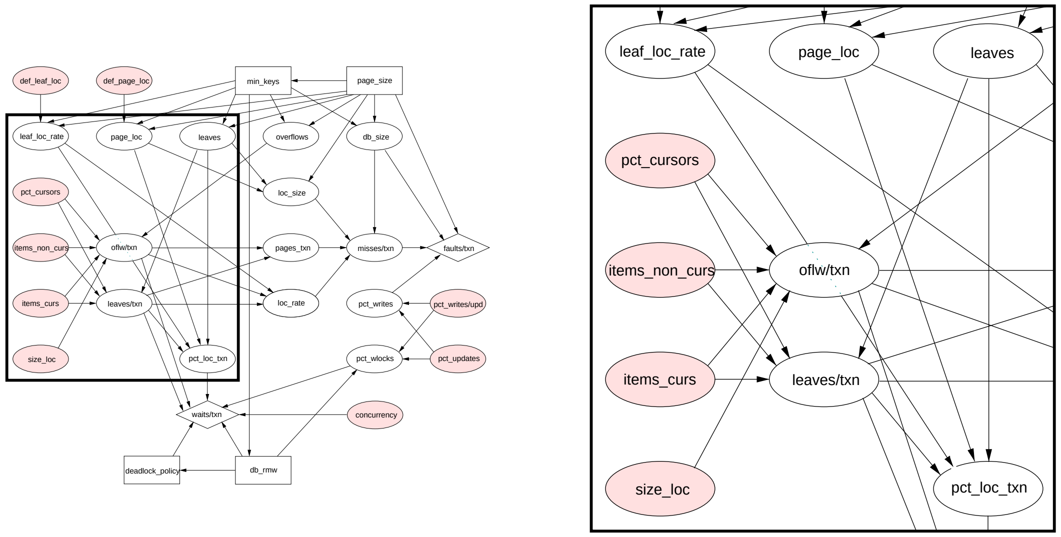


Figure 4.6. The *pct_loc/txn* node and its parents. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. An explanation of the arcs into the *pct_loc/txn* node is provided in Section

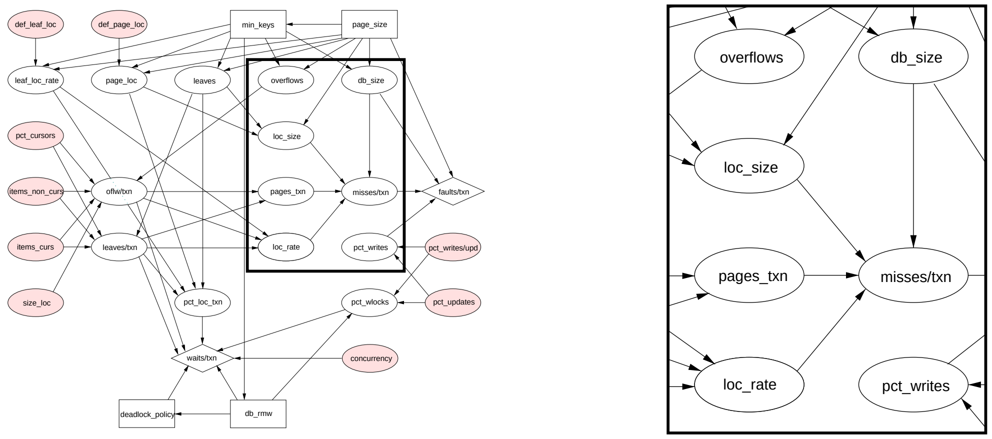


Figure 4.7. The *misses/txn* node and its parents. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. An explanation of the arcs into the *misses/txn* node is provided in Section 4.4.3.1.

The expected number of misses per committed transaction (*misses/txn*) depends on the expected number of leaf and overflow pages accessed per transaction (*pages/txn*), and the degree to which these pages are present in the cache. This latter factor depends on what percentage of the accessed pages are in the locality set (*loc_rate* as described above) and on the sizes of both the locality set (*loc_size*) and the database as a whole (*db_size*). I therefore draw arcs from these three nodes and from *pages/txn* to *misses/txn* (Figure 4.7).

Update transactions can affect both I/O performance and the degree of lock contention in the system. Two intermediate chance nodes that are related to update transactions are highlighted in Figure 4.8. As described in Section 4.2.1, one impact of update transactions on I/O performance stems from the fact that evicting dirty pages from the memory pool can produce extra page faults. I thus include a node representing the percentage of accessed items that are modified (*pct_writes*), because this variable gives an indication of the likelihood that an evicted page is dirty. The value of this node is proportional to the product of *pct_updates and pct_writes/upd*, and thus it inherits from these two nodes.

Update transactions also increase the likelihood of lock contention because they acquire write locks, which can be held by only one thread at a time. The *pct_wlocks* node represents the percentage of accesses to data items that involve

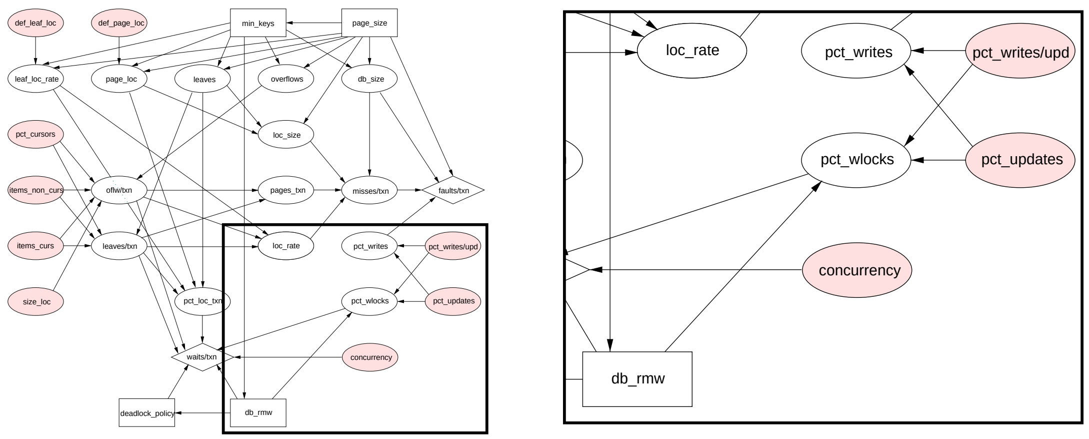


Figure 4.8. The *pct_writes* and *pct_wlocks* nodes and their parents. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. An explanation of the arcs into the *pct_writes* and *pct_wlocks* nodes is provided in Section 4.4.3.1.

acquiring a write lock. If the DB_RMW flag is not being used, *pct_wlocks* is computed by multiplying *pct_updates* and *pct_writes/upd*; if the DB_RMW flag is being used, a write lock is acquired for every leaf page accessed by an update transaction, and thus *pct_wlocks* is equal to *pct_updates*. Given these methods of determining the value of *pct_wlocks*, it naturally inherits from *pct_updates*, *pct_writes/upd*, and *db_rmw*.

4.4.3.2 Parents of the Value Nodes

To conclude the presentation of the influence diagram, this section explains the arcs into the value nodes by considering how each value node is affected by its parents.

The *faults/txn* value node has four parents, as shown in Figure 4.9. The variable with the most obvious relevance is the number of misses in DB's memory-pool cache (*misses/txn*); it is naturally one of its parents. However, it is possible to miss in the memory pool and still avoid a page fault by finding the corresponding file block or blocks in the operating system buffer cache; the overall size of the database file (*db_size*) influences the likelihood of this happening, and thus it is also a parent of *faults/txn*. Finally, if the database page size is less than the filesystem block size, the eviction of a dirty page from the memory pool may require a page fault in order to update the contents of the block containing the page. Therefore, *pct_writes* (which affects the likelihood of a dirty evict) and *page_size* are also parents of *faults/txn*.

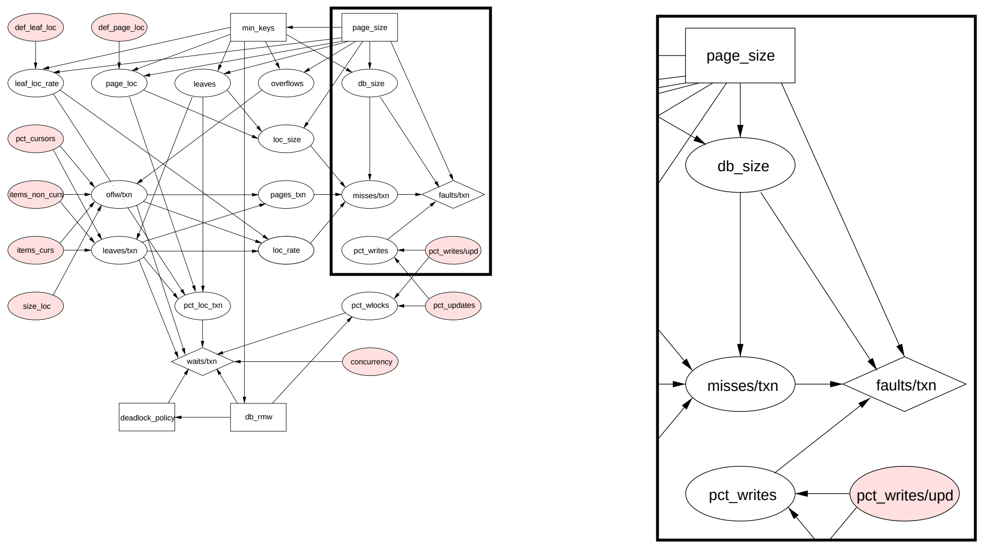


Figure 4.9. The *faults/txn* value node and its parents. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. An explanation of the arcs into the diamond-shaped *faults/txn* node is provided in Section 4.4.3.2.

The parents of the *waits/txn* value node, which are shown in Figure 4.10, reflect the two main reasons for lock waits: (1) a thread attempts to acquire or upgrade a lock for a page for which another thread already holds a lock; and (2) a thread attempts to update an item in an overflow page and thus needs to access the DB metadata page, but another thread already holds a write lock on the metadata page (see Section 4.2.1). In the first case, only leaf pages are locked for the duration of a transaction and most of the leaves accessed are in the locality set. Therefore, larger values of *pct_loc/txn* make locks waits more likely, and this node is a parent of *waits/txn*. In the second case, what matters is the number of overflow pages accessed per transaction, and thus *oflow/txn* is also a parent of *contention*. Both types of lock waits only occur when some of the threads are acquiring write locks, and both are more likely under higher degrees of concurrency. Therefore, *waits/txn* inherits from *pct_wlocks* and *leaves/txn*—which together give an indication of the number of write locks acquired per transaction—and from *concurrency*. When combined with *db_rmw* (which is also a parent of *waits/txn*), the *leaves/txn* and *pct_wlocks* nodes also give an indication of the *total* number of locks—both read and write locks—acquired per transaction. As more locks are acquired, there are more opportunities for waits to

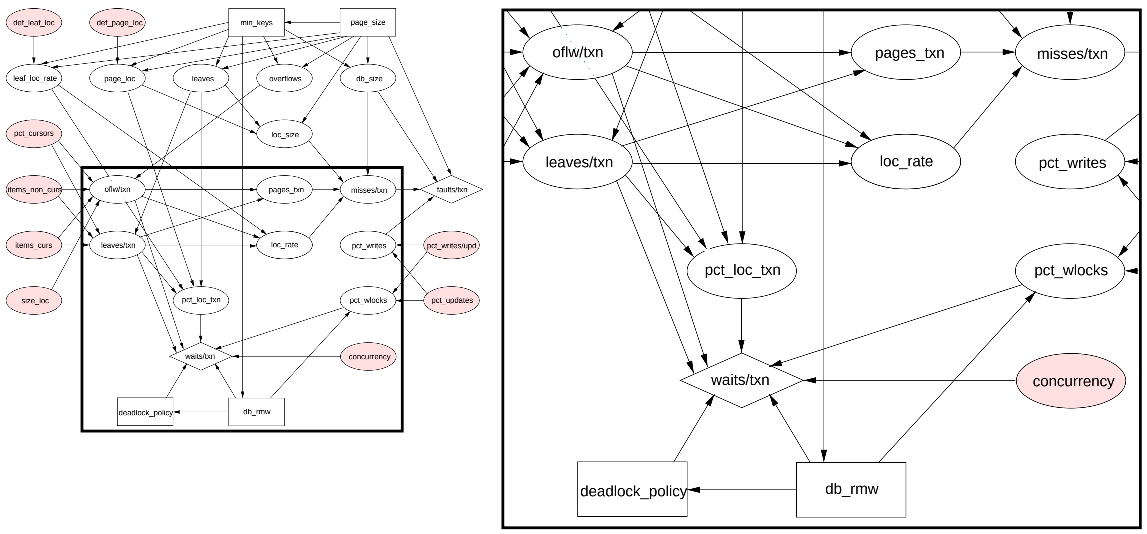


Figure 4.10. The *waits/txn* value node and its parents. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. An explanation of the arcs into the diamond-shaped *waits/txn* node is provided in Section 4.4.3.2.

occur. Finally, *waits/txn* will increase when deadlocks occur and transactions are aborted, because the locks held by an aborted transaction are reacquired when the transaction is retried. For a given level of lock contention, both the number of aborted transactions and the average number of leaves locked per aborted transaction can depend on the settings for the *db_rmw* and *deadlock_policy* knobs. Therefore, these knobs are also parents of *waits/txn*.

4.5 Conclusions

The Berkeley DB embedded database system is one example of a software system that could benefit from an automated approach to software tuning. The influence diagram presented in the previous section can serve as the foundation of a model-based software tuner for Berkeley DB, allowing such a tuner to determine the optimal knob settings for an arbitrary combination of workload characteristics. The next chapter explains how to construct and use this type of model to tune an arbitrary software system; it draws on lessons learned in designing and applying the influence diagram for Berkeley DB. Chapter 7 presents the results of experiments that use this influence diagram to tune Berkeley DB.

Chapter 5

Using an Influence Diagram for Software Tuning

This chapter presents a methodology for applying the probabilistic reasoning techniques described in Chapter 3 to software tuning. More specifically, it explains how an influence diagram and related learning and inference techniques can be used as the basis of an effective, model-based software tuner that satisfies the criteria described in Chapter 2.

The methodology consists of six steps: (1) designing the structure of the influence diagram model used by the tuner, (2) gathering the necessary training data, (3) discretizing the continuous variables in the model, (4) learning the model's parameters, (5) using the model to dynamically adjust the knob settings as the system runs, and (6) updating the parameters of the model over time. The following sections address each of them in turn.

5.1 Designing the Structure of the Model

Designing the structure of an influence diagram is a challenging task. It requires the knowledge of a domain expert—in this case, someone familiar with the design and inner workings of the software system being tuned—to determine which variables and arcs to include in the model. However, the design process only needs to be performed once for a given software system. The resulting model structure can be hard-coded into the tuner associated with the software system and shipped as part of the product, and the tuner itself can run without human intervention. The following sections offer some guidance about how to design an influence diagram for software tuning.

5.1.1 Choosing Variables for the Model

As discussed in Section 3.2.1, there are three types of nodes in an influence diagram—decision nodes, value nodes, and chance nodes. The chance nodes can be further subdivided into those with parents (*intermediate chance nodes*) and those without parents (*root chance nodes*). In an influence diagram for software tuning, the decision nodes are the knobs to be tuned, the value nodes are the performance measures to be optimized, and the root chance nodes are the relevant characteristics of workloads experienced by the software system. When choosing the variables to include in the model, it is helpful to begin with these three types of variables. This section first discusses each of these variable types in turn, and it then covers other guidelines for selecting variables.

5.1.1.1 Decision Nodes

The decision nodes—the knobs—will typically be well-defined, although it may be necessary to conduct preliminary experiments to determine whether it makes sense to tune a particular knob. For example, in my work with Berkeley DB, I performed a series of experiments to determine if the *db_rmw* and *deadlock_policy* knobs (Sections 4.2.3 and 4.2.4, respectively) have enough of an impact on performance to justify the effort needed to tune them. I used the workload generator described in Chapter 6 to simulate workloads with varying degrees of lock contention and I ran the workloads using different settings for these two knobs. These experiments demonstrated that different knob settings were optimal for different workloads, and that the performance of the system could be improved significantly by tuning these knobs.

5.1.1.2 Value Nodes

The variables chosen for the value node or nodes should reflect the goal of the tuning process—for example, to maximize the throughput of the system or to minimize its response time. However, even if the tuning goal involves a single variable such as throughput, it may not be feasible to design the model with a single value node. The reason for this stems from the need to limit the number of parameters for each node in the model, so that the model can be trained and reasoned with efficiently. The number of parameters associated with a node is exponential in the number of its parents, and a value node that corresponds to a single, overarching performance metric will typically have too many parents to be practical.

The potential problems associated with a single value node can be avoided by using multiple value nodes, each of which reflects one aspect of the system's performance. In particular, it can be helpful to consider variables that reflect different types of performance *losses* that the system can incur, as I have done with the *faults/txn* and *waits/txn* value nodes in the influence diagram for Berkeley DB (Figure 4.2). The advantage of this approach is that a given type of performance loss is typically affected by only a subset of the knobs and workload characteristics, which makes it easier to limit the number of its parents. This approach assumes that minimizing some linear combination of the performance-loss variables is equivalent to maximizing the overall performance of the system. Section 5.4 discusses the process of determining the coefficients of this linear combination.

In selecting variables to reflect performance losses, one possible approach is to use variables that measure the time wasted on activities that degrade performance. For example, a measure of the time spent waiting to acquire locks could be used to assess the impact of lock contention on performance. Ideally, minimizing an unweighted sum of these wasted-time measurements would maximize the performance of the system—i.e., the coefficients of the linear combination of the value nodes would all have a value of 1.0. I considered this approach in my work with Berkeley DB, but I concluded that it was impractical for at least two reasons. First, it can be difficult to accurately measure the times involved. For example, the operating system provides no record of the time spent resolving page faults, only a count of the number of times that page faults occur. Second, even when accurate time measurements are possible, they typically cannot be translated directly into performance. This is especially true in the case of multithreaded software systems, because the time that one thread spends waiting can be used by another thread to perform useful work. Thus, it would still be necessary to learn a set of coefficients for the linear combination of these nodes.

As a result of the difficulties involved in using time-based value nodes, I opted instead to use count-based statistics that reflect the relevant performance losses. Such counts are often provided by the operating system or by the software system being tuned. For example, I was able to obtain a count of the number of page faults from the Solaris operating system and a count of the number of lock waits from Berkeley DB. Indeed, the statistics provided by the software system being tuned and by the operating systems on which the software system runs can be a good source of ideas for value nodes to include in the model. If the necessary statistics are not available, it may be possible to instrument the software system to maintain the necessary counts. Typically, adding a counter to a system can be done easily and with minimal overhead. Adding a time-based statistic, on the other hand, is usually much more costly, because each time measurement incurs the overhead of making a system call into the operating system.

5.1.1.3 Root Chance Nodes

The root chance nodes are used to represent the relevant characteristics of workloads experienced by the software system. Workload characteristics are relevant if they affect the performance metrics that the tuner is attempting to optimize. In choosing these variables, it can be helpful to consider the workload characteristics that a human expert would need to consider in order to tune the knobs. For example, as discussed in Section 4.2.3, the *db_rmw* knob specifies whether a Berkeley DB application should use a special flag to acquire write locks when reading items that may later be modified; doing so eliminates the need to upgrade read locks to write locks and can thus reduce the number of deadlocks that occur. If a thread needs to read an item before it can decide whether to modify it, the desirability of using this flag depends on the probability that an item will be modified after it is read. Thus, one of the workload characteristics included in Figure 4.2 is *pct_writes/upd*, the percentage of items accessed by update transactions that are written after they are read. The other variables discussed in Section 4.3 provide additional examples of the types of workload characteristics that can be included in an influence diagram for software tuning.

When defining a workload characteristic, it is important to do so in a way that does not depend on the knob settings that are currently in use by the system; this ensures that the workload characteristic will have a unique value for a given workload. Otherwise, the corresponding chance node will not be a root chance node (because it will need one or more of the decision nodes as parents), and it will not be possible to enter its value as evidence before the influence diagram is evaluated. For example, in the influence diagram for Berkeley DB (Figure 4.2, page 69), the values of the *page_loc* and *page_loc_rate* variables—which measure the degree of access locality that a workload exhibits (Section 4.3.2)—*do* depend on the current knob settings, and thus they cannot be root chance nodes. To obtain workload characteristics with unique values for a given workload, I defined additional variables (*def_page_loc* and *def_loc_rate*) that represent the values of *page_loc* and *page_loc_rate* that would be measured under the default knob settings.

5.1.1.4 Unnecessary Variables

An influence diagram for automated software tuning does *not* need to include variables whose values are fixed or slowly changing. In particular, we can omit variables that describe either the platform on which the system runs (e.g., the amount of physical memory) or configurable aspects of the system that are not being tuned dynamically (e.g., the size of Berkeley DB's memory pool; see Section 4.1.2). Examples of slowly changing variables include ones that describe the data stored in a database (e.g., the mean item length). We can exclude such variables because they are implicitly captured by the parameters of the model, which are learned for a specific environment (Section 2.1). If any of the excluded variables change (e.g., if the contents of a database change over time), the process of updating the parameters over time should capture these changes. Section 5.6 describes how these parameter updates are performed.

5.1.1.5 Using Normalized Variables

When gathering training data for the model or observing the characteristics of a workload for which the system needs to be tuned, the relevant statistics are often measured over some interval. This interval can be either time-based (e.g., measure the number of page faults that occur during a two-minute interval) or work-based (e.g., measure the number of page faults that occur during the time needed to complete 1000 transactions). In either case, it is helpful to normalize these measurements so that the values of the variables in the model do not depend on the exact interval used for the measurements. I have used two types of normalization in my work with Berkeley DB: expressing a statistic as a percentage, and creating a per-transaction average by dividing a statistic by the number of transactions completed during the measurement interval. Examples of the former normalization method include *pct_updates* and *pct_wlocks*. Examples of the latter include *faults/txn* and *waits/txn*. See Section 4.4 for more detail about these nodes.

Using normalized variables takes on an added importance when the measurement interval is time-based rather than work-based. In such cases, normalization eliminates non-essential dependencies between the variables that can arise as a result of using a time-based interval. For example, the number of page faults that occur in a Berkeley DB application over a two-minute interval depends on the number of transactions that are completed during that interval, which in turn depends on the number of lock waits that occur during the interval. Normalizing the number of page faults (e.g., by dividing by the number of completed transactions) allows us to avoid including this dependency in the model.

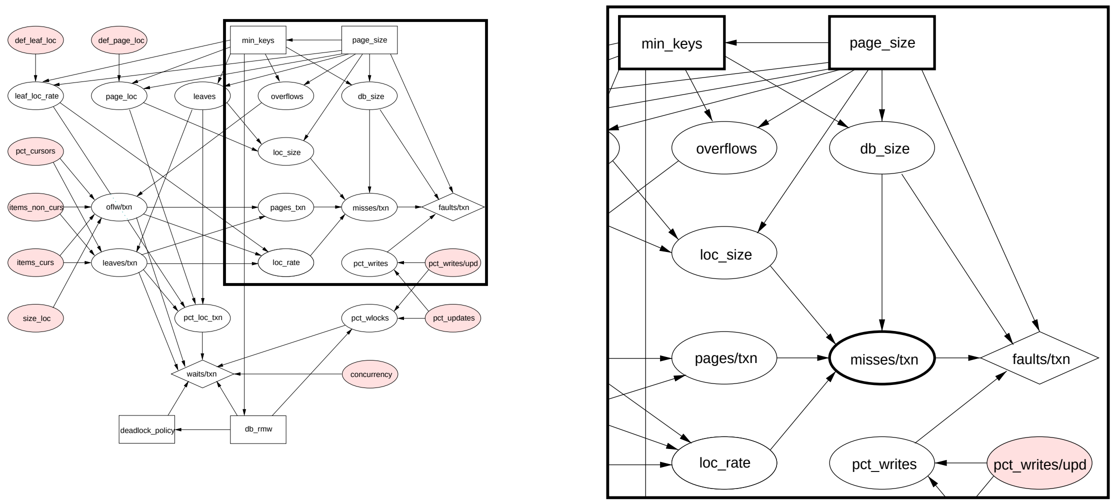


Figure 5.1. The *misses/txn* node and surrounding nodes. The figure on the right provides a closeup of a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. The *misses/txn* variable does not need the *min_keys* and *page_size* knobs as parents. A node representing the miss rate, on the other hand, would need to inherit from those knobs for the reasons discussed in Section 5.1.1.6.

5.1.1.6 Avoiding Unnecessary Dependencies

In general, care should be taken to avoid defining variables in a way that introduces unnecessary dependencies. The use of normalization described in the previous section is one example of this general rule. For example, to measure the degree to which Berkeley DB is able to find database pages in its memory pool (Section 4.1.2), one variable that could be used is *miss rate*, the percentage of page accesses that incur a miss in the memory pool. However, the miss rate depends on the height of the Btree, because taller trees lead to more internal-page accesses and thus have more page accesses overall. More specifically, if tree T_1 and tree T_2 each produce the same number of misses but T_1 is taller than T_2, then T_1's miss rate will be lower than T_2's miss rate because of the larger total number of page accesses that T_1 requires. And because the settings of the *page_size* and *min_keys* knobs affect the height of the Btree, a *miss rate* chance node would need to have the corresponding decision nodes as parents, whereas the *misses/txn* variable that I have used in the influence diagram for Berkeley DB does not need those nodes as parents (Figure 5.1).

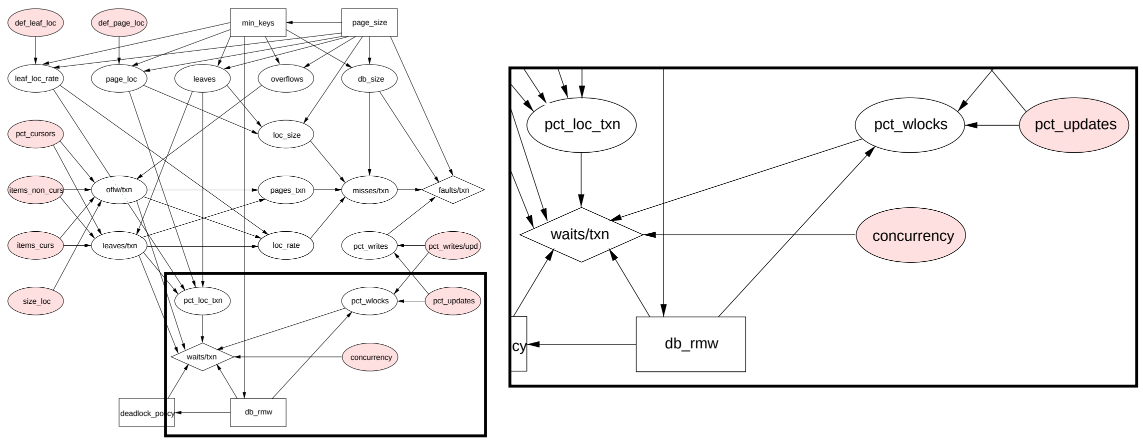


Figure 5.2. The *waits/txn* value node and some of the chance and decision nodes that affect it. The figure on the right focuses on a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. The nodes shown include variables like *concurrency* that directly affect *waits/txn*, as shown by the conditional arcs from those nodes to *waits/txn*, as well as variables like *pct_updates* that affect *waits/txn* only indirectly and for which no conditional arc to *waits/txn* has been drawn.

5.1.2 Adding Arcs to the Model

As discussed in Section 3.2.1, there are two types of arcs in an influence diagram—conditional arcs and informational arcs. The conditional arcs—arcs into chance or value nodes that indicate probabilistic relevance—can typically be added on the basis of intuitive notions of causality. For example, the lock waits in a Berkeley DB application are “caused” by multiple threads attempting to acquire locks for the same database pages at the same time. As the number of threads accessing the database increases, it becomes increasingly likely that lock waits will occur. Therefore, it makes sense to include an arc from *concurrency*—which measures the number of threads accessing the database—to *waits/txn*, as shown in Figure 5.2.

If the influence of one node on another is purely indirect, a conditional arc between the two nodes is not needed. For example, in Figure 5.2, the percentage of update transactions (*pct_updates*) also affects the likelihood of lock waits, but its influence stems from the fact that increasing *pct_updates* can increase the percentage of database accesses that involve write locks (*pct_wlocks*). Therefore, the fact that *pct_wlocks* is already a parent of *waits/txn* means that no conditional arc is needed from *pct_updates* to *waits/txn*. On the other hand, the setting chosen for *db_rmw* affects *waits/txn* both indirectly—through its impact on *pct_wlocks*—and directly—through the ways in which it and the setting chosen for *deadlock_policy* influence both the likelihood that deadlocks will occur and the types of transactions that are chosen to resolve them—and thus the influence diagram includes two directed paths from *db_rmw* to *waits/txn*.

As discussed in Section 3.2.1, informational arcs—arcs into decision nodes that indicate the information known when a decision is made—must be used to impose an ordering on the decisions. For example, in the influence diagram for Berkeley DB (Figure 4.2), the decision nodes are ordered as follows: *page_size*, *min_keys*, *db_rmw*, *deadlock_policy*. However, because we assume that the tuner knows the values of the observable workload characteristics before tuning the knobs, *any* ordering of the decision nodes will produce the same optimal policies. This is a special case of the rule for reordering decisions discussed in Section 3.2.1: because the set of known chance-node values is the same for each of the decisions, the decisions can be arbitrarily reordered.

Although the ordering of the decision nodes will not affect the recommended knob settings, it *can* affect the efficiency of the model. In particular, if the optimal policy for a given decision node depends on only a subset of the observed chance nodes or the other decision nodes, it can be more efficient to order the decisions in a way that takes advantage of this fact. Both Shachter [Sha98] and Nielsen and Jensen [Nie99] present methods for determining the nodes that are relevant to a given decision.

To indicate the observations that are made before the knobs are tuned, informational arcs can also be drawn from the observable workload characteristics to the decision node that comes first in the imposed ordering. However, provided that the nodes in question will always be observed and entered as evidence before the influence diagram is evaluated (i.e., provided that we are only evaluating the model for a particular set of observations, rather than determining the full optimal policies for the knobs), these arcs are not strictly necessary.

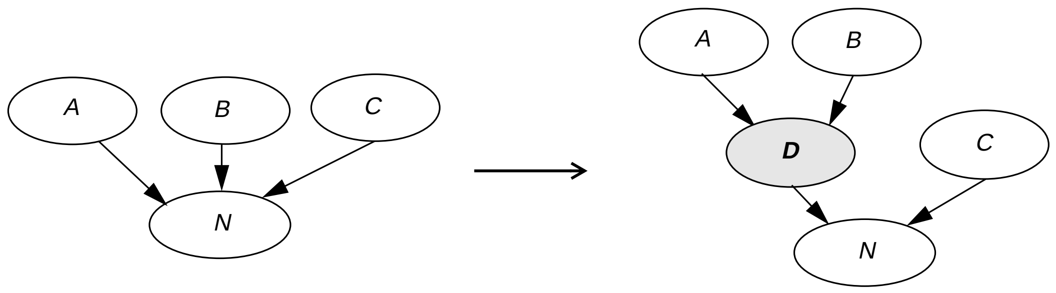


Figure 5.3. Adding an intermediate chance node. This figure shows a fragment of an influence diagram before (*left*) and after (*right*) the addition of an intermediate node, *D*, between node *N* and two of its original parents, *A* and *B*.

As mentioned in Section 3.2.1, the algorithm for evaluating an influence diagram (Section 3.2.4) makes the reasonable assumption that all information known when a decision is made is available for all subsequent decisions. Thus, it is never necessary to include more than one informational arc that begins at a given chance or decision node, *N*; rather, it suffices to include a single informational arc from *N* to the first decision made after *N*'s value is known.

5.1.3 Limiting the Number of Parents of a Node

As discussed in Section 5.1.1.2, the number of parameters associated with a value node or intermediate chance node is exponential in the number of its parents. More precisely, the number of parameters associated with a node is linear in the number of instantiations of its parents, which is in turn exponential in the number of its parents. As the number of parameters associated with a node increases, so does the amount of data needed to train the model and the cost of evaluating the model. Therefore, it is important to limit the number of parent instantiations associated with each node by limiting the number of its parents.

One technique for reducing the number of parents of a node, *N*, is to introduce an intermediate chance node between *N* and two or more of its parents, as shown in Figure 5.2. This technique assumes that the node in question is conditionally independent of its original parents given the intermediate node. In the influence diagram for Berkeley DB, for example, the *pct_writes* node captures the impact of both the *pct_writes/upd* and *pct_updates* nodes on our beliefs about the *faults/txn*

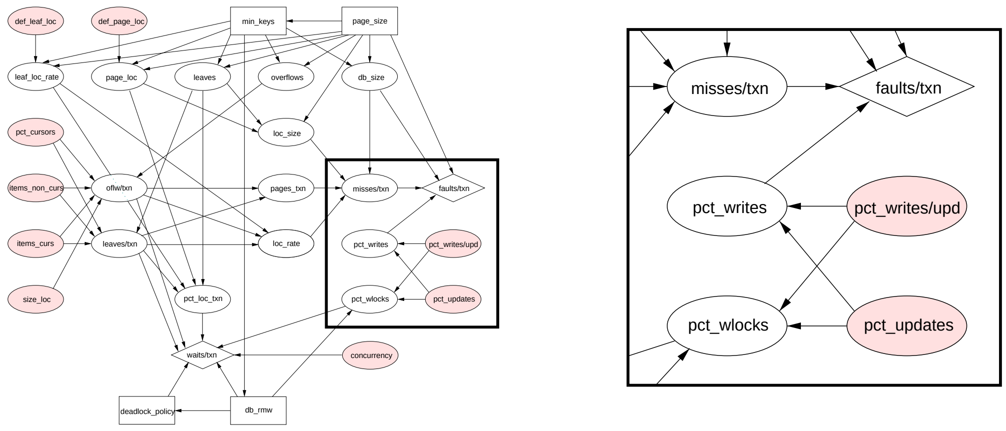


Figure 5.4. Using an intermediate node to limit the number of a node's parents. The figure on the right provides a closeup of a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box in the figure on the left. The intermediate chance node *pct_writes* captures the influence of *pct_writes_upd* and *pct_updates* on *faults/txn*, and thus takes the place of the latter nodes as parents of *faults/txn*, reducing the total number of parents of this node.

node (Figure 5.4) and thereby reduces the number of parents of this node. And provided that there are fewer possible values for *pct_writes* than there are instantiations of *pct_writes/upd* and *pct_updates*, the number of parameters associated with *faults/txn* will also be reduced. Moreover, introducing an intermediate chance node in this way typically reduces the total number of parameters in the model.

