Automatic Diagnosis of Performance Problems in Database Management Systems

by

Darcy G. Benoit

A thesis submitted to the

School of Computing

in conformity with the requirements for

the degree of Doctor of Philosophy

Queen's University

Kingston, Ontario, Canada

June 2003

Abstract

Database performance is directly linked to the allocation of the resources used by the Database Management System (DBMS). The complex relationships between numerous DBMS resources make problem diagnosis and performance tuning complex and time-consuming tasks. Costly Database Administrators (DBAs) are currently needed to initially tune a DBMS for performance and then to retune the DBMS as the database grows and workloads change. Automatic diagnosis and resource management removes the need for DBAs, greatly reducing the cost of ownership for the DBMS. An automated system also allows the DBMS to respond more quickly to changes in the workload as performance can be monitored 24 hours a day. An automated diagnosis and resource management system allows the DBMS to improve performance for both static and dynamic workloads.

One of the key issues in automatic resource management is the capability of the system to diagnose resource problems. Diagnosis of the resource allocation problem is the first step in the process of tuning the resources. In this dissertation, we propose an automatic diagnosis framework and algorithm that can be used to diagnose DBMS resource problems. We formally define the DBMS diagnosis problem and analyze problem complexity. We develop a model to diagnosis the DBMS and demonstrate the ability of the model to correctly identify system bottlenecks for a generic OLTP workload. We

modify the OTLP workload to further demonstrate the ability of the diagnosis system to handle changing workloads.

The diagnosis system is evaluated by comparing the performance of the DBMS workload tuned by the diagnosis system to the performance of the same workload tuned by an expert and by the Performance Tuning Wizard software included with our test database. Achieving workload performance that is close to or better than these tuning methods will deem the diagnosis system a success.

The contributions of this dissertation include the formalization of the diagnosis problem, an analysis of the complexity of the problem, the development and implementation of models to demonstrate that the diagnosis process can be successfully automated and the presentation of a generic diagnosis system that can be adapted to other software systems that rely on resource feedback for performance tuning.

Acknowledgements

My journey through university has been a long a winding road which has taken me across the paths of many interesting and helpful people, of which there are too many to mention. It is due to these people that my journey has been worthwhile, for the knowledge that I have received from books has paled in comparison to all that I have learned from those close to me. It is with this in mind that I would like to thank those who have helped me though this process.

First and foremost, I would like to thank my supervisor Patrick Martin, who has been extremely supportive and patient in dealing with me. My journey has not always been easy, but he has been with me every step of the way – prodding when I needed prodding, dispensing advice when needed, and generally guiding me along my path. I hope that he is pleased with the final results.

I would also like to thank Wendy Powley, who has helped me solve many problems during the course of my stay at Queen's. Without her help my journey would have been significantly more difficult. She deserves more praise that I will ever be able to bestow upon her.

Along a research vein, I would like to thank IBM Canada and the IBM Center for Advanced Studies for their support on this project.

To everyone else in the School of Computing at Queen's, I would like to express my gratitude. Irene, Debby, Lynda and Sandra have all gone beyond the call of duty to help me when needed, as have Tom, Gary, Dave and Richard when I was having computer troubles. To the softball team and those involved in lively debates in the coffee room – you are in some of my best memories of Queen's.

Last, but not least, I would like to thank my family and my wife, Elizabeth. Their love, support and sometimes gentle prodding were key in helping me complete my journey. It is their faith in me that made my journey worthwhile.

Table of Contents

Abstract i

Acknowledgements iii

List of Figures viii

List of Tables x

Chapter 1 - Introduction 1

1.1 Definitions 2

1.2 Motivation 3

1.3 Contributions 6

1.4 Evaluation 7

1.5 Organization of Thesis 7

Chapter 2 - Related Work 8

2.1 Previous Efforts 8

2.1.1 Automated Diagnosis 9

2.1.2 Automated Tuning 11

2.2 Diagnosis 14

Rule-based diagnosis 14

Model-based Diagnosis 16

2.2.1 Optimization 18

Generic Optimization 19

Dynamic Programming Model 19

Linear Programming Model 20

Queuing Network Models 21

2.2.2 Case-Based Reasoning 22

2.2.3 Expert Systems 24

2.3 How related work can apply to our problem 25
2.4 Database Benchmarks 26
Chapter 3 - Diagnosis Framework 30
3.1 Modeling the Problem 30
3.1.1 DBMS Assumptions 30
3.1.2 DBMS Diagnosis Modeling 31
3.2 Resource Model 34
3.2.1 Resources 35
3.2.2 Resource Relationships 38
3.2.3 Generating Resource Subtrees 41
3.3 Workload Model 44
3.4 Diagnosis Rules 47
3.5 Diagnosis Tree 48
3.6 The Diagnosis System 50
Chapter 4 - Building the Diagnosis Tree 51
4.1 The Initial Diagnosis Tree 51
4.2 Tuning the Tree 54
4.2.1 Interpreting the Data 57
4.3 Modifying the Diagnosis Tree 63
4.4 A Generic Tuning Tree 66
Chapter 5 - Evaluation of Diagnosis Framework 69
5.1 The Test Environment 70
5.2 The Evaluation Process 72
5.3 Typical Tuning Scenarios 77
5.4 Scenario 1 – Size Increase 79
5.5 Scenario 2 – Modified Workload 91
5.6 Scenario 3 – Workload Change 95
5.7 Diagnosis Tree Lifespan 101

5.8 Summary 103

Chapter 6 - Conclusions 105

6.1 Contributions 106

6.2 Future Work 107

References 112

Appendix A - Test Environment 119

Appendix B - TPC-C Benchmark 120

Appendix C - DBMS Resources 123

Appendix D - Performance Data Collected 125

Appendix E - Glossary of Terms 127

Appendix F - Confidence Intervals 131

Appendix G - Performance Monitor Database Schema 133

Appendix H - Decision Database Schema 136

Appendix I - Forward and Reverse Resource Trees 140

Appendix J - Statistical Analysis Data 145

Vita 148

List of Figures

Figure 1 - Example decision tree. 16

Figure 2 – Overview of the Diagnosis Models 33

Figure 3 - The Quartermaster Architecture. 34

Figure 4 - ER diagram of the formal resource definition. 37

Figure 5 - ER diagram of the resource model. 40

Figure 6 - An example resource model. 40

Figure 7 - An example of a generated forward resource tree. 43

Figure 8 - An example of a generated reverse resource tree. 44

Figure 9 - ER diagram for the workload model. 46

Figure 10 - Sample Diagnosis Tree. 49

Figure 11 - The initial diagnosis tree 52

Figure 12 - The impact of I/O cleaners on performance. 58

Figure 13 – The impact of deadlock check time on performance. 61

Figure 14 - The impact of the changed pages threshold resource on performance. 62

Figure 15 - The tuned diagnosis tree. 64

Figure 16 - The effects of the Log Buffer Size on performance. 66

Figure 17 - Proposed generic diagnosis tree. 67

Figure 18 - Evaluation Process 73

Figure 19- Throughput results for the original workload on a small database 84

Figure 20 - Reverse resource tree with Locklist Size as root. 87

Figure 21 - Reverse resource tree with I/O Cleaners as root 88

Figure 22 - Throughput results for the diagnosis of the original workload on a large database. 90

Figure 23 - Throughput results for the diagnosis of the modified workload on a small database 93

Figure 24 - Throughput results for the diagnosis of the modified workload on a large database. 95

Figure 25 - The effect of sort overflows on sort query response time. 97
Figure 26 - Throughput results for the changed workload on a small database. 99
Figure 27 - Throughput results for the changed workload on a large database. 101
Figure 28 - TPC-C table schema. 121
Figure 29 - Confidence interval equation. 131
Figure 30 - Forward resource tree for the number of I/O cleaners resource. 140
Figure 31 - Reverse resource tree for the number of I/O cleaners resource. 141
Figure 32 - Forward resource tree for the deadlock check time resource. 141
Figure 33 - Reverse resource tree for the deadlock check time resource. 141
Figure 34 - Forward resource tree for the lock timeout resource. 142
Figure 35 - Reverse resource tree for the lock timeout resource. 142
Figure 36 - Forward resource tree for the locklist size resource. 142
Figure 37 - Reverse resource tree for the locklist size resource. 143
Figure 38 - Forward resource tree for the sort heap size resource. 143
Figure 39 - Reverse resource tree for the sort heap size resource. 143
Figure 40 - Forward resource tree for the sort heap threshold resource. 144
Figure 41 - Reverse resource tree for the sort heap threshold resource. 144