However, adding an intermediate chance node can also reduce both the accuracy and efficiency of the model. One of the main reasons for a potential reduction in accuracy stems from the assumption of conditional independence that is made when an intermediate node is introduced. This assumption is often only approximately correct, especially in domains like software tuning that require the discretization of continuous variables. For example, although the undiscretized value of *pct_writes* may provide as much information about the value of *faults/txn* as the undiscretized values of *pct_writes/upd* and *pct_updates*, the discretized value of *pct_writes* may not provide as much information as the discretized values of *pct_writes/upd* and *pct_updates*. As a result, adding intermediate nodes can reduce the accuracy of the model's predictions about the impact of the knobs and workload characteristics on the performance of the system. In addition, adding an intermediate node may increase the cost of performing inference in some cases. Therefore, before adding an intermediate chance node to an influence diagram, it is necessary to weigh the potential costs and benefits of doing so. The method of model refinement discussed in the next section provides one method of assessing the impact of adding a given intermediate node.

5.1.4 Checking and Refining the Model

It is important to verify that an influence diagram is *regular* as defined by Shachter [Sha86]. This involves confirming that the graph has no directed cycles, that the value nodes have no children, and that there is a directed path that passes through all of the decision nodes. In addition, the accuracy of the conditional independence statements that the model encodes should also be checked. In particular, for each value node and intermediate chance node, the designer of the model should ensure that it is reasonable to assert that the node is conditionally independent of its nondescendants given the values of its parents. If such an assertion is not valid, the structure of the model should be reconsidered. Provided that the conditional arcs are added on the basis of intuitive notions of causality as described in Section 5.1.2, the conditional independence assertions of the model should typically pass this test, but there may still be problematic cases.

Examples of invalid conditional-independence assertions can be found in an earlier version of the influence diagram for Berkeley DB; the relevant portion of the model is shown in Figure 5.5. This model differs from the final model in its inclusion of an extra intermediate chance node, *prob(wait)*, that represents the probability that a given lock request will cause a thread to wait. The value of this variable is simply the ratio of the number of lock waits to the number of lock requests. Although this node reduces the number of parents of *waits/txn*, it also leads to at least two invalid assertions of conditional independence. Because taller Btrees require more internal-page accesses, the number of lock requests that are produced by a given workload is larger in taller Btrees. Therefore, settings of *page_size* and *min_keys* that produce

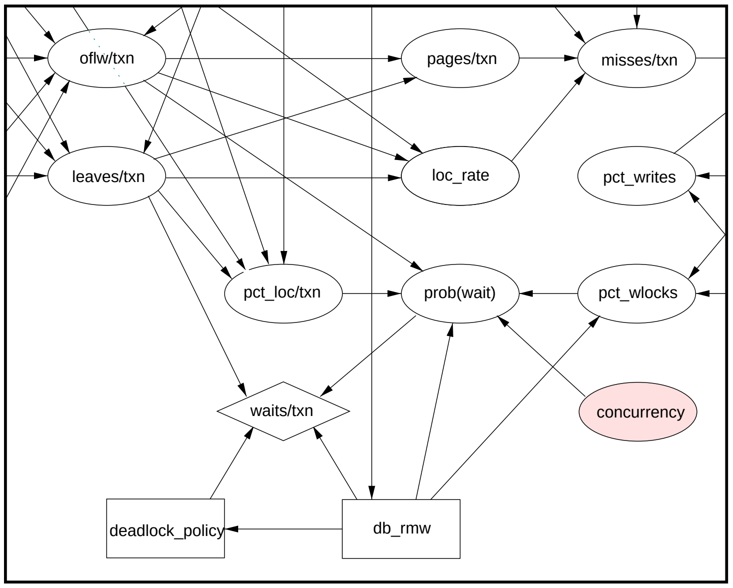


Figure 5.5. Invalid conditional-independence assertions. The figure above highlights a portion of an earlier version of the influence diagram for Berkeley DB. The model includes an intermediate chance node, *prob(wait)*, that was added to reduce the number of parents of *waits/txn*. However, adding this node also leads the model to make invalid assertions of conditional independence, as described in Section 5.1.4.

taller Btrees will tend to produce smaller values of *prob(wait)*, which means that *prob(wait)* is not conditionally independent of *page_size* and *min_keys* given the values of its parents, as the model suggests. Moreover, the impact of a given value of *prob(wait)* on the value of *waits/txn* also depends on the height of the Btree, and thus *waits/txn* is not conditionally independent of *page_size* and *min_keys*. Therefore, I revised the model to remove the *prob(wait)* node.

In addition to performing qualitative checks of the model's soundness, the designer of the model should attempt to perform quantitative checks as well. Measurements of the variables in the model for a range of different workloads and knob settings can be used to assess the quality of the model's recommendations—and to guide possible refinements—before the model is actually deployed as part of an automated tuner. If the designer is uncertain about one or more design choices (e.g., whether to add a particular intermediate chance node to the model), data can be used to compare the performance of different candidate models. Measurements of the variables in the model can also allow the designer to quantitatively verify the assertions of conditional independence encoded in the model's structure.

The data needed to evaluate one or more models should consist of two sets of measurements that are gathered using the process described in Section 5.2. The first set is used to train the model or models being assessed, as described in Sections 5.3 and 5.4.[1] The second set—the *validation set*—is used to evaluate the performance of the models on a variety of workloads. When gathering this validation data, a large number of different knob settings (all of them, if possible) should be tried for each workload; this provides empirical evidence of which knob settings are optimal for each workload in the set. By comparing the system's maximal measured performance for a given workload with the performance it achieves using the knob settings recommended by the models, the quality of the models' recommendations can be assessed.

5.2 Gathering the Training Data

The parameters of the influence diagram used by the tuner are learned from a collection of training data. The process of gathering the training data can vary depending on the system being tuned. In all cases, this process involves measuring the performance of the system for various combinations of knob settings and workload characteristics. More specifically, it involves measuring the values of the intermediate chance nodes and value nodes that result from different combinations of values for the decision nodes and root chance nodes. Each tuple of measurements of the variables in the model is referred to as a *training example* or *training instance*. The performance metric that the tuner will attempt to optimize (which may or may not be explicitly represented in the model) should also be measured as part of each training example.

As discussed in Section 2.4, it may be impractical to gather training data on a deployed software system as it runs, because a randomly selected knob combination could lead to a significant performance loss. Therefore, it may be preferable to gather

1. It is worth noting that the discretizations and parameters learned during the validation process would not be retained in most cases. The actual training data for a given installation of the tuner must ordinarily be gathered on a per-platform basis, as described in Section 5.2.

training examples offline using simulated workloads run on a separate version of the system, perhaps using a workload generator. This is the approach that I have taken in my work with Berkeley DB; Chapter 6 discusses the design of workload generators for software tuning. Developing a workload generator is a non-trivial task, but the work involved—like the work involved in designing an influence diagram for tuning—need only be performed once, by the developers of the software system. No effort by end-users is required.

When using a workload generator, the training workloads are drawn from distributions over the workload characteristics. These distributions can be estimated or anticipated by someone familiar with the system, or they can simply be uniform distributions over the ranges of possible values. Each of the selected workloads is run under one or more of the possible knob configurations. Once the system reaches a steady state for a given run, the values of the variables in the influence diagram are measured over some interval and together constitute a training example. Neither the estimates of the workload distributions from which the simulated workloads are drawn nor the simulated workloads themselves need to precisely match the actual workloads run on the system. The training examples can be seen as providing reasonable starting estimates of the model's parameters, and the parameters can be refined over time as the system uses the influence diagram for tuning. Section 5.6 explains how this process of parameter refinement is performed.

It is worth noting that the training examples do *not* need to include instances of all possible combinations of workload characteristics and knob settings. The conditional independence relationships encoded in the influence diagram make it unnecessary to see all combinations, and they give the model its ability to generalize. Provided that the training instances include examples of most of the possible instantiations of the parents of each intermediate chance node and value node in the model, the model should be able to recommend optimal or near optimal knob settings for combinations of workload characteristics that are not present in the training data—as well as combinations of workload characteristics for which only some of the knob settings have been tried.

Because the parameters of the model may depend upon the hardware platform on which the software system runs, the process of gathering the training data will typically need to be performed on a platform-specific basis. One possible approach would be for the software developer to gather training data for a wide range of hardware platforms and precompute the initial parameters of the model for each of these platforms. Alternately, the workload generator and a script for driving it could be included with the software system and run by the end-user on the appropriate platform. Here again, no expertise would be needed by the end-user. The workload generator—like the tuner itself—would run “out of the box.”

Other variations on the process of gathering the training data are also possible. For example, the manufacturer of an information appliance that runs an embedded software system could obtain the tuner and workload generator from the software developer, configure the workload generator to produce workloads that the appliance is likely to face, and run the workload generator on the appliance to gather the training data needed by the tuner. In all cases, the work involved is amortized over the number of installations of the software system, each of which is able to tune itself without human intervention.

5.3 Discretizing the Continuous Variables

Researchers have developed methods for incorporating continuous variables in influence diagrams in ways that involve explicitly representing and reasoning with continuous probability distributions [e.g., Sha89]. However, there are at least two problems associated with these methods. First, they are limited to certain families of distributions—typically Gaussians—which means using them makes potentially incorrect assumptions about the nature of the data [Fri98]. Second, existing tools for working with influence diagrams are limited in their ability to handle continuous distributions.

An alternative method for dealing with continuous variables—or discrete variables with enough possible values that they are effectively continuous—is to convert each continuous variable into a discrete variable. This process of *discretization* involves specifying a sequence of *cutoff values* that divide the range of possible values for the variable into subranges referred to as *bins*. Values of the continuous variable that fall into the same bin are given the same value in the discretized version of the variable. For example, a cutoff-value sequence of 2, 4, and 10 creates four bins: one for values in $(-\infty, 2)$, one for values in $[2, 4)$, one for values in $[4, 10)$, and one for values in $[10, \infty)$.

More formally, a sequence of cutoff values, $s = (c_1, c_2, ..., c_k)$, where $c_1 < c_2 < ... < c_k$, defines a function f_s that maps real numbers to elements of the set $\{0, ..., k\}$ as follows:

$$f_s(x) = \begin{cases} 0 \text{ if } x < c_1 \\ i \text{ if } c_i \le x < c_{i+1} \text{ and } 1 \le i < k \\ k \text{ if } x \ge c_k \end{cases} \quad \textbf{(EQ 18)}$$

When using a sequence of cutoff values to discretize a variable, each value x of the undiscretized variable is given the value $f_s(x)$ in the discretized version of the variable.

Determining an effective discretization for a variable is a challenging task. The need to discretize multiple variables simultaneously in a way that captures the interactions between them makes the problem that much more difficult. The paragraphs that follow begin by presenting two simple approaches to discretization and discussing their limitations. The section continues with an introduction to a method developed by Friedman and Goldszmidt [Fri96] for learning the discretization of continuous variables in a related probabilistic model known as a Bayesian network, and it concludes with a description of how I have adapted their approach to learn the discretization of continuous variables in an influence diagram.

5.3.1 Simple Discretization Methods

5.3.1.1 Equal-width Bins

One simple approach to discretization is *equal-width bins*, in which the distance between any two adjacent cutoff values is the same. More specifically, if s is the smallest value observed in the training data (or the smallest possible value for the variable, if that value is known *a priori*), l is the largest observed (or largest possible) value, and n is the desired number of bins, the cutoffs are spaced w units apart, where w is computed as follows:

$$w = \frac{l - s}{n} \qquad \textbf{(EQ 19)}$$

The resulting cutoff-value sequence is $s + w$, $s + 2 \cdot w$, ... , $s + (n - 1) \cdot w$.

A significant limitation of an equal-width approach is that outliers in the training data can skew the range of values of the variable and produce bins that contain either too many or too few training instances. When a bin contains too many training instances, important distinctions between values can be lost. When a bin contains too few training instances, it becomes likely that the children of the node in question will have parameters for which no training data is available.

One example of the skewing effect of outliers is found in the training data from my experiments with Berkeley DB, which includes the distribution of values for *misses/txn* shown in Figure 5.6. Using equal-width bins with $s = 0$, $l = 25$, and $n = 5$ produces cutoffs spaced $w = 5$ units apart, as shown by the vertical lines on the graph. This discretization puts over 68% of the values in the first bin, which means that all distinctions between two-thirds of the training data for this variable would be lost. In addition, there are only 20 training instances in the fifth bin.

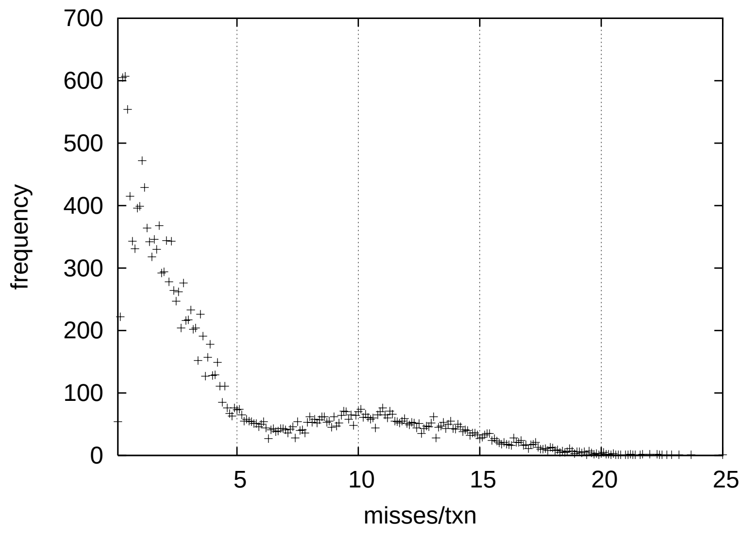


Figure 5.6. Example of an equal-width discretization. This graph shows the distribution of values of the *misses/txn* variable in the training data for the influence diagram for Berkeley DB. Using five equal-width bins produces the cutoff values shown by the vertical lines. The counts of the number of training instances in the individual bins are, from left to right: 12797, 2581, 2585, 734, and 20. In determining the frequencies, the *misses/txn* values were rounded to one decimal place.

5.3.1.2 Equal-height Bins

Another common discretization approach is *equal-height bins*, in which the cutoffs are chosen so that each bin ends up with approximately the same number of training instances. The number of instances in a given bin is referred to as the *height* of the bin. One reason that the heights of the bins may be only approximately equal is that a given value may appear multiple times, and all instances of a given value are placed in the same bin. Figure 5.7 presents the pseudocode for the algorithm that I have used to determine equal-height cutoffs. It adjusts the target value for the bin height as needed to handle situations in which repeated values cause one or more of the bins to receive extra training instances. For example, if we were dividing 100 values into five equal-height bins and 40 of the values were identical, the identical values would be placed in one bin and the remaining 60 values would be divided equally among the remaining four bins. If some of the bins end up with too few values, the user can reduce the desired number of bins and rerun the algorithm.

An equal-height approach tends to avoid creating bins that contain too many or too few training instances. Figure 5.8 shows the same distribution of values for

```
equal_height_discretize(V, nbins) {
      nvals = 0;
      count = a hashtable
      for each unique value v in V {
            count{v} = the number of times that v appears in V;
            nvals = nvals + count{v};
      }
      orig_target = nvals/nbins;            // original target height

      // adjust for values that get their own bin
      for each unique value v in V {
            if (count{v} > orig_target) {
                  nvals = nvals - count[v];
                  nbins = nbins - 1;
            }
      }
      target = nvals/nbins;

      // consider the values in ascending order
      total = total_in_bins = ncutoffs = 0;
      A = an array containing the values in V (without repeats),
          sorted in ascending order;
      for (i = 0; i < size(A); i++) {
            total = total + count{A[i]};
            height = total - total_in_bins;
            new_cutoff = false;
            if (count{A[i]} > orig_target || i = size(A) - 1) {
                  cutoffs[ncutoffs] = A[i];  // A[i] gets its own bin
                  new_cutoff = true;
            } else {
                  next_height = height + count{A[i+1]};
                  if (|height - target| < |next_height - target|) {
                        cutoffs[ncutoffs] = (A[i] + A[i+1])/2;
                        new_cutoff = true;

                        // adjust the target height
                        nvals = nvals - height;
                        nbins = nbins - 1;
                        if (nbins == 1) {
                              return cutoffs;
                        }
                        target = nvals/nbins;
                  }
            }
            if (new_cutoff) {
                  total_in_bins = total;
                  ncutoffs = ncutoffs + 1;
            }
      }
}
```

Figure 5.7. Pseudocode for an algorithm to perform equal-height discretization. The algorithm takes a collection V of real numbers and a desired number of bins, `nbins`, and determines cutoffs that partition V into `nbins` equal-height bins, where `nbins` ≥ 2. It adjusts for cases in which one or more bins receive extra values because of repeated values.

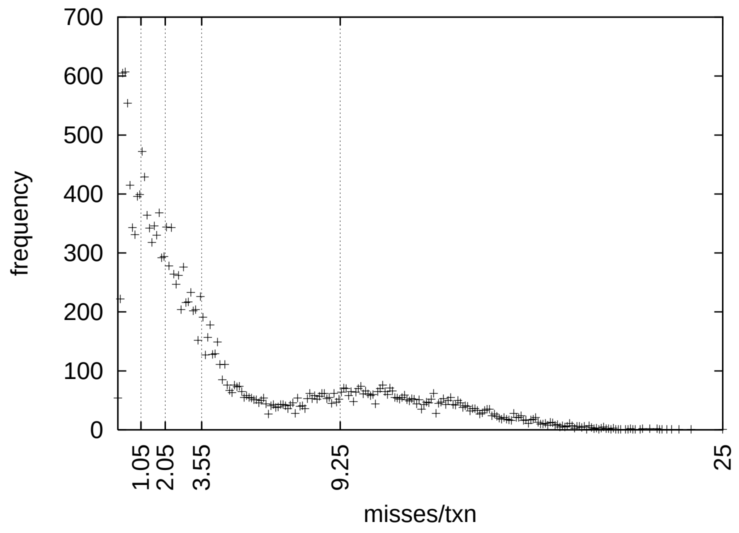


Figure 5.8. Example of an equal-height discretization. This graph shows the distribution of values for the *misses/txn* variable in the training data for the influence diagram for Berkeley DB. Using five equal-height bins produces the cutoff values shown by the vertical lines. The counts of the number of training instances in the individual bins are 3926, 3555, 3668, 3789, and 3779. The counts are not exactly equal because all instances of a given value are placed in the same bin. In determining the frequencies, the *misses/txn* values were rounded to one decimal place.

misses/txn as the one shown in Figure 5.6, but partitioned into five equal-height bins. The training instances are distributed in a fairly uniform fashion among the five bins.

In addition, equal-height discretizations tend to draw distinctions between the values—or the narrow subranges of values—that occur most frequently in the training data, because the equal-height criterion makes it less likely that frequently occurring values will be grouped together. And because frequently occurring values tend to be associated with the most common workloads, the ability to distinguish these values may allow the influence diagram to make more accurate tuning recommendations for those workloads.

5.3.1.3 Limitations

Both equal-width and equal-height discretizations suffer from two significant limitations. First, both of these methods employ a set of parameters—the desired numbers of bins for the variables—and it can be difficult to predetermine appropriate values for these parameters. Second, and more important, both methods discretize individual variables in isolation, without regard to how the resulting discretizations affect the influence diagram as a whole. It is thus likely that the discretizations will fail to capture distinctions that are relevant to the interactions between neighboring variables in the model. For example, in the influence diagram for Berkeley DB, it may be the case that lock waits become much more likely when the value of *concurrency* exceeds some threshold. Neither an equal-width nor an equal-height approach would have any way of detecting this, and thus the resulting discretization may end up grouping values from above and below the threshold into the same bin.

If a human being determines the discretizations, it may be possible to employ an approach in which either an equal-width or equal-height approach provides an initial set of discretizations that are then tweaked by hand—perhaps on the basis of a process that tests the accuracy of the model (Section 5.1.4). However, determining the necessary modifications to a set of discretizations is non-trivial, and this process would need to be performed separately for each installation of the software system. As a result, hand-tweaking the discretizations violates the goal of creating a fully automated tuner. Without an automated means of discretizing the continuous variables in the model, we end up replacing one tuning process (the tuning of the knobs) with another (the tuning of the discretizations). Clearly, what is needed is an automated method of *learning* the appropriate discretizations from the training data. The remainder of this section presents one such method.

5.3.2 Learning Discretizations of Variables in a Bayesian Network

A Bayesian network [Pea88] is a probabilistic graphical model that is closely related to an influence diagram. In a Bayesian network, there are no decision or value nodes and there are only conditional arcs, but the syntax and semantics are otherwise the same as the syntax and semantics of an influence diagram. Bayesian networks can be used to perform various types of probabilistic inference—such as determining the most probable explanation for a set of observations—and they have been applied to problems in a variety of domains.

Friedman and Goldszmidt [Fri96] have developed a method based on the *Minimal Description Length (MDL)* principle for learning the discretizations of continuous variables in a Bayesian network. This section begins by explaining the MDL principle and how it can be used to define a metric that balances the complexity of a graphical model with the degree to which it captures interactions among related variables in the network, and it continues with a presentation of Friedman and Goldszmidt's algorithm for using this metric to learn the discretizations of variables in a Bayesian network. Section 5.3.3 presents an extended and modified version of this algorithm that can be used to discretize continuous variables in an influence diagram.

5.3.2.1 The MDL Principle

The MDL principle is used to distinguish between different methods for encoding a collection of data. Encoding data involves replacing each unit of data with a corresponding code; the goal is to compress the data, producing a representation that is more compact than the original one. A Bayesian network or influence diagram can be seen as a mechanism for encoding the data used to train the model. More specifically, the joint probability distribution stored in the model can be used to guide Huffman encoding [Cor90], which assigns codes to units of data on the basis of the frequencies with which they appear. Because we want to be able to recover the original data from the encoded version, we need to store a representation of the model along with the encoded data. If the continuous variables are being discretized, we also need to store enough information to recover the undiscretized values from the discretized ones. The *description length* of the encoding is thus the sum of the number of bits used to represent the model (including the discretization) and the number of bits used to represent the encoded, discretized data. For a given collection of training data, different models produce different description lengths. The MDL principle states that the optimal model is the one that minimizes the description length.

Although the MDL principle makes sense when the goal is to compress a collection of data, its relevance to the process of learning discretizations in a

Bayesian network or influence diagram is less obvious. However, as shown in the next section, an argument can be made that the discretizations that minimize the description length of the encoded training data are also the discretizations that best capture the interactions between the variables in the model.

5.3.2.2 Computing the Description Length

The argument for using the MDL principle to learn the discretizations of the continuous variables in the model becomes more compelling when we examine the formula for the description length of a Bayesian network, which Friedman and Goldszmidt derive in their work [Fri96]. The total description length is the sum of the description lengths of the four elements of the encoding: the model (including the parameters); the sequences of cutoff values that are used to discretize the continuous variables; the discretized training data; and the information needed to recover the undiscretized data from the discretized data. The paragraphs below consider each of these elements in turn.

In the discussion that follows, $\mathbf{U} = \{ X_1, ..., X_n \}$ will refer to the set of variables in the model, and $\mathbf{U}_c$ will denote the subset of them that are continuous. $\mathbf{U}^* = \{ X_1^*, ..., X_n^* \}$ will represent a set containing both the discrete variables in $\mathbf{U}$ and the discretized versions of the continuous variables in $\mathbf{U}$. More specifically, if X_i is discrete, $X_i^* = X_i$; if X_i is continuous, X_i^* is the discretized version of X_i. Π_i will refer to the parents of X_i in the model, and Π_i^* to the discretized versions of these parents. N will represent the number of training instances. $|\mathbf{A}|$ will refer to the number of values in the set $\mathbf{A}$; $\Omega(\boldsymbol{X})$ will refer to the set of all possible instantiations of the variables in $\boldsymbol{X}$; and $\|\boldsymbol{X}\|$ will refer to the number of values in $\Omega(\boldsymbol{X})$. Finally, the notation established in Chapter 3 for probabilities will also be used below.

The component of the description length that deals with the encoding of the model itself can be computed as follows:

$$DL_M = \sum_i \log_2 \|X_i^*\| + \frac{\log_2 N}{2} \sum_i \|\Pi_i^*\| (\|X_i^*\| - 1) \qquad \text{(EQ 20)}$$

The first element of this expression computes the number of bits needed to encode the number of values of each of the variables in $\mathbf{U}^*$. The second element computes the number of bits needed to encode the parameters of the model; each variable X_i needs $\|X_i^*\| - 1$ parameters to specify each of its associated probability distributions, of which there are $\|\Pi_i^*\|$—one for each instantiation of its parents. Additional bits would also be needed to encode the number of variables in the model and a description of the parents of each variable, but these values can be omitted from consideration because they do not depend on the discretizations.

The second element of the total description length is the number of bits needed to encode the cutoff-value sequences used to discretize the continuous variables. Friedman and Goldszmidt show that this value can be computed as follows:

$$DL_C = \sum_{X_i \in \mathbf{U}_c} (N_i - 1) H\left(\frac{k_i - 1}{N_i - 1}\right) \qquad \text{(EQ 21)}$$

where N_i is the number of unique, undiscretized values of X_i in the training set, k_i is the number of bins in the discretized version of X_i, and H is the function defined as follows: $H(p) = -p\log_2 p - (1-p)\log_2(1-p)$. The intuition here is that we need to store a cutoff-value sequence for each variable X_i. The possible cutoff values for X_i are the N_i - 1 midpoints between successive values in its set of N_i unique values, and there are $\binom{N_i - 1}{k_i - 1}$ ways of selecting a sequence of k_i - 1 of these values. The expression to the right of the summation in equation 21 provides an upper bound on the number of bits needed to describe which of these sequences has been chosen.

The third and fourth elements of the total description length are DL_D, the number of bits needed to encode the discretized training data, and DL_R, the number of bits needed to encode the values used to recover the undiscretized data from the discretized data. Friedman and Goldszmidt show that the relevant elements of the sum of these two description lengths can be expressed compactly using an information-theoretic metric known as *mutual information* [Cov91]. The mutual information between two sets of random variables, $\boldsymbol{X}$ and $\boldsymbol{Y}$, is defined as follows:2

$$[\boldsymbol{X};\boldsymbol{Y}] = -\sum_{\substack{\boldsymbol{x} \in \Omega(\boldsymbol{X}), \\ \boldsymbol{y} \in \Omega(\boldsymbol{Y})}} P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y}) \log_2 \frac{P(\boldsymbol{X} = \boldsymbol{x}, \boldsymbol{Y} = \boldsymbol{y})}{P(\boldsymbol{X} = \boldsymbol{x}) P(\boldsymbol{Y} = \boldsymbol{y}} \qquad \textbf{(EQ 22)}$$

Mutual information is a measure of the degree to which knowing the values of one set of variables decreases our uncertainty about ("gives us information about") the values of a second set of variables. If $\boldsymbol{X}$ and $\boldsymbol{Y}$ are independent—i.e., if knowing the values of the variables in $\boldsymbol{Y}$ gives us no information about the value of $\boldsymbol{X}$, and vice versa—then $I[\boldsymbol{X};\boldsymbol{Y}] = 0$. The larger the mutual information is between a node and its neighbors in a Bayesian network or influence diagram, the more likely it is that the model accurately captures the interactions between those variables.

In their derivation of the description length of a Bayesian network, Friedman and Goldszmidt use basic properties of the mutual information metric and other related measures to show the following:

$$DL_D + DL_R = -N\sum_i I[X_i^*;\Pi_i^*] \qquad \textbf{(EQ 23)}$$

where terms that are not affected by the process of learning the model have been omitted, and the probabilities used to perform the mutual information computations are based on the frequencies with which values appear in the training data (Section 5.4.1.1). The total description length can thus be computed as follows:

$$\begin{aligned} DL = {} & \sum_i \log_2 \|X_i^*\| + \frac{\log_2 N}{2} \sum_i \|\Pi_i^*\|(\|X_i^*\| - 1) \\ & + \sum_{X_i \in \mathbf{U}_c} (N_i - 1) H\left(\frac{k_i - 1}{N_i - 1}\right) - N\sum_i I[X_i^*;\Pi_i^* \end{aligned} \qquad \textbf{(EQ 24)}$$

When the description length is expressed in the form above, it becomes evident that this metric balances the complexity of the model—which is represented by the first three terms of the sum—with the mutual information between each node and its parents—which is represented by the final term. This characteristic of the description length is the reason why the MDL principle is a useful guide to the process of learning the optimal discretizations. Because increasing $\sum_i I[X_i^*;\Pi_i^*]$

2. If either $\boldsymbol{X}$ or $\boldsymbol{Y}$ is the empty set, $I[\boldsymbol{X};\boldsymbol{Y}]$ is defined to be zero.

decreases the description length, the MDL principle leads us to favor cutoff values that increase the mutual information between a node and its parents and children. However, adding additional cutoff values also increases the description length (through the first three terms of the sum), and thus the MDL principle leads us to avoid discretizations that are overly complex. This balancing of mutual information with complexity guards against overfitting the model to the training data, and it should also help to limit the number of parameters for which no training data is available—a problem that becomes more likely as the complexity of the model increases.

5.3.2.3 Friedman and Goldszmidt's Algorithm

To learn the discretizations of variables in a Bayesian network, Friedman and Goldszmidt employ a greedy, iterative approach in which they repeatedly attempt to improve the discretization of one of the continuous variables while holding fixed the discretizations of the other variables. When attempting to improve a variable's discretization, their algorithm begins by removing all of the variable's cutoff values and resetting its discretization to a single bin [Fri03]. It then considers all possible cutoff values, adding the one that leads to the largest decrease in description length. It continues adding cutoff values in this way until there are no remaining cutoff values that can reduce the description length. The algorithm then compares the description length produced by this new discretization to the description length produced by the previous discretization for that variable. If the new description length is smaller, it adopts the new discretization.

When using the MDL principle to improve the discretization of a single variable, X_i, Friedman and Goldszmidt note that we can restrict our attention to the elements of the description length that are affected by X_i's discretization. More specifically, we can minimize the *local description length* for X_i, which is defined as follows:

$$L(X_i) = \log_2\|X_i^*\| + \frac{\log_2 N}{2}\left(\|\Pi_i^*\|(\|X_i^*\|-1) + \sum_{j,\,X_i\in\Pi_j}\|\Pi_j^*\|(\|X_j^*\|-1)\right)$$
$$+ (N_i-1)H\left(\frac{k_i-1}{N_i-1}\right) - N\left(I[X_i^*;\Pi_i^*] + \sum_{j,\,X_i\in\Pi_j} I[X_j^*;\Pi_j^*]\right)$$

(EQ 25)

Here again, the probabilities used to perform the mutual information computations are derived from the frequencies with which values appear in the training data. Note that the above expression, like the one for the total description length (equation 24), has four components, the first three of which measure model complexity and the last of which measures mutual information between neighboring variables. However, in computing the local description length for X_i, we only consider quantities related to X_i and the variables in what is known as X_i's *Markov blanket*—its parents, its children, and the parents of its children.

As discussed above, after the algorithm resets a variable's discretization to a single bin, it only considers changes to the discretization that involve adding a new cutoff value to the existing cutoff values for that variable. And because *any* new cutoff value has the same effect on the complexity of the model (the first three components of $DL(X_i)$), the algorithm can determine the optimal candidate for a new cutoff value by finding the value that leads to the largest increase in mutual information (the fourth component of $DL(X_i)$). More precisely, it chooses the cutoff value, c, that maximizes the following expression:

$$ain(X_i, c) = \left(I[X_i^*[c];\Pi_i^*] + \sum_{j,\,X_i\in\Pi_j} I[X_j^*;\Pi_j^*[c]]\right) - \left(I[X_i^*;\Pi_i^*] + \sum_{j,\,X_i\in\Pi_j} I[X_j^*;\Pi_j^*\right.$$

(EQ 26)

where the notation [c] indicates that the cutoff values used to discretize X_i include the value c. Then, once the optimal candidate value is found, the algorithm determines whether it should be added to the discretization by seeing if it reduces $DL(X_i)$, which is equivalent to determining whether the increase in mutual information from adding the cutoff value is greater than the corresponding increase in the model's complexity.

```
learn_disc_FG(B,D) {
      assign initial cutoff values to the continuous variables;
      push the continuous variables onto a queue Q;
      while (Q is not empty) {
            remove the first element from Q and store it in X;

            // try to improve X's discretization, starting from scratch
            old_disc = disc(X);
            reset disc(X) to a single bin;
            do {
                  determine the cutoff c that maximizes Gain(X,c);
                  delta = DL(X,disc(X) + c) - DL(disc(X));
                  if (delta < 0) {
                        disc(X) = disc(X) + c;
                  }
            } while (delta < 0)

            // if we succeed, make sure X's Markov blanket is in Q
            if (DL(X,disc(X)) < DL(X,old_disc) {
                  for (all Y in X's Markov blanket) {
                        if (Y is not in Q) {
                              add Y to Q
                        }
                  }
            } else {
                  disc(X) = old_disc;
            }
      }
}
```

Figure 5.9. Pseudocode for Friedman and Goldszmidt's algorithm. The algorithm takes a Bayesian network, `B`, and a collection of training data, `D`, and determines the discretizations of the continuous variables in `B`. `DL(X,disc)` is the local description length for the variable `X` when it is discretized using the cutoff values in `disc`; it is computed using equation 25. `disc(X) + c` refers to the sequence of cutoff values formed when `c` is added to the cutoff values in `disc(X)`. `Gain(X,c)` is the gain in mutual information from adding cutoff value `c` to the discretization of variable `X`; it is computed using equation 26. See Section 5.3.2 for more details of the algorithm.

To fully define the algorithm, two additional features need to be specified. The first is the choice of the initial discretizations for the continuous variables. These initial cutoff values are needed to ensure that it is possible to assess the impact of adding new cutoff values during the initial stages of the algorithm. Friedman and Goldszmidt use a method called *least square quantization* [Cov91] that they describe as essentially attempting to fit k Gaussians to the distribution of values. The second feature of the algorithm that needs to be specified is the order in which variables are considered. The approach taken by Friedman and Goldszmidt is shown in the pseudocode for their algorithm in Figure 5.9. It guarantees that the changes made to the discretization of variable X_i in one pass of the algorithm have a chance to affect the discretizations of the variables in X_i's Markov blanket before X_i's discretization is reconsidered. Friedman and Goldszmidt do not specify the order in which nodes are initially added to the queue used by the algorithm; I assume that they use a random ordering.

Because the algorithm outlined in Figure 5.9 takes a greedy approach, it is only guaranteed to converge to a local minimum of the description length. Thus, it may not actually find the optimal discretizations. An exhaustive search for the optimal discretizations is typically infeasible, but other, non-exhaustive search strategies could be used in place of greedy search.

5.3.3 Learning Discretizations of Variables in an Influence Diagram

Friedman and Goldszmidt's algorithm cannot be applied directly to an influence diagram, because the algorithm does not take into account the decision and value nodes that are found in such models. This section describes an extended and modified version of their algorithm (hereafter referred to as *the modified algorithm*) that is able to handle influence diagrams. It also explains how the presence of value nodes in an influence diagram can simplify the process of specifying initial cutoff values for the variables being discretized, and it describes an additional modification that I have made to their algorithm.

5.3.3.1 Dealing with Decision Nodes

The modified algorithm ignores the informational arcs in an influence diagram. It treats decision nodes as if they were root chance nodes (i.e., chance nodes without parents) that have only chance and value nodes as their children.

Typically, a decision node will already be a discrete variable, and thus it will only be considered by the algorithm when it is the parent of a continuous chance node or a value node. However, if a decision node represents a continuous variable, it could also be discretized by the algorithm. In the context of software tuning, this could be useful when dealing with numerical-valued knobs that have a large range of possible settings. By discretizing such a knob, the influence diagram could be used to recommend an interval of possible settings that corresponds to one of the bins in the discretized version of the knob, and then other criteria—possibly including feedback from attempts to adjust the knob within the recommended interval—could be used to select a setting from that interval.

5.3.3.2 Dealing with Value Nodes

In and of itself, a value node never needs to be discretized, because such a node is represented and manipulated in the same way whether it is continuous or discrete. However, in order to capture the relevant interactions between a value node and its parents, the modified algorithm begins by assigning a temporary fixed discretization to each value node, basing the cutoff values on the values of the node that appear in the training data. Doing so allows the algorithm to treat a value node as a chance node so that it can measure the mutual information between a value node and its parents and thereby discretize the parents more effectively.

A number of methods could be used to determine the fixed discretizations assigned to the value nodes; I have chosen to use an equal-height approach (Section 5.3.1.2). The limitations of this approach, as described in Section 5.3.1.3, should be less of an issue in this context. In particular, the need to choose the number of bins for these nodes should be less problematic. Ordinarily, we need to avoid specifying too many bins for a node, because doing so can lead to situations in which there is insufficient training data to learn some of the node's parameters. But because the discretizations of the value nodes are not used as part of the final model, this concern does not apply here. Instead, we can afford to give each value node a large number of bins—and doing so makes it more likely that the discretizations will capture the interactions between the value nodes and their parents. The experiments presented in Section 7.5.3 explore the impact of varying the number of equal-height bins used for this purpose.

Because the discretizations added to the value nodes do not affect the complexity of the final model, the modified algorithm does not consider these discretizations when it computes the complexity-related components of the description length. More specifically, the modified algorithm uses the following modified expression for the local description length:

$$L'(X_i) = \log_2\|X_i^*\| + \frac{\log_2 N}{2}\Bigg(\|\Pi_i^*\|(\|X_i^*\| - 1) + \sum_{\substack{j,\, X_i \in \Pi_j, \\ X_j \notin \mathbf{V}}} \|\Pi_j^*\|(\|X_j^*\| - 1) + \sum_{\substack{j,\, X_i \in \Pi_j, \\ X_j \in \mathbf{V}}} \|\Pi_j^*\|$$

$$+ (N_i - 1)H\left(\frac{k_i - 1}{N_i - 1}\right) - N\left(I[X_i^*;\Pi_i^*] + \sum_{j,\, X_i \in \Pi_j} I[X_j^*;\Pi_j^*]\right)$$

(EQ 27)

where **V** represents the set of value nodes in the model. Note that the second component of this expression—the one that measures the length of the encoded parameters of the discretized versions of a node X_i and its children—now has three terms instead of two. If a child of X_i is *not* a value node, we compute the number of its parameters as before—multiplying the number of instantiations of its parents by one less than the number of values of the node. If a child of X_i *is* a value node, we count only the number of instantiations of its parents, ignoring the number of values of the node itself. This reflects the fact that a value node will ultimately have one expected-value parameter—rather than multiple probability parameters—for each instantiation of its parents.

5.3.3.3 Simplifying the Specification of Initial Cutoff Values

As discussed in Section 5.3.2.3, Friedman and Goldszmidt's algorithm requires that initial cutoff-value sequences be assigned to the nodes being discretized. These preliminary cutoff values are needed to ensure that the algorithm can assess the impact of adding new cutoff values to nodes considered at the start of the algorithm.[3] How-

3. Without initial cutoff values, we could end up with a situation in which all of the nodes in the Markov blanket of a given node, X, have only one value (because they all have the trivial discretization of a single bin). When this is the case, $Gain(X, c) = 0$ for every candidate cutoff value c, and thus the algorithm cannot distinguish between candidate cutoff values.

ever, because these initial cutoff values are not chosen on the basis of mutual information, they may prevent the algorithm from discovering the most effective discretizations for the nodes.

The modified algorithm takes advantage of the fixed discretizations specified for the value nodes (see Section 5.3.3.2) to avoid specifying initial cutoff values for the nodes being discretized. Because these value-node discretizations are in place at the start of the algorithm, the algorithm is able to capture interactions between the value nodes and their parents, even if the parents begin with no cutoff values. Then, once the algorithm begins to learn cutoff values for the parents of the value nodes, it is able to learn cutoff values for other neighbors of those nodes, and so on. Therefore, there is no need to specify initial discretizations for the nodes being discretized.

The modified algorithm begins by initializing the discretizations of the continuous chance nodes to a single bin (i.e., an empty cutoff-value sequence). Then, during the initial pass of the algorithm, nodes are visited according to their distance from the value nodes. More specifically, let $l(X_i)$ be the length of the shortest directed path from node X_i to a value node. When initializing the queue of nodes to be discretized, the algorithm orders the nodes according to their l values so that all nodes with $l = 1$ (i.e., the parents of the value nodes) are visited first, then all nodes with $l = 2$, and so on. After the initial pass, nodes are visited according to the order in which they are added back to the queue by the algorithm. The full, modified algorithm is shown in Figure 5.10.

Because the fixed discretizations specified for the value nodes are not based on mutual-information considerations, they, like the initial discretizations specified in the original algorithm, can also degrade the quality of the learned discretizations. However, the potential negative impact of the value-node discretizations should typically be less than that of the discretizations required by the original algorithm, for two reasons. First, the cutoff values added to the value nodes do not affect the complexity-related components of the description length, and thus they should not reduce the number of cutoff values that the algorithm is able to learn for the

```
learn_disc_ID(I,D) {
      assign initial cutoff values to the value nodes;
      for each continuous chance or decision node n {
            let l(n) = the length of shortest path from n to
               a value node;
            assign the trivial discretization to n;
      }
      add the continuous chance and decision nodes to a queue Q;
      sort Q according to the nodes’ l values, smaller values first;

      while (Q is not empty) {
            remove the first element from Q and store it in X;

            // try to improve X’s discretization
            determine the cutoff value c that maximizes Gain(X,c);
            delta = DL’(X,disc(X) + c) - DL’(X,disc(X));
            if (delta < 0) {
                  disc(X) = disc(X) + c;
            }

            // if we succeed, add X’s Markov blanket--and X--to Q
            if (delta < 0) {
                  for (all Y in X’s Markov blanket) {
                        if (Y is not in Q) {
                              add Y to Q;
                        }
                  }
                  add X to Q;
            }
      }
}
```

Figure 5.10. Pseudocode for an algorithm to learn the discretizations of continuous variables in an influence diagram. The algorithm—which is based on the algorithm of Friedman and Goldszmidt (Figure 5.9)—takes an influence diagram, `I`, and a collection of training data, `D`, and determines the discretizations of the continuous variables in `I`. `DL’(X,disc)` is the modified local description length for the variable `X` when it is discretized using the cutoff values in `disc`; it is computed using equation 27. `disc(X) + c` refers to the sequence of cutoff values formed when `c` is added to the cutoff values in `disc(X)`. `Gain(X,c)` is the gain in mutual information from adding cutoff value `c` to the discretization of variable `X`; it is computed using equation 26. See Section 5.3.3 for more details of the algorithm.

continuous chance and decision nodes. Second, we can afford to give the discretizations of the value nodes a large number of bins, and this should reduce the risk that these discretizations will prevent the algorithm from capturing significant interactions between the variables. Even if the initialization procedure from the original algorithm were used, initial discretizations of the value nodes would still need to be specified. The modified algorithm simply takes advantage of these discretizations to avoid specifying initial discretizations for the continuous chance and decision nodes.

5.3.3.4 An Additional Modification

To improve the discretization of a node, Friedman and Goldszmidt's algorithm removes the node's current discretization and repeatedly adds cutoff values until there are no remaining values that reduce the description length of the model. The modified algorithm takes a different approach to improving a node's discretization. It uses a node's current discretization as the starting point for possible improvements rather than starting over from scratch, and it adds at most one new cutoff value to a node per iteration. Because this approach never reconsiders cutoff values added in earlier rounds of the algorithm, it may produce models with larger description lengths and less accuracy than the models produced using Friedman and Goldszmidt's approach. However, their approach often takes considerably more time, because it tends to produces large cutoff-value sequences at the start of the algorithm—when the increase in description length from adding a cutoff value is small—only to scale them back in later iterations of the algorithm. Section 7.5.3 compares the performance of models produced using these two approaches.

If the modified algorithm is able to add a cutoff value to a node, it adds the node itself back to the queue along with the node's Markov blanket, so that it can subsequently determine if additional cutoff values should be added. In Friedman and Goldszmidt's algorithm, a node is only reconsidered after the discretization of one of its neighbors is changed, because their algorithm always adds as many cutoff values as possible given the current discretizations of the neighbors.

5.4 Learning the Parameters of the Model

Once the training data has been collected and the continuous variables have been discretized, we can learn the parameters of the model from the training data. This section first explains how to estimate parameters for which training data is available. It then presents methods for estimating parameters for which no training data exists.

Finally, it explains how regression can be used to learn appropriate weights for the value nodes so that optimizing their sum is equivalent to optimizing the overall performance measure that the tuner is attempting to optimize.

5.4.1 Estimating Probabilities

As discussed in Section 3.2.2, each chance node in an influence diagram has an associated set of parameters that specify one or more probability distributions. For a root chance node, these parameters specify a marginal distribution over the possible values of the node. For an intermediate chance node, these parameters specify a collection of conditional distributions, one for each instantiation of its parents. If we assume that the continuous variables in the model have all been discretized, these distributions are all simple, multinomial distributions, and each parameter represents the probability that a variable will take on a particular value. The notation θ_{ijk} will refer to the probability that chance node X_i takes on its kth value given the jth instantiation of its parents (if any), $\overline{\theta}_{ij}$ will represent the vector of parameters describing the probability distribution for X_i given the jth instantiation of its parents[4], and Θ will refer to the entire collection of parameters.

5.4.1.1 Maximum Likelihood Estimation

One of the most common methods of estimating probability parameters is *maximum likelihood estimation*, which selects the parameters that maximize the probability of observing the training data. If we assume that the individual training instances are independent of each other and that the vectors of parameters $\overline{\theta}_{ij}$ are also mutually independent, then determining the maximum-likelihood estimates simply involves maintaining counts and using them to compute ratios of the following form:

$$\theta_{ijk} = \frac{N_{ijk}}{N_{ij}} \qquad \text{(EQ 28)}$$

4. If X_i has no parents, j will always be 0.

where N_{ijk} is the number of training examples in which X_i takes on its kth value *and* X_i's parents take on the jth combination of their values, and N_{ij} is the total number of training examples in which X_i's parents take on the jth combination of their values (i.e., $N_{ij} = \sum_k N_{ijk}$). For root chance nodes, $N_{ij} = N$, the total number of training instances.

One drawback of maximum likelihood estimation is its tendency to overfit the training data. In particular, if a given instantiation of a node's parents is seldom seen in the training data (i.e., if N_{ij} is small), the associated parameter estimates are often unreliable. In addition, if a particular parent instantiation is *never* seen, maximum likelihood estimation cannot be performed for the parameters associated with that instantiation. Section 5.4.3 presents a method for estimating probability parameters in such cases.

5.4.1.2 Incorporating Prior Estimates

One way to avoid overfitting is to take a Bayesian approach to learning the probabilities, in which we treat Θ as a (vector-valued) random variable with a prior distribution. A full treatment of this approach is beyond the scope of this thesis; readers are encouraged to consult Heckerman's tutorial on learning in Bayesian networks [Hec95] for more detail. In the context of learning parameters for multinomial distributions in an influence diagram, this approach can be reduced to what Mitchell refers to as computing an *m-estimate* of a probability [Mit97]. Such an estimate is computed as follows:

$$\theta_{ijk} = \frac{N_{ijk} + m_{ijk}\theta_{ijk}^{\circ}}{N_{ij} + m_{ijk}} \qquad \textbf{(EQ 29)}$$

where θ_{ijk}° is a prior estimate of the value of θ_{ijk} and m_{ijk} is a value that specifies how much weight to give to that estimate. m_{ijk} is often referred to as the *equivalent sample size*, because if we use the same m_{ijk} value for all of the prior probabilities associated with a given instantiation of a node's parents (i.e., for all of the probabilities in the vector $\bar{\theta}_{ij}°$), it is as if we had observed m_{ijk} additional training examples for that instantiation that were distributed according to the probabilities in $\bar{\theta}_{ij}°$. Typically, a single equivalent sample size is chosen for the entire model.

To choose the prior probabilities $\bar{\theta}_{ij}°$ associated with a given parent instantiation, it is often assumed that the values of the variable are either uniformly distributed or distributed according to the marginal distribution of the variable in the training data [Fri97]. In the former approach, we set $\theta_{ijk}° = \frac{1}{r_i}$, where r_i is the number of possible values of the variable X_i. In the latter approach, we set $\theta_{ijk}° = \frac{M_{ik}}{N}$, where M_{ik} is the number of the N training examples in which X_i takes on its kth value.

Maximum a posteriori (MAP) estimation is another method of estimating probabilities that begins with a prior distribution for the parameters. It selects the most probable parameter values in light of the training data—i.e., the values of the parameters that have the maximum posterior probability given the training data. For many types of probability distributions, MAP estimation is significantly simpler than a full Bayesian approach.

5.4.1.3 Dealing with Incomplete Data

All of the estimation methods described above assume that the training data is *complete*, which means that each training example includes a value for each of the variables in the model. In the context of software tuning, this is typically a reasonable assumption, especially if a workload generator is used to produce the training examples. Even if a particular variable cannot be observed during training, it is often possible to estimate its value.

When the training data is incomplete, other methods for estimating the probabilities are needed. One such method is the *expectation maximization (EM)* algorithm [Dem77]. Heckerman [Hec95] explains how to use this and other techniques to estimate probability parameters in a Bayesian network, and the same techniques could be applied to an influence diagram as well.

5.4.2 Estimating Expected Values

As discussed in Section 3.2.2, each value node in an influence diagram has an associated set of parameters that specify the conditional expected value of the node for each possible instantiation of its parents. The notation E_{ij} will refer to the conditional expected value of value-node variable X_i given the jth instantiation of its parents.

5.4.2.1 Using Simple Averages

The simplest method of estimating the conditional expected values of a value node involves computing averages of the following form:

$$E_{ij} = \frac{\sum_k V_{ijk}}{N_{ij}} \tag{EQ 30}$$

where V_{ijk} is the value of X_i in the kth training example in which X_i's parents take on the jth combination of their values, and N_{ij} is again the total number of training examples in which X_i's parents take on the jth combination of their values.

Like the maximum-likelihood method of estimating probabilities (Section 5.4.1.2), this method of using averages to estimate expected values also risks overfitting the training data. If a given instantiation of a value node's parents is seldom seen, the corresponding average may underpredict or overpredict the true conditional expected value of the node for that instantiation. If a particular parent instantiation is *never* seen, this method cannot be used for the expected value associated with that instantiation. Section 5.4.3 presents methods for estimating expected values in such cases.

5.4.2.2 Incorporating Prior Estimates

If reasonable prior estimates can be determined for the expected-value parameters, the risk of overfitting can be reduced by adopting an approach similar to the m-estimate method of estimating probabilities (Section 5.4.1.2). Using this approach, the expected values would be computed as follows:

$$E_{ij} = \frac{\sum_{k} V_{ijk} + m_{ij} E_{ij}°}{N_{ij} + m_{ij}} \qquad \text{(EQ 31)}$$

where $E_{ij}°$ is a prior estimate of the value of E_{ij} and m_{ij} is the equivalent sample size for that estimate.

One limitation of this method is that can be difficult to devise reasonable prior estimates of the expected values. One possibility would be to average all of the values of the node that appear in the training set (i.e., $E_{ij}° = \frac{\Sigma_{j,k}(v_{ijk})}{N}$). Another possibility would be to average values from training examples that include one or more components of the parent instantiation associated with the parameter. For example, if the jth parent instantiation of the *waits/txn* node in Figure 4.2 included *db_rmw* = 1 and *deadlock_policy* = random, then the prior estimate $E_{ij}°$ associated with that instantiation could be computed as the average of all training examples that include those two knob settings.

5.4.2.3 Dealing with Incomplete Data

Both of the methods described above for estimating expected-value parameters assume that all of the training examples include values for the value nodes and their parents. As mentioned in Section 5.4.1.3, we should ordinarily be able to measure or estimate the values of variables in an influence diagram for software tuning, so this assumption is not an unreasonable one.

If the training data for the parents of a value node is incomplete, it should be possible to adapt the methods described in Section 5.4.1.3 to determine the expected-value parameters associated with the value node. However, these methods are not useful if data is lacking for a value node itself. Fortunately, this is not a significant limitation, because it is unlikely that a tuner would lack measurements for the performance metrics that it is attempting to optimize.

5.4.3 Estimating Unseen Parameters

When a given instantiation of a node's parents is never seen in the training data, it becomes difficult to accurately estimate the parameters associated with that instantiation. In the discussion that follows, parameters that are associated with unseen parent instantiations will be referred to as *unseen parameters*. Neither maximum-likelihood estimation (Section 5.4.1.1) nor the method of using simple averages to estimate expected values (Section 5.4.2) can be used to compute unseen parameters. The *m*-estimate methods (Sections 5.4.1.2 and 5.4.2.2) can be used; they assign an unseen parameter the value of its prior estimate. However, these prior estimates tend to be extremely imprecise.

This section presents a method of estimating unseen parameters that bases its estimates on small subsets of the parameters for which training data is available—parameters that will be referred to as *seen parameters*. This method takes a nearest-neighbor approach to parameter estimation, in which an estimate of an unseen parameter is computed by averaging the values of the seen parameters associated with the parent instantiations are determined to be "nearest" to the unseen parent instantiation. The distance metric used to determine the nearest neighbors is learned from the training data. The proposed method also takes advantage of the monotonic relationship that tends to exist between a node and its chance-node parents in an influence diagram developed for software tuning, using this monotonicity to determine constraints on the values of unseen parameters. These constraints are used to validate the nearest-neighbor estimate and possibly to modify it. The resulting estimates should typically be more accurate than estimates that ignore the training data entirely (e.g., by using a uniform distribution for one of the conditional distributions of a chance node) or that employ coarser-grained summaries of the training data (e.g., by using a node's marginal distribution for one of its conditional distributions).

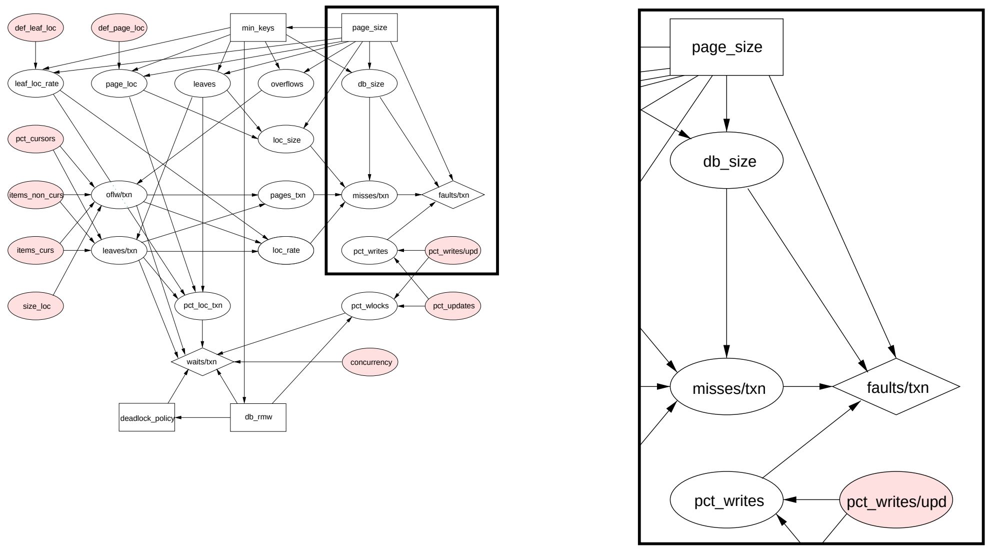


Figure 5.11. The *faults/txn* value node and its parents. The figure on the right provides a closeup of a portion of the influence diagram for Berkeley DB; the relevant portion of the diagram is indicated by the box on the figure on the left. The four parents of *faults/txn* are the nodes that are connected to it by a directed arc.

To understand the material that follows, it is important to recall that an instantiation of a node's parents is a vector of value assignments to the parents. For example, the *faults/txn* value node in the influence diagram for Berkeley DB has the vector of parents (*page_size*, *misses/txn*, *pct_writes*, *db_size*), as shown in Figure 5.11. One possible instantiation of these parents is the vector (8192, 0, 0, 0), where all of the nodes except *page_size* have been discretized.

5.4.3.1 Determining Nearest-Neighbor Estimates for Expected Values

If an instantiation π of the parents of a node is never seen in the training data, the estimates of the parameters associated with π can be based on the parameters associated with the seen parent instantiations that are most similar to the π. This type of approach to estimation is often referred to as a *nearest-neighbor* approach [Mit97]. The degree to which two instantiations are similar to each other is measured using some sort of distance metric, $d(\pi_1, \pi_2)$; smaller values for d indicate higher degrees of similarity. For a given instantiation π_1, the instantiation π_2 that minimizes $d(\pi_1, \pi_2)$ is referred to as π_1's *nearest neighbor*.

If an unseen parent instantiation π for the variable X_i has a unique nearest neighbor, π_n, in the set of seen instantiations of X_i's parents, the expected value associated with π_n (i.e., $E(X_i | \Pi_i = \pi_n)$) is used as the nearest-neighbor estimate. If π has multiple nearest neighbors (i.e., if there are multiple seen instantiations that produce the same minimal value for d), the average of their associated expected values is used as the estimate. If X_i is a value node, the expected value associated with a seen parent instantiation is simply the parameter associated with that instantiation, as computed by one of the methods described in Section 5.4.2. If X_i is a chance node, the expected value associated with a seen parent instantiation is computed as follows:

$$E_{ij} = \sum_k V_{ik} \theta_{ijk} \qquad \text{(EQ 32)}$$

where V_{ik} is the average of the undiscretized values of X_i that fall into the kth bin when X_i is discretized, and θ_{ijk} is the probability that X_i takes on a value from that kth bin given the jth instantiation of its parents, as computed by one of the methods described in Section 5.4.1.

One challenge involved in employing a nearest-neighbor approach is determining a suitable distance metric. In some domains, it is appropriate to use the Euclidean distance—or its square:

$$d(\pi_1, \pi_2) = \sum_i (\pi_1[i] - \pi_2[i])^2 \qquad \text{(EQ 33)}$$

where $\pi[i]$ denotes the ith component of the vector π. However, this metric assumes that a displacement in one of the vector's components has approximately the same effect as the same displacement in another component. This assumption will often be violated in the context of estimating unseen parameters. For example, in Figure 5.11, a one-unit increase in the discretized value of *misses/txn* may have a very different effect on the value of *faults/txn* than a one-unit increase in the value of *pct_writes*. Therefore, an alternative distance metric is needed.

The proposed estimation algorithm learns the distance metric from the training data. It does so by determining, for each chance-node parent of a node *N*, the average effect of each possible displacement in the value of that parent on the expected value of *N*.[5] The algorithm does *not* attempt to learn the impact of modifications to a decision-node parent (i.e., a parent that represents a knob), because the tuner needs to distinguish the effects of different knob settings on performance, and thus we do not want the algorithm to base its estimates for parameters associated with one knob setting on data gathered for another knob setting. If two parent instantiations have different values for a decision-node parent, they are given an infinite distance. In the unlikely case that there are *no* seen parent instantiations with decision-node values that match the values in the unseen instantiation, the algorithm uses a special value (e.g., ∞) as the nearest-neighbor estimate, and the remainder of the estimation algorithm handles this value appropriately.

Given a variable X_i with parents Π_i, the algorithm learns a distance metric by considering all pairs (π_1, π_2) of seen instantiations of Π_i that differ only in the value of a single chance-node parent. For example, the instantiation (8192, 0, 0, 0) of the parents of *misses/txn* (Figure 5.11) would be considered together with the instantiations (8192, 1, 0, 0), (8192, 0, 1, 0), and (8192, 0, 0, 1); because 8192 is the value of a decision node, it is not varied. As it considers the relevant pairs of instantiations, the algorithm maintains a collection of running sums, one for each possible displacement of each chance-node parent; S_{ijk} denotes the running sum for displacements of $\pm k$ in the value of X_i's *j*th parent. Given a pair of instantiations, (π_1, π_2), that differ in only the value of the *j*th parent, the algorithm increments the appropriate running sum as follows:

$$S_{ij(|\pi_1[j]-\pi_2[j]|)} = S_{ij(|\pi_1[j]-\pi_2[j]|)} + |E(X_i|\Pi_i = \pi_1) - E(X_i|\Pi_i = \pi_2)| \quad \textbf{(EQ 34)}$$

5. Only the magnitude of the displacements is considered.

The algorithm also maintains a count c_{ijk} of the number of times that each running sum is updated. After considering all relevant pairs of instantiations, the ratio $\frac{s_{ijk}}{c_{ijk}}$ provides an estimate of the change in the expected value of X_i that results from a displacement of $\pm k$ in the value of X_i's jth parent. These estimates are imprecise: they do not consider the direction of the displacements, and they ignore both the initial and final values of the parent in question, as well as the values of the other parents. However, they should be precise enough to allow us to compute a reasonable distance metric.

Given a pair (π_{i1}, π_{i2}) of instantiations of the parents of node X_i, the algorithm computes the distance between π_{i1} and π_{i2} as follows:

$$d(\pi_{i1}, \pi_{i2}) = \begin{cases} \sum_j \frac{s_{ij(|\pi_1[j] - \pi_2[j]|)}}{c_{ij(|\pi_1[j] - \pi_2[j]|)}} & \text{if } \pi_{i1} \text{ and } \pi_{i2} \text{ have the same decision-node values} \\ \infty & \text{otherwise} \end{cases} \quad \text{(EQ 35)}$$

where the summation iterates over X_i's chance-node parents. In other words, the distance of a multi-dimensional displacement is the sum of the distances associated with the corresponding single-dimensional displacements. Although adding distances in this way may not always be accurate—e.g., the effects of the individual displacements may actually cancel each other out and produce a parent instantiation that is closer to the original instantiation than instantiations that involve a single displacement—it avoids the extra overhead that would be needed to actually learn the distance values for multi-dimensional displacements.

5.4.3.2 Testing for Monotonic Parent-Child Relationships

In an influence diagram developed for software tuning, a chance or value node typically has a *monotonic* relationship with each of its chance-node parents.[6] This means that when the value of a parent increases, the value of the node either: (a) increases or stays the same, but never decreases; or (b) decreases or stays the same, but never increases. The influence diagram for Berkeley DB (Figure 4.2) provides a number of

6. This claim obviously does not hold if the nodes involved take on nominal—as opposed to numeric—values. However, because the chance nodes in an influence diagram for software tuning typically represent variables that are measured or computed, they should ordinarily have numeric values.

examples of this phenomenon. Indeed, *all* of the value nodes and intermediate chance nodes in that model have monotonic—or nearly monotonic—relationships with their chance-node parents. For example, increasing the average number of pages accessed per transaction (*pages/txn*) tends to increase the average number of memory-pool misses per transaction (*misses/txn*), unless the locality set fits in the memory pool, in which case *misses/txn* stays the same as *pages/txn* increases. On the other hand, increasing the percentage of accesses that go to items in the locality set (*loc_rate*) tends to decrease *misses/txn*, unless the locality set fits in the memory pool or is large enough that almost all accesses result in a miss, in which case *misses/txn* stays the same as *loc_rate* increases.

Although there is no guarantee that a monotonic relationship will exist between a chance or value node and its chance-node parents, this type of relationship occurs frequently enough that it makes sense to attempt to exploit it. The algorithm for estimating unseen parameters learns from the training data whether a monotonic relationship exists. When such a relationship does exist, the algorithm attempts to devise constraints on the unseen parameters, as described in the following section, and it uses these constraints to supplement the nearest-neighbor estimates.

Like the procedure for determining the nearest-neighbor distance metric, the procedure for testing the relationships between a node and its parents does not consider parents that are decision nodes. As stated earlier, we do not want the algorithm to base its estimates for parameters associated with one knob setting on data gathered for another knob setting, and thus there is no reason to attempt to learn whether a node has a monotonic relationship with its decision-node parents.

For a variable X_i with parents Π_i, the algorithm tests for monotonic relationships between X_i and the chance nodes in Π_i by considering all pairs of seen instantiations of Π_i that differ only in the value of one chance-node parent. Given a pair of instantiations, (π_1, π_2), that differ in only their kth component, the algorithm computes the following ratio:[7]

7. See Section 5.4.3.1 for an explanation of how the expected value associated with a given parent instantiation is determined.

$$\frac{E(X_i | \Pi_i = \pi_2) - E(X_i | \Pi_i = \pi_1)}{\pi_2[k] - \pi_1[k]} \quad \text{(EQ 36)}$$

where $\pi[k]$ denotes the kth component of the vector π. If this ratio is positive, the pair of instantiations (π_1, π_2) can be considered a piece of evidence that the relationship between X_i and its kth parent is *monotone increasing*, meaning that the value of X_i either increases or stays the same when the value of the parent increases. If this ratio is negative, this pair of instantiations can be considered a piece of evidence that the relationship between X_i and its kth parent is *monotone decreasing*, meaning that the value of X_i either decreases or stays the same when the value of the parent increases.

To weigh the evidence provided by all of the relevant pairs of instantiations, the algorithm increments the value of a counter variable C_{ik} when the ratio in equation 36 is positive for instantiations of Π_i that differ in their kth component, and it decrements the same counter when the ratio is negative. If the ratio is 0, the algorithm does nothing. After all of the instantiations have been compared, the algorithm determines the nature of the relationship between X_i and its kth parent according to the following decision rule:

```
if (Cik > t, a positive threshold)
        Xi is monotone increasing with respect to its kth parent
else if (Cik < -t)
        Xi is monotone decreasing with respect to its kth parent
else
        the relationship between Xi and its kth parent is not monotonic.
```

By learning the nature of the relationships between a node and its parents, the algorithm is able to devise constraints on the conditional expected value of the node given an unseen instantiation of its parents.