List of Tables

Table 1 – Example resources for diagnosis tree tuning. 55

Table 2 - Statistical analysis of test workload. 71

Table 3 - DBMS resource values. 75

Table 4 – Tuning strategy 76

Table 5 - DBMS tuning scenarios 77

Table 6 – Transaction frequencies for the original and modified workloads. 78

Table 7 - Diagnosis of the original workload on a small database 80

Table 8 - Diagnosis of the original workload on a large database. 90

Table 9 - Diagnosis of the modified workload on a small database. 92

Table 10 - Diagnosis of the modified workload on a large database. 94

Table 11 - Diagnosis of the changed workload on a small database. 98

Table 12 - Diagnosis of the changed workload on a large database. 100

Table 13 - TPC-C data relations 121

Table 14 - Transaction requirements 122

Table 15 - Data used for standard deviation calculation. 132

Chapter 1

Introduction

A **D**ata**B**ase **M**anagement **S**ystem (**DBMS**) is an application that allows a user to create, access, and maintain a collection of related data. A DBMS is a complex system that is composed of a collection of subsystems, each with a specific task. It is the job of the DBMS software to control each of these smaller subsystems during the life of a database. Due to the inherent competition for system resources, it is understandable that achieving a high level of performance from a DBMS is a difficult task. System resources are allocated for use by the DBMS through DBMS resource settings. The initial difficulty confronted when tuning a DBMS is determining which of the numerous resources need to be adjusted in order to solve the performance problem. In this dissertation, we consider the difficulties associated with diagnosing DBMS performance problems and propose a method for automating the diagnosis process.

1.1 Definitions

In order to discuss the various issues involved with DBMS performance, several concepts must first be introduced.

Resource – A resource is a piece of software or hardware that is in limited supply. An example of a hardware resource is the physical memory in the system. An example of a software resource is a logical limit placed on the number of allowed concurrent processes.

Resource Tuning – Resource tuning is the process of determining how to adjust the setting for a particular resource in order to alleviate a bottleneck in the DBMS. Determining how to adjust a resource involves knowledge of how that particular resource affects the running system as well as how adjusting that resource affects other resources.

DBMS Tuning – DBMS tuning is the process of increasing or decreasing the performance of the DBMS by altering the amount of physical and logical resources available to the DBMS.

Diagnosis – DBMS diagnosis is the process of determining which of the database resources needs to be adjusted in order to solve a performance problem. Once the offending resource has been identified, we perform resource tuning to determine how to adjust the problem resource.

1.2 Motivation

A DBMS has the responsibility for accessing and maintaining large amounts of data. Maintaining data integrity and supporting concurrent users introduces a significant amount of overhead to a DBMS. This overhead decreases the ability of the DBMS to serve the data to the users quickly. We must decrease overhead while maintaining data integrity and providing information to the users as quickly as possible.

DBMS performance is regulated by adjusting DBMS resource parameters. The large number of tuning parameters and the complexity of workloads makes achieving and maintaining peak DBMS performance a non-trivial task [SHI00] [CHA00] [WEI94]. **D**ata**B**ase **A**dministrators (**DBA**s), who are the people with the knowledge and expertise needed to tune DBMSs, are scarce and expensive to employ [CHA99] [LOM99] [WEI94].

The process of DBMS tuning can be broken down into two distinct tasks: diagnosis and resource adjustment. Diagnosis involves determining which of the resources in the DBMS is responsible for the performance problem. Resource adjustment involves altering the settings for a particular resource (and others that may be related to it) in order to achieve better performance. Resource adjustment is also referred to as "resource tuning". As databases increase in size and complexity, the ability to manually control performance becomes "impractical" [BRO94] [BRO95]. Several calls for the automation of the diagnosis and tuning processes have been made in recent years [BER98] [BRO94]

[CHA00] [CHA99] [LOM99] [MAR00] [WEI94]. Automation would allow the DBMS to quickly achieve peak performance without any human interaction.

It is important to clarify that two different levels of "tuning" exist for DBMSs. In one case, DBMS resources are adjusted in order to increase or maintain performance. In the other case, performance tuning consists of application optimization, data placement concerns, hardware issues and other factors external to the DBMS. This dissertation will focus on the adjustment of DBMS resources as the method of affecting performance.

DBMS tuning involves the collection and analysis of DBMS performance statistics in order to determine the cause of the performance problem [IBM00]. The statistics collected may be simple to read and understand, or they may need to be calculated from other data and then analyzed. It is a time-consuming task for a DBA to analyze the large amount of performance data that can be collected from a running DBMS. A DBA must narrow down the amount of data to be analyzed by considering the type of performance problem and then rule out some of the resources.

By automating the analysis of the performance data, it is possible to consider a large amount of data in a very short period of time. Automatic diagnosis should lead to a more thorough inspection of all of the data while quickly producing a list of possible culprit resource allocations.

The inevitable increase in hardware performance will ultimately lead to more powerful computers embedded in various systems and Internet appliances. Appliances that store and manage information will be candidates for embedded DBMSs. The interfaces associated with such Internet appliances will not likely provide the option for adjusting DBMS parameters, so the underlying DBMS in such a device will have to be self-managing [BER98].

A DBMS is a natural choice as an interface to provide large amounts of data on the Internet. Unfortunately, the Internet does not provide a stable workload for a DBMS. The workload changes as the number of people browsing increases and decreases throughout the day. It is impossible for a DBA to tune a database quickly enough to keep up with a consistently shifting workload. Automating the diagnosis and tuning processes will enable the DBMS to dynamically manage the available resources in these situations.

Cost is another consideration in the quest for a self-managing DBMS. DBAs are expensive to hire, even for short durations. A full-time DBA is a heavy burden for small and medium-sized businesses. Automation of the tuning process can remove much of the need for a DBA. It can also mean less hardware cost since the DBMS can make the best use of available resources. At present, many companies use overpowered machines to run their DBMSs in order to compensate for inadequate performance tuning. An overpowered machine is able to support mediocre performance tuning while handling shifting workloads with the extra hardware resources. Automatic tuning provides better usage of the hardware resource, thereby eliminating the need to buy an overpowered machine.

1.3 Contributions

An algorithm is proposed to automatically diagnose DBMS performance problems. The algorithm uses a diagnosis tree and a resource model along with hardware and workload models to diagnose resource bottlenecks. The diagnosis algorithm is constructed as part of the Quartermaster framework, which is a goal-oriented framework for diagnosing and tuning DBMS resources [BEN99]. Such a diagnosis and tuning framework can be applied to other types of software systems where performance is an issue.

The contributions of this dissertation are the following:

- a formal description of the DBMS diagnosis problem;
- an analysis of the complexity of the diagnosis problem;
- the development and implementation of models to demonstrate successful automation;
- the development of a generic diagnosis system that can be adapted to other software systems
- a systematic experimental evaluation of our approach as compared to an experienced DBA and the DB2 Tuning Wizard

This dissertation shows that the collection of underlying performance data can be used to diagnose performance problems, allowing an automated system to manage and control the resources.

1.4 Evaluation

The diagnosis model was implemented and tested using IBM's DB2/UDB. The throughput resulting from each complete diagnosis was compared to various throughput values obtained by using the DB2 Tuning Wizard, a performance application included with the DBMS. Although DB2 was used as an example throughout, the principles used in the creation of the diagnosis system can be applied to other DBMSs as the workloads are not inherently linked to the DB2 software.

1.5 Organization of Thesis

Chapter 2 of this dissertation describes approaches to DBMS diagnosis proposed in the literature. Chapter 3 describes the models used for DBMS diagnosis and explains how the models work together to diagnose a DBMS. Chapter 4 explains the process used to create and tune the diagnosis tree. Chapter 5 presents the results returned from testing our system on a working DBMS. Results are discussed along with the effectiveness of the diagnosis algorithm. Chapter 6 concludes the dissertation by summarizing the results of the research and presenting additional areas of research in the area of automatic diagnosis.