5.4.3.3 Determining Constraints on Expected Values

If an instantiation π_{unseen} of the parents of node X_i is never seen in the training data, the algorithm searches for constraints on the conditional expected value of the variable given that instantiation (i.e., on the value of $E(X_i | \Pi_i = \pi_{unseen})$) by considering all seen parent instantiations that differ from the unseen instantiation in only the value of one chance-node parent.[8] If the relationship between X_i and that parent is monotonic, the expected values associated with these instantiations can be used to determine constraints on $E(X_i | \Pi_i = \pi_{unseen})$. The portion of the estimation algorithm that is responsible for determining these constraints is shown in Figure 5.12. It attempts to determine the tightest possible lower and upper bounds on the value of $E(X_i | \Pi_i = \pi_{unseen})$, but there will be cases in which it cannot determine nontrivial values for one or both of these bounds.[9] In particular, if the parent values in the unseen instantiation are all equal to the lowest possible values of their respective parents or all equal to the highest possible values of those parents, the algorithm will be able to determine at most one bound. Other factors that can limit the number of constraints include the absence of a monotonic relationship between X_i and one or more of its parents and the presence of additional unseen parent instantiations.

Even parameters associated with seen parent instantiations have limited accuracy. Therefore, it is possible for the constraints learned by the algorithm to be inconsistent, with the upper bound being smaller than the lower bound. If this occurs, the algorithm uses the constraint that was obtained from the parent instantiation that is deemed closest to the unseen instantiation by the distance metric described in the Section 5.4.3.1, and it discards the other constraint.

5.4.3.4 Finalizing Estimates for Unseen Expected Values

When estimating the conditional expected value of a node given an unseen parent instantiation, the estimation algorithm combines the nearest-neighbor estimate (Section 5.4.3.1) and the constraints on the expected value (Section 5.4.3.3) to compute a final estimate. Let l be the lower bound (possibly $-\infty$) and u be the upper bound (possibly ∞) determined by the algorithm in Figure 5.12, and let π_l and π_u be the parent

8. The algorithm does not consider parent instantiations that differ from the unseen instantiation in multiple dimensions. It is often not possible to determine constraints from the parameters associated with such instantiations, because the node in question may be monotone increasing with respect to one parent and monotone decreasing with respect to another.
9. The trivial constraints are a lower bound of $-\infty$ and an upper bound of ∞.

```
find_constraints(X_i, p_unseen) {
      lower = -∞;
      upper = ∞;
      for each seen instantiation, p_seen, of X_i’s parents that
       differs from p_unseen in only the value of one chance-node parent
      {
            k = the index of the component in which
               p_seen and p_unseen differ;
            if (p_seen[k] < p_unseen[k]) {
                  if (C_ik > t) {
                        //
                        // Monotone increasing and below,
                        // so E(X_i|p_seen) <= E(X_i|p_unseen)
                        //
                        if (E(X_i|p_seen) > lower)
                              lower = E(X_i|p_seen);
                  } else if (C_ik < -t) {
                        //
                        // Monotone decreasing and below,
                        // so E(X_i|p_seen) >= E(X_i|p_unseen)
                        //
                        if (E(X_i|p_seen) < upper)
                              upper = E(X_i|p_seen);
                  }
            } else {
                  // p_seen[k] > p_unseen[k]
                  if (C_ik > t) {
                        //
                        // Monotone increasing and above,
                        // so E(X_i|p_seen) >= E(X_i|p_unseen)
                        //
                        if (E(X_i|p_seen) < upper)
                              upper = E(X_i|p_seen);
                  } else if (C_ik < -t) {
                        //
                        // Monotone decreasing and above,
                        // so E(X_i|p_seen) <= E(X_i|p_unseen)
                        //
                        if (E(X_i|p_seen) > lower)
                              lower = E(X_i|p_seen);
                  }
            }
      }
}
```

Figure 5.12. Pseudocode for an algorithm to learn constraints on the expected value of a chance or value node in an influence diagram. The algorithm—which is based on the algorithm of Friedman and Goldszmidt (Figure 5.9)—takes the node in question, X_i, and an unseen instantiation of its parents, p_unseen, and attempts to determine a lower bound, lower, and an upper bound, upper, on the conditional expected value of X_i given p_unseen. p[k] denotes the *k*th component of the vector p. C_ik is the counter variable for X_i and its *k*th parent; it is computed using the procedure described in Section 5.4.3.2. E(X_i|p) refers to the the conditional expected value of X_i given the instantiation p of its parents. See Section 5.4.3.3 for more details of the

instantiations associated with l and u, respectively. Let n be the nearest-neighbor estimate, and let π_n be its corresponding parent instantiation. The estimation algorithm determines if the nearest-neighbor estimate is consistent with the constraints (i.e., if $l \le n \le u$). If it is consistent, the algorithm uses n as the final estimate.

If the nearest-neighbor estimate is not consistent, the algorithm computes the final estimate from the constraints. If only one of the constraints is non-trivial (i.e., if l = -∞ or u = ∞), the algorithm uses the value of that constraint as the final estimate. Otherwise, it uses a weighted average of the constraints as the final estimate, with the distance values of the corresponding parent instantiations serving as the weights. Thus, the final estimate for an unseen expected value is determined as follows:[10]

$$E(X_i|\Pi_i = \pi_{unseen}) = \begin{cases} n \text{ if } n \in [l, u] \\ l \text{ if } n \notin [l, u] \text{ and } u = \infty \\ u \text{ if } n \notin [l, u] \text{ and } l = -\infty \\ l + \dfrac{d(\pi_{unseen}, \pi_l)}{d(\pi_{unseen}, \pi_l) + d(\pi_{unseen}, \pi_h)} \cdot (u - l) \text{ otherwise} \end{cases} \qquad \text{(EQ 37)}$$

5.4.3.5 *Estimating Unseen Probability Parameters*

The estimation algorithm also uses a nearest-neighbor approach supplemented by constraints to estimate unseen probability parameters. Given a chance node X_i with an unseen parent instantiation π, the algorithm first uses the process described in the preceding sections to estimate the expected value, e, of X_i given π. As before, this process involves determining a nearest-neighbor estimate n, a lower bound l, and an upper bound u, and using these values to compute e. The algorithm then constructs a probability distribution that has an expected value equal to e, and it uses the parameters of this distribution as the estimates of the unseen probability parameters. In the unlikely case that the estimation algorithm is unable to determine a reasonable

10. Note that no special case is needed for situations in which both constraints are trivial, because it will always be the case that $-\infty \le n \le \infty$.

estimate for e from the training data (e.g., if the only seen parent instantiations have different decision-node values than the unseen instantiation), the algorithm assigns parameters that constitute a uniform distribution.

If the value of e was derived from the probability distribution associated with a single, seen parent instantiation, the algorithm can simply reuse the vector of probability parameters for that distribution. More specifically, if e equals the lower bound l, it can simply use $\overline{\theta}_l$, the vector of probability parameters used to compute l. Similarly, if e equals the upper bound u, it can simply use $\overline{\theta}_u$, the vector of probability parameters used to compute u. Finally, if e equals the nearest-neighbor estimate n and n was computed from a single nearest neighbor, the algorithm can simply use $\overline{\theta}_n$, the probability parameters associated with that neighbor.

If the estimate of the expected value, e, was derived from multiple probability distributions, the algorithm creates a new probability distribution with an expected value equal to e. In such cases, the algorithm essentially "averages" the probability distributions by computing the average of their associated expected values and creating a distribution whose expected value is equal to the average.

The new distribution is constructed in two steps. First, the algorithm finds the bin in the discretized version of X_i that contains the estimated expected value, e. In other words, it uses equation 18 to determine the discretized value, *e_disc*, that corresponds to e. The algorithm gives at least some of the probability mass for the new distribution to the probability parameter associated with that bin. To ensure that the new distribution has the desired expected value, the algorithm may also give some probability mass to one of the neighboring parameters. The other parameters are all given a value of 0. The actual algorithm for assigning probability mass to the components of the distribution (a function that I have named *assign_probs*) is shown in Figure 5.13.

In summary, given a nearest-neighbor estimate n, a lower bound l, an upper bound u, and a final estimate e for the expected value of chance node X_i given the (unseen) jth instantiation of its parents—as well as the probability vectors associated

```
assign_probs(X_i, V_i, e, e_disc) {
      initialize the probs array to all 0s;
      if (e < V_i[e_disc] && e_disc > 0) {
            //
            // Reduce the expected value by assigning some
            // probability mass to the bin before e_disc.
            //
            probs[e_disc] = (e - V_i[e_disc]) /
               (V_i[e_disc] - V_i[e_disc-1]);
            probs[e_disc - 1] = 1.0 - probs[e_disc];
      } else if (e > V_i[e_disc] && e_disc < ‖X_i‖ - 1) {
            //
            // Increase the expected value by assigning some
            // probability mass to the bin after e_disc.
            //
            probs[e_disc] = (V_i[e_disc + 1] - e) /
               (V_i[e_disc + 1] - V_i[e_disc]);
            probs[e_disc + 1] = 1.0 - probs[e_disc];
      } else
            probs[e_disc] = 1.0;
      return (probs);
}
```

Figure 5.13. Pseudocode for an algorithm to construct a discrete probability distribution with a desired expected value for a chance node in an influence diagram. The algorithm takes a chance node, `X_i`; an array of real values, `V_i`, where `V_i[k]` is the average of the undiscretized values of `X_i` that fall into the `k`th bin when `X_i` is discretized; the desired expected value, `e`, of the probability distribution; and the discretized value, `e_disc`, that corresponds to `e` under `X_i`'s discretization. It produces an array of probability parameters, `probs`, where `probs[j]` is the probability mass assigned to the `j`th bin of `X_i`. ‖`X_i`‖ is the number of possible values of `X_i`. See Section 5.4.3.5 for more details of the algorithm.

with n, l, and u—the vector of parameters $\bar{\theta}_{ij}$ describing the probability distribution associated with the unseen instantiation is determined as follows:

$$\bar{\theta}_{ij} = \begin{cases} \bar{\theta}_n \text{ if } e = n \text{ and there is one nearest neighbor} \\ \bar{\theta}_l \text{ if } e = l \\ \bar{\theta}_u \text{ if } e = u \\ \bar{\theta}_{uniform} \text{ if an estimate for } e \text{ could not be determined} \\ \text{the vector returned by } assign\_probs(X_i, V_i, e, e\_disc) \text{ otherwise} \end{cases} \tag{EQ 38}$$

where V_i is an array of real values such that $V_i[k]$ is the average of the undiscretized values of X_i that fall into the kth bin when X_i is discretized, and e_disc is the discretized value that corresponds to e under X_i's discretization.

5.4.4 Learning Weights for the Value Nodes

As discussed in Section 5.1.1.2, it is often impractical to use a single value node to represent the performance metric that the tuner is attempting to optimize. Therefore, it may be necessary to employ multiple value nodes—each of which reflects one aspect of the system's performance—and to learn appropriate weights for these nodes so that optimizing their sum is equivalent to optimizing the overall performance metric. For example, in the influence diagram for Berkeley DB (Figure 4.2), the *misses/txn* and *faults/txn* value nodes are used in place of a single value node for the throughput of the system, and the tuner needs to learn weights for these nodes so that minimizing their sum (i.e., maximizing the opposite of their sum) is equivalent to maximizing throughput.

Regression techniques provide one means of learning the appropriate weights for the value nodes from the training data. More formally, the tuner can use multivariate linear regression [Net96] to learn the coefficients c_1, c_2, ..., c_n of a function of the following form[11]

$$y = c_0 + c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \qquad \textbf{(EQ 39)}$$

where x_1, x_2, ..., x_n are the n value nodes and y is either the performance metric that the tuner is attempting to optimize or some transformed version of that metric. y is known as the *dependent variable* in the regression function, and the xs are known as the *independent variables*.

Using the overall performance metric, M, as the dependent variable in equation 39 assumes that M can be expressed as a linear combination of the value nodes. However, this assumption may often be unrealistic, especially in cases in which the value nodes represent performance losses that the system experiences. In such cases, the coefficients learned will be negative—reflecting the fact that increasing the performance losses reduces the performance of the system—and thus the resulting function will produce negative y values for large values of the xs.

11. The value of c_0 will also be determined by the regression techniques, but it is not needed by the tuner.

However, performance metrics such as throughput can never be negative, regardless of the performance losses incurred. In the Berkeley DB influence diagram, for example, the throughput of the system can never be negative, no matter how large the values of *misses/txn* and *faults/txn* become.

To handle cases in which the overall performance metric cannot be expressed as a linear combination of the value nodes, the tuner can employ one of several remedial measures that have been developed by statisticians [Net96, Section 3.8]. Some of these measures involve abandoning equation 39 in favor of a nonlinear regression model; others transform one or more of the variables so that linear regression can still be used.

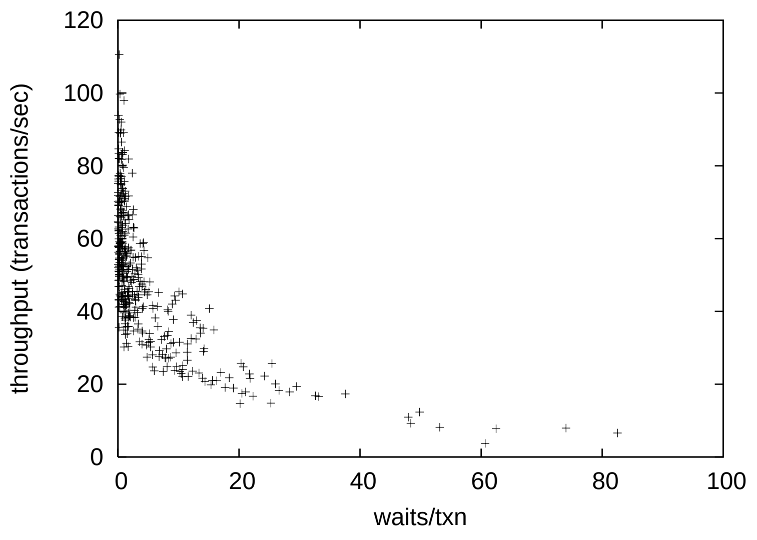
To determine the appropriate remedial measure for my work with Berkeley DB, I considered a subset *S* of training examples that all had a particular pair of discretized values for *pages/txn* and *pct_writes*; focusing on examples with similar values for these variables made it easier to isolate the impact of *faults/txn* and *waits/txn* on throughput. To consider the impact of each type of performance loss separately, I first selected the examples in *S* with discretized *faults/txn* values from a particular bin and plotted throughput as a function of *waits/txn* for those examples. I then selected those examples in *S* with discretized *waits/txn* values from a particular bin and plotted throughput as a function of *faults/txn* for those examples.

The resulting scatterplots (Figure 5.14) show that throughput appears to decay exponentially in response to increases in *faults/txn* and *waits/txn*. Thus, it seems reasonable to assume that the relationship between throughput (t) and the value nodes (w and f) could be captured by a function of the following form:

$$t = \alpha_0 e^{-\alpha_1 w - \alpha_2 f}$$

where α_0, α_1, and α_2 are constants. Taking the natural logarithm of both sides of this equation restores a linear combination of the value nodes:

$$\ln t = \ln \alpha_0 - \alpha_1 w - \alpha_2 f$$



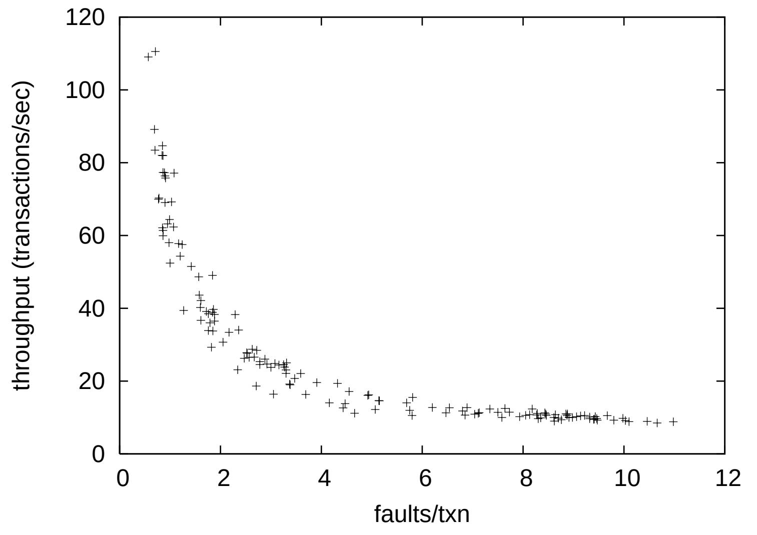


Figure 5.14. Throughput as a function of *waits/txn* and *faults/txn*. These graphs show the relationship between *throughput* and *waits/txn* (*top*) and the relationship between *throughput* and *faults/txn* (*bottom*) in subsets of the training data for the influence diagram for Berkeley DB. The training examples used to construct both graphs were selected to have comparable values for the *pages/txn* and *pct_writes* variables in the model. More specifically, they all had *pages/txn* values in [19, 37] and *pct_writes* values in [0, 4]. In the top graph, the training examples were further restricted to have *faults/txn* values in [0.64, 1.24]. In the bottom graph, the training examples were further restricted to have *waits/txn* values in [0.09, 0.23]. The ranges of values for *pages/txn* and *pct_writes* correspond to bins from discretizations learned by the algorithm presented in Section 5.3.3. The ranges of values for *faults/txn* and *waits/txn* correspond to bins from equal-height discretizations (Section 5.3.1.2), where each variable was given eight bins.

Therefore, I used the natural logarithms of the throughput values as the y values in equation 39 and applied simple linear regression to learn the weights of the value nodes.

More generally, the domain expert who designs the influence diagram for a given automated tuner should determine which transformations of the variables (if any) should be performed before the tuner uses regression to learn the value nodes' weights. When the value nodes represent performance losses, taking the natural logarithm of the dependent variable will often be an appropriate transformation to employ.

Another phenomenon that can prevent linear regression from capturing the relationship between the independent variables and the dependent variable is the presence of outliers in the training data—points that are clearly separated from the rest of the data. Not all outliers have a significant effect on the coefficients determined for the regression equation. Those that do are referred to as influential cases, and statisticians have developed remedial measures for dealing with such cases [Net96, Section 10.3]. One approach is to simply discard the outliers, as I have done in my work with Berkeley DB (Section 7.2.1). Other methods known as robust regression procedures have also been developed; these methods reduce the impact of outlying cases without removing them from the training data. Significantly, Neter et al. [Net96] point out that robust regression procedures are especially useful in situations in which the regression process must be automated, as is the case for an automated tuner.

Once the training data has been appropriately transformed by the necessary remedial measures, the tuner uses linear regression to learn the weights associated with the value nodes. These weights can be applied to the model in one of two ways. In the first approach, the tuner uses the weights to scale the value-node values in the training data before it estimates the expected-value parameters associated with the value nodes. In the second approach, the tuner first estimates the expected-value parameters from the unweighted training data and then uses the weights to scale the parameters. Because the weights do not depend on the discretization of the continuous nodes, the regression itself can be performed at any point after the training data has been gathered.

It is important to note that the regression equation does *not* need to accurately predict the value of the overall performance metric for a given instantiation of the value nodes. Rather, the regression process need only produce weights that accurately reflect the relative impacts of the value nodes on performance. The results presented in Chapter 7 demonstrate that regression techniques are able to learn weights that fulfill this criterion.

5.5 Using the Model

Once the influence diagram has been constructed and trained, it can be used by the tuner to recommend appropriate knob settings for each workload faced by the system, dynamically adjusting the knobs as the characteristics of the workload change over time. This section first explains how the model can be extended to handle situations in which one or more of the workload characteristics cannot be observed, and it then considers the process of using the model to recommend knob settings for a given workload.

5.5.1 Dealing with Unobservable Workload Characteristics

As outlined in Section 2.1, the tuner periodically receives a summary of the current state of the system from a separate monitor process. This summary should include the current knob settings and the values of whatever chance and value nodes that the monitor is able to measure. The observable chance nodes will typically include all or most of the workload characteristics (the root chance nodes). However, it may be difficult or even impossible to measure the values of some of these workload characteristics. This section explains how to deal with this problem.

The influence diagram for Berkeley DB (Figure 4.2, page 69) provides two examples of workload characteristics that cannot always be measured directly: *def_page_loc* and *def_leaf_loc* (Section 4.3.2). These variables describe the access locality of a workload by specifying the locality that *would be measured* if the default knob settings for the *page_size* and *min_keys* knobs were used.[12] Given the definitions of these variables, it is not always possible to determine their current values. However, it *is* possible to measure the values of two related variables that describe the workload's locality under the current knob settings: *page_loc*, which corresponds to *def_page_loc*; and *leaf_loc_rate*, which corresponds to *def_leaf_loc*. And the current values of these variables, along with the current settings for *page_size* and *min_keys*, provide information about the possible current values of *def_page_loc* and *def_leaf_loc*.

To reflect the information gained from the current values of *page_loc*, *leaf_loc_rate*, *page_size* and *min_keys*, we can extend the influence diagram for Berkeley DB, as shown in Figure 5.15. The extended diagram has new root chance nodes that represent the current values of these four variables—i.e., the values that hold before the next tuning decisions are made. To distinguish these nodes from the ones that represent the values of the variables after the next set of knob adjustments, the superscript ^{0} has been appended to the name of each variable. In addition, the new $page\_size^0$ and min_keys^0 nodes are chance nodes rather than decision nodes to reflect the fact that they are decisions that have already been made. Each of the unobservable workload characteristics has three of the new nodes as parents—the node that corresponds to the current value of its child, and the nodes representing the current values of *page_size* and *min_keys*.

More generally, if an influence diagram developed for software tuning has a root chance node, *N*, that represents an unobservable workload characteristic, and if there are other observed nodes in the model that provide information about the value of *N*, the influence diagram can be extended to reflect this fact. If *N*'s children are all observable, then the new influence diagram is derived from the old one by adding a set of root chance nodes that represent the current values of *N*'s children and of all observable parents of those children. Conditional arcs are drawn from each of the new nodes to *N*, and informational arcs can also be drawn from these nodes to the

12. As discussed in Section 4.3.2, defining *def_page_loc* and *def_leaf_loc* in this manner allows these variable to describe locality in a way that does not depend on the current knob settings.

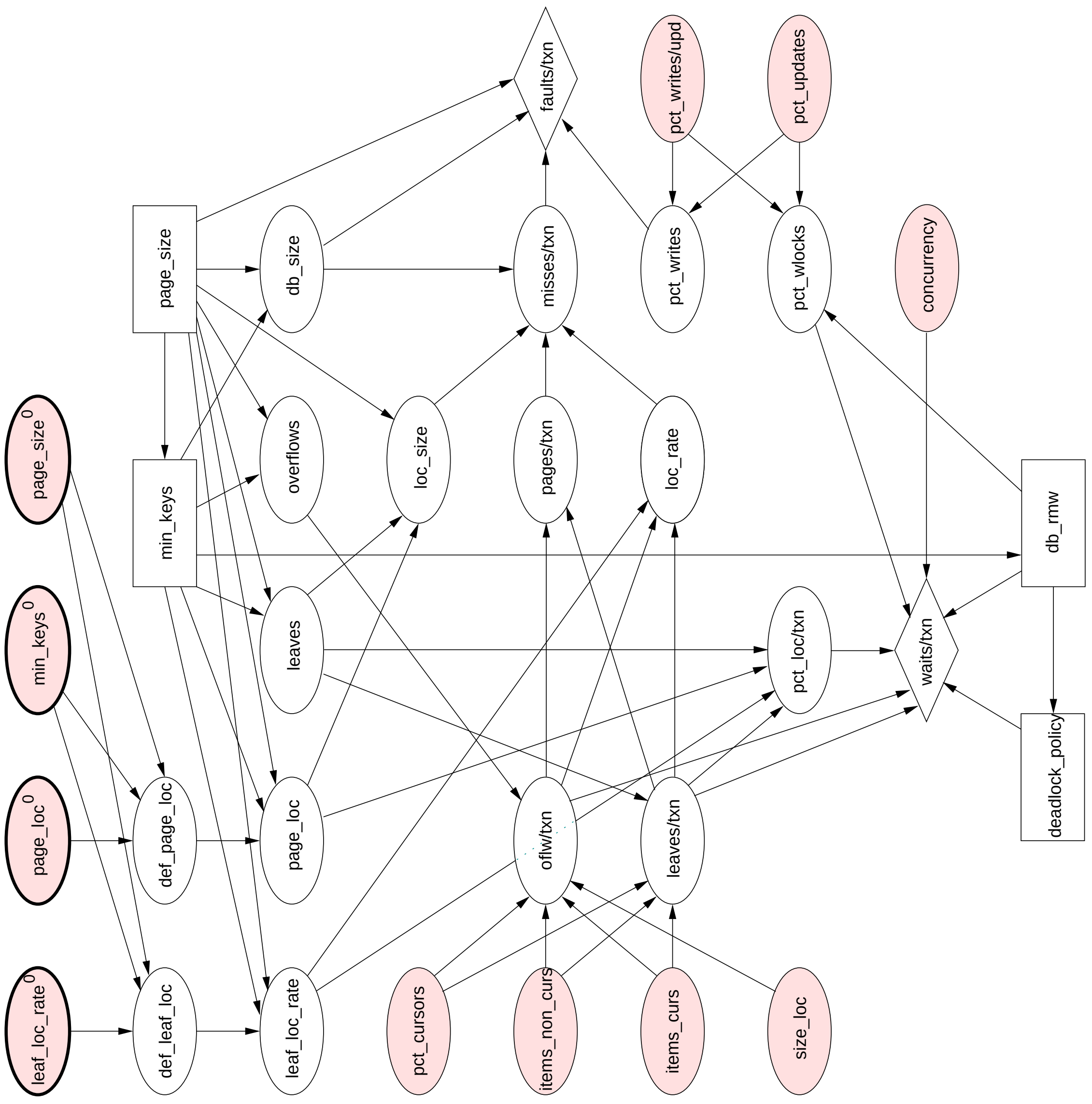


Figure 5.15. An extended influence diagram for Berkeley DB. The figure shows a version of the influence diagram for Berkeley DB that includes four additional nodes. The new nodes are found at the top of the diagram and are shown with bold outlines. These nodes represent the values of the *leaf_loc_rate*, *page_loc*, *min_keys*, and *page_size* nodes before the next set of tuning decisions are made. The values of these variables can be used to provide information about the current values of the *def_leaf_loc* and *def_page_loc* workload characteristics when they cannot be measured directly.

first decision node to indicate that their values are known before the next set of decisions is made. If one or more of *N*'s children are not observable, the process of extending the model is somewhat more complicated, but it still involves adding root chance nodes that represent the current values of all of the observable nodes that provide information about the current value of *N*.

The parameters of the new root chance nodes—as well as the parameters of the unobservable workload characteristics, given their new parents—can be derived from the training data using one of the methods described in Section 5.4.1. Although the workload characteristics in question cannot be measured consistently on a running system, it may be possible to measure their values during training, especially if the training workloads are produced by a workload generator that selects the values of the workload characteristics. If this is the case, one of the methods of learning parameters from complete data (Sections 5.4.1.1 and 5.4.1.2) can be used to estimate the parameters associated with these characteristics. If an unobservable workload characteristic cannot be measured during training, it may be possible to use an estimation method designed for incomplete data (Section 5.4.1.3).

5.5.2 Tuning the Knobs for a Given Workload

Given the observed characteristics of the current workload—and, if the influence diagram has been extended, the observed values of the additional root chance nodes (Section 5.5.1)—the tuner evaluates the influence diagram to determine the optimal knob settings for that workload (Section 3.2.4). Because the observed values are available as evidence, the evaluation algorithm does not need to determine the full optimal policies for the decision nodes; rather, it can simply determine the optimal decisions given the observed values.

In general, only the values of *root* chance nodes should be entered as evidence in influence diagrams designed for software tuning. The monitor may be able to provide the tuner with the current values of other variables in the model, but the nodes in the model represent the values of these variables after the tuning decisions have been made.

If the influence diagram recommends knob settings that differ from the settings currently in use, the tuner must decide whether the knobs should be retuned. If there is a cost associated with adjusting the knobs, the tuner should weigh the expected gain in performance from retuning—which can be estimated by using the influence diagram to determine the expected utility of both the current settings and the recommended ones—against an estimate of the cost of adjusting the knobs. If the tuner decides that the benefits of retuning outweigh the costs, it modifies the knobs accordingly.

5.6 Updating the Model Over Time

The monitor's measurements of the variables in the model can also be used to update the model over time, allowing the tuner to improve its predictions and to adapt to changes in the environment. Each set of measurements effectively constitutes a new training example. When the tuner decides to adjust the knobs, an additional training example is also be obtained for the combination of the current workload and the new knob settings.

To update the parameters of the model, the tuner needs to maintain the counts and value totals that appear in the equations in Section 5.4. In a simplistic approach, new training data would simply be used to increment these variables, and the probabilities and expected values would be recomputed accordingly. However, if aspects of the system or the environment change over time (e.g., if the size of a database file grows), this simple approach can prevent the tuner from adapting to the changes. To avoid this problem, a real-valued fading factor from the interval (0,1) can be used to reduce the impact of prior training examples over time [Jen01]. If *f* is the value of this factor, then for each count or value-node total *v* that is associated with the values in a new training example, we perform the following update:

$$v \leftarrow f \cdot v + 1 \qquad \textbf{(EQ 40)}$$

For example, if the monitor observed a value from the kth bin of chance node X_i in conjunction with the jth instantiation of X_i's parents, the count N_{ijk} of the number of times that these values are seen together would be updated as follows:

$$N_{ijk} \leftarrow f \cdot N_{ijk} + 1$$

In addition to updating the model's parameters, it may also make sense to periodically reconsider the discretizations of the continuous variables. Doing so requires access to the undiscretized training data, and it may thus be necessary for the tuner or monitor to log the undiscretized training examples—including the measurements gathered as the system runs. Given this data, the tuner could periodically run the discretization algorithm offline, allowing it to determine new discretizations from some number of the most recent training examples. These discretizations could then be used to determine new parameters for the model and new values for the counts and value totals maintained by the tuner.

To improve the performance of the model and to fully capture the impact of environmental changes, it may also be necessary to periodically gather additional training data. This can be done offline as described in Section 5.2. During this supplemental training, special focus could be given to workload characteristics that have been encountered most frequently, allowing the tuner to improve its performance on the most common workloads. In addition to this offline training, it may also make sense to occasionally try non-optimal knob settings—i.e., settings that the tuner does *not* recommend—on the system itself as it runs, because the resulting measurements may enable the tuner to make more accurate tuning recommendations overall.

5.7 Related Work

Little has been written about the process of designing good probabilistic, graphical models. Jensen presents an overview of this process in his introductory text on Bayesian networks and influence diagrams [Jen01], and he includes a number of useful examples and exercises. However, some of his advice—such as the suggestion that variables be classified as hypothesis variables and information variables—seems better suited to other problem domains, such as medical diagnosis. Another useful reference is the collection of papers edited by Oliver and Smith [Oli90], which includes a paper by Howard on the design and use of influence diagrams [How90] and example applications of influence diagrams by Agogino and Ramamurthi [Ago90] and Shachter et al. [Sha90].

A large body of research has been devoted to *learning* the structure of a probabilistic model from training data. For example, Friedman and Goldszmidt's algorithm for using the MDL principle to learn discretizations is part of a larger MDL-based approach that includes learning the structure of a Bayesian network [Fri96]. Heckerman's tutorial [Hec95] provides an overview of several methods for learning the structure of these models. And although Jensen claims that these techniques can rarely be applied to real-world problems [Jen01], it may be feasible to use them to improve upon a baseline model that has been constructed by a domain expert.

A number of methods have been developed for discretizing continous variables in the context of supervised machine learning. Dougherty et al. [Dou95] and Kohavi and Sahami [Koh96] together provide an overview of many of these methods. However, because these techniques were not designed for use in the context of probabilistic reasoning and decision-making, they are not able to capture the interactions between neighboring variables in an arbitrary Bayesian network or influence diagram. The method that comes closest to doing so is one developed by Fayyad and Irani [Fay93], who attempt to increase the mutual information between each of the variables being discretized and the class variable (i.e., the variable that the machine-learning algorithm is attempting to predict). Smith [Smi93] discusses the process of discretizing variables in the context of decision analysis, but the method that he presents—which involves constructing a discrete distribution that matches the moments (mean, variance, etc.) of the underlying continuous distribution—also does not take into account interactions between variables.

In addition to determining optimal knob settings, an influence diagram for software tuning can also be used to pose value-of-information questions. This would allow the tuner to determine which workload characteristics should be continuously measured and which can be left unobserved—or observed infrequently—after the initial training runs. Matheson [Mat90] explains how to use an influence diagram for this purpose.

To the best of my knowledge, the methodology presented in this chapter represents the first application of probabilistic reasoning techniques to the problem of software tuning. The closest related applications that I am aware of are those of Horvitz et al., who apply graphical, probabilistic models to a variety of applications that involve software systems, including software debugging [Bur95] and intelligent user interfaces [Hec98, Hor98]. Jameson et al. [Jam99, Jam00] have also used these models to design intelligent, adaptive user interfaces.

The FAST expert system developed by Irgon et al. [Irg88] also employs techniques from artificial intelligence to tune the performance of a complex software system. However, FAST uses frame-based knowledge representation [Rus95, Section 10.6] and heuristic problem-solving techniques rather than an approach based on probabilistic reasoning. In addition, FAST is focused primarily on the diagnosis and repair of performance problems (e.g., improving poor I/O performance by moving files from one disk to another), rather than on the problem of finding the optimal knob settings for a given workload.

5.8 Conclusions

The methodology presented in this chapter provides a step-by-step approach to using an influence diagram and related learning and inference techniques as the basis of an effective, automated software tuner. The methodology includes solutions to three challenges associated with using an influence diagram for tuning: incorporating performance measures in the model (Sections 5.1.1.2 and 5.4.4), dealing with continuous variables in the model (Section 5.3.3), and estimating parameters for which no training data is available (Section 5.4.3). Chapter 6 discusses the design of workload generators that can be used to gather the training data for an influence diagram offline, and Chapter 7 presents experiments that assess the ability of the methodology to tune an actual software system.

Chapter 6

Designing a Workload Generator for Use in Software Tuning

As discussed in Sections 2.4 and 5.2, it may often be preferable to use a workload generator to produce the training data needed by an automated software tuner. This chapter discusses some of the issues involved in creating an effective workload generator for software tuning, including the importance of ensuring that it captures the steady-state performance of the system. It also presents *db_perf*, a workload generator for Berkeley DB.

6.1 Architecture

A workload generator for software tuning needs to perform two distinct functions. First, it needs to produce workloads that simulate potential real-life workloads faced by the system being tuned. Second, it needs to measure the performance of the system as it responds to the generated workload. Depending on the nature of the system being tuned, these two functions may be performed by a single module or divided among two or more modules. For example, if the system employs a client-server architecture, it may be necessary to produce the workloads on one or more client machines and to measure the performance of the system on a separate server machine. If the system is embedded—like Berkeley DB—then the workload generator can simply be an application that links the system into its address space. In such cases, the same module can both generate the workloads and measure the relevant values. In the discussion that follows, I will simply refer to a single workload generator, without specifying whether its functions are concentrated in a single application or divided among multiple modules.

6.2 Criteria

To be effective, a workload generator for software tuning should meet several criteria. First, it should be capable of producing a wide range of workloads for the software system being tuned. Ideally, it should be dynamically configurable, so that an arbitrary workload can be specified at runtime. The actual workload specification can take different forms, such as a trace of the operations that constitute the workload or a configuration file that specifies the values of the relevant workload characteristics.

Second, the workload generator should ensure that it measures the steady-state performance of the system. When a system confronts a new workload, it often enters a transition period in which its performance gradually changes. For example, when a workload is generated for Berkeley DB, it can take time for the pages containing the most frequently accessed items to be brought into memory, and the performance of the system gradually changes as more and more accesses are satisfied without the overhead of going to disk. Once this transition period is complete, the system enters a steady state in which its performance remains at a relatively stable level over time.[1] The workload generator should wait for the system to stabilize before measuring its performance. Otherwise, the performance statistics that it produces may lead to non-optimal tuning recommendations. In the extreme case, the optimal knob settings may only be evident once a steady state has been reached, and a non-optimal combination of knob settings may appear to be optimal if measurements are made before the system stabilizes. Section 6.3 provides more detail about the process of determining a steady state.

Lastly, the workload generator should have reasonable time costs. Because a large number of training examples are typically needed, every effort should be made

1. The performance statistics may still undergo minor fluctuations during a steady state, but time averages of the statistics should be relatively stable.

to minimize the time taken to generate each example. However, there is a tension between the desire for short running times and the need to ensure that the system has stabilized. Moreover, some techniques for speeding up the process of gathering training data can actually prevent the workload generator from capturing the true steady-state performance of the system; Section 6.4.3 provides one example of this phenomenon. Care must be taken to ensure that steps taken to reduce the workload generator's time costs do not sacrifice the accuracy of its measurements.

6.3 Determining a Steady State

To obtain an accurate assessment of the impact of different knob settings on the system's performance, it is essential that the workload generator capture the system's steady-state performance. If the relevant variables are measured before a steady state is reached, the measurements may suggest that one combination of knob settings, s_1, outperforms another combination, s_2, on a particular workload, when in fact the steady-state performance of s_2 is equal to or better than that of s_1. Different techniques can be used to test for a steady state. The paragraphs that follow first discuss a type of technique that often fails to capture the steady-state performance, and they then present a more accurate approach that explicitly measures the variance of the relevant variables.

6.3.1 Focusing on the Requisite Conditions

One approach to determining a steady state is to focus on conditions that must be met before a steady state can be achieved. Typically, this involves associating threshold values with variables that are indicators of the requisite conditions. For example, in my work with Berkeley DB, I originally focused on the need to warm the memory pool in order to reach a steady state. I thus configured the workload generator to declare a steady state when either the rate of evictions from the memory pool rose above 0 (i.e., when the memory pool was filled), or when the memory-pool miss rate

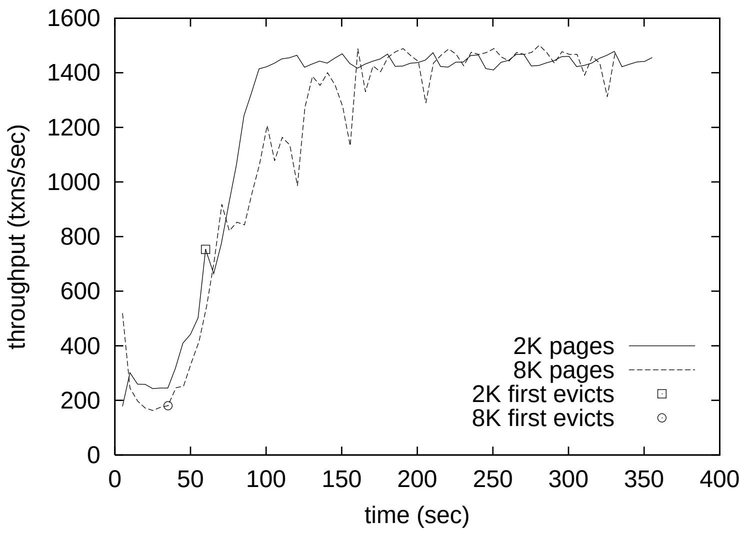


Figure 6.1. A faulty method of capturing steady-state performance. This graph traces the evolution of the throughput of a particular Berkeley DB workload on two different database configurations, one with 2K pages and one with 8K pages. The symbol plotted on each curve indicates the point at which the first memory-pool evictions occurred and a steady state was falsely declared. Because the 2K configuration is closer to a steady state when the first evictions occur, its measured throughput is higher, even though the steady-state throughputs of the two configurations are actually comparable.

dropped below a certain value. The problem with techniques of this type is that they do not explicitly test if the performance of the system has truly stabilized, and they can thus lead to faulty measurements.

One illustration of the inadequacy of focusing on requisite conditions is shown in the graph in Figure 6.1. Both curves in this graph trace the throughput of the same Berkeley DB workload as it evolves over time, starting from a state in which the memory pool is empty and none of the database is in the operating system's buffer cache. In one run, the system used 2K pages, and in the other it used 8K pages. The symbol plotted on each curve indicates the point at which the first memory-pool evictions occurred and a steady state was falsely declared; the statistics recorded by the workload generator were measured from that point to the end of the run. Although the two page-size settings produce comparable steady-state throughputs for this workload, the measured throughputs falsely suggest that 2K pages outperform 8K pages, because the 2K configuration is closer to a true steady state when the first evictions occur.

More generally, using evictions as the sole indicator of a steady state favors page sizes that are smaller than the filesystem's block size. Each memory-pool miss brings only a single new page into the memory pool, and thus the use of smaller pages tends to lengthen the time that it takes for the memory pool to fill and the first evictions to occur. However, reading a page from disk that is smaller than the block size still causes a full block to be read into the operating system's buffer cache; as a result, using a smaller page size does *not* typically increase the time needed to reach a true steady state. Thus, databases with smaller page sizes tend to be closer to a steady state when the first evictions occur, and measurements of throughput that begin with the first evictions tend to larger for these databases. This phenomenon is evident in Figure 6.1. In general, focusing solely on the conditions required for a steady state is often insufficient. Instead, the workload generator should explicitly measure the values of the relevant variables and wait for them to stabilize.

6.3.2 Testing for Stability Over Time

One method of testing for stability in the value of a variable is to compute the variance in its recent values; if the variance is below some threshold, the variable is considered stable. This type of stability test can be implemented by having the workload generator store a collection of n equally spaced samples from the past t seconds for each of the relevant variables. Conceptually, it can be helpful to picture a sliding, t-second measurement window for each variable, as shown in Figure 6.2. For a given variable s_i, the samples $\{ s_{i1}, s_{i2}, ..., s_{in} \}$ measure the value of s_i over each of the $\frac{t}{n}$-second subintervals in the window. As time advances, the measurement window slides forward: a new sample is measured and stored, and the oldest sample is discarded.

Given a collection of samples for a particular variable, the workload generator computes the variance of the samples and determines if the value of the variable has been sufficiently stable over the past t seconds. Once the variance in each collection of

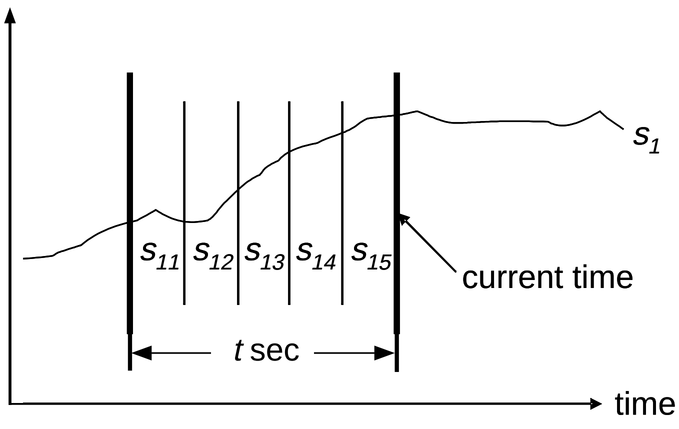


Figure 6.2. Maintaining a sliding measurement window. To test for a steady state, the workload generator can store a collection of n samples of each of the relevant variables over the past t seconds and compute the variance of each set of samples. The values collected form a sliding t-second measurement window, as visualized in the diagram above for a window that includes five samples of the variable s_1. As time advances, the window slides forward, adding a new sample and discarding the oldest one.

samples is sufficiently small, the workload generator stops the run and outputs the averages of the variables of interest over the current measurement window.

It is worth noting that the proposed method determines a steady state after the fact. In other words, rather than beginning to measure the necessary values after determining that a steady state has been reached—as is the case for methods that focus on requisite conditions—the proposed method constantly maintains the measurements needed to compute the values of the variables of interest, and it outputs the current version of those values when it determines that they reflect a steady state. This after-the-fact approach guarantees that the values in the training examples provide accurate measurements of the system's steady-state performance, and it can also reduce the time needed to collect the examples.

The workload generator need not test the stability of all of the variables that it measures, but the performance metric being optimized should typically be included. When implementing the stability test, the values of the following parameters must be specified: the size of the measurement window, t; the number of samples to maintain, n; and the variance threshold used to determine if the value of a variable has stabilized. The appropriate values of these parameters will depend in part on the nature of the system being tuned. For example, if the performance variables tend to experience frequent statistical fluctuations, the values of t and n should be such that the presence of these fluctuations will not prevent the workload generator from detecting a steady state. In addition, the value of t should be large enough to ensure that no further transitions in the performance of the system will occur, yet small enough to produce reasonable running times.

6.4 *db_perf:* A Workload Generator for Berkeley DB

To obtain training data for Berkeley DB, I developed *db_perf*, a workload generator that accepts detailed specifications of both the database and the workloads to be modeled, and that accurately captures the steady-state performance of the system on the resulting workloads. This section begins with an overview of *db_perf*, and it then provides more detail about various aspects of this tool.

6.4.1 Overview

db_perf is a dynamically configurable workload generator. It reads configuration files that are written using a simple yet powerful language that can be used to specify the relevant aspects of the database and workload, including such things as the types of access locality described in Section 4.3.2. *db_perf* supports the random selection of workload parameters at run time, and its configuration files can include variables, switch statements, and simple arithmetic expressions. Each configuration file is divided into four sections that specify: (1) the configuration of the Berkeley DB environment (including any initialization that should be performed) and the values of various miscellaneous parameters; (2) the contents of the database file or files to be used during the run; (3) the types of transactions to be performed; and (4) the number of threads that should access the database and the types of transactions to be performed by each thread.

To capture the steady-state performance of the system, *db_perf* uses a separate *stat-check thread* that tests the relevant variables for stability using the method described in Section 6.3.2. After the relevant variables have stabilized, *db_perf* outputs statistics that describe the performance of the system during the final measurement window. Many of these measurements come from the statistics that Berkeley DB maintains for each of its modules. In addition to testing for stability, the *stat-check thread* can also be configured to output a trace file that records the values of various variables at regular intervals. These trace files can be converted to graphs to confirm that a steady state has been reached and to gain insight into the performance of the system for a given workload.

6.4.2 Specifying the Database and Workload

Specifying a database in *db_perf* involves describing both the keys of the data items and one or more distributions over the sizes of the corresponding values. Keys must be specified and subsequently created in a way that allows the benchmark to generate successful queries. For example, although it might be desirable to use a scheme in which the mean and standard deviation of the key sizes could be specified, such a scheme would make it difficult to generate queries for keys that are known to be in the database. *db_perf* solves the problem of key specification by allowing users to create one or more *keygroups*, each of which consists of keys of the same length.

Users can define a keygroup by specifying either: (1) the number of distinct characters for each position in the key (e.g., “key_chars_per_slot 1 2 10 10 10” creates five-character keys that begin with the same first character and have two possible second characters, ten possible third characters, etc., for a total of $1 \cdot 2 \cdot 10 \cdot 10 \cdot 10 = 2000$ keys); or (b) the total number of keys. Users can also specify the first key in the keygroup, from which all of the remaining keys are derived. For example, if the first key is “aaaa,” subsequent keys would be “baaa,” “caaa,” and so on. Given the first key of a keygroup and n, the number of keys that the keygroup contains, a random key from the keygroup can be selected by choosing a value from 0 to $n - 1$ and converting it to the appropriate string through a process akin to converting the base of a number (Figure 6.3).

```
num_to_key(keygrp, num) {
   keystr = copy(keygrp.start_key);
   place = 0;
   place_val = 1;

   // Determine the largest place value with a non-zero value--i.e.,
   // the largest element of the key that will differ from
   // keygrp.start_key.
   while ((place_val * keygrp.chars_per_slot[place]) <= num) {
      place_val = place_val * keygrp.chars_per_slot[place];
      place++;
   }

   // Convert the number to the equivalent key.
   for (i = place; i < keygrp.key_size; i--) {
      keystr[i] = keygrp.start_key[i] + floor(num / place_val);
      num = num % place_val;
      if (i - 1 < keygrp.key_size)
         place_val /= keygrp.chars_per_slot[i - 1];
   }

   return keystr;
}
```

Figure 6.3. Pseudocode for converting a number to a key string. The algorithm takes a keygroup, `keygrp`, and a number, `num`, and converts the number to the corresponding key from the keygroup using a process similar to converting the base of a number. `keygrp.start_key` is an array of characters corresponding to the smallest key in the keygroup according to a lexicographic comparison function. `keygrp.chars_per_slot[n]` is the number of possible characters for the `n`th position in the key (i.e., the `n`th element of the array of characters). `keygrp.key_size` is the length of the keys in the keygroup. In this version of the function, the leftmost character of the key is the least significant digit of the corresponding number. A similar function can be written to produce keys in which the rightmost character is the least significant digit.

Keygroups can also be used to specify the types of access locality described in Section 4.3.2. To create a workload with a particular value for page locality, one or more keygroups can be designated as the locality set, and the majority of accesses can be directed to items in those keygroups. To create a workload with a particular value for size locality, one or more keygroups can be created with items that are large enough to be forced into overflow pages, and the appropriate percentage of accesses can be directed to those keygroups. Section 7.1.4 explains how keygroups were used to control access locality in the workloads generated for the experiments.

A *db_perf* configuration file also includes some number of user-defined transaction types and thread types. Each transaction type specifies a sequence of one or more operations, including such actions as reading, updating, or adding a collection of items. For each operation, probability distributions can be specified over the number of items to be accessed and the keygroups from which the items should be selected. Each thread type includes a number of threads to be created and a probability distribution over the types of transactions that each thread of that type should perform.

6.4.3 Using Initial Scans to Reduce Running Times

Because some workloads take a long time to reach a steady state, *db_perf* allows a user to specify preliminary scans of all or part of the database in an attempt to quickly warm Berkeley DB's memory pool and the operating system's buffer cache. These initial scans—which are not included in the measurements that *db_perf* reports—can significantly reduce the average time needed for a *db_perf* run. However, care must be taken to ensure that these initial scans do not prevent *db_perf* from capturing the true steady-state performance of the system. In fact, if initial scans are not configured properly, they can actually create a false steady state—i.e., an interval in which the statistics measured by *db_perf* are stable but do not accurately reflect the long-term behavior of the system for the specified workload.

One way that an initial scan can create a false steady state is to dirty the pages that it accesses at a rate that is substantially different than the rate at which the actual workload dirties them. For example, consider a workload that includes some percentage of update transactions (Section 4.3.1). If the initial scan for this workload dirties none of the pages that it brings into memory (i.e., if it only performs reads), none of the pages evicted at the start of the workload will be dirty.[2] However, once the system begins to evict the pages accessed by the actual workload, the dirty-evict rate will begin to rise and the throughout of the system will typically decrease. If *db_perf* declares a steady state before the dirty-evict rate begins to increase, the

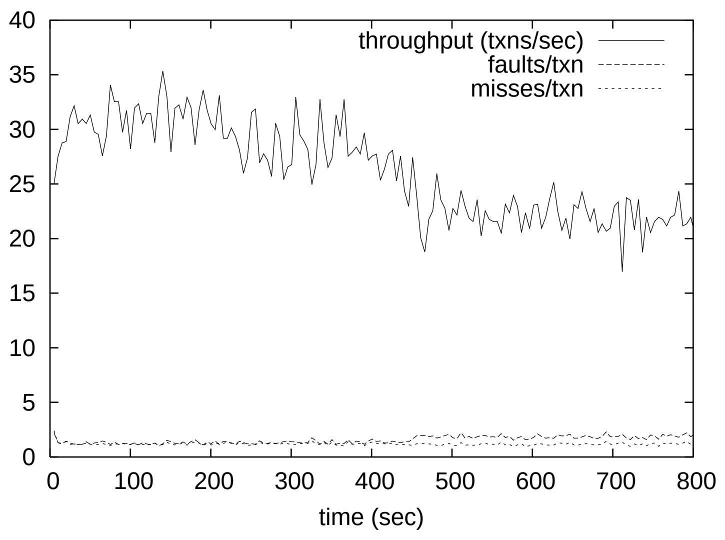
2. This results from the fact that the memory pool evicts pages using an approximation of the LRU algorithm (Section 4.1.2), and thus all pages evicted at the start of the cache will be pages accessed solely by the initial scan.

resulting measurements will be inaccurate. Therefore, *db_perf* allows the user to specify the rate at which items are modified by an initial scan.

Another, more subtle way that an initial scan can create a false steady state is to employ a substantially different access pattern than the one used by the actual workload. For example, I initially used a sequential scan of the database to warm the memory pool and buffer cache. However, an analysis of results like the ones displayed in Figure 6.4 led me to question the wisdom of this approach. The run portrayed in these graphs used a database with 2K pages, and the initial scan dirtied approximately 70% of the pages that it accessed. Because of the size of the database used and the amount of memory available for the buffer cache, pages evicted from the memory pool by this workload are often not present in the buffer cache. Therefore, when a dirty page is evicted, it should often be necessary to read the file block containing the page into the buffer cache so that the contents of the unmodified portions of the block can be preserved. As discussed in Section 4.1.2, reading a file block translates into a page fault on the operating system used in the experiments; therefore, there should often be two page faults per memory-pool miss for this workload—one to fault in the block containing the evicted page and one to fault in the block containing the page being read into the memory pool.

At the start of the run displayed in Figure 6.4, the expected ratio of faults to misses does not materialize: the value of *faults/txn* is only slightly higher than the value of *misses/txn*. Then, after a significant amount of time has elapsed, the *faults/ txn* value jumps to the expected level and throughput decreases accordingly. The false steady state at the start of the run is a result of the initial scan. Because the scan is sequential, the leaf pages evicted at the start of the run are also sequential. And because four 2K pages fit on each 8K file block, one page fault is able to satisfy as many as four dirty evicts. In other words, we see event sequences like the following:

evict dirty page 100, which faults in the block containing pages 100-103
evict dirty page 101; no page fault—the block is already in the buffer cache
evict clean page 102; no page fault—the block is already in the buffer cache
evict dirty page 103; no page fault—the block is already in the buffer cache
evict dirty page 104, which faults in the block containing pages 104-107 ...



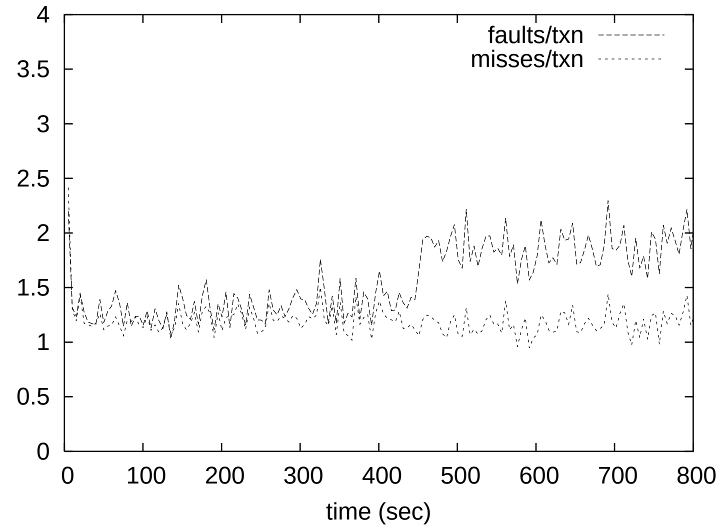


Figure 6.4. A false steady state. The above graphs trace three statistics—*throughput*, *faults/txn*, and *misses/txn*—for a Berkeley DB workload generated by *db_perf*. The top graph shows all three statistics. The bottom graph reduces the range of the vertical axis so that the evolution of *faults/txn* and *misses/txn* can be seen more clearly. Before the workload began, a sequential scan of the database was performed. This scan created a false steady state at the start of the run for reasons described in Section 6.4.3.

Once the system starts to evict pages that were accessed by the actual workload, this pattern no longer holds, and the value of *faults/txn* increases. However, before this can happen, *db_perf* is likely to conclude that the initial performance of the system represents a steady state. As a result, the statistics that it reports can end up falsely suggesting that a page size of 2K performs best for this workload.

To prevent the creation of a false steady state, I modified *db_perf* to support nonsequential initial scans. More specifically, these scans start at the *m*th item in a keygroup and access every *n*th subsequent item in that keygroup, where the values of *m* and *n* are specified in the definition of the scan. Through experimentation, I determined that the optimal values for *n* are ones that cause a scan to access every eighth leaf page. Skipping eight pages between accesses eliminates the phenomenon seen in the above example, because pages are no longer accessed sequentially. In addition, accessing every eighth page ensures that the file blocks read to satisfy the dirty evicts are also not accessed sequentially. Sequential file-block reads can cause the filesystem to read the blocks that come next in the sequence before they are requested by the application, and this can also result in a temporary decrease in the value of $faults/txn$.

The examples presented in this section illustrate the care that must be taken in devising methods to reduce the time needed to generate and measure a given workload. Although such methods may allow training data to be collected more efficiently, it is crucial that they do not prevent the workload generator from capturing the steady-state performance of the system.

6.5 Related Work

A number of prior research efforts on automated software tuning have employed one or more simulators to evaluate the performance of their proposed techniques [e.g., Bro95 and Mat97]. Simulators have also been used to obtain training data for software tuning [Fei99]. Many of these prior efforts have used simulators to simulate aspects of the software system itself, and occasionally even the hardware platform on which the system runs. By contrast, workload generators produce workloads that employ the actual software system being tuned and that run on a real hardware platform. Therefore, I have chosen to refer to them as workload *generators* rather than workload *simulators*.

The fact that a workload generator uses the actual software system makes capturing the system's steady-state performance that much more challenging. The frequent use of simulated systems by prior research efforts may explain why little attention has been given to the problem of ensuring that the measured statistics are stable. Kurt Brown raises this issue in his work [Bro95], but he proposes an approach that focuses on the criteria needed to reach a steady state (e.g., waiting until the buffer cache has filled or some number of database files or indices have been opened), rather than on the stability of the actual statistics. The example presented in Section 6.3.1 demonstrates that such an approach is often inadequate.

Workload generators are similar in spirit to macrobenchmarks, programs that run a workload on a system and measure its performance. Macrobenchmarks have been developed for a number of types of software systems, including database systems [Tra90] and Web servers [Sta00]. However, these benchmarks typically have a limited degree of configurability, and thus it is difficult to use them to obtain meaningful performance comparisons. Research in application-specific benchmarking has begun to address this limitation by developing benchmarks that combine a characterization of the performance of the underlying system with a characterization of the workload of interest [Bro97, Sel99b, Smi01]. Among the application-specific benchmarks that have been developed to date, HBench-Web [Man98] is most similar to *db_perf*. It generates stochastic workloads that simulate the workloads experienced by a particular Web server. HBench-Web derives the characteristics of the generated workloads from an analysis of Web server logs. A similar approach could be taken to derive the characteristics of at least some of the workloads that *db_perf* generates for a particular Berkeley DB application. However, to enable an automated software tuner to determine appropriate knob settings for previously unseen workloads, the workload characteristics derived from logs of previous usage would need to be supplemented by other, randomly selected workload characteristics.

6.6 Conclusions

A workload generator allows the training data needed by an automated software tuner to be collected offline, without disrupting the performance of the system to be tuned. This chapter outlined the criteria that a workload generator for software tuning should meet, and it presented *db_perf*, a workload generator for Berkeley DB. It also discussed possible techniques for measuring the steady-state performance of a software system, illustrating the limitations of one class of techniques and advocating an approach that explicitly tests the variance of the relevant statistics. The next chapter presents the results of experiments that employ data collected using *db_perf*, and it explains how *db_perf* was configured to generate the workloads used in these experiments.

Chapter 7

Evaluation

This chapter presents the results of experiments that evaluate various aspects of the proposed methodology for automated software tuning. In particular, the experiments assess the ability of an influence diagram to make accurate tuning recommendations for Berkeley DB after being trained using the algorithms for variable discretization and parameter estimation presented in Chapter 5. They also evaluate the impact of various aspects of these algorithms on the performance of the model, and they compare the influence diagram's tuning recommendations to ones based solely on regression.

7.1 Gathering the Data

The collection of data used in the experiments was gathered using *db_perf*, the workload generator presented in Chapter 6. The results of one set of *db_perf* runs were used for training, and the results of a second set of runs were used to assess the tuning recommendations of the models considered in the experiments. This section presents the relevant details of the procedure used to generate and collect the data.

7.1.1 Test Platform

db_perf was run on five 333-MHz Sun UltraSPARC IIi machines that used the Solaris 7 operating system (also known as SunOS 5.7) and that were configured more or less identically. Each machine had a single Seagate ST39140A Ultra ATA hard drive and 128 MB of RAM. Preliminary tests confirmed that Berkeley DB performed comparably on all five machines: running the same series of workloads on each of the machines produced comparable sets of performance statistics.

db_perf used the 4.0.14 release of Berkeley DB with a 64-MB memory pool (Section 4.1.2). The memory-mapped files used by Berkeley DB's subsystems (including the memory-pool file) were locked into memory to prevent them from being evicted by the operating system. To accommodate a large number of concurrent transactions, Berkeley DB was also configured to use a non-default value of 4000 for both the maximum number of locks and the maximum number of lock objects in the lock subsystem. A small number of workloads with extremely high levels of lock contention needed an even larger number of locks; these workloads were rerun with 8000 locks and 8000 lock objects. With the exception of the knobs being tuned and the settings mentioned above, all configurable aspects of Berkeley DB were run using the default values.

7.1.2 Test Database

The database used by *db_perf* consisted of 490,000 randomly-generated data items, all of which had small (six- or seven-byte) keys. One-seventh of the items had values with lengths drawn from a normal distribution with a mean of 450 and a standard deviation of 10 (the *large items*), and the remaining items had values with lengths drawn from a normal distribution with a mean of 100 and a standard deviation of 5 (the *small items*). The large items were uniformly distributed across the pages of the database—every seventh item was a large item. The method used to achieve this pattern of small and large items is described in Section 7.1.4.

The database was created once at the start of the experiments and the data items were stored in a flat file using DB's *db_dump* utility. Then, at the start of each run, the *db_load* utility loaded the data into a Btree using the values chosen for the *page_size* and *min_keys* knobs.

7.1.3 Knob Settings

As described in Sections 4.2.1 and 4.2.2, the *page_size* and *min_keys* knobs govern the physical layout of the database. The experiments considered two possible values for each of these knobs: *page_size* values of 2K and 8K, and *min_keys* values of 2 (for both 2K and 8K pages) and either 4 (for 2K pages) or 16 (for 8K pages). The database layouts that result from the (2K, 2) and (8K, 2) knob combinations fit all of the items on leaf pages. The (2K, 4) and (8K, 16) layouts force the large items into overflow pages, which may improve the performance of workloads that primarily access the small items. The experiments did not consider a (2K, 16) knob combination because it forces both the small and large items into overflow pages, and they did not consider an (8K, 4) knob combination because it leads to the same layout as the (8K, 2) combination. Therefore, *min_keys* values greater than 2 were grouped together in a single "4/16" option in the model, because otherwise the algorithm for evaluating the influence diagram would attempt to consider (2K, 16) and (8K, 4) knob combinations, for which there is no data.

As described in Sections 4.2.3 and 4.2.4, the *db_rmw* and *deadlock_policy* knobs are designed to reduce the degree of lock contention in the system. The experiments considered both of the possible values of *db_rmw*: 0, which indicates that the DB_RMW flag should *not* be used when reading an object that may later be written; and 1, which indicates that the DB_RMW flag should be used in such cases. For the *deadlock_policy* knob, which determines the policy used when selecting a transaction to resolve a deadlock, the experiments tried four different values: random (reject a random lock request), minlocks (reject the lock request made by the transaction holding the fewest locks), minwrite (reject the lock request made by the transaction holding the fewest write locks), and youngest (reject the lock request made by the transaction with the newest locker ID).

There are 32 possible combinations of the knob settings discussed above. A given combination of knob settings will be written as a tuple of values of the form

Table 7.1. Distributions of the workload characteristics used in the experiments. N(μ,σ) represents a normal distribution with mean μ and standard deviation σ. The numbers on the right side of each column represent the probability with which a given distribution was chosen. See Section 4.3 for more information about the workload characteristics.

concurrency		***pct_cursors***	
N(10,2)	1/3	0	2/3
N(40,5)	1/3	N(20,3)	1/3
N(100,10)	1/3	***items_non_curs***	
pct_updates		N(10,2)	1/3
N(30,5)	1/2	N(40,5)	2/3
N(75,5)	1/2	***items_curs***	
def_page_loc		N(50,5)	1/2
N(10,2)	1/4	N(100,10)	1/2
N(50,5)	1/4	***pct_writes/upd***	
N(90,1)	1/2	N(10,2)	1/3
size_loc		N(50,5)	1/3
N(3,1)	2/3	100	1/3
N(20,3)	1/3		

(*page_size*, *min_keys*, *db_rmw*, *deadlock_policy*). For example, the default settings used by Berkeley DB would be written as (8K, 2, 0, random).

7.1.4 Workload Characteristics

The workload characteristics were drawn from the distributions shown in Table 7.1, which were designed to generate a wide range of workloads. In particular, an effort was made to include workloads that benefit from non-default knob settings. For example, to ensure that the non-default settings of the *page_size*, *db_rmw*, and *deadlock_policy* knobs would be optimal for some of the generated workloads, I included distributions that tend to produce workloads with large amounts of lock contention (e.g., the N(100,10) distribution for *concurrency* and the N(75,5) distribution for *pct_updates*). After an initial set of distributions had been selected, refinements were made on the basis of preliminary experiments to ensure that the distributions produced, for each of the four knobs being tuned, examples of workloads that benefit from non-default settings of that knob. Constructing workload distributions in this way ensured that the generated workloads would provide a good test of the model's ability to make accurate tuning recommendations.

To prepare for a given *db_perf* run, a driver script selected a distribution for each workload variable from among the possible distributions for that variable and encoded these distributions in a *db_perf* configuration file. *db_perf* then chose the values of the workload variables according to the distributions. The value of the *def_leaf_loc* workload characteristic (Section 4.3.2) was fixed at 90% for the experiments; different degrees of page locality were considered by varying the value of the *def_page_loc* variable. In addition, because the small and large items were evenly distributed throughout the database, the *leaf_loc_rate* variable also had a fixed value of 90%. Therefore, the *def_leaf_loc* and *leaf_loc_rate* nodes were not able to affect the recommendations of the model for these workloads, and they were removed from the model for the purposes of the experiments.

db_perf was configured to run up to four different types of transactions in each workload: (1) non-sequential read-only transactions, which read a non-sequential collection of items; (2) sequential read-only transactions, which use a cursor to read a sequential collection of items; (3) non-sequential update transactions, which read a non-sequential collection of items and modify some percentage of them; and (4) sequential update transactions, which use a cursor to read a sequential collection of items and modify some percentage of them. For a given workload, *db_perf* determined the percentage of each type of transaction from the values of the *pct_cursors* and *pct_updates* workload characteristics. It determined the number of items accessed per transaction from the *items_curs* and *items_non_curs* workload characteristics, which specify the means of normal distributions over the numbers of items accessed by sequential and non-sequential transactions, respectively. The standard deviations of these distributions were fixed at 20% of the mean. When executing an update transaction, *db_perf* modified items with a frequency specified by the *pct_writes/upd* workload variable. More information about all of the transaction characteristics mentioned above can be found in Section 4.3.1.