Chapter 2

Related Work

Related literature has provided information in two distinct areas: previous work on automating DBMS tuning and approaches to the general problem of diagnosing faults in systems. The chapter is divided into four sections. The first section reviews DBMS tuning literature, focusing on the issues of general resource management as well as tuning algorithms for specific resources. The second section introduces the area of diagnosis and presents three different approaches to diagnosing faults in generic systems. The third section discusses how the related work can apply to our proposed system. The final section presents DBMS benchmarks, invaluable tools for evaluation and tuning.

2.1 Previous Efforts

Several efforts have been made in the area of automating the control of DBMS resources. Automating resource management requires that the automated system be able both to diagnose the resource causing the performance problem as well as to properly adjust the

resource to remove the bottleneck. Each previous effort falls either into the category of automated diagnosis or automated tuning.

2.1.1 Automated Diagnosis

Automated DBMS resource management is the ultimate goal for work in the area of resource management. The following papers address the issue of automating the diagnosis process.

Chaudhuri and Weikum present the idea that the current method of controlling resources in DBMSs is outdated and that a new database system architecture must be considered [CHA00]. They argue that the present model has an overloaded feature set, a query language that is difficult to use, unpredictable performance, overly difficult tuning and various other problems. They believe that the best way to solve the problems facing DBMSs today is by recreating the database management system with a RISC-style architecture. They believe that the performance tuning problems will be solved by restructuring DBMSs into better defined components that are easier to tune. By reducing the number of components involved in the tuning process, automatic resource tuning for DBMSs will be achievable.

Hellerstein has proposed an architecture for a generic automated tuning system (ATS) that uses "a feedback control loop that is layered on top of a target system" [HEL97]. This approach recognizes the complexity associated with performance tuning for all types of computer systems. Other work by Hellerstein and others assesses the application of

"control theory to the evaluation of controllers" used for software management [PAR01]. Hellerstein also points out the desirability of having a proactive resource management system that can detect problems before they occur as opposed to a reactive system that merely fixes the performance of a degraded system.

Hart *et al* propose a method to isolate performance problems of systems where performance data is stored in multidimensional databases (MDDBs) [HAR99]. This system is designed to use performance data stored in an MDDB to determine the source of the performance problem. The proposed diagnosis system is applicable to any computer system where performance data is stored in an MDDB. The diagnosis system does not address the problem of adjusting the resources once a problem has been diagnosed.

Bigus *et al* have recently proposed a generic agent for automated performance tuning [BIG00]. The generic agent is designed to support tuning for systems where no prior knowledge is known to systems where effective resource controllers exist. The generic agent used in the automatic tuning process relies heavily on intelligent control. The test system presented in the paper depends on a neural prediction agent that learns the system model, a neural prediction agent that is adapted to determine the appropriate control settings and an agent responsible for monitoring the workload and performance. The test system is a Lotus Notes server. The automated resource management system is able to reduce the queue length in the server over time.

Weikum *et al* address the need for automatic memory management in data servers [WEI99]. The paper surveys the possible approaches to memory management such as the self-tuning of cache memory and exploiting distributed memory and speculative prefetching for data and web servers.

2.1.2 Automated Tuning

Resource management can only be completely automated if system resources can be automatically adjusted to increase system performance. Many papers and approaches exist in the area of resource tuning. This section overviews the various approaches to this problem.

Several papers exist in the area of automatic memory tuning for DBMSs. **Brown *et al*** explore the area of automatic memory management [BRO93] [BRO94] [BRO95]. The main focus of the research work presented in these papers is the relationship between the user-defined goals associated with classes of transactions and the allocation of memory resources in the buffer pool. Brown also considers adjusting multiprogramming levels instead of memory to achieve the same result [BRO94]. Brown's work approaches the idea of tuning as a goal-oriented problem where the "optimal" resource allocation is when all of the workload goals are achieved.

Chung *et al* are also interested in goal-oriented buffer pool management [CHU95]. The approach by Chung [CHU95] differs from Brown's [BRO95] in that the performance index is not calculated based on the I/O response times for transaction classes but by

measuring the response time of the buffer pool. Buffer pool performance indexes are calculated and they attempt to achieve a "lexicographically minimal performance index vector" by adjusting the size of the buffer pools [CHU95].

Martin *et al* present a "dynamic reconfiguration algorithm" to resize automatically buffer pools based on class goals set by an administrator [MAR00]. The reconfiguration algorithm takes into consideration the goals set for each of the transaction classes in the workload and uses response times as a basis for reallocating buffer pool memory. An "Achievement Index" is used to determine if a transaction class meets its goals by comparing actual response times with goal response times. Cost estimate equations are used to estimate the effect of moving memory from one buffer pool to another. Buffer pool memory is reallocated until all response times fall within a specified percentage of the required goals.

Xu *et al* approach the problem of automated memory management by addressing the issue of buffer pool configurations [XU02]. One key memory management issue is the assignment of tables and indexes to particular buffer pools. Assigning two fundamentally different tables to the same buffer pool may result in contention for memory and adversely affect performance. The approach taken involves defining a *feature vector* for each database object and then using a data-clustering algorithm to define similar groups of database objects. The resulting groups are then assigned to buffer pools that are sized appropriately, resulting in a configuration that performs as well as one designed by an expert DBA.

Chaudhuri and Narasayya approach the area of dynamic resource allocation from the perspective of automating statistics management for the DBMS query optimizers [CHA00-2]. DBMSs use statistics about the data stored in the database to determine the query plan used. Knowing which statistical information is needed is currently left to the DBA. This paper presents techniques for automatically determining which statistics are essential and which statistics are non-essential.

Agrawal *et al* investigate automating the selection of indexes and materialized views for DBMSs [AGR00]. Both indexes and materialized views can greatly increase DBMS performance if the correct set exist during query execution. Maintaining every possible index or materialized view is not possible, resulting in an incomplete set in the DBMS. Choosing the indexes or materialized views that will best serve the workload is a difficult decision. Agrawal *et al* present algorithms and an architecture that can identify a small set of candidate materialized views and indexes.

Weikum *et al* explore the subject of automated tuning systems for DBMSs with the "Comfort Automatic Tuning Project" [WEI94]. The project explores system design principles needed to create an automated tuning system for DBMSs. The paper reiterates the need for dynamically adjustable resource parameters to allow the feedback loop to adjust resources while the workload is running. Several tuning algorithms for specific problems such as load control for locking and self-tuning memory management are described.

2.2 Diagnosis

"Diagnostic reasoning requires a means of assigning credit or blame to parts of the model based on observed behavioral discrepancies" [deK92]. Using this definition of diagnostic reasoning, we should be able to use diagnostic reasoning to determine what resources are affecting the performance of a DBMS. We consider two different types of diagnostic systems – rule-based diagnosis and model-based diagnosis.

Rule-based diagnosis

Building a traditional rule-based diagnosis system for troubleshooting first involves the accumulation of data from experts [DAV92] [PAU98] [RYM92]. Empirical associations and rules about objects are created by the experts most familiar with the system at hand. This information is then used to build a rule-based diagnosis system to troubleshoot the system. Such rule-based diagnosis systems are very dependent on the device for which they are designed and require a new set of rules for each new device or version of the device. Gathering information from experts can also be a difficult task, as a large body of information and experience may be needed before a useful algorithm can be devised [DAV92]. An example of a rule-based system is XCON, an expert system used to configure DEC computers. Approximately 500 rules were needed to configure the VAX 780 computer. The number of rules increased to 6000 as additional models were added to the rule base [LUG93]. Gathering and programming rules of this type is a time-consuming and difficult task.

A variation on rule-based diagnosis involves the use of decision trees [DAV92] [RYM92]. Decision trees stem from the state space representation of some problems [LUG93]. In a state space representation of a problem, states of the solved problem are stored at each node in the tree. The tree is then traversed, using the rules stored at the nodes along with facts about the current world state to solve the problem. The tree traversal can be either goal-directed, where the goal is known and the tree is traversed to find the data, or data-directed, where the data is known and we traverse the tree to determine the goal [LUG93].