The locality of accesses to the database was controlled using *db_perf*'s keygroup abstraction (Section 6.4.2). For each workload, the items were divided into

Table 7.2. Using keygroups to control locality in the generated workloads. *db_perf*'s keygroup abstraction was used to control access locality in the workloads generated for the experiments. Shown below are regular expressions that provide a partial specification of the keygroups for a workload in which the locality set consists of 10% of the leaf pages. The leaf pages in the locality set contain keys whose second character is a letter from the set `[A-E]`—10% of the 50 possible characters for that position in the key—and they thus constitute approximately 10% of the leaf pages. See Figure 7.1 for a explanation of how keygroups were also used to produce the desired interleaving of large and small items on a page. The actual method used to specify the keygroups differed in minor ways from the one shown here.

Keygroup	Regular expression for its keys
small items on pages in a 10% locality set	`?[A-E]???[a-f]`
large items on pages in a 10% locality set	`?[A-E]???aa`
small items on pages not in the locality set	`?[F-r]???[a-f]`
large items on pages not in the locality set	`?[F-r]???aa`

four keygroups: (1) small items from pages in the locality set (Section 4.3.2), (2) large items from pages in the locality set, (3) small items from pages not in the locality set, and (4) large items from pages not in the locality set. The exact division of the items was governed by the value of the *def_page_loc* variable, which specifies the percentage of leaf pages in the locality set. Table 7.2 illustrates how the keygroups were defined for a given value of this variable. Because the small and large items were evenly distributed throughout the database, assigning a given percentage of both the small and large items to the first two keygroups is equivalent to selecting that same percentage of leaf pages for the locality set. 90% of the accesses were then directed to the items in those keygroups.

When specifying which items would be included in the locality set, the *second* character of the keys was used as the distinguishing character, as shown in the example in Table 7.2. Using the second character rather than the first spreads the locality-set leaf pages throughout the database, which reduces the degree of contention for the internal pages of the Btree. When there is too much contention for internal pages, it becomes difficult to find workloads that are positively affected by the non-default setting of the *db_rmw* knob—which is used to prevent deadlocks due to lock upgrades (Section 4.2.3). This difficulty stems from the fact that lock upgrades only occur on locks for leaf pages, and lock conflicts on the internal pages reduce the

```
key: gbncef   value: gfrwoipwla... (length = 101)
key: gbncfa   value: okuyqhajsm... (length = 104)
key: gbncfaa  value: dnujqertsafdgzkklwlhdgwgss... (length = 453)
key: gbncfb   value: uitrfgeamb... (length = 97)
key: gbncfc   value: lorewqmfva... (length = 100)
key: gbncfd   value: qznhyfasjt... (length = 107)
key: gbncfe   value: pyteqwngha... (length = 95)
key: gbncff   value: rndwzkcbgw... (length = 102)
key: gbncga   value: ouwbnfdsqa... (length = 99)
key: gbncgaa  value: ytewjsmaskhwypowkawjadxwls... (length = 445)
...
```

Figure 7.1. Interleaving small and large items on a page. The keygroups were specified so that every seventh item is a large item, as shown in the above example of items from a particular leaf page. The small items have keys that end in one of the six characters `[a-f]`, and the large items have keys that are one character longer and always end with the substring `aa`. This produces the desired interleaving of large and small items.

probability that two threads will both attempt to upgrade a lock for the same leaf page.

The keygroups were also used to provide the desired interleaving of small and large items in the database; Figure 7.1 provides an example of the items that might appear on a given leaf page. The value of the *size_loc* workload variable governed the frequency with which the keygroups containing the large items were accessed. Given that 90% of the accesses went to pages in the locality set, the four keygroups were accessed with the following frequencies, where *s* is the value of *size_loc*:

small items from locality-set pages:	90(100 - *s*)/100 %
large items from locality-set pages:	90*s*/100 %
small items from non-locality-set pages:	10(100 - *s*)/100 %
large items from non-locality-set pages:	10*s*/100 %

Because the *db_perf* runs were conducted on a uniprocessor, transactions were also configured to yield the processor after every operation that reads an item.[1] Doing so increases the likelihood that a given page will already be locked when a thread attempts to lock it. Similar levels of contention would be possible on multiprocessor machines, which enable multiple threads to be active at the same time.

1. More specifically, the transactions yielded the processor after every call to the `DB->get()` and `DBcursor->get()` methods from the Berkeley DB C API.

7.1.5 Running the Workloads

Two sets of *db_perf* runs were conducted. The *training runs* generated the data used to train the models considered in the experiments, and the *test runs* generated the data used to assess the accuracy of the models' tuning recommendations. For each of the randomly selected workloads in the training runs, *db_perf* was run using a single, randomly selected combination of knob settings. For each of the randomly selected workloads in the test runs, *db_perf* was run 32 times, once for each possible combination of knob settings.

In both sets of runs, *db_perf* generated each workload until the values of the following variables had stabilized: (1) *throughput*, the overall performance metric that the tuner is attempting to optimize; (2) *misses/txn* and (3) *faults/txn*, two I/O-related variables from the influence diagram that often take a long time to stabilize; and (4) *dirty_evicts/txn*, a variable that measures the number of dirty pages evicted from the memory pool per transaction. Although *dirty_evicts/txn* is not explicitly included in the influence diagram, its value must stabilize before the values of *faults/txn* and *throughput* can be fully stable. To test the stability of these variables, *db_perf* used a five-minute measurement window (Section 6.3.2) with five samples per variable, and it considered a variable to be stable if the standard deviation of its samples was either less than 0.01 in absolute magnitude or less than five percent of the mean.

In an attempt to pre-warm the memory pool and buffer cache, *db_perf* also performed a series of initial scans of portions of the database before the start of each workload (Section 6.4.3). Each of these initial scans read a set of items and modified some percentage of them. The probability of modifying an item i during an initial scan was computed as follows:

$$P(i) = \begin{cases} u \cdot w \text{ if } i \text{ is on an overflow page} \\ 1 - (1 - u \cdot w)^2 \text{ if } i \text{ is on a leaf page} \end{cases} \qquad \text{(EQ 41)}$$

where u is the value of the *pct_updates* workload characteristic expressed as a decimal, and w is the value of the *pct_writes_upd* workload characteristic expressed as a decimal. These probabilities were designed to produce an initial dirty-evict rate that would be comparable to the rate seen once a steady state is reached, as discussed in Section 6.4.3. They are based on two assumptions: (1) that an overflow page will typically be accessed once before it is evicted (and thus the probability of modifying an overflow-page item during an initial scan is the same as the probability that the workload will modify an item); and (2) that a leaf page will typically be accessed twice before it is evicted (and thus the probability of modifying a leaf-page item during an initial scan is the same as the probability that a leaf page will be dirty after being accessed twice by the workload). Although these assumptions may not always hold, the initial dirty-evict rate should typically be fairly close to the steady-state dirty-evict rate, and which should allow a steady state to be reached more quickly.

Scans involving leaf pages were configured to access one item from every eighth leaf page to avoid the potential effects of sequential scans on performance discussed in Section 6.4.3, and the collection of pages accessed was configured to take up approximately 128 MB—the size of the test machines' physical memories. A bug in the script used to drive the runs in the test set caused the initial scans for test runs that used (*page_size*, *min_keys*) combinations of (2K, 4), (8K, 2) and (8K, 16) to skip fewer than eight pages between accesses. As a result, these scans read some pages multiple times and fewer pages overall; this means that smaller portions of the database were read into memory and that the accessed pages were dirtied at a higher rate. Although this bug lengthened the time needed to reach a steady state in some of the test runs, an analysis of a random sample of these runs showed that the bug did not prevent *db_perf* from accurately capturing the steady-state performance of the system.

db_perf executed 18717 training runs and 14176 test runs (443 test workloads, with 32 runs per workload). Of the 443 test workloads, 92 of them were used to refine the model (the *validation set*), and the remaining 351 workloads (the

Table 7.3. Summary of the variables included in the data files. This table explains how the values of the variables were obtained and it lists the precision used to represent them. Most values were rounded to the nearest integer; this is indicated by a precision value of 0. If more precision was needed, two places after the decimal were retained; this is indicated by a precision value of 2. More information about the individual variables can be found in Chapter 4. See Section 7.1.4 for information about the distributions of the variables that were chosen at random by the driver script.

Variable	How obtained	Precision
concurrency	chosen at random by the driver script	0
deadlock_policy	chosen at random by the driver script	N/A
def_page_loc	chosen at random by the driver script	0
db_rmw	chosen at random by the driver script	0
db_size	size of file (in MB) measured by the filesystem	0
faults/txn	measured by *db_perf* using the *getrusage* system call	2
items_curs	chosen at random by the driver script	0
items_non_curs	chosen at random by the driver script	0
leaves	counted the number of headers for leaf pages in a file of page headers produced using Berkeley DB's *db_dump* utility	0
leaves/txn	instrumented Berkeley DB to count the number of leaf pages locked by transactions at the point at which they are committed	0
loc_rate	computed using equation 17 on page 75, with *leaf_loc_rate* = 90	0
loc_size	computed using equation 16 on page 74	0
min_keys	chosen at random by the driver script	0
misses/txn	measured by *db_perf* from Berkeley DB's mpool statistics	2
overflows	counted the number of headers for overflow pages in a file of page headers produced using Berkeley DB's *db_dump* utility	0
oflw/txn	estimated as follows: $oflow/txn = \frac{n}{70000}\left(\frac{c \cdot \gamma}{7} + (1-c) \cdot i \cdot s\right)$ where *n* is the value of *overflows*, *c* is *pct_cursors* expressed as a decimal, γ is the value of *items_curs*, *i* is the value of *items_non_curs*, and *s* is the value of *size_loc* expressed as a decimal. *n* / 70000 is the percentage of large items in overflow pages, and the quantity in parentheses estimates the number of large items accessed per transaction.	2
page_loc	equal to *def_page_loc* for the test database	0
page_size	chosen at random by the driver script	0
pages/txn	*pages/txn* = *leaves/txn* + *oflw/txn*	0
pct_cursors	chosen at random by the driver script	0
pct_loc/txn	estimated as follows: $pct\_loc/txn = \frac{leaves/txn}{\frac{page\_loc}{100} \cdot leaves}$	2
pct_wlocks	if *db_rmw* = 0, *pct_wlocks* = *pct_writes*; if *db_rmw* = 1, *pct_wlocks = pct_updates.*	0
pct_writes	estimated as follows: $pct\_writes = \frac{pct\_updates \cdot pct\_writes/upd}{100}$	0
pct_writes/upd	chosen at random by the driver script	0
pct_updates	chosen at random by the driver script	0
size_loc	chosen at random by the driver script	0
throughput	measured by *db_perf*	2
waits/txn	measured by *db_perf* from Berkeley DB's lock statistics	2

Table 7.4. Optimal knob settings for the test-set workloads. For each combination of knob settings, the number shown is the number of test-set workloads for which those settings produced the maximal measured throughput. There are 351 test-set workloads in total.

db_rmw	*deadlock policy*	(*page_size*, *min_keys*)			
		(2K,2)	**(2K,4)**	**(8K,2)**	**(8K,16)**
0	**random**	10	2	29	11
0	**minlocks**	17	4	17	25
0	**minwrite**	11	2	33	19
0	**youngest**	6	3	13	22
1	**random**	0	0	15	3
1	**minlocks**	4	0	15	4
1	**minwrite**	19	0	30	18
1	**youngest**	5	0	11	3

test set) were used to assess the performance of the final model and the other models considered in the experiments. Perl scripts were used to process the output of the *db_perf* runs and to create a data file for each set of runs. Table 7.3 summarizes the variables included in the data files and the methods used to measure or estimate them.

The training runs executed for an average of just over 14 minutes per run; gathering the entire training set took five machines an average of 37 days per machine. Spending this much time collecting training data is clearly undesirable. However, Section 7.5.1 will show that only a fraction of the training data is needed to achieve a reasonable level of performance on the test-set workloads.

The test set includes a wide range of different workloads. As shown in Table 7.4, many of these workloads benefit from non-default knob settings. 23.6% of them did best with 2K pages, 33% did best with *min_keys* values greater than 2, 36.2% did best when the DB_RMW flag was used in conjunction with update transactions, and 80% did best with a deadlock policy other than the default, random policy. Almost all of the 32 possible combinations of knob settings were optimal for at least one of the these workloads. In addition to the optimal knob settings for a given workload, there are often a handful of near-optimal settings—settings that produce throughputs that are within 5% of the optimal throughput for that workload.^{2}

7.2 Creating the Models

A number of different influence diagrams were created in the course of preparing for and conducting the experiments. To determine the final model—the one used to assess the overall effectiveness of the methodology—a number of candidate models were compared using the data in the validation set; see Section 7.3 below for more detail. Once the final model was selected and evaluated, subsequent experiments were conducted to test the impact of various aspects of the training procedures; these experiments required that additional models be constructed with the same structure as the final model but with different parameters. This section describes the procedure used to create each of the models.

7.2.1 Learning the Weights of the Value Nodes

The weights of the value nodes were determined using linear regression. As discussed in Section 5.4.4, analyzing a subset of the training data led me to use the natural logarithm of throughput as the dependent variable rather than throughput itself. The value-node variables (*faults/txn* and *waits/txn*) served as the independent variables.

Further analysis of the training data revealed that there were outliers with extremely high values for the *waits/txn* value node. As shown in Figure 7.2, the vast majority (98.5%) of the training instances have *waits/txn* values of less than 100, but there are a small number of instances with values that are much larger. Most of these outliers used the random setting of the *deadlock_policy* knob, and I suspect that the high *waits/txn* values stem from situations in which transactions became trapped in cycles of repeated deadlocks. Because the random deadlock-resolution policy does not distinguish between transactions on the basis of their age or the number of locks that they hold, it is possible for a group of conflicting transactions to remain deadlocked

2. For the 351 test-set workloads, the average number of near-optimal knob settings for a given workload is 3.95. This average includes 23 workloads for which there are no near-optimal settings.

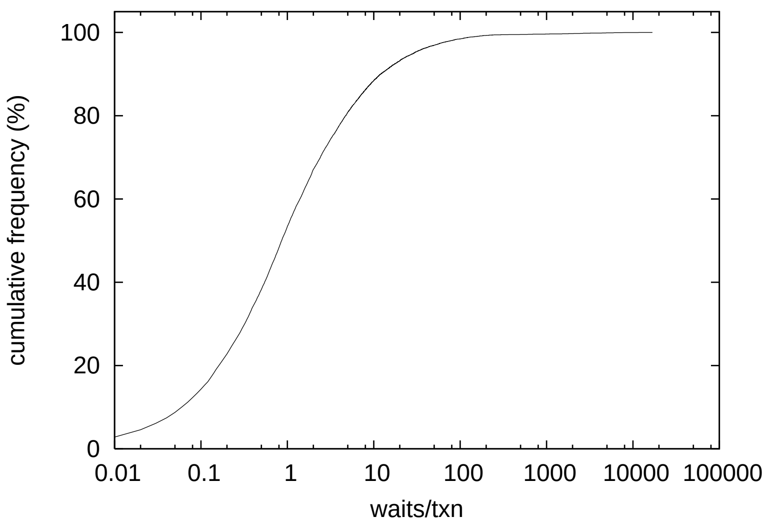


Figure 7.2. Outliers in the training data. This graph shows the cumulative distribution of the *waits/txn* variable in the training data. The *x*-axis has a log scale. The vast majority of the training instances (98.5%) have *waits/txn* values of less than 100, but there are a small number of outliers with much higher values.

for an extended period of time as they effectively take turns getting aborted. This type of scenario produces high *waits/txn* values as transactions repeatedly redo operations and attempt to reacquire the necessary locks. However, the same scenario does not affect the value of *faults/txn*, because the pages needed to redo an operation are typically in the memory pool from the last time that the operation was performed.

When regression is performed on the full training set (including the outliers), the following equation is produced:

$$\ln t = 3.56749 - 0.00084w - 0.15148f \qquad \text{(EQ 42)}$$

where t is *throughput*, w is *waits/txn*, and f is *faults/txn*. The regression produces an extremely small coefficient for *waits/txn* in an attempt to accommodate the large *waits/txn* values of the outliers, and the equation thus underestimates the impact of *waits/txn* on throughput. To obtain a more realistic coefficient, I eliminated the training examples with *waits/txn* values of 100 or more before performing the regression. Doing so produced the following equation:

$$\ln t = 3.74601 - 0.02778w - 0.15851f \qquad \text{(EQ 43)}$$

The coefficients of w and f in this equation were used as the final weights of the value nodes: the *waits/txn* values in the training data were multiplied by -0.02778, and the *faults/txn* values were multiplied by -0.15851.

Several of the candidate models included an additional value node representing the number of bytes written to the database log per transaction. The same procedure was used to learn the weights of the value nodes in these models, including the use of the natural logarithm of throughput as the dependent variable and the exclusion of outliers with *waits/txn* values of 100 or more. The following weights were produced: -0.00003206 for the number of bytes written to the log per transaction, -0.01609 for *waits/txn*, and -0.14888 for *faults/txn*.

7.2.2 Discretizing the Continuous Variables

The algorithm presented in Section 5.3.3 was used to learn the discretizations of the continuous variables in the models. I wrote a Perl script, *learn_disc*, that operates on three files: a *model-structure file* that specifies the arcs in the model; an *initial discretization file* that provides an initial cutoff-value sequence for each node in the model and indicates which nodes should be discretized; and a file containing the training data. The script runs the algorithm on the data and produces a *final discretization file* that contains the learned cutoff-value sequences.

learn_disc could be made to determine the order in which the variables are considered by the algorithm, as described in Section 5.3.3.3, but the current version requires that a (possibly partial) ordering be specified in the initial discretization file. In all of the specified orderings, nodes were sorted according to their distance from the value nodes, as required by the algorithm. In particular, the following ordering was used to discretize the variables in the final model: *pct_loc/txn*, *misses/txn*, *leaves/txn*, *pct_writes*, *oflw/txn*, *pct_wlocks*, *concurrency*, *page_loc*, *loc_rate*, *loc_size*, *pages/txn*, *pct_updates*, *pct_writes/upd*, *items_non_curs*, *items_curs*, *size_loc*, *pct_cursors*, and *def_page_loc*.[3]

3. The *leaves*, *overflows*, and *db_size* variables all had a small enough number of values that they did not need to be discretized.

Because the algorithm considers all possible cutoffs for a node on each iteration, the running time of the algorithm depends on the number of unique values that appear in the training data for each variable. To reduce the number of values, I used a version of the training data that uses less precision to represent non-integral values. More specifically, the values of variables that ordinarily have a precision of two places after the decimal (see Table 7.3) were rounded to one place after the decimal for the purpose of discretization. The learned discretizations are coarse enough that reducing the precision in this way should not significantly affect the quality of the resulting cutoff-value sequences.

7.2.3 Constructing and Manipulating the Models Using the Netica Toolkit

The Netica toolkit (version 2.15 for Solaris) [Nor03] was used to construct and manipulate the influence diagrams used in the experiments. I wrote a C program, *build_id*, that uses the Netica API to construct an influence diagram from the information contained in three files: the model-structure file described in the previous section; the final discretization file produced by *learn_disc*; and the file containing the training data. The program learns the parameters of the model and stores the influence diagram in a file using Netica's DNET format.

By default, Netica computes *m*-estimates (Section 5.4.1.2) for the probability parameters in the model; it uses uniform prior distributions and equivalent sample sizes equal to the number of possible values of the variable.[4] Preliminary experiments indicated that the prior distributions were having too much influence on the estimated parameters of certain chance nodes in the model—in particular, on nodes with a large number of possible values. To avoid this problem, I set the value of Netica's `BaseExperience_bn` variable to 1/2, which is equivalent to setting the equivalent sample size for all of the priors to 0. Doing so causes Netica to perform maximum likelihood estimation (Section 5.4.1.1).

4. The reference manual for the Netica API states that Netica uses an equivalent sample size equal to the value of the `BaseExperience_bn` variable, which by default is 1 ([Nor97], page 40). In practice, however, Netica uses an equivalent sample size equal to the number of possible values of the variable.

build_id computes the expected-value parameters associated with the value nodes using simple averages (Section 5.4.2.1), and it uses the method outlined in Section 5.4.3 to estimate the unseen parameters associated with both the chance and value nodes. Variants of *build_id* that use alternate methods of estimating unseen parameters were also used to assess the impact of these methods on the accuracy of the models.

I also used the Netica API to create two programs that manipulate the influence diagrams created using *build_id*. The first program, *best_settings*, uses an influence diagram to recommend knob settings for one or more workloads. More specifically, it reads a combination of workload characteristics (i.e., the values of the root chance nodes) from the keyboard or a file, enters those characteristics as evidence, and evaluates the influence diagram in light of those characteristics. The second program, *enter_evidence*, allows the user to enter values for both the workload characteristics and knob settings, and it outputs the posterior probabilities of the intermediate chance nodes and the conditional expected values of the value nodes given the specified evidence. This program can be used to determine why a given model ends up recommending a particular combination of knob settings for a given workload.

7.3 Determining the Final Model

To select the model used to assess the overall effectiveness of the methodology, a number of candidate models were compared using the process of model refinement described in Section 5.1.4. The candidate models differed in either their structure, their parameters (i.e., in the process used for training), or both. Figure 7.3 shows two of the alternate model structures that were considered. Other model variants were produced by changing the number of initial bins given to the value nodes for the purpose of discretization, and by considering alternate methods of discretizing the continuous variables and estimating the unseen parameters.

The candidate models were compared on the basis of the knob settings that they recommended for the workloads in the validation set. Each model was assessed using the metrics described in Section 2.2: *accuracy*, the percentage of workloads for which the model recommends optimal or near-optimal knob settings; and *average slowdown*, the average percent difference between the performance achieved using the model's non-optimal knob recommendations and the maximal measured performance for the corresponding workloads. The final model performed best on both of these metrics. It used the basic model structure presented in Chapter 4, but the *def_leaf_loc* and *leaf_loc_rate* nodes were removed from the model for the reasons discussed in Section 7.1.4. The resulting model structure is presented in Figure 7.4.[5] The final model was trained using the process outlined in Section 7.2. It also used the discretizations shown in Table 7.5, which were determined by *learn_disc* from the training data using initial discretizations that gave ten equal-height bins to the value nodes.

It is worth noting that there is minimal overlap between the validation and test sets. None of the workloads in the test set have combinations of undiscretized workload characteristics that also appear in the validation set. Even when the workloads' *discretized* characteristics are used as the basis of comparison, the vast majority of the test-set workloads (313 of the 351 workloads, or 89%) do not appear in the validation set, and the remaining 11% of the workloads were not singled out in any way when determining the final model.

7.4 Evaluating the Final Model

7.4.1 Performance of the Recommended Knob Settings

The final model recommends optimal or near-optimal knob settings for 270 of the 351 test workloads (76.9%); 90 of the recommended settings are optimal, and 180 produce non-optimal throughputs that are within 5% of the maximal measured throughput

5. It was not necessary to extend the model as discussed in Section 5.5.1 because the test-set data included the values of all of the characteristics of the workload.

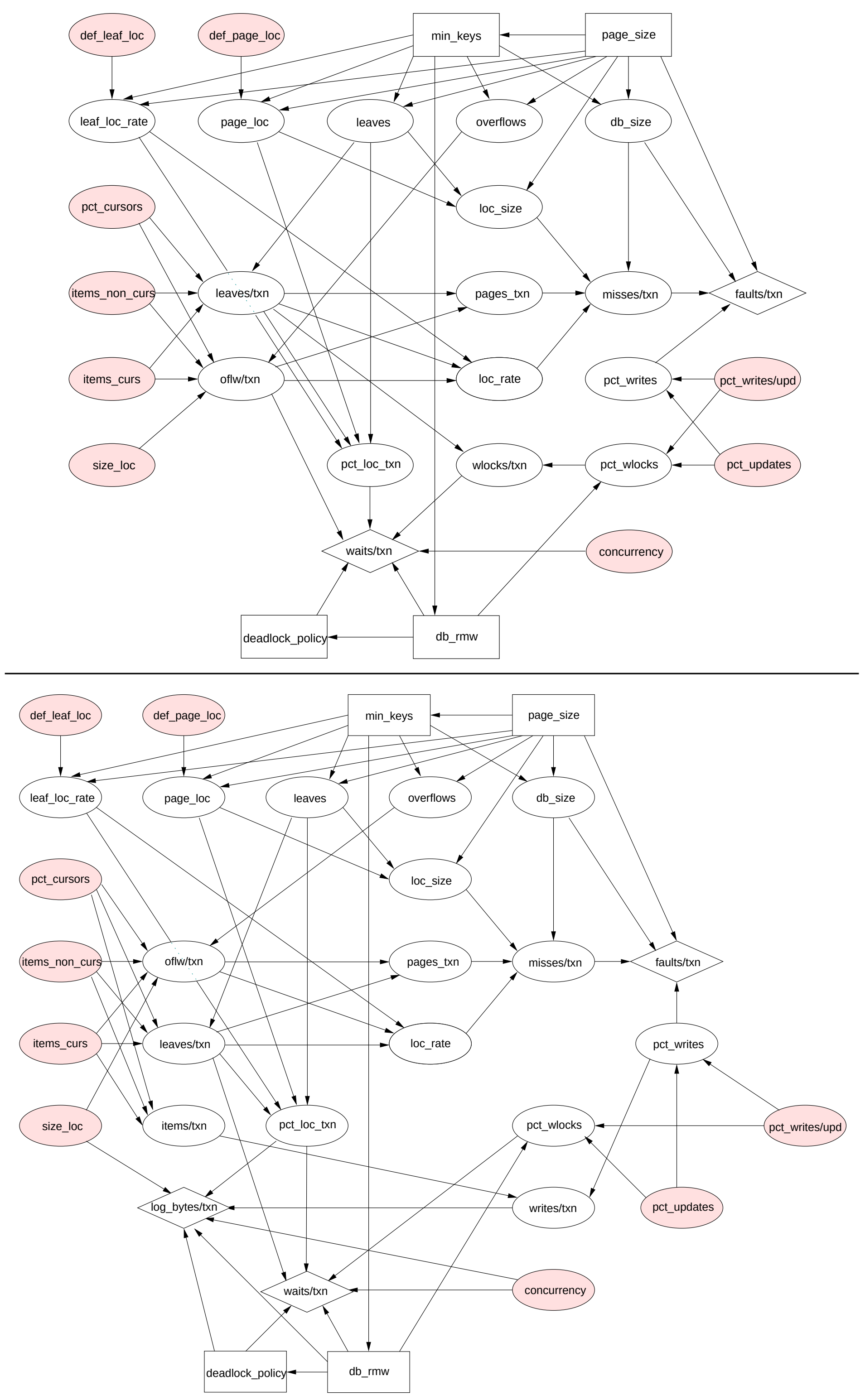


Figure 7.3. Other candidate model structures. The top model adds an intermediate chance node (*wlocks/txn*) to the final model (Figure 7.4) to reduce the number of parents of *waits/txn*. The bottom model adds an additional value node (*log_bytes/txn*) and two associated chance nodes (*items/txn* and *writes/txn*) in an attempt to better capture one impact of the *deadlock_policy* and *db_rmw* knobs on performance, and it removes the arc from *oflw/txn* to *waits/txn*. These models did not perform as well as the final model on the workloads in the validation set.

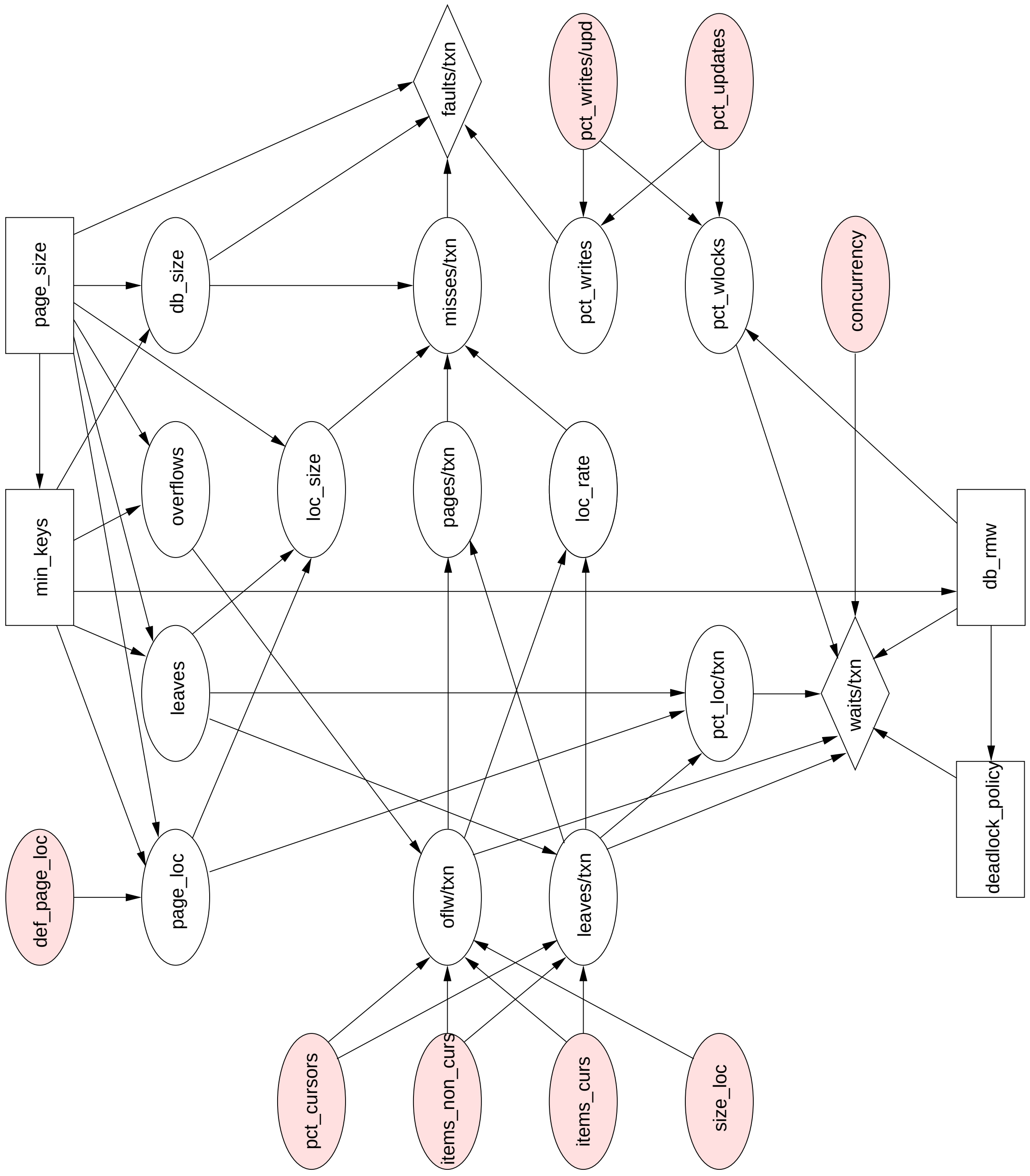


Figure 7.4. The final model. The shaded nodes are the root chance nodes, which represent characteristics of the workloads encountered by the system. All of these characteristics are observed before a tuning decision is made, but the informational arcs that would make this fact explicit have been omitted for the sake of readability. Brief descriptions of the nodes in the influence diagram are given in Table 4.1, and additional detail about both the nodes and arcs is provided in Section 4.4.

Table 7.5. Discretizations of variables in the final model. This table presents the discretizations of the continuous chance nodes in the influence diagram used to evaluate the overall effectiveness of the methodology. The discretizations were learned from the training data using the algorithm presented in Section 5.3.3; the initial discretizations used by the algorithm gave the value nodes 10 equal-height bins and the continuous chance nodes a single bin.

Variable	Number of bins	Cutoff-value sequence
concurrency	3	(27.5, 57.5)
def_page_loc	6	(38.5, 48.5, 53.5, 58.5, 66.5)
items_curs	1	( )
items_non_curs	5	(10.5, 11.5, 23.5, 30.5)
leaves/txn	3	(11.5, 24.5)
loc_rate	5	(68.5, 78.5, 84.5, 89.5)
loc_size	7	(19.5, 32.5, 44.5, 48.5, 53.5, 78.5)
misses/txn	7	(0.55, 0.95, 1.55, 2.85, 6.25, 9.75)
oflw/txn	4	(0.05, 0.65, 2.55)
page_loc	6	(38.5, 48.5, 53.5, 58.5, 66.5)
pages/txn	3	(11.5, 24.5)
pct_cursors	2	(10.5)
pct_loc/txn	4	(1.15, 2.85, 11.35)
pct_wlocks	3	(19.5, 43.5)
pct_writes	11	(2.5, 5.5, 11.5, 17.5, 24.5, 34.5, 39.5, 56.5, 70.5, 77.5)
pct_writes/upd	6	(7.5, 32.5, 44.5, 51.5, 66.5)
pct_updates	8	(19.5, 24.5, 34.5, 39.5, 43.5, 70.5, 77.5)
size_loc	3	(1.5, 6.5)

for the corresponding workloads. The throughputs achieved by the influence diagram's non-optimal knob recommendations (including the near-optimal cases) are 5.04% slower than optimal on average.

By contrast, the default (8K, 2, 0, random) knob settings achieve optimal or near-optimal throughputs on only 33.9% of the test workloads (119 workloads, with 29 optimal recommendations and 90 near-optimal ones). When the default settings are non-optimal, they lead to throughputs that are an average of 37.1% slower than optimal. Figure 7.5 compares the overall performance of the influence diagram's recommendations with the performance of the default knob settings. Only 7.7% of the model's recommendations have slowdowns of 10% or more—with the largest slowdown being 50.2%. The default settings, on the other hand, have slowdowns of

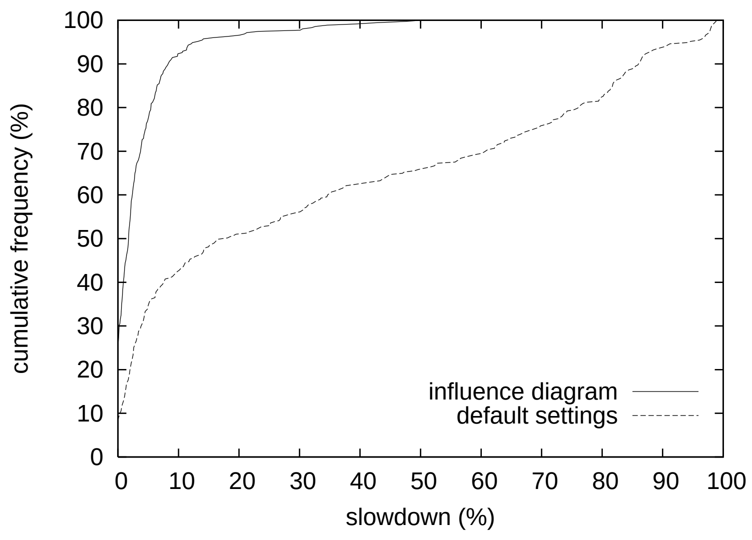


Figure 7.5. Performance of the final model on the test workloads. The above graph shows the cumulative distribution of slowdown values associated with the influence diagram's recommended knob settings for the test-set workloads, and it compares the performance of the influence diagram's recommendations to the performance obtained using the default knob settings. 76.9% of the influence diagram's recommendations are optimal or near-optimal, meaning that they have slowdowns of less than 5%. 92.3% of the recommendations have slowdowns of less than 10%.

10% or more on over half of the test workloads, including slowdowns of over 50% on 120 of the 351 workloads.

7.4.2 Ability of the Model to Generalize

Of the 351 test workloads, *none* have exactly the same combination of workload characteristics as the workloads in the training set. However, in assessing the ability of the influence diagram to generalize, it is perhaps more meaningful to consider only the test workloads that involve previously unseen combinations of the values of the discretized workload variables. Using this criterion for whether a workload is seen in the training set, 41 of the 351 test workloads are previously unseen. The influence diagram recommends optimal or near-optimal knob settings for 33 of these workloads (80.5%), and its non-optimal recommendations have an average slowdown of 6.94%. By comparison, the model recommends optimal or near-optimal settings for 237 of 310 test workloads that are seen in the training data (76.5%), with non-optimal recommendations that are an average 4.76% slower. Figure 7.6 compares the overall performance of the influence diagram's recommendations for the seen and unseen test workloads.

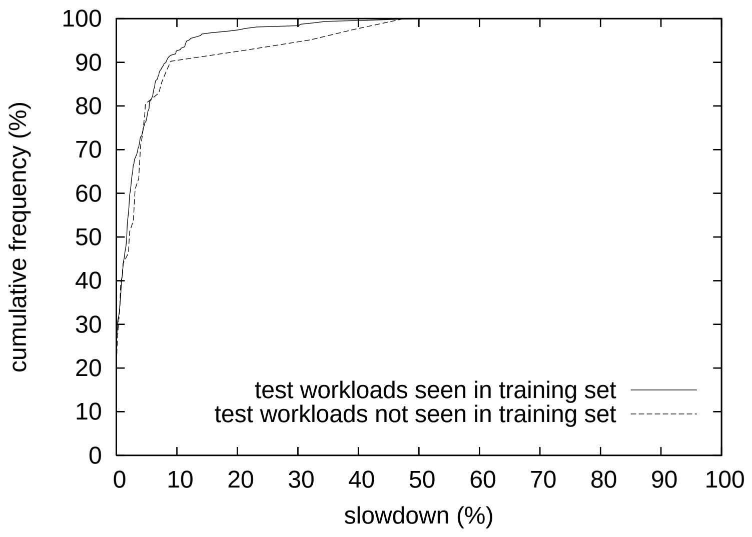


Figure 7.6. Ability of the model to generalize. The above graph shows the cumulative distributions of slowdown values associated with the influence diagram's recommended knob settings for two subsets of the test workloads: the workloads that are seen in the training set, and the previously unseen workloads. A workload is considered seen or unseen on the basis of its *discretized* workload characteristics. Most of the seen workloads are seen in conjunction with only a fraction of the possible knob settings.

Although the knob settings that the influence diagram recommends for the unseen workloads have somewhat poorer performance than the settings that it recommends for the seen workloads, the model's overall performance on the unseen workloads still demonstrates that it is able to generalize from training data to previously unseen workloads. Moreover, most of the seen test workloads are seen in the training set in conjunction with only a fraction of the 32 possible knob settings; the average number of knob settings that appear together with a given combination of discretized workload characteristics in the training set is 9.72. Thus, the model also needs to generalize to a certain degree to obtain knob recommendations for these workloads.

7.4.3 Comparison with a Regression-Only Approach

One possible alternative to using an influence diagram for tuning is to employ models based solely on regression. This type of approach was taken by Brewer in his work on tuning library subroutines [Bre94, Bre95]. To compare the two approaches, 32 equations were derived from the training data using multiple linear regression; each

Table 7.6. Terms considered as possible independent variables in the regression equations. The list includes both the workload characteristics that were varied in the experiments and potentially relevant combinations of those characteristics. For each regression, terms that were deemed irrelevant—because their coefficients had confidence intervals that included zero—were discarded.

workload characteristics

- concurrency (*c*)
- def_page_loc (*p*)
- items_non_cursor (*i*)
- items_cursor (*I*)
- pct_cursors (c)
- pct_updates (*u*)
- pct_writes/upd (*w*)
- size_loc (*s*)

combinations of workload characteristics

ci, $\frac{i}{p}$, $\frac{ci}{p}$, uw, iuw, $ciuw$, si

equation maps workload characteristics to throughput for one of the 32 possible configurations of the knobs. These equations were then used to determine the optimal knob settings for the workloads in the test set. For a given combination of workload characteristics, ***w***, the recommended knob settings are the ones associated with the regression equation that predicts the largest throughput value for the values in ***w***.

To learn the regression equation for a given combination of knob settings, multiple linear regression was performed on the subset of the training examples that used those settings. The regression process began with a set of possible independent variables and iteratively removed any variable that was deemed irrelevant because the 95% confidence interval for its coefficient included zero. If there were multiple irrelevant variables after a given round of regression, the one with the highest *P*-value was discarded.[6] The regression was then rerun on the remaining variables. This process continued until all of the remaining variables were relevant.

The list of possible independent variables included all of the workload variables, as well as combinations of these variables that seemed relevant to the

6. The *P*-value associated with a variable is a measure of the likelihood that the variable does not explain any of the variation observed in the independent variable beyond the variation that is already explained by the other variables.

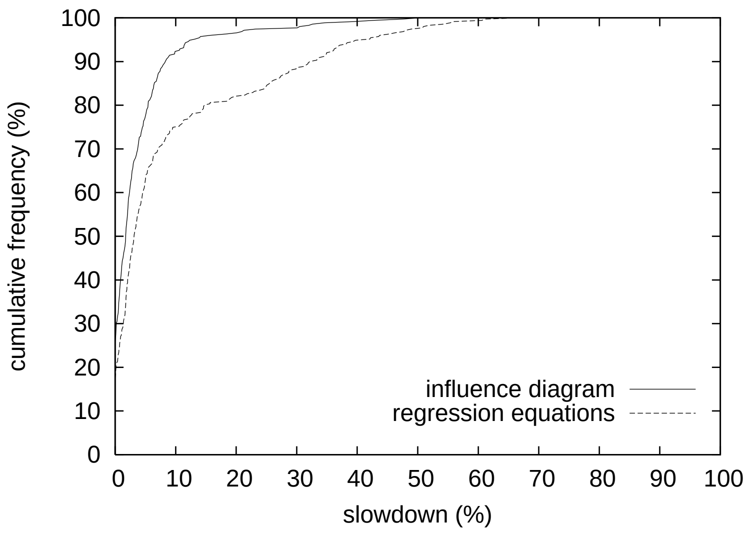


Figure 7.7. Comparing with a regression-only approach. The above graph shows cumulative distributions for the slowdown values associated with the knob settings recommended by both the final influence diagram for Berkeley DB and a set of regression equations derived from the training data.

throughput of the system (Table 7.6). For example, the percentage of items that are modified by a given workload can be estimated as the product of the values of the *pct_updates* and *pct_writes_upd* variables for that workload, and thus this product was included as a possible independent variable. For the reasons discussed in Section 5.4.4, the natural logarithm of throughput was used as the dependent variable rather than throughput itself.

The resulting regression equations have R^2 values that range from 0.701 to 0.966.[7] They recommend optimal or near-optimal knob settings for 219 of the 351 test workloads (62.4%); 61 of the recommended settings are optimal, and 158 have non-optimal throughputs that are within 5% of the maximal measured throughput for the corresponding workload. The throughputs achieved by the regression equations' non-optimal knob recommendations (including the near-optimal cases) are 11.38% slower than optimal on average. Figure 7.7 compares the overall performance of the regression equations' recommendations with the performance of the influence

7. The R^2 value associated with a regression equation provides a measure of how well the equation fits the training data. It can take on any real value in the interval [0.0, 1.0]. Higher values indicate a better fit.

diagram's recommendations on the same workloads. The influence diagram clearly outperforms the regression equations on these workloads.

It might be possible to improve the performance of the regression equations by starting with a different set of terms for the independent variables. However, it is important to note that the process of determining appropriate terms to consider during regression is a difficult one. Indeed, this process involves challenges that are similar to the ones encountered in designing the structure of an influence diagram, and it must be performed without the advantages provided by the graphical structure of influence diagrams and their ability to exploit intuitive notions of causality. And unlike influence diagrams, regression equations fail to take advantage of the conditional independencies that exist between variables, and they may thus require more training data to achieve comparable levels of accuracy.

7.5 Additional Experiments

This section presents the results of experiments that evaluate specific aspects of the proposed methodology. These experiments were run on a set of samples of the training data that will be referred to as the *training samples*. These samples had ten different sizes—ranging from 10 to 100 percent of the total training set—and there were a total of 91 samples: ten random samples of each size less than 100%, and a single sample containing the entire training set.

All of the models derived from the training samples used the same model structure as the final model (Figure 7.4). In addition, the training algorithms used to create the final model were also used to create the models derived from the samples unless the description of a particular experiment states otherwise. New discretizations were learned for each sample, and—unless otherwise stated—the initial discretizations used ten equal-height bins for the value nodes. The only step in the training process that was not performed on a per-sample basis was the learning of weights for the value nodes; instead, the weights learned from the entire training set were reused. This should not affect the results appreciably; separate experiments showed that performing regression on even 10 percent of the training data yields value-node weights that are almost identical to the ones obtained from the full training set.

7.5.1 Evaluating the Impact of Training-Set Size

The process of gathering training data is one of the costlier aspects of any model-based approach to software tuning. Therefore, we would like the model to be useful after a limited amount of training. To evaluate the impact of the number of training examples on the quality of the final model's recommendations, variants of the model were derived from each of the training samples and used to recommend knob settings for the workloads in the test set.

Figure 7.8 summarizes the performance of the influence diagrams derived from the samples. The mean performance of the models increases as the sample size increases from 10% to 60%, at which point it levels off. Some of the models derived from the 30%, 40%, and 50% samples perform comparably to models derived from the larger samples, but others do much more poorly. The models derived from 60 to 90% of the training data have much smaller variances in performance: they consistently make optimal or near-optimal recommendations for approximately 70 to 80% of the workloads, and they have average slowdowns of approximately 4 to 7%. The 90% samples produce models with the highest mean accuracy and lowest mean slowdown; one-tailed t-tests indicate that there are statistically significant differences between the mean results for these models and the corresponding mean results for models derived from the 60%, 70%, and 80% samples. However, the resulting differences in the performance of the system are small enough that any sample size from 60 to 100% could be used.

The time needed to gather 60% of the training data is still quite substantial: it would take the five machines used in the experiments roughly 22 days, rather than the 37 days needed to gather the entire training set. However, *all* of the influence diagrams derived from the samples exceed the performance of the default settings,

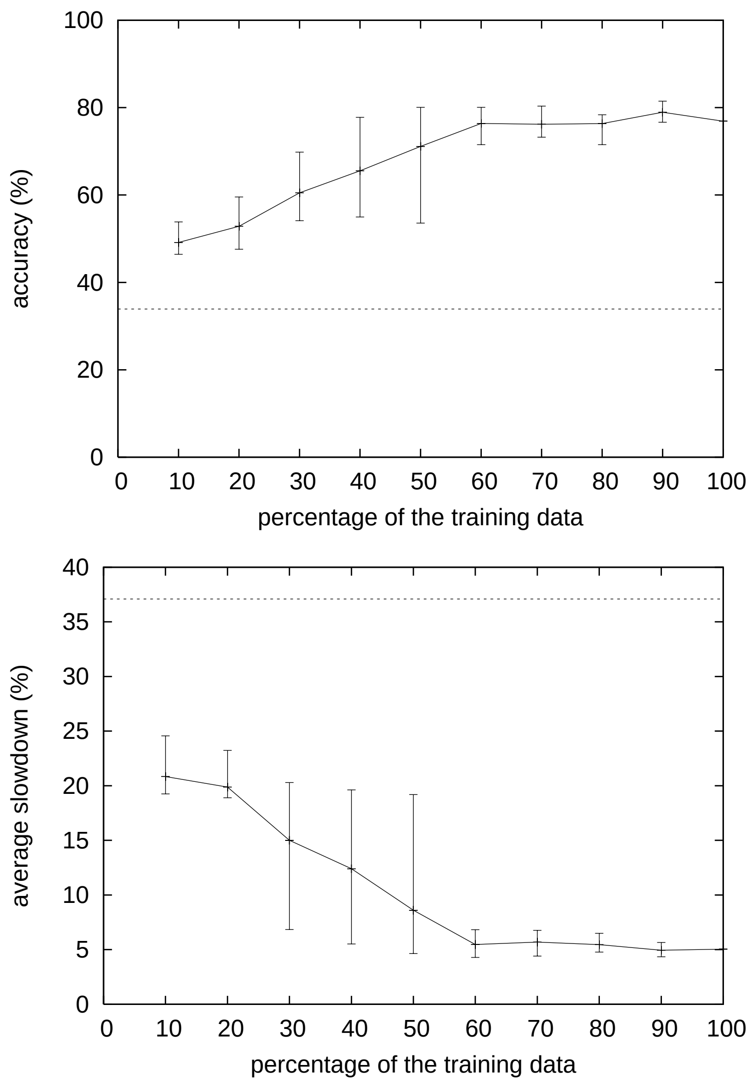


Figure 7.8. Varying the number of training examples. The top graph shows how the accuracy of the models is affected by the size of the training set, and the bottom graph shows how the average slowdown of the models—which does not include workloads for which the slowdown is 0—is affected. For sample sizes less than 100%, the points plotted are the average performance of ten models derived from training sets of that size, and the error bars show the minimum and maximum values for that sample size. The dotted line in each graph indicates the performance of the default knob settings.

which is indicated by the dotted lines on the graphs in Figure 7.8. Therefore, it would be possible to deploy the tuner after gathering only a small amount of training data (e.g., after the four days needed to gather 10% of the data) and to have it begin tuning the system while additional training data is collected offline. The tuner would begin improving the overall performance of the system when it is first deployed, and the quality of its recommendations would increase as additional training data became available, both from offline training and from the training examples that the tuner collects as the system runs.

The influence diagram does significantly better than the default settings after seeing only 10% of the training data, which corresponds to 1871 training examples. To represent this as a percentage of the number of possible combinations of workload characteristics and knob settings, we need to discretize the workload characteristics in some way. When the discretizations learned from the entire training set (Table 7.5) are used, there are 829,440 possible combinations of workload characteristics and knob settings; 10% of the training set thus corresponds to 0.23% of the possible combinations, if we assume that there are no repeated workloads in the set. Another method of counting the number of workload/knob-setting combinations is to discretize the workload characteristics according to the distributions from which they are drawn (Table 7.1). Although this approach ignores the possibility that two values from the same distribution can have different effects on the performance of the system, it has the advantage of not depending on the learned discretizations. When the workload characteristics are discretized in this way, there are 20,736 possible workload/knob-setting combinations; 1871 examples constitute just over 9% of this value. It is impossible to say how much training data is needed for an arbitrary influence diagram to achieve a reasonable level of performance, but these values show that it is possible for an influence diagram to improve the performance of a software system after seeing only a small percentage of the possible combinations of workload characteristics and knob settings.

7.5.2 Evaluating the Procedure for Estimating Unseen Parameters

To assess the effectiveness of the proposed procedure for estimating unseen parameters (Section 5.4.3), variants of the final model were created using different methods of estimating these parameters. Four methods of estimating unseen expected-value parameters were compared: (1) the proposed method, which uses both a nearest-neighbor estimate and one or two constraints to derive its final estimate; (2) a method that always uses a nearest-neighbor estimate as the final estimate; (3) a method that

Table 7.7. Procedures for estimating unseen parameters. The methods listed below were compared on different sized samples of the training data.

Method of estimating an unseen expected value	Method of estimating an unseen probability distribution
nearest-neighbor estimate and constraints	derive from an estimate of its expected value
nearest-neighbor estimate and constraints	use a uniform distribution
nearest-neighbor estimate only	derive from an estimate of its expected value
nearest-neighbor estimate only	use a uniform distribution
constraints only	derive from an estimate of its expected value
constraints only	use a uniform distribution
unseen expected values = 0	use a uniform distribution

uses only the constraints to derive the final estimate, averaging them as needed as described in Section 5.4.3.4; and (4) a method that sets all unseen expected values to 0.[8] In addition, two methods of estimating unseen probability parameters were compared: (1) the proposed method, which derives an unseen probability distribution from an estimate of the distribution's expected value (computed using one of the three methods above) and from the distributions associated with the nearest-neighbor estimate and with the constraints; and (2) a method that uses a uniform distribution for all unseen probability distributions. The latter method of estimating unseen probability parameters was always used in conjunction with the fourth method of estimating unseen expected values. Estimation methods that use either nearest-neighbor estimates or constraints or both to derive their estimates will be referred to as *informed* methods because they base their estimates on the training data, and methods that do not used the training data will be referred to as *uninformed* methods.

Combining the methods for estimating unseen expected values with the methods for estimating unseen probability distributions gives seven different procedures for estimating unseen parameters, as shown in Table 7.7. Each of these procedures was used to derive an influence diagram for each of the training samples, and the resulting models were used to recommend knob settings for the test-set

8. The fourth method (setting unseen expected values to 0) is what happens to unseen expected values by default when the expected values are estimated using simple averages.

Table 7.8. Accuracies of different procedures for estimating unseen parameters. Shown are the accuracies of influence diagrams trained using different procedures for estimating unseen parameters and different sized samples of the training data. For sample sizes less than 100%, each value is the average accuracy of ten models derived from training sets of that size.

Sample size (%)	Accuracy (%)						
	nearest neighbor		constraints		NN + constraints		E = 0
	derive Ps	uniform	derive Ps	uniform	derive Ps	uniform	uniform
10	48.58	48.55	49.17	49.14	49.15	49.09	44.07
20	52.99	52.99	52.91	52.91	52.85	52.85	50.34
30	60.43	60.42	60.51	60.51	60.51	60.51	57.35
40	65.70	65.70	65.44	65.41	65.53	65.50	62.08
50	70.88	70.88	71.42	71.42	71.11	71.11	68.77
60	76.35	76.38	76.38	76.41	76.38	76.41	74.76
70	76.15	76.18	76.21	76.18	76.18	76.15	74.18
80	76.29	76.24	76.38	76.38	76.35	76.35	75.24
90	78.80	78.80	79.00	79.00	78.95	78.95	77.21
100	76.64	76.64	76.90	76.90	76.90	76.90	74.90

workloads. I anticipated that the informed estimation methods would do better than the uniformed methods, and that the method proposed in Section 5.4.3 (which uses both a nearest-neighbor estimate and a set of constraints) would do better than the methods that use either the nearest-neighbor estimate or the set of constraints but not both. I also expected that the differences between the methods would be greater for smaller sample sizes, because it seemed likely that smaller samples would have a larger number of unseen parameters.

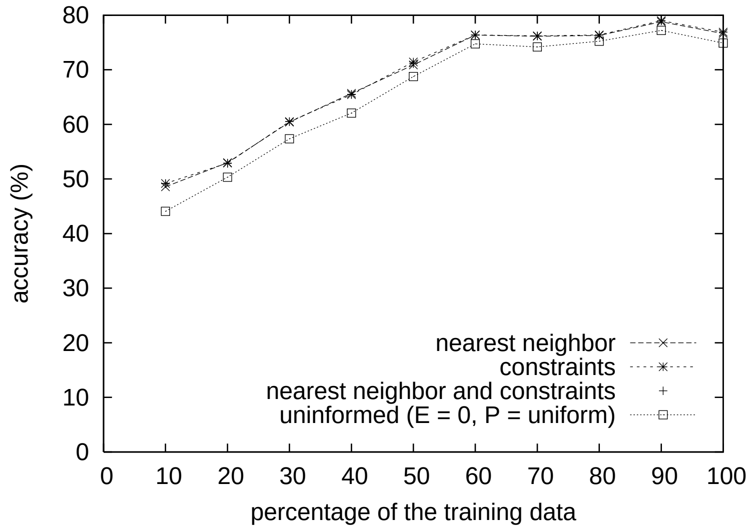
The average accuracies of the influence diagrams for the various combinations of estimation procedures and sample sizes are presented in Table 7.8. These results suggest that an informed method of estimating unseen probability parameters may not be necessary. This can be seen by comparing the results in the pairs of columns labeled “nearest neighbor,” “constraints,” and “NN + constraints” in the table. For a given method of estimating unseen expected values (i.e., a given pair of columns), the models produced using uniform distributions for unseen probability distributions (“uniform”) have almost identical accuracies to the models produced by deriving unseen distributions from estimates of their expected values (“derive Ps”). The average slowdowns (not shown in the table) are also nearly the same. One reason for the similar results is that none of models have a significant number of nontrivial unseen distributions.[9] Later in this section, I explain why the number of unseen probability distributions remains relatively small for all of the sample sizes.

Given the lack of significant differences between the two methods for estimating unseen probabilities, the rest of this section compares only the methods for estimating unseen expected values. It does so by focusing on four of the seven estimation procedures: the three procedures that use informed methods to estimate both unseen expected values and unseen probabilities, and the procedure that uses uniformed methods to estimate both types of unseen parameters.

The average accuracy and slowdown results of the models produced using each of the four procedures are graphed in Figure 7.9. The three informed procedures consistently produce better average results than the uninformed procedure. Because the results can vary widely for models derived from different samples of the same size, not all of the observed differences pass one-tailed t-tests for statistical significance. However, the results for the individual samples suggest that the informed procedures are clearly preferable to the uninformed procedure: for all but three of the 91 samples, the models produced using the informed procedures have higher accuracies than the model produced using the uniformed procedure, and the slowdown results are similarly skewed. Therefore, it seems worthwhile to employ one of the informed procedures to estimate unseen expected values. The choice of which informed procedure to use does not seem to matter. The differences between the average results for these procedures are not statistically significant, and the results for the individual samples are not dominated by any one procedure.

My hypothesis that the informed methods would outperform the uniformed methods is borne out by the results for the methods of estimating unseen expected values, but not for the methods of estimating unseen probabilities. My other two hypotheses are not supported by the results. The proposed method of using both

9. A trivial unseen parameter is one associated with an impossible instantiation of the parents of a node. For example, both *page_size* and *db_size* are parents of *faults/txn*, but it impossible for a page size of 2K to be seen together with the size of one of the 8K database configuations, and vice versa.



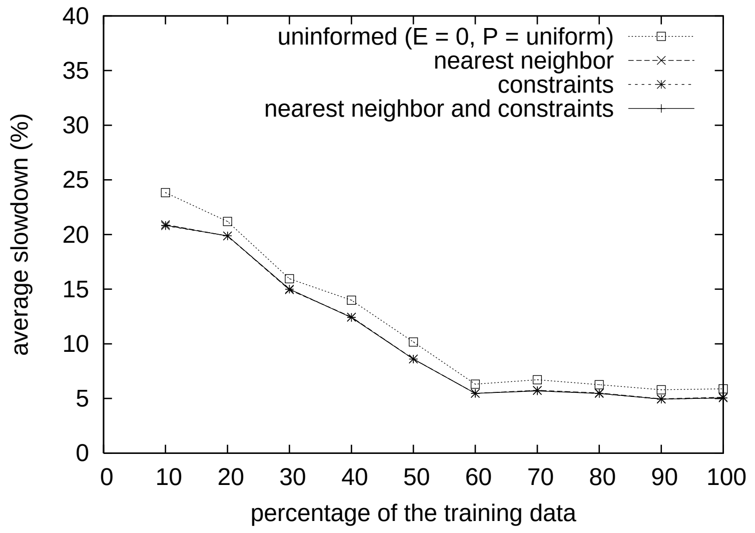


Figure 7.9. Comparing four methods of estimating unseen parameters. The top graph shows the accuracies of models produced using four different procedures for estimating unseen parameters, and the bottom graph shows the average slowdown values of the same models. For sample sizes less than 100%, the points plotted are the average performance of ten models derived from training sets of that size. The *uniformed* procedure sets unseen expected values to 0 and uses a uniform distribution for unseen probability distributions. The other three procedures estimate unseen expected values from a nearest-neighbor estimate, a set of constraints, or both, and they derive unseen probability distributions from estimates of their expected values.

nearest-neighbor estimates and constraints does not outperform the methods that use one or the other. And it is not the case that the differences between the informed and uninformed methods increase significantly as the amount of training data decreases, because the number of unseen parameters does not increase as the sample size decreases. In fact, smaller samples tend to produce fewer unseen parameters than larger samples. This is because the discretization algorithm learns fewer bins per variable for smaller training sets, which prevents the number of unseen parameters from growing too large.

For this model, informed methods of estimating unseen expected values have an impact on performance, but informed methods of estimating unseen probability distributions do not. This difference may stem in part from the particular nodes involved. For example, the *waits/txn* value node, which has the most parents of any node in the model, also has the largest number of parent instantiations and the largest number of unseen parameters. Because it is a value node, the methods for estimating unseen expected values are used to estimate its parameters, and this may explain why the informed methods of estimating unseen expected values have a larger impact on the performance of the models than the informed methods of estimating unseen probabilities. In models for which more of the unseen parameters are probability parameters, the informed methods of estimating probabilities might have more of an effect on the performance of the models.

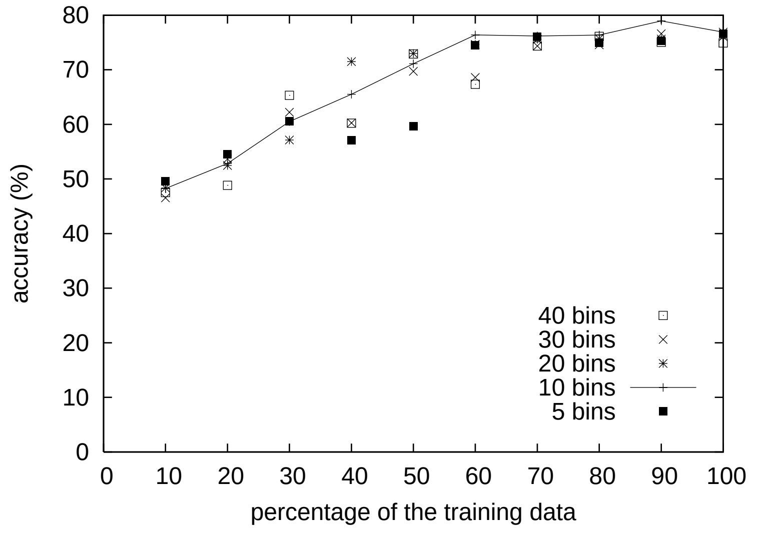
However, it may also be the case that it is generally more important to make informed estimates of unseen expected values than it is to make informed estimates of unseen probabilities, especially when the seen expected values are negative (i.e., when the value nodes represent performance losses or other quantities that we are trying to minimize). Using a value of 0 for unseen expected values in such models can lead the model to compute artificially high expected values for one or more of the knob settings, and to thus recommend non-optimal knob settings. Other possible uninformed methods (e.g., using a negative value with a large magnitude for unseen expected values) can cause the model to artificially *reduce* the expected values of one or more knob settings, and to thus fail to recommend the optimal settings. By comparison, uninformed estimates of probability distributions are not likely to have as large of an effect on the model's recommendations.

7.5.3 Evaluating Aspects of the Discretization Algorithm

Two sets of experiments were conducted to evaluate aspects of the proposed algorithm for discretizing continuous variables in an influence diagram (Section 5.3.3). The first set of experiments assessed the impact of the number of equal-height bins, k, given to the value nodes in the initial discretizations. Models were created from the training samples for initial discretizations that used k = 5, 10, 20, 30, and 40. The performance of these models on the test-set workloads is summarized in Figure 7.10.

No single value of k consistently outperforms the other values. The five values produce models with comparable accuracies and slowdowns from both the 10% samples and those that include 70% or more of the training data. For the intermediate-sized samples (20-60%), the differences in the mean accuracies and slowdowns are more pronounced, although some of these differences do not pass one-tailed t-tests for statistical significance. k = 40 performs significantly better than the other k values on the 30% samples, and k = 20 does significantly better than the other values on the 40% samples. k = 5 does significantly worse than the other values on the 50% samples, and k = 30 and k = 40 do significantly worse than the other values on the 60% samples. In addition, k values of 5, 30, and 40 also perform poorly on the 40% samples. It is possible that using only five bins for the value nodes may at times prevent the discretization algorithm from capturing relevant interactions between the value nodes and their parents, whereas using 30 or 40 bins may increase the risk of overfitting the training data. Additional experiments would be needed to verify these hypotheses.

The final model was produced using k = 10. As shown by the curves in Figure 7.10, this value of k does reasonably well across the full range of sample sizes. There are sample sizes for which other values have better average performance, but k = 10 is usually competitive and never substantially worse. Using k = 20 also produces competitive results for most of the sample sizes. The fact that both k = 10 and k = 20 generally perform well—and that all five values perform comparably on the larger



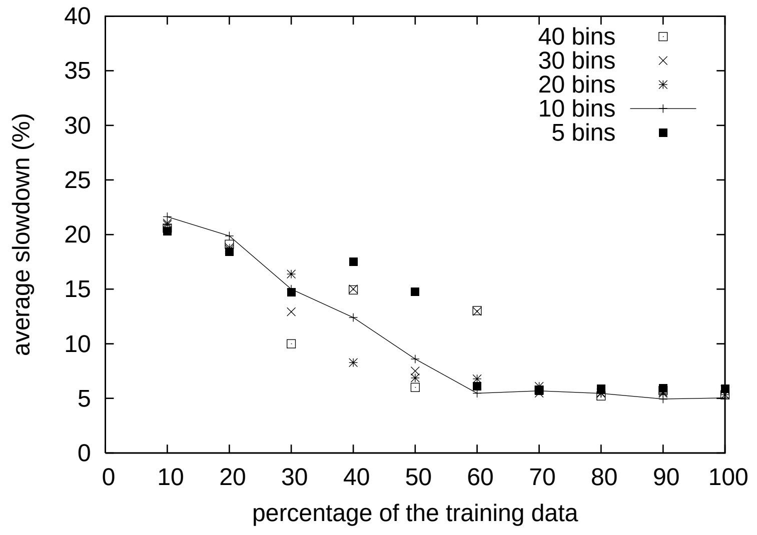
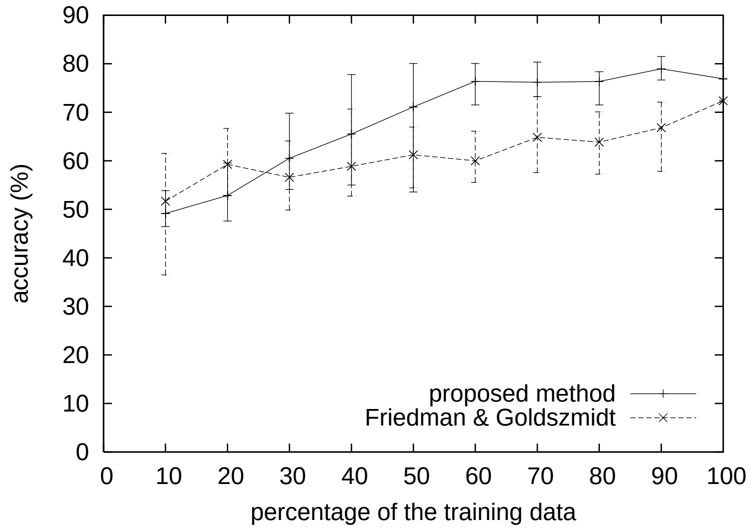


Figure 7.10. Comparing different initial discretizations for the value nodes. The above graphs compare the performance of models produced using initial discretizations that give different numbers of bins to the value nodes. The top graph displays the accuracies of the models, and the bottom graph displays their average slowdown values. For sample sizes less than 100%, the points plotted are the average performance of ten models derived from training sets of that size. The curves shown connect the results for models that were produced using 10 bins—the number used in producing the final model. For the sake of readability, only points are plotted for the other options.

training samples—suggests that it should be possible for the developer of an automated software tuner to select an appropriate *k* value during the process of refining the model and to hard-code that value into the tuner.