In a decision tree, rules are stored at each node. As the tree is traversed, each node is evaluated and the result determines which branch of the tree will be followed. As nodes in the tree are evaluated and branches of the tree are traversed, other branches and nodes in the tree are excluded or "pruned" during the traversal [LUG93]. Pruning the tree can quickly reduce the number of possible solutions while focusing on those solutions most likely to solve the problem. For example, consider the decision tree in Figure 1. If the decision made at Node 1 causes Node 5 to be next to be traversed, then in that single step the entire left side of the tree under Node 2 is pruned from the search space.

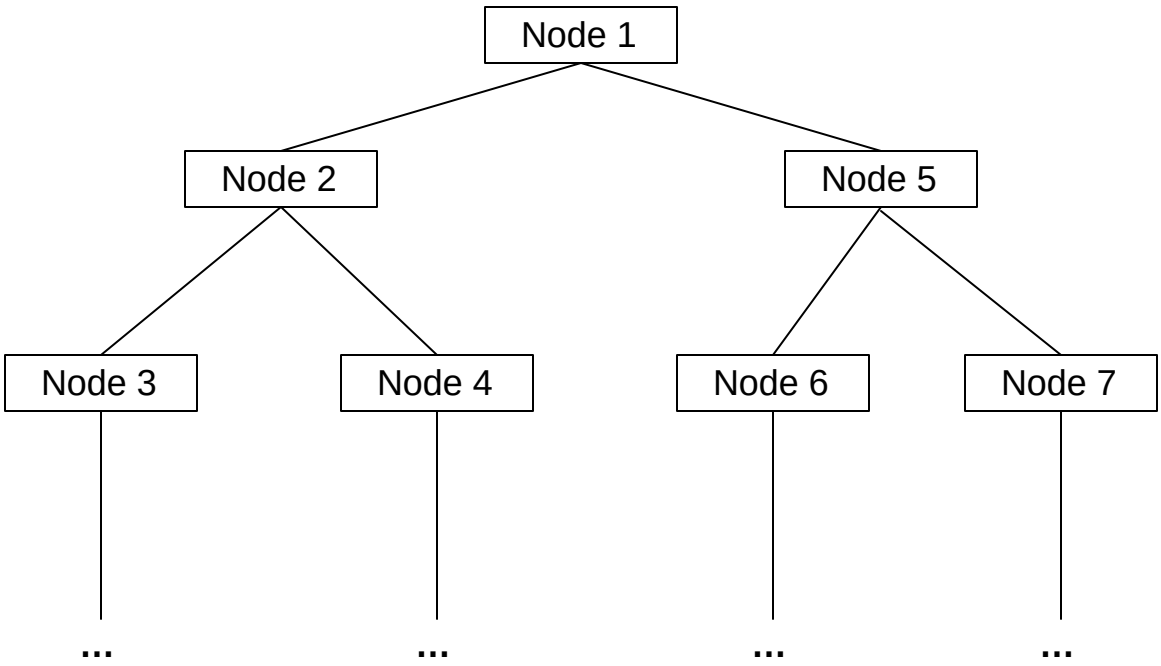


Figure 1 - Example decision tree.

Model-based Diagnosis

In diagnostic reasoning, a model of the system is used to determine what parts of the system are not performing correctly. The model of the system is presumed to be correct and any differences between the model and the actual system are used to point out malfunctions in the system. Model-based diagnosis is acknowledged as a wide ranging area [deK92]. Model-based diagnosis includes troubleshooting mechanical devices, circuits, and modeling physical or biological systems [deK92].

The primary application area for diagnostic reasoning is electronics, specifically circuits and other multi-component systems [deK89], [deK92], [MOZ91]. The key issue with systems of this nature is to find the component or components that are causing the

problem and to replace or repair them. This requires the component to be performing incorrectly, as diagnosis in a circuit is really a test of correctness. Problems in circuits can usually be traced to the malfunction of a component – such as an adder that is not adding or a broken XOR switch. Component error can be quantified and isolated, making it identifiable to the diagnostic process. This is not the case when diagnosing a DBMS. In DBMSs, resources do not have a quantifiable "broken" state; instead, they do not perform to capacity. We do not know the upper performance limit of many resources as the capacity of the resource is unknown and is dependent on many factors. Since there is no "broken" part in the DBMS, the traditional methods of testing for malfunctioning components do not apply.

When diagnosing circuits and other component-based systems, the solution to the problem usually involves the replacement of the broken component. After replacing the broken component the circuit is retested to ensure that replacement part is functioning properly. Alleviating a bottleneck in a DBMS system involves the reallocation of resources. There is no "correct" allocation for any one particular resource in a DBMS. What may be an optimal allocation for one workload and computer system may not work for a different workload or computer system.

A model of the DBMS system is needed to use model-based diagnostic reasoning for system diagnosis. Davis and Hamscher [DAV92] discuss the issue of systems that are either too simple or too complex to model. The complex end of the spectrum is bound by problems "involving subtle and complicated interactions in the device, interactions whose

outcome is too hard to predict…" [DAV92]. The relationships between resources in a DBMS are complex and not well understood. Resources can be related either by the sharing of an underlying physical resource, or by having a software dependence on another resource in the system. The complex web of relationships between physical and logical resource allocations results in the relationship between resources being unclear. This already complex model is further complicated by the workload the DBMS is expected to run. Accurately modeling the DBMS is not presently possible due to the complexities of the system.

2.2.1 Optimization

"Optimization is a technology for calculating the best possible utilization of resources needed to achieve a desired result" [EOP02]. Determining the best utilization of resources depends on the boundaries set by those trying to solve a given problem. In one case, the best utilization of resources may mean solving the problem in the least amount of time. Another case may require that the problem be solved with the least amount of resources. A third case may involve maximizing the throughput for the given problem. In all of these cases, optimization involves maximizing or minimizing one aspect of the problem such as time, throughput or resources. With respect to the DBMS resource problem, we could apply optimization techniques to maximize the throughput of the system or to minimize the amount of resources used by the system. Either method would produce the desired effect of better performance with fewer resources. Several different optimization algorithms are reviewed in here – generic optimization, dynamic programming and linear programming. Some of these optimization algorithms are very

specific to a particular problem, while others are more generic and can be applied to several different types of problems [MOL89].

Generic Optimization

Optimization usually involves finding a maximum or minimum value for the presented problem by solving a series of equations that are used to model the system [GAS75]. We must therefore first be able to create a series of equations to model the system. This is possible only if the relationships between the various resources are documented and well defined.

In the case of a DBMS system, there are simply too many resources (typically hundreds) to consider defining every interdependency between all of the resources. The complexity of creating a series of equations for a generic optimization algorithm to use far outweighs the cost associated with diagnosing the DBMS.

Dynamic Programming Model

Dynamic programming is a technique for solving many different types of optimization problems [CUR97]. Dynamic programming was introduced by Richard Bellman in 1957 [BEL57]. He introduced the idea of the *Principle of Optimality* that states:

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision"

Bird and de Moor [BIR93] state that dynamic programming can be used to solve an optimization problem if the solution to that problem is composed of optimal solutions to subproblems. This requires that the initial problem be divided into smaller subproblems that have optimal solutions. In general, dynamic programming is used where a sequence of decisions is needed to solve a particular problem. By computing various solutions to smaller subproblems, dynamic programming reuses the solutions to various subproblems as a way of avoiding unnecessary computation [CUR96].

We are not able to divide the initial DBMS diagnosis problem into smaller, solvable optimization problems because of the level of interaction between the various DBMS resources. It is not possible to find the optimal allocation for one resource without taking into consideration the allocations of other resources. Without the ability to break the larger problem into smaller, solvable subproblems, the dynamic programming solution is impractical.

Linear Programming Model

Linear programming problems are a subset of general mathematical problems in which the description of the *mathematical model* of the problem can be stated using linear equations [GAS75]. Linear equations are those equations which, when plotted on a graph, are straight lines. Linear programming was developed and introduced by American George B. Dantzig in 1947 as a method for solving linear problems presented by the U.S. Air Force. Once introduced to the world, it became clear that linear programming could be used for a wide range of production and optimization problems. The original simplex

method that was introduced by Dantzig has since been replaced with a faster method presented by Naranda Karmarker in 1983[COL00].