The second set of experiments compared two methods of improving a variable's discretization on a given iteration of the algorithm: the method used by Friedman and Goldszmidt [Fri96], in which a variable's current discretization is



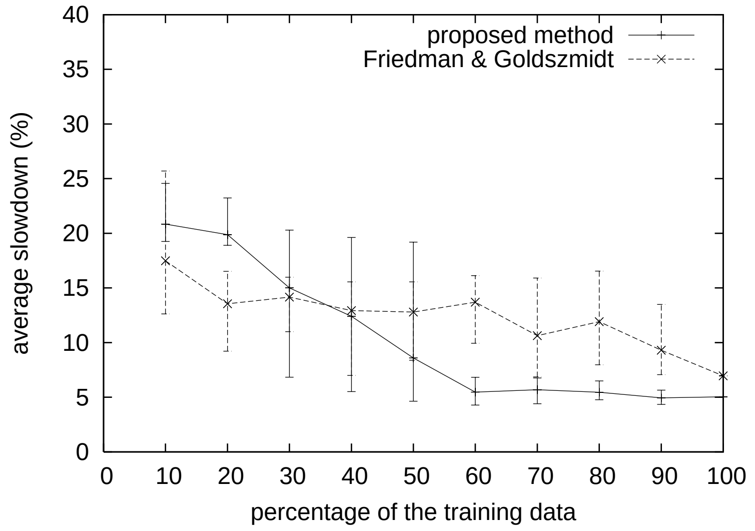


Figure 7.11. Comparing two methods of improving a node's discretization. The above graphs compare the performance of models produced using two methods of improving the discretization of a node on a given iteration of the algorithm presented in Section 5.3.3. The top graph displays the accuracies of the models, and the bottom graph displays their average slowdown values. For sample sizes less than 100%, the points plotted are the average performance of ten models derived from training sets of that size, and the error bars show the minimum and maximum values for that size.

removed and cutoff values are added from scratch until there are no remaining values that reduce the description length of the model; and the proposed method, which starts with the variable's current discretization and adds at most one cutoff value per iteration. These methods are discussed more fully in Section 5.3.3.4. Models were created from the training samples using each of these methods. Figure 7.11 summarizes the performance of these models on the test-set workloads.

Friedman and Goldszmidt's method outperforms the proposed method for a majority of the 10% and 20% samples; it leads to higher accuracies for eight of ten samples in each case and smaller slowdowns for eight of the ten 10% samples and all of the 20% samples. The proposed method leads to higher accuracies for the majority of the 30%, 40%, and 50% samples and to smaller slowdowns for most of the 50% samples, but the slowdown results for the 30% and 40% samples are comparable, and the variance in the results for all of these sample sizes is large. For the larger sample sizes—60% to 100%—the proposed method consistently outperforms the method of Friedman and Goldszmidt on both measures, although the differences become less pronounced as the sample size increases from 60% to 100%. The results produced by the proposed method also have smaller variances for the larger samples.

Further experiments are needed to determine why the two methods perform differently for different sample sizes. However, these results suggest that the proposed method is at least competitive with Friedman and Goldszmidt's method in terms of the quality of the models that are produced. And because the proposed method typically takes significantly less time to learn the discretizations, using it should allow the tuner to save time without sacrificing performance.

7.6 Conclusions

The experiments presented in this chapter demonstrate that the proposed methodology for software tuning can be used to construct a tuner that makes effective tuning recommendations for a wide range of workloads. The final influence diagram for tuning Berkeley DB—designed according to the guidelines presented in Section 5.1 and trained using the algorithms presented in Sections 5.3 and 5.4—makes optimal or near-optimal recommendations for over three quarters of the test workloads. Its recommendations are within 10% of optimal for over 90% of the test workloads, and its non-optimal recommendations lead to an average slowdown of just over 5%. The influence diagram is also able to generalize from experience, as shown by the results presented in Section 7.4.2. Although the performance of the influence diagram is not perfect, it outperforms both the default knob settings and an alternative approach based solely on regression, and it leads to improved performance after seeing only a small percentage of the possible combinations of workload characteristics and knob settings.

The results presented in Section 7.5.2 show that it can be beneficial to make informed estimates of the unseen parameters associated with the value nodes of an influence diagram used for tuning. The three informed methods considered in the experiments—using a nearest-neighbor estimate, a set of constraints on the value of a parameter, or both—perform comparably on training sets of different sizes, and they all outperform an uninformed method that sets an unseen expected value to zero. However, the experiments do not demonstrate any benefit to using an informed method of estimating unseen probability parameters; using a uniform distribution for unseen distributions works as well as the informed methods.

The experiments also demonstrate that the discretization algorithm presented in Section 5.3.3—including the modified method of improving the discretization of a given variable—is able to learn effective discretizations of the continuous variables in an influence diagram. Moreover, the results show that the algorithm performs well for a range of possible values for the parameter that governs the number of bins given to the value nodes. This suggests that this parameter can be hard-coded into the tuner and that the discretization algorithm—and the tuner that uses it—can run without human intervention.

Chapter 8

Conclusions

This chapter provides an overall assessment of the proposed methodology for automated software tuning—an assessment that is informed by my experience of using the methodology to tune Berkeley DB. The chapter then discusses possible directions for future work and summarizes the thesis.

8.1 Assessment of the Proposed Methodology

Section 2.2 outlines a list of criteria that an effective automated software tuner should meet. One way to assess the proposed methodology is to consider the degree to which a tuner constructed using the methodology satisfies these criteria.

The first criterion, accuracy, can be addressed by considering the performance of the final influence diagram for Berkeley DB, as presented in Section 7.4. The final model recommends optimal or near-optimal knob settings for 76.9% of the test-set workloads, and knob settings that are within ten percent of optimal for 92.3% of those workloads. Even when its recommendations are non-optimal, the average slowdown is just over five percent. Ideally, the model would have provided optimal or near-optimal recommendations for *all* of the test-set workloads. The fact that it fails to do so is disappointing, and the existence of workloads for which its recommendations have substantial slowdowns is particularly troubling. However, models of complex systems are seldom perfect, and the fact that the final influence diagram outperforms an alternative approach based on linear regression suggests that influence diagrams can produce tuning recommendations of reasonable accuracy.

The proposed methodology also allows a tuner to satisfy another criterion mentioned in Section 2.2: the ability to respond to changes in the environment. The parameters of an influence diagram for software tuning are derived from training data, and they are continually updated using data gathered from the software system as it runs—and possibly from additional offline training. This continual updating of the parameters—in particular, the use of a fading factor to reduce the impact of older examples—should allow a tuner to respond to environmental changes as they occur. However, this thesis has not evaluated the ability of an influence diagram to adapt over time, and refinements to the procedure outlined in Section 5.6 for updating the model may still be needed.

The results presented in Section 7.4.2 demonstrate the ability of the methodology to produce a tuner that can generalize from experience. The final influence diagram for Berkeley DB produces tuning recommendations for previously unseen workloads that are of comparable quality to its recommendations for workloads that do appear in the training data, although the average slowdown for these workloads is somewhat larger. In addition, it is worth noting that the definition of an unseen workload used to compute these results is fairly restrictive: it only considers a workload to be unseen if its *discretized* workload characteristics *never* appear in the training data. The influence diagram also has to generalize to a certain degree on the remaining test-set workloads, since none of them have *undiscretized* combinations of workload characteristics that appear in the training data, and most of them appear in the training data in conjunction with only a fraction of the possible knob settings.

The results also show that an influence diagram is able to tune multiple knobs simultaneously. By contrast, most prior work in automated software tuning has tuned at most one or two knobs in isolation. In general, the ability of influence diagrams to explicitly model the interactions between the relevant variables and to exploit conditional independencies between them should enable influence diagrams to tune more knobs than models that lack these abilities. For example, in the regression-only approach considered in Section 7.4.3, separate sets of training examples are needed for each combination of knob settings, which limits the number of combinations that can reasonably be considered.[1] This limitation does not apply to an influence diagram, because a single training example can provide information that is relevant to multiple combinations of knob settings.

Another criterion listed in Section 2.2 is the need for an effective tuner to have reasonable time costs. The runtime costs of using a trained influence diagram to recommend knob settings are potentially large in theory, but in practice they are typically quite small—at most one or two seconds per workload for the model used in the experiments. However, as discussed in Section 2.3.3, there can be substantial time costs associated with the process of collecting the training data needed by this and other model-based approaches. This was the case in my work with Berkeley DB: over a month was needed to collect the full training set used in the experiments. However, as shown in Section 7.5.1, it would have been possible to begin obtaining performance improvements from this model after only a small percentage of the training data had been collected. In addition, the bulk of the training time was devoted to warming Berkeley DB's memory pool and the operating system's buffer cache. In situations in which cache-related effects are less significant, it should be possible to collect training data much more efficiently. In any case, the use of a workload generator allows training data to be collected offline without disrupting the performance of the software system, and the time costs associated with training are amortized over the life of the tuner.

Finally, Section 2.2 requires that an effective tuner be fully automated, and the proposed methodology satisfies this requirement. Although there is a considerable amount of up-front work that must performed to design the model used by the tuner and, when one is needed, the workload generator used to produce

1. It may be possible in some domains to construct regression equations that include the knob settings as independent variables, in which case this limitation would not hold. However, it seems unlikely that this approach would work well in general, especially given the fact that knobs often have nomimal values rather than numeric ones. The *deadlock_policy* knob considered in the experiments is one example of a knob with nominal values.

training data, this work only needs to be performed once for a given software system—either by a system or application developer or someone else who is familiar with the workings of the system. Once the tuner and workload generator are designed, they can run unattended. The discretization algorithm presented in Section 5.3.3, which builds on the work of Friedman and Goldszmidt [Fri96], overcomes one potential barrier to complete automation by ensuring that the continuous variables in the model can be discretized automatically.

As discussed in Section 2.3.3, another potential limitation of model-based approaches to software tuning is the difficulty of devising a good model. I experienced this difficulty firsthand in designing the influence diagram for Berkeley DB, which involved many months of refinement. However, the design process became easier with time. I originally constructed an influence diagram for tuning just the *page_size* and *min_keys* knobs, and it took over a year to come up with a model that made reasonable predictions. However, I started with only limited knowledge of the system, and the failure of my initial workload generator to capture the steady-state performance of the system also impeded my attempts to refine the model. Once I had improved the measurements made by the workload generator and completed the two-knob model, I was able to extend it to incorporate two additional knobs in only a few weeks' time. This is not to say that designing a good model of this type will ever be easy; *all* approaches to software tuning that can accommodate unseen workloads—i.e., all approaches based on feedback techniques or analytical models—require both expertise and experimentation during the design process. But my hope is that the guidelines presented in Section 5.1 will facilitate the design of influence diagrams for software tuning and other related applications.

In summary, the proposed methodology has the potential to create effective, automated software tuners. It will be necessary to actually deploy a tuner based on this methodology to fully confirm its effectiveness, but the work presented in this thesis suggests that it is worth taking that next step.

8.2 Future Work

One aspect of the proposed methodology that remains to be tested is the process of updating the parameters of the model over time (Section 5.6). Experiments are needed to determine appropriate values for the fading factor used to reduce the impact of older training examples over time, as well as to assess whether multiple sources of new training data are needed to maintain the tuner's accuracy. As mentioned in Section 5.6, training examples obtained from the software system as it runs—which I will refer to as *runtime examples*—may not be sufficient. The primary reason for this is that all runtime examples involve the tuner's recommended knob settings. If the optimal knob settings for a particular workload change over time because of changes in the environment, it may be impossible to detect this fact from the runtime examples alone. As a result, it may be necessary to supplement the runtime examples with periodic offline training, or to occasionally experiment with non-optimal settings on the system itself.

Important insights could also be gained by applying the proposed methodology to other software systems. For example, both of the value nodes in the final influence diagram for Berkeley DB (Figure 7.4) have at least one decision node as a parent. The *waits/txn* node inherits from both the *db_rmw* and *deadlock_policy* nodes, and the *faults/txn* node inherits from the *page_size* node, which together with the *db_size* node also provides information about the value of the *min_keys* node. Giving the value nodes decision-node parents may enable an influence diagram to more accurately predict the performance of different knob settings, because it allows the model to learn a different expected value for each combination of a value node's decision-node parents. However, this type of design may also lead to overfitting by reducing the number of training examples that contribute to each expected value. Having the value nodes inherit from one or more decision nodes seems to work well in the influence diagram for Berkeley DB; applying the methodology to other systems could help to determine whether this type of design is generally preferable.

There are at least two possible extensions to the methodology that I believe would be worth pursuing. One is to extend it to handle numerical-valued knobs with many possible settings. To handle such knobs efficiently, it may be necessary to discretize them along with the continuous chance nodes, as discussed in Section 5.3.3.1. The influence diagram would then be used to recommend an interval of possible settings rather than a precise value. The challenge is to devise a procedure for selecting a particular setting from the recommended interval. This may necessitate supplementing the influence diagram with an additional model of some sort, or empirically exploring possible values within the recommended interval using a feedback-based approach.

A second worthwhile extension would be to enable the methodology to handle asymmetric tuning problems. Standard influence diagrams assume that a decision problem is symmetric: that the set of possible values for each chance or decision node in the model does not depend on the instantiation of the node's parents. Among other things, this means that all possible combinations of knob settings must be considered, even though some combinations may never make sense. The need to treat an asymmetric decision problem as if it were symmetric can lead to both unnecessary time devoted to training (as data is collected for combinations of knob settings that are never worth considering) and to unnecessary computation when the influence diagram is evaluated. Methods have been proposed for dealing with asymmetric decision problems within the framework of an influence diagram [Qi94, Nie00], and it might make sense to use of one of these methods in the context of software tuning.

It may also be worth considering modifications to the methods used to train the influence diagram. For example, it may be beneficial to use the proposed procedure for estimating unseen parameters (Section 5.4.3) to improve the estimates of *seen* parameters for which only a small number of training examples are available. Doing so might reduce the likelihood of overfitting. Other modifications to the proposed training methods could also be explored.

Finally, it would be interesting to explore the ability of the proposed methodology to tune other types of complex systems. Examples of such systems include server applications that need to determine appropriate modifications to their default resource allocations [Sul00a] and teams of rational software agents that need to balance their individual interests with the needs of the group [Sul00b, Gro02].

8.3 Summary

The need for an automated approach to software tuning has existed for some time because of the difficulties involved in manually tuning complex software systems. Current trends, including the growing complexity of software systems and the increasing deployment of software systems in settings where manual tuning is either impractical or overly costly, only exacerbate this need. Prior efforts at automated tuning have primarily focused on tuning one or two of the many knobs that systems expose for tuning. In addition, these efforts have been closely tied to particular systems, which makes it difficult to transfer this earlier work to other systems.

To facilitate the construction of self-tuning software systems, this thesis has proposed a methodology for automated software tuning that provides step-by-step guidance for building an automated tuner for an arbitrary software system. The resulting tuners employ a model known as an influence diagram and related learning and inference algorithms to determine the optimal knob settings for each workload encountered by the system. The methodology includes guidelines for designing the structure of an influence diagram for software tuning, as well as procedures for learning the initial parameters of the model from training data and updating the values of these parameters over time.

This thesis has also addressed several challenges associated with using an influence diagram for tuning. First, it has explained that it is often necessary, for reasons of efficiency, to replace a single overarching performance measure with multiple metrics that each reflect one aspect of the system's performance, and it has shown that regression techniques can be used to learn weights for these metrics so that optimizing their sum is equivalent to optimizing the overall performance of the system. Second, it has presented an algorithm, based on prior work by Friedman and Goldszmidt [Fri96], for discretizing continuous variables in an influence diagram. The proposed algorithm learns the discretizations from training data, and it balances the complexity of the model with the degree to which the discretizations capture the interactions between related variables. Third, the thesis has proposed a method for estimating unseen parameters in an influence diagram—probabilities and expected values for which no training data is available. The proposed method supplements a nearest-neighbor approach with constraints that are derived from the monotonic relationship that typically exists between related variables in an influence diagram for software tuning. Fourth, the thesis has shown how to design a workload generator that can be used to produce training data for software tuning, and it has presented a technique for ensuring that a workload generator captures the steady-state performance of the system being tuned.

By addressing these challenges and presenting the proposed methodology, this thesis has made it possible for probabilistic reasoning and decision-making techniques to serve as the foundation of an effective, automated approach to software tuning. Moreover, the thesis has shown that an influence diagram created using the proposed methodology is able to produce considerable performance improvements for a varied set of workloads for the Berkeley DB embedded database system, including workloads that are not encountered during training. Although there are still issues that must be addressed before a tuner based on the proposed methodology can be deployed, the results demonstrate that this methodology has the potential to meet the need for a fully automated approach to software tuning.

References

[Ago90] Alice M. Agogino and K. Ramamurthi. Real time influence diagrams for monitoring and controlling mechanical systems. In Robert M. Oliver and James Q. Smith, eds., *Influence Diagrams, Belief Nets and Decision Analysis.* John Wiley & Sons, New York, NY, 1990.

[Agr00] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes for SQL databases. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB '00)*, Cairo, Egypt, September 2000.

[Bad75] Marc Badel, Erol Gelenbe, Jacques Leroudier, and Dominique Potier. Adaptive optimization of a time-sharing system's performance. *Proceedings of the IEEE* 63(6):958-956, June 1975.

[Bar93] Joseph S. Barrera III. Self-tuning systems software. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, IEEE Computer Society, Napa, CA, October 1993.

[Ber98] Philip A. Bernstein, Michael L. Brodie, Stefano Ceri, David J. DeWitt, Michael J. Franklin, Hector Garcia-Molina, Jim Gray, Gerald Held, Joseph M. Hellerstein, H. V. Jagadish, Michael Lesk, David Maier, Jeffrey F. Naughton, Hamid Pirahesh, Michael Stonebraker, and Jeffrey D. Ullman. The Asilomar report on database research. *SIGMOD Record* 27(4):74-80, 1998.

[Ble76] Parker R. Blevins and C.V. Ramamoorthy. Aspects of a dynamically adaptive operating system. *IEEE Transactions on Computers* C-25(7):713-725, July 1976.

[Bre94] Eric A. Brewer. Portable high-performance supercomputing: high-level platform-dependent optimization. Ph.D. thesis, Massachusetts Institute of Technology, September 1994.

[Bre95] Eric A. Brewer. High-level optimization via automated statistical modeling. In *Proceedings of the Fifth Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, Santa Barbara, CA, July 1995.

[Bro93] Kurt P. Brown, Michael J. Carey, and Miron Livny. Towards an autopilot in the DBMS performance cockpit. In *Proceedings of the Fifth International High Performance Transaction Systems Workshop (HPTS '93)*, Asilomar, CA, September 1993.

[Bro94] Kurt P. Brown, Manish Mehta, Michael J. Carey, and Miron Livny. Towards automated performance tuning for complex workloads. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB '94)*, Santiago, Chile, September 1994.

[Bro95] Kurt P. Brown. Goal-oriented memory allocation in database management systems. Ph.D. thesis, University of Wisconsin, Madison, 1995 (available as technical report CS-TR-1995-1288).

[Bro96] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Madison, WI, June 1996.

[Bro97] Aaron Brown. A decompositional approach to performance evaluation. Harvard University Computer Science Technical Report TR-03-97, April 1997.

[Bur95] Lisa Burnell and Eric Horvitz. Structure and chance: melding logic and probability for software debugging. *Communications of the ACM* 38(3):31-41, 1995.

[Cha97] Surajit Chaudhuri and Vivek Narasayya. An efficient cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd International Conference on Very Large Databases (VLDB '97)*, Athens, Greece, August 1997.

[Cha99] Surajit Chaudhuri, Eric Christensen, Goetz Graefe, Vivek R. Narasayya, and Michael J. Zwilling. Self-tuning technology in Microsoft SQL server. *Data Engineering Journal* 22(2):20-26, June 1999.

[Cha00] Surajit Chaudhuri and Gerhard Weikum. Rethinking database system architecture: towards a self-tuning, RISC-style database system. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB '00)*, Cairo, Egypt, September 2000.

[Com79] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys* 11(2):121-137, June 1979.

[Cor62] Fernando J. Corbato, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the AFIPS Fall Joint Computer Conference*, 1962.

[Cor90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[Cov91] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley & Sons, New York, NY, 1991.

[Coz01] Fabio G. Cozman. JavaBayes toolkit. http://www-2.cs.cmu.edu/~javabayes/Home/index.html.

[Dem77] A.P. Dempster, N.M. Laird, and D.B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society*, Series B, 39(1):1-38, 1977.

[Dou95] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In Armand Prieditis and Stuart Russell, eds., *Machine Learning: Proceedings of the Twelfth International Conference*, Morgan Kaufmann, San Francisco, CA, 1995.

[Dra67] Alvin W. Drake. *Fundamentals of Applied Probability Theory*. McGraw-Hill, New York, NY, 1967.

[Fay93] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference in Artificial Intelligence (IJCAI '93)*, Chambéry, France, August-September 1993.

[Fei99] Dror G. Feitelson and Michael Naaman. Self-tuning systems. *IEEE Software* 16(2):52-60, March/April 1999.

[Fra01] Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*, fourth edition. Pearson Education, Upper Saddle River, NJ, 2001.

[Fri96] Nir Friedman and Moises Goldszmidt. Discretizing continuous attributes while learning Bayesian networks. In Lorenza Saitta, ed., *Machine Learning: Proceedings of the Thirteenth International Conference*. Morgan Kaufmann, San Francisco, CA, 1996.

[Fri97] Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Machine Learning* 29(2-3):131-163, 1997.

[Fri98] Nir Friedman, Moises Goldszmidt, and Thomas J. Lee. Bayesian network classification with continuous attributes: Getting the best of both discretization and parametric fitting. In Jude W. Shavlik, ed., *Machine Learning: Proceedings of the 15th International Conference*. Morgan Kaufmann, San Francisco, CA, 1998.

[Fri03] Nir Friedman, personal communication, April 15, 2003.

[Goe99] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. Adaptive resource management via modular feedback control. CSE Technical Report CSE-99-003, Oregon Graduate Institute, January 1999.

[Gra93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Francisco, CA, 1993.

[Gro02] Barbara J. Grosz, Sarit Kraus, David G. Sullivan, and Sanmay Das. The influence of social norms and social consciousness on intention reconciliation. *Artificial Intelligence* 142(2002):147-177.

[Hec95] David Heckerman. A tutorial on learning with Bayesian networks. Technical report TR-95-06, Microsoft Research, March 1995 (revised November 1996).

[Hec98] David Heckerman and Eric Horvitz. Inferring informational goals from free-text queries. In *Proceedings of 14th Conference on Uncertainty in Artificial Intelligence (UAI '98)*, Madison, WI, July 1998.

[Hor98] Eric Horvitz, Jack Breese, David Heckerman, David Hovel, and Koos Rommelse. The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI '98)*, Madison, WI, July 1998.

[How84] Ronald A. Howard and James E. Matheson. Influence diagrams. In R. A. Howard and J. E. Matheson, eds., *The Principles and Applications of Decision Analysis, Vol. II*, Strategic Decisions Group, Menlo Park, CA, 1984.

[How90] Ronald A. Howard. From influence to relevance to knowledge. In Robert M. Oliver and James Q. Smith, eds., *Influence Diagrams, Belief Nets and Decision Analysis*. John Wiley & Sons, New York, NY, 1990.

[Hug01] Hugin Expert A/S. Hugin decision-engine toolkit. www.hugin.dk.

[Irg88] Adam E. Irgon, Anthony H. Dragoni, Jr., and Thomas O. Huleatt. FAST: A large scale expert system for application and system software performance tuning. In *Proceedings of the 1998 ACM Conference on the Measurement and Modeling of Computer Systems (SIGMETRICS '98)*, Sante Fe, New Mexico, June 1998.

[Jac88] Van Jacobson. Congestion avoidance and control. In *Proceedings of the 1988 ACM Symposium on Communications Architectures and Protocols (SIGCOMM '88)*, Stanford, CA, August 1988.

[Jam99] Anthony Jameson, Ralph Schäfer, Thomas Weis, André Berthold, and Thomas Weyrath. Making systems sensitive to the user's time and working memory constraints. In *Proceedings of the 1999 International Conference on Intelligent User Interfaces (IUI '99)*, Los Angeles, CA, January 1999.

[Jam00] Anthony Jameson, Barbara Großmann-Hutter, Leonie March, and Ralf Rummer. Creating an empirical basis for adaptation decisions. In *Proceedings of the 2001 International Conference on Intelligent User Interfaces (IUI 2000)*, New Orleans, LA, January 2000.

[Jen94] Frank Jensen, Finn V. Jensen, and S.L. Dittmer. From influence diagrams to junction trees. In *Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence (UAI '94)*, Seattle, WA, July 1994.

[Jen01] Finn V. Jensen. *Bayesian Networks and Decision Graphs*. Springer Verlag, 2001.

[Kan02] Michael Kanellos. IBM leads charge on holistic computing. CNET News.com, April, 11, 2002, http://news.com.com/2100-1001-881279.html.

[Kes91] Srinivasan Keshav. A control-theoretic approach to flow control. In *Proceedings of the 1991 ACM Symposium on Communications Architectures and Protocols (SIGCOMM '91*), Zurich, Switzerland, September 1991.

[Koh96] Ron Kohavi and Mehran Sahami. Error-based and entropy-based discretization of continuous features. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD '96)*, Portland, OR, August 1996.

[Laz84] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Upper Saddle River, NJ, 1984.

[Mak98] Spyros Makridakis, Steven C. Wheelwright, and Rob J. Hyndman. *Forecasting: Methods and Applications*, third edition. John Wiley & Sons, New York, NY, 1998.

[Man98] Stephen Manley, Margo Seltzer, and Michael Courage. A self-scaling and self-configuring benchmark for web servers. In *Proceedings of the 1998 ACM Conference on the Measurement and Modeling of Computer Systems (SIGMETRICS '98)*, Madison, WI, June 1998.

[Mas90] Henry Massalin and Calton Pu. Fine-grain adaptive scheduling using feedback. *Computing Systems* 3(1):139-173, Winter 1990.

[Mat90] James E. Matheson. Using influence diagrams to value information and control. In R.M. Oliver and J.Q. Smith, eds., *Influence Diagrams, Belief Nets and Decision Analysis*, John Wiley & Sons, New York, NY, 1990.

[Mat97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randy Wang, and Thomas Anderson. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP '97)*, Saint Malo, France, October 1997.

[Men01] Daniel A. Menascé, Daniel Barbará, and Ronald Dodge. Preserving QoS of e-commerce sites through self-tuning: A performance model approach. In *Proceedings of the 2001 ACM Conference on Economic Commerce (EC '01)*, Tampa, FL, October 2001.

[Mit97] Tom M. Mitchell. *Machine Learning*. WCB McGraw-Hill, Boston, MA, 1997.

[Nar00] Dushyanth Narayanan, Jason Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications*, Monterey, CA, December 2000.

[Net96] John Neter, Michael H. Kutner, Christopher J. Nachtsheim, and William Wasserman. *Applied Linear Statistical Models*, fourth edition. Richard D. Irwin, Inc., Chicago, IL, 1996.

[Nie99] Thomas D. Nielsen and Finn V. Jensen. Welldefined decision scenarios. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI '99)*, Stockholm, Sweden, July 1999.

[Nie00] Thomas D. Nielsen and Finn V. Jensen. Representing and solving asymmetric Bayesian decision problems. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI 2000)*, Stanford, CA, July 2000.

[Nob97] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, J. Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP-16)*, Saint-Malo, France, October 1997.

[Nor97] Norsys Software Corp. *Netica API Programmer's Library Reference Manual*, version 1.06. June 14, 1997. http://www.norsys.com/dl/NeticaAPIMan.ps.

[Nor03] Norsys Software Corp. Netica API. http://www.norsys.com/netica.html.

[Oli90] Robert M. Oliver and James Q. Smith, eds. *Influence Diagrams, Belief Nets and Decision Analysis*. John Wiley & Sons, New York, NY, 1990.

[Ols99] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the 1999 USENIX Annual Technical Conference*, Monterey, California, June 1999.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, revised second printing. Morgan Kaufmann, San Francisco, CA, 1988.

[Pfe99] Avi Pfeffer, Daphne Koller, Brian Milch, and Ken T. Takusagawa. SPOOK: a system for probabilistic object-oriented knowledge representation. In *Proceedings of the 15th Annual Conference on Uncertainty in AI (UAI '99)*, Stockholm, Sweden, July 1999.

[Pfe00] Avi Pfeffer and Daphne Koller. Semantics and inference for recursive probability models. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*, Austin, TX, July-August, 2000.

[Pra94] Malcolm Pradhan, Gregory Provan, Blackford Middleton, and Max Henrion. Knowledge engineering for large belief networks. In *Proceedings of the 10th Annual Conference on Uncertainty in Artificial Intelligence (UAI '94)*, Seattle, WA, July 1994.

[Qi94] Runping Qi, Nevin Lianwen Zhang, and David Poole. Solving asymmetric decision problems with influence diagrams. In *Proceedings of the 10th Annual Conference on Uncertainty in AI (UAI '94)*, Seattle, WA, July 1994.

[Rei81] David Reiner and Tad Pinkerton. A method for adaptive performance improvement of operating systems. In *Proceedings of the 1981 ACM Conference on the Measurement and Modeling of Computer Systems (SIGMETRICS '81)*, Las Vegas, NV, September 1981.

[Ros01] Sheldon Ross. *A First Course in Probability*, sixth edition, Prentice-Hall, Upper Saddle River, NJ, 2001.

[Rus95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Upper Saddle River, NJ, 1995.

[Sch99] K. Bernhard Schiefer and Gary Valentin. DB2 universal database performance tuning. *Data Engineering Journal* 22(2):12-19, June 1999.

[Sel96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI '96)*, Seattle, WA, October 1996.

[Sel97] Margo I. Seltzer and Christopher Small. Self-monitoring and self-adapting operating systems. In *Proceedings of the Sixth Workshop on Hot Topics on Operating Systems (HotOS-VI)*, Chatham, MA, May 1997.

[Sel99a] Margo Seltzer and Michael Olson. Challenges in embedded database system administration. In *Proceedings of the First Workshop on Embedded Systems*, Cambridge, MA, March 1999.

[Sel99b] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HOTOS-VII)*, Rio Rico, AZ, March 1999.

[Sha86] Ross D. Shachter. Evaluating influence diagrams, *Operations Research* 34:871-882, November-December 1986.

[Sha89] Ross D. Shachter and C. Robert Kenley. Gaussian influence diagrams. *Management Science* 35(5):527-550, May 1989.

[Sha90] Ross D. Shachter, David M. Eddy, and Vic Hasselbad. An influence diagram approach to medical technology assessment. In Robert M. Oliver and James Q. Smith, eds., *Influence Diagrams, Belief Nets and Decision Analysis*. John Wiley & Sons, New York, NY, 1990.

[Sha98] Ross D. Shachter. Bayes-ball: the rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI '98)*, Madison, WI, July 1998.

[Sle01] Sleepycat Software, Inc. *Berkeley DB*. New Riders Publishing, Indianapolis, IN, 2001. Also available online at www.sleepycat.com/docs.

[Smi93] James E. Smith. Moment methods for decision analysis. *Management Science* 39(3):340-358, March 1993.

[Smi01] Keith Arnold Smith. Workload-specific file system benchmarks. Ph.D. thesis, Harvard University, January 2001.

[Spi98] Peter M. Spiro. Ubiquitous, self-tuning, scalable servers. In *Proceedings of the 1998 ACM International Conference on the Management of Data (SIGMOD '98)*, Seattle, WA, June 1998.

[Sta00] Standard Performance Evluation Council. SPECweb99 Release 1.02. On-line whitepaper. http://www.spec.org/osg/web99/docs/whitepaper.html.

[Ste99] David C. Steere, Ashvin Goel, Joshua Gruenberg, Dylan McNamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, February 1999.

[Sul00a] David G. Sullivan and Margo I. Seltzer. Isolation with flexibility: a resource management framework for central servers. In *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[Sul00b] David G. Sullivan, Barbara J. Grosz, and Sarit Kraus. Intention reconciliation by collaborative agents. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS 2000)*, Boston, MA, July 2000.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Upper Saddle River, NJ, 1992.

[Tra90] Transaction Processing Performance Council. *TPC Benchmark B Standard Specification*. Waterside Associates, Fremont, CA, 1990.

[Vap95] Vladimir N. Vapnik. *The Nature of Statistical Learning Theory*, Springer-Verlag, Heidelberg, Germany, 1995.

[Vud01] Richard Vuduc, James W. Demmel, and James Bilmes. Statistical models for automatic performance tuning. In *Proceedings of the 2001 International Conference on Computational Science (ICCS 2001)*, San Francisco, CA, May 2001.

[Wei94] Gerhard Weikum, Christof Hasse, Axel Moenkeberg, and Peter Zabback. The COMFORT automatic tuning project, *Information Systems* 19(5):381-432, 1994.

[Wel01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP-18)*, Banff, Canada, October 2001.

[Zil01] Daniel C. Zilio, Sam Lightstone, Kelly A. Lyons, and Guy M. Lohman. Self-managing technology in IBM DB2 universal database. In *Proceedings of the 2001 ACM International Conference on Information and Knowledge Management (CIKM '01)*, Atlanta, GA, November 2001.