The strength of the linear programming method is its ability to find the optimal solution quickly. Linear programming is able to handle large numbers of variables (in the thousands [COL00]) and still produce a solution in a reasonable amount of time. Although this method may seem tempting for our diagnosis problem, the problem in using this method lies not in its ability to solve a linear system of equations, but in being able to generate the appropriate system of equations. The system of equations must define all of the relationships between the various DBMS resource, a task that is presently too complex to complete.

Queuing Network Models

Queuing network models have been used for many years to predict the effects of changes to a computer system. These models are able to estimate the impact of hardware, software, and load changes on a particular system. The amount of CPU time needed to process a queuing network model algorithm is very small, and the results are available quickly [LAZ84].

Information about the workload components is needed to use a queuing network model. For each workload component, the system load for that component and the resource demands for that component are needed [SEV81]. Although queuing network models work well for small and mid-range problems, very large systems with diverse workload

components become problematic, forcing approximate solutions to be returned from the algorithm.

One of the issues with using queuing network models for predicting the performance of a DBMS involves the inputs used by the queuing network model. Queuing network models are able to predict the performance of a system when additional hardware such as disks or CPUs are added. The queuing network model deals specifically with the interaction between the workload and the hardware resources, allowing the model to predict the effect of adding additional hardware. The queuing network model does not deal with two key issues in database system performance, namely the relationships between the transactions in the workload and the relationships between the transactions and the data. In order to tune a DBMS we need to increase the ability of the DBMS to process particular queries, and not just increase the performance of the system in general. Queuing network models do not have the granularity needed to predict the performance of the queries within the DBMS – they are only able to predict the performance of the system as a whole [SEV81].

2.2.2 Case-Based Reasoning

Case-based reasoning is a method of solving problems using the specific knowledge from problems that have already been solved [AAM94]. Case-based reasoning does not use generalized rules derived from previous solutions, but uses information derived from actual stored cases of previously solved problems [LEA96]. Case-based reasoning algorithms recommend a plan of action by matching the new problem to other previously

encountered problems. It is assumed that knowledge of how the previous case was solved will be helpful for solving the problem at hand.

Generally, case-based reasoning algorithms must go through four steps: retrieve, reuse, revise and retain [AAM94]. In the retrieve step, the cases that are most similar to the present problem are retrieved from the collection of cases. In the reuse step, information from these retrieved cases is used to help solve the problem at hand. During the revision step, the proposed solution is checked for accuracy to make sure that it actually does solve the problem at hand. If minor revisions are needed, they are implemented to achieve the desired goal. Once the solution has been revised, it is then retained as a new case in the collections of cases. This allows the algorithm to retrieve this case, along with others, if a similar problem reoccurs.

Case-based reasoning is based on two assumptions: similar problems have similar solutions, and the same types of problems reoccur in a system [LEA96]. It is possible for a single DBMS problem to be caused by several different resource allocations. This means that a single DBMS performance problem may have multiple solutions. DBMS diagnosis does not follow a basic assumption needed for case-based reasoning, that similar problems have similar solutions. It may be difficult for a case-based reasoning method to differentiate between the various solutions to determine the one that is the best.

Case-based reasoning does hold some possibility for DBMS diagnosis. Reusing knowledge from previous solved problems can help to narrow down different

performance issues and may be a benefit for diagnosis. Case-based reasoning may be possible if the previously mentioned problems can be overcome.

2.2.3 Expert Systems

Expert systems often make use of extensive knowledge bases to solve a problem [LUG93]. The knowledge base is a collection of rules and other information collected from human experts in the subject at hand. Each knowledge base usually covers a specific domain, allowing the expert system to focus on a narrow set of problems. All of the information in the knowledge base is extracted from humans; expert systems do not learn from their experiences, they only make decisions based on their present knowledge [LUG93].

One key problem associated with expert systems is the quality of the "expert" knowledge and the heuristic algorithms used to interpret the data and the knowledge in order to calculate the output of the system. The quality of the knowledge and the heuristics is related to how well defined the subject area is and how well it is understood. In the well-defined area of VAX computer hardware configuration, the XCON expert system had to maintain over 6000 rules in its knowledge base in order to properly configure several lines of VAX hardware. Experts modified up to 50% of the rules each year due to the introduction of new machines [BAC84] [LUG93].

The creation of an expert system to solve the DBMS diagnosis problem has several drawbacks. The first drawback is related to the lack of consistent expert information on

how to tune the system properly. Information found in manuals and retrieved from experts often contradicts information collected in testing and retrieved from other experts. Information on how to tune a system depends on the hardware configuration and the workload running on the DBMS. As the hardware and workload change, so does the advice given by the experts. Due to the lack of consistency in the information retrieved from experts and the variability associated with the unlimited number of hardware and workload combinations, the creation of an expert system is not possible.

2.3 How related work can apply to our problem

After studying the above approaches, we conclude that no one approach is sufficient for our problem. Optimization algorithms or a model of the DBMS for model based-diagnosis are very complex solutions. A simple rule-based system depends on associating various symptoms with a particular fault, which is not immediately possible with a DBMS. Many poor resource allocations can cause the same symptoms, meaning that the rules may be too broad and may not help in diagnosing the problem. Knowledge of the underlying system can be used to assist the rule-based diagnosis, and the addition of this information may allow such a system to effectively diagnose a DBMS. Building a knowledge base for an expert system is also complex. Finally, decision trees tend only to guide tests for the system and do not usually use system-specific information to help with the diagnosis.

We believe that a rule-based decision tree can provide an effective method for diagnosing DBMSs. The rule-based portion of the system allows us to test certain parts of the

DBMS, taking information about DBMS performance and, with knowledge of the structure of the DBMS, use the information to diagnose the system. General performance questions are usually not sufficient to diagnose such a complex system. A diagnosis tree allows us to build a picture of the "state" of the DBMS because at every point in the diagnosis tree we know how previous questions were answered. The diagnosis tree stores rules in each node and uses performance information to evaluate the rules at the node. The results of evaluation will determine the path in which the diagnosis tree will be traversed. The performance-based navigation results in ignoring some portions of the diagnosis tree in favour of other sections to me more closely scrutinized.

Our approach is similar to expert systems in that we have a "knowledge base" and an "inference engine". Our "knowledge base" is the information that we store about the database. Our "inference engine" is the code that we use to view the data and to determine which action to take. Unlike the examples of expert systems referenced in [AAM94], our knowledge base is not programmed in an IF … THEN logic rule structure. Our approach is more like the "belief networks" and "influence diagrams" found in [HOR88], but without the probabilities used in their belief networks. Our implementation does not use existing expert system shells and the traditional Artificial Intelligence languages such as LISP and PROLOG because of their scalability problems [MYL95].

2.4 Database Benchmarks

An invaluable tool when tuning a DBMS is a realistic, repeatable workload that can be used to measure performance. To insure that a realistic workload and database were used

for our experiments, we use a standard DBMS benchmark. The most popular industry database benchmark standards are maintained by the Transaction Processing Performance Council (TPC). The TPC is a non-profit corporation that was founded to create hardware and software independent benchmark standards and to publish audited performance reports [TPC2]. The TPC bases each of its benchmarks on a business model. Each of the different benchmarks is meant to mimic a business model. It is expected that when comparing systems, a user will compare the results from the benchmark that most closely resembles their business.

There are many benefits when using a TPC benchmark for performance tuning. The consistency of the workload is the first benefit, allowing multiple tests with comparable (although not identical) workloads. The clear performance metric provides a method to compare the performance from one run to the next. TPC benchmarks measure performance with a throughput or response time metric and a price/performance metric. The following sections briefly explain the business model the benchmark is modeled after and each of the performance metrics.

TPC-C:

The TPC-C benchmark is modeled after actual production **O**n-**L**ine **T**ransaction **P**rocessing (**OLTP**) workloads. The order-entry workload consists of five transactions used to simulate entering and delivering orders, recording payments, checking order status and checking the level of stock. The performance metric of the TPC-C benchmark is the number of "new order" transactions per minute that can be completed while

executing the four other transactions at a predefined ratio. The number of new order transactions per minute create the throughput performance metric called "tpmC" or "*Transactions Per Minute C*" [TPC] [TPC2].

The TPC-C benchmark also uses a price/performance metric, taking into account the total price of the system used to generate the throughput results. The price and the throughput are used to determine a dollar cost for each transaction per minute. This price then allows a consumer to compare not only the throughput performance, but also the cost associated with that performance [TPC2].

TPC-H

The TPC-H benchmark is modeled after an On-Line Analytical Processing (OLAP) application. TPC-H is designed to simulate a large database with various ad-hoc decision support queries. The ad-hoc nature of the benchmark implies that the queries are unknown to the DBMS until runtime. TPC-H benchmarks are reported for different database sizes, producing a result for each size. The TPC-H performance metric is called the "TPC-H Composite Query-per-Hour Performance Metric" (QphH@Size) [TPC2]. This composite performance metric takes into account performance values collected for the queries submitted in both single and concurrent streams. We measure the price/performance metric by determining the dollar cost per QphH@Size.

TPC-R

The TPC-R benchmark is modeled after a business reporting system. TPC-R is similar to TPC-H with the exception that it is expected that the DBMS has previous knowledge of the queries, allowing for database optimizations. The performance metric is the "TPC-R Composite Query-per-Hour Performance Metric" (QphR@Size) [TPC2] and is reported based on the size of the database. The price/performance metric is determined by the cost per QphR@Size.

TPC-W

The TPC-W benchmark is modeled after a web-based e-commerce application. Unlike previous benchmarks in which well-defined business transactions are modeled, the TPC-W benchmark simulates an internet business where web browsing and online purchasing occur. The TPC-W workload is characterized by multiple, online browser sessions, dynamic page generation, consistent objects, data contention, and transaction integrity [TPC2]. The performance metric for TPC-W is measured on the number of "Web Interactions Per Second" (WIPS) [TPC2]. The initial performance metric is based on a workload model that consists of mostly shopping. In order to model other scenarios, the TPC has also provided mainly ordering (WIPSo) and mainly browsing (WIPSb) options. As with TPC-H and TPC-R, TPC-W results are also based on a particular size. The price/performance metric used is the dollar cost per WIPS.

Chapter 3

Diagnosis Framework

We begin this chapter by defining the DBMS diagnosis problem. The diagnosis problem definition includes an explanation of the assumptions made when modeling the DBMS. We then explain the resource tree model, diagnosis tree model and workload model and why each of them is required for the diagnosis process.

3.1 Modeling the Problem

This section contains assumptions made about the DBMS and definitions used to describe the diagnosis problem.

3.1.1 DBMS Assumptions

The following assumptions were used during the development of the diagnosis system and system models.

- We assume that a DBMS has access to a limited supply of hardware and software resources. Optimal performance is a maximum amount of performance achievable

by the DBMS given the limited resources. Diagnosing a DBMS involves determining which of the resources is causing the performance bottleneck.

- Overall DBMS performance is directly related to the performance of underlying DBMS resources. Performance such as transaction throughput and response time is ultimately related to the performance of underlying resources such as the buffer pools and the input/output subsystem. Improving the performance of the underlying resources will result in improving the performance of the DBMS workload.
- We assume that the hardware used in these experiments is functioning well and is not the system bottleneck. It is expected that DBMS diagnosis will be different for the situation where the hardware, not the software, is the bottleneck.
- We assume that DBMS performance can be measured in two ways – the throughput of the workload and the performance of the underlying resources. For example, adjusting a resource may result in an increase in throughput, a measurable increase in performance. Adjusting a resource may also result in a decrease of some underlying performance measure, such as the amount of time required to do a sort. We regard such a reallocation to be beneficial to the DBMS even if there is no significant increase in throughput.

3.1.2 DBMS Diagnosis Modeling

Our approach to DBMS diagnosis and tuning involves several steps. In consultation with DBMS documentation and expertise, we create a resource model, a workload model, diagnosis rules and finally we define a diagnosis tree. The creation of the diagnosis tree is

a complex task that requires input from both the workload model and the diagnosis rules. The resource model and the diagnosis tree are then used to diagnose the working DBMS. The diagnosis produces a set of resources where tuning each resource is a possible solution to the performance problem. Tuning algorithms are used to determine the resource that will provide the greatest performance increase and that resource is adjusted. The running system is then observed and performance data is collected for the next diagnosis. This diagnosis and tuning loop continues until the diagnosis algorithm is unable to diagnose any more poorly performing resources or the system performance is determined to be adequate. Figure 2 gives an overview of the diagnosis system.

The core of the diagnosis process is the use of the diagnosis tree and the resource model. System diagnosis is accomplished by traversing the diagnosis tree. Starting at the root node of the tree, questions are posed about the performance of the DBMS. Depending on the values of particular performance indicators within the DBMS, a decision is made to traverse either the left or right branch of the tree. This continues until a leaf node in the diagnosis tree is reached. The leaf node contains a list of one or more resources that should be considered for tuning. A sample tree is located in Figure 10 on page 49.

Once the list of resources has been acquired, the resource model can then be used either to expand the list of resources to consider for tuning, or to generate a list of resources that may be affected by adjusting a resource from the current list. This is done with the use of resource trees that are generated from the resource model. The resource model is also

available to any tuning algorithm that may wish to access information about resource relationships.

The diagnosis system is designed to fit into Quartermaster, a framework for automating performance management of DBMS systems [BEN99] [MAR00]. Quartermaster supports the collection and storage of performance data and the monitoring of performance goals. An overview of the Quartermaster framework is presented in Figure 3. The "Planner" module in the Quartermaster framework is responsible for determining the resource that should be tuned to solve a performance problem. The diagnosis framework defined in this dissertation provides the Planner module.

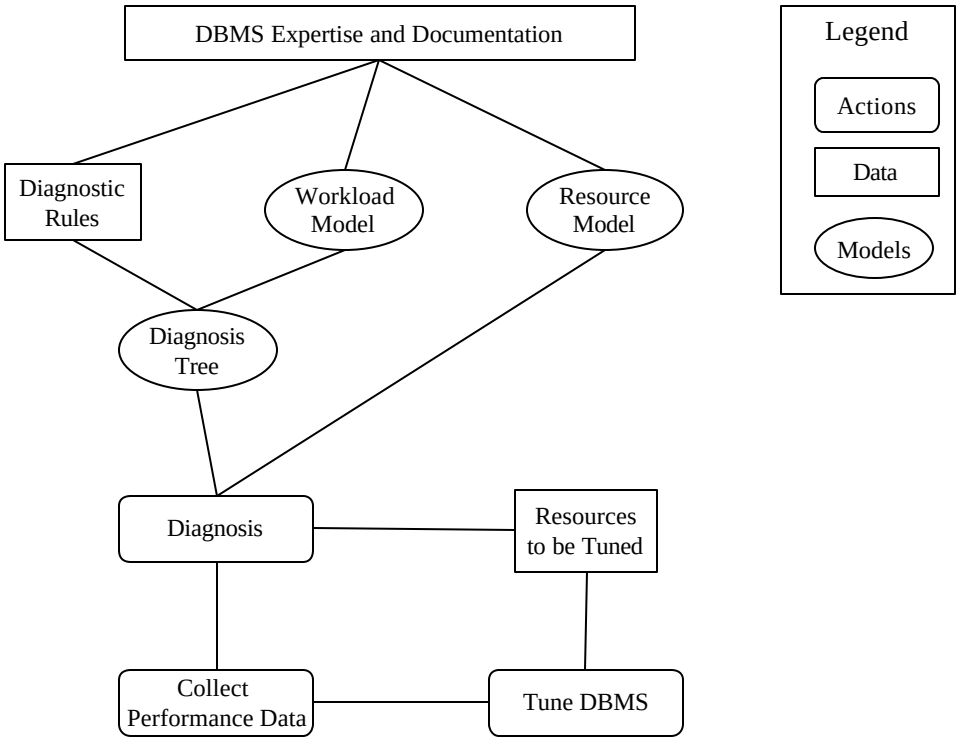


Figure 2 – Overview of the Diagnosis Models.

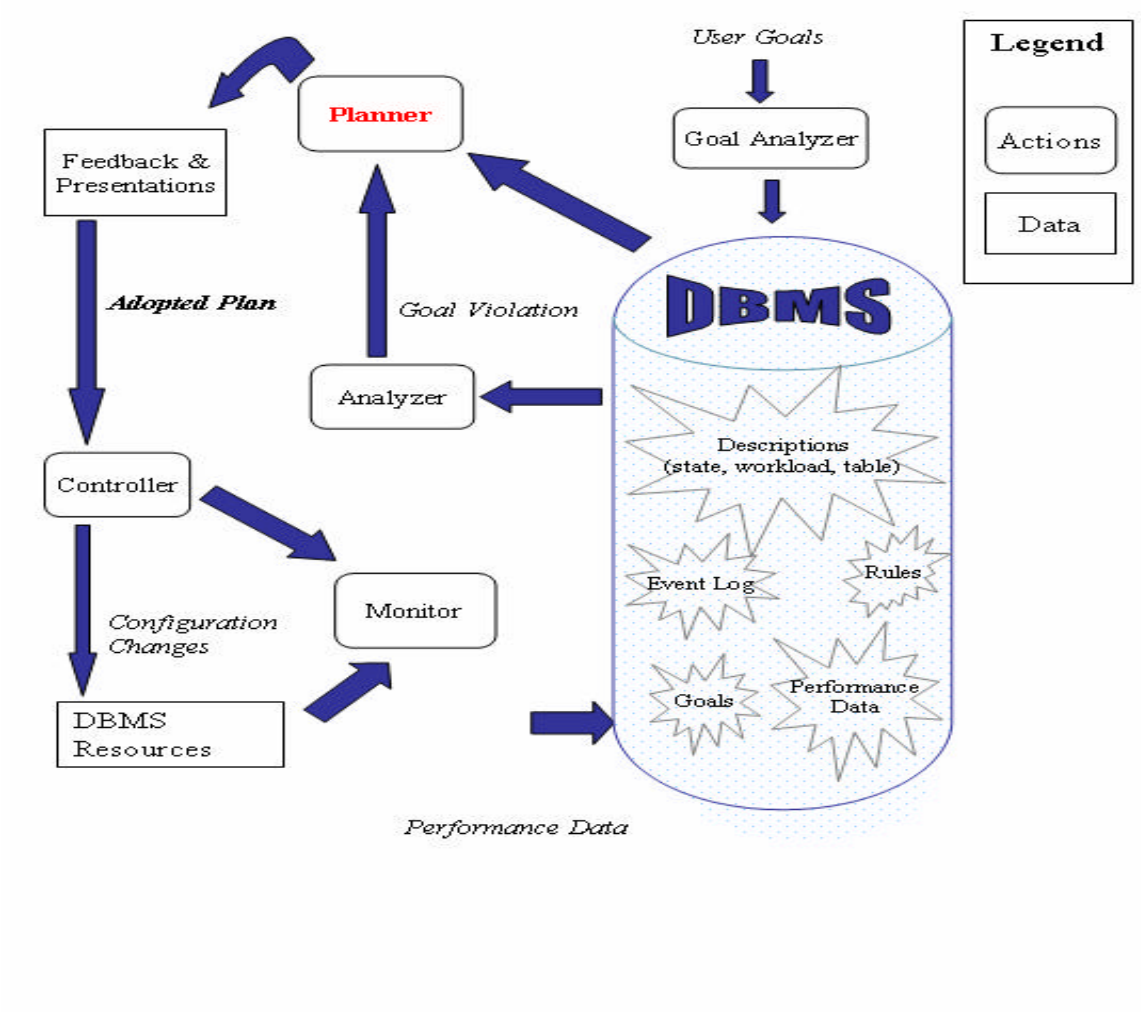


Figure 3 - The Quartermaster Architecture.

3.2 Resource Model

The resource model is a collection of information about the DBMS, DBMS resources, and the relationships between those resources. Information about the resources is gathered from several different sources. DBMS manuals provide much of the information about the DBMS and resources. This information is further expanded using other DBMS documentation and technical reference books. A more in-depth view of the DBMS resources can be produced by consultation with DBMS programmers and DBAs. Consultation allows for the inclusion of undocumented information in the model. The

resource model is used to present a consolidated view of DBMS information from many sources and make this information available to for diagnosis purposes. The focus of the resource model is to establish two types of information – information about the resources and information about the relationships between the various resources. We now consider the two types of information.

3.2.1 Resources

Resources are defined as any object used by the DBMS where the amount of the object can be adjusted. Resources are further refined into two categories – physical and logical resources. **Physical resources** are hardware-oriented resources whose allocations have a direct effect on the physical hardware. Examples of physical resources are main memory or disk space. Allocations of physical resources are limited by the hardware available from the system for the DBMS. In general, the more physical resources available to the DBMS the better it will perform. **Logical resources** are those resources that are provided by the DBMS. An example of a logical resource is the number of processes allocated to write data to disk. Logical resource allocations are limited by the DBMS. Some of the limits may be indirectly linked to the amount of physical resources available (such as the DBMS denying the creation of a process due to a lack of memory). Logical resource allocations do have an effect on physical resources, as the DBMS must use memory and CPU to maintain these processes. Some logical resources, such as the number of I/O processes, will also have an affect on system resources such as disk drive performance.

In our model, a resource in the DBMS has the following attributes:

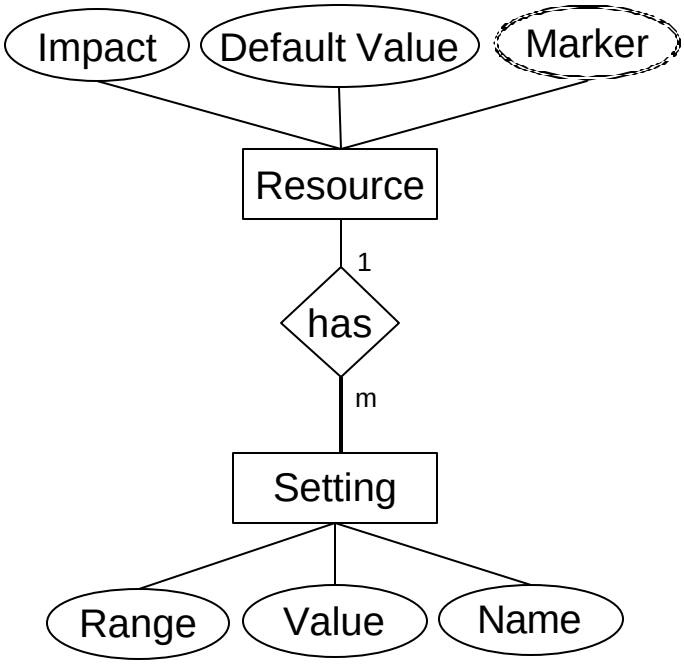
- Impact – The impact that this resource has on DBMS performance. Impact is categorized as either high, medium or low. High impact resources will have a greater effect on performance than low impact resources.
- Allowable range – The allowable (legal) range of values that the resource may be assigned. Lower and upper limits of the range are specified by the DBMS documentation. The allowable range of resource values is strictly a software or hardware limitation; the allowable range is not based on performance.
- Default value – The default value assigned to the resource by the DBMS.
- Marker values – A list of marker names and values. Markers are observed or calculated values that can be used to determine how the system is performing with respect to the resource in question.
- Setting values – A list of setting values associated with the resource. A single resource may be associated with several tuning parameters. A setting value is a tuning parameter and its current value.

A resource can be represented by a single tuple

$$R = <M, I, <S, A>, D>$$

where $M = \{M_1, M_2, \ldots, M_n\}$; M_i is a marker of the form <mname, mvalue>; I is the impact that the particular resource has on performance; <S, A> = a set of tuples $<S_i, A_i>$ where S_i is a setting of the form <sname, svalue> and A_i is the range of possible values that setting may have in the form of {set}; D is the default value assigned by the DBMS.

This formal resource definition is used to create the ER diagram of a resource (shown in Figure 4), which is the basis of the relational model used for our specific implementation.



1

Figure 4 - ER diagram of the formal resource definition.

An example of a DB2 database resource is the number of Input/Output (I/O) cleaner processes allocated in the DBMS. The I/O Cleaners can be expressed in the tuple:

num_iocleaners = < <% of async writes, 95>, <num_iocleaners, 10, {0-255}>, HIGH, 1>

In this example, the marker value used to determine the performance of the number of I/O cleaners resource is the percentage of asynchronous writes made by the DBMS. If the percentage of asynchronous writes is low, then the I/O cleaners are not properly writing dirty pages back to disk and slower synchronous writes are being used. The setting value used in this example is 10, signifying the number of I/O cleaners presently allocated in the DBMS. Increasing the number of I/O cleaners will increase concurrency but may

overload a system that cannot handle more concurrency. The impact of the I/O cleaners resource on the DBMS has been rated as HIGH by the documentation. The HIGH rating signifies that adjusting this resource can have a significant impact on the performance of the system. The range of legal values for the I/O cleaners resource is between 0 and 255. The DBMS software will allow a DBA to specify anywhere from 0 I/O cleaner processes to 255 I/O cleaner processes. Internal structures in the DBMS will not allow more I/O cleaners to be allocated, limiting the maximum number of I/O cleaners in the system. We are allowed to specify a lower number of 0 I/O cleaners for the situation where the database is read-only, removing the need to write updates back to the database and rendering the I/O cleaner processes unneeded. The final attribute for the I/O cleaner resource is the default value used by IBM when the DBMS is initially installed. The default value for this resource is 1, indicating that a single I/O cleaner process will be allocated under the default settings. We store the default DBMS resource settings as a reference point to an out-of-the-box resource allocation. All of the information used to define the tuple for the I/O cleaner resource was extracted from DB2 documentation and experience using DB2.

3.2.2 Resource Relationships

Key data required for the construction of the resource model pertains to the relationships between various DBMS resources. Relationship information is critical because adjusting the value of one resource will have an effect on those resources that are closely related to it. Information about resource relationships can be extracted from DBMS documentation or from DBA experience. DB2 documentation specifies the relationship between various

resources. Consider, for example, the I/O cleaners resource. Documentation for DB2 specifies that if the number of I/O cleaners is modified, we should also consider adjusting the related parameters buffer pool size and changed pages threshold. Buffer pool size is the amount of cache memory that we have allocated for use by the DBMS. Changed Pages Threshold is the percentage of updated pages required in the buffer pool before the I/O cleaner processes are started to write the changed pages back to disk.

Determining the relationships between various resources is important for DBMS diagnosis. Knowing that adjusting one resource may have an effect on other resources allows us to make a more knowledgeable decision when diagnosing the DBMS. It should be noted that resource relationship information is *directional.* DB2 documentation suggests that while adjusting the buffer pool size we should also consider adjusting the changed pages threshold parameter, but while adjusting the changed pages threshold the documentation does not recommend adjusting the size of the buffer pool.

The resource model can be represented by the set

$$RM = \{<R, \{E\}>\}$$

where in each tuple of the set, R in the set is a resource and $E = \{E_1, E_2, \ldots E_i\}$; E_i is a directed edge from that resource to another resource in the resource model. A directed edge from R_i to R_j indicates that a change in resource R_i can have a direct effect on resource R_j. The resource model is visually represented as a directed graph. It is possible to have two edges between a pair of nodes – one in each direction. We simplify our resource model by reducing these pairs of edges to a single edge with arrows on both

ends. Relationship information must be gathered before the resource model is constructed. Collecting resource relationship information is a one-time task for a DBMS. The ER diagram for the resource model is shown in Figure 5. This ER model is the basis for our relational implementation. Figure 6 is an example resource model.

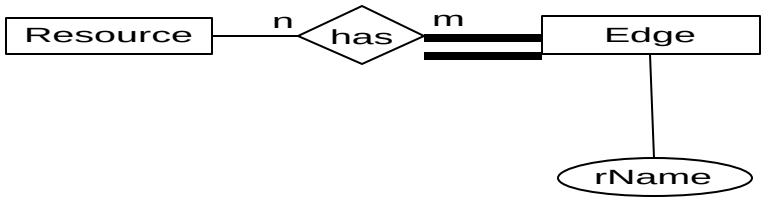


Figure 5 - ER diagram of the resource model.

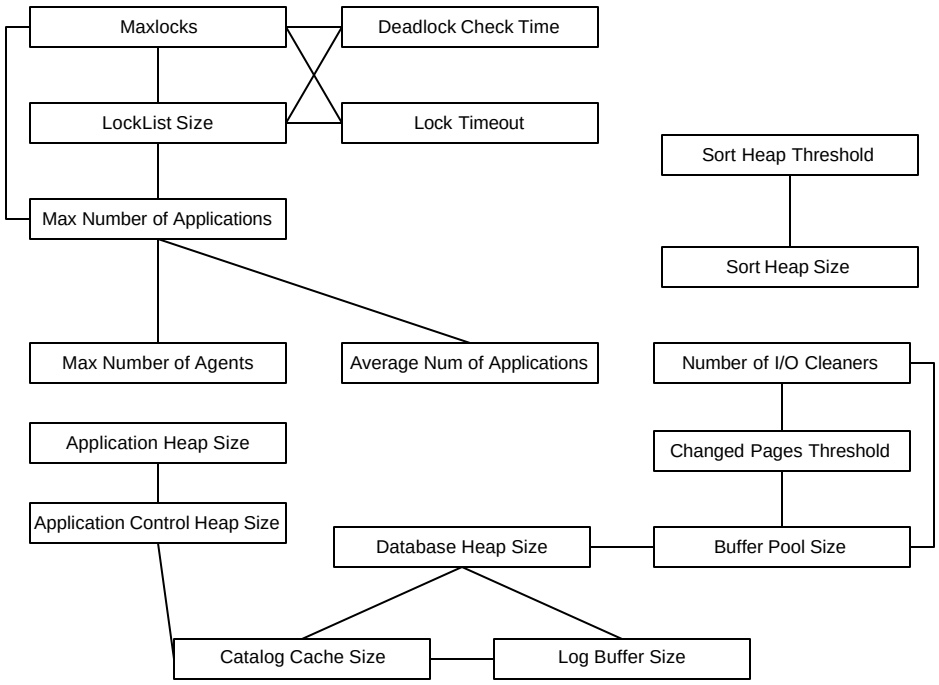


Figure 6 - An example resource model.

3.2.3 Generating Resource Subtrees

The resource model is used to store information about relationships between various resources in the DBMS. The considerable size of the resource model makes it beneficial to generate smaller resource trees when dealing with a DBMS resource. Smaller resource trees are used to simplify resource information for a given resource. The smaller resource "subtrees" can be used during the diagnosis process. Cycles from the original resource model are removed during the generation of the smaller resource subtrees. Eliminating cycles from the graph removes the possibility of a resource occurring in the subtree multiple times.

The resource model stores directional information relating to the impact of one resource on another. Knowing that one resource has an impact on another resource allows us to predict the impact of adjusting each DBMS resource. For each resource adjusted in the DBMS, we can determine the "ripple effect" this adjustment will have on other resources in the system. This allows us to predict the effects of resource adjustment. To predict the effects of resource adjustment, we can generate a **forward resource tree**. A forward resource tree will have, as the root node, the resource that is to be adjusted. Each node in the forward resource tree that is one edge away from the root will be directly affected by an adjustment of the root resource. A forward resource tree can be generated as many levels deep as desired, further determining the effect of adjusting the resource at the root of the tree.

Forward resource trees are generated by first determining a node to act as the root of the forward resource tree. Next, all of the resources that are directly affected by the root node are included in the forward resource tree, with arrows pointing from the root node to each node added to the tree. Each of these new nodes is a "Level 1" node. For each individual resource in Level 1, we determine all of the resources that are directly affected by that individual resource and include them in the tree as part of "Level 2", with arrows from the Level 1 resource to these new resources. It should be noted that any individual resource should only appear in a forward resource tree once, eliminating circular references. Duplicate resource nodes should be ignored and not included in the forward resource tree. After this process has been completed for all of the nodes in Level 1, then a full forward resource tree will have been generated to two levels. If further levels are desired, all of the nodes in Level 2 can be examined and new resource nodes can be added to create Level 3. A sample forward resource tree is found in Figure 7. The forward resource tree in Figure 7 is generated from the resource model example in Figure 6 with the Buffer Pool Size as the root node. The forward resource tree in Figure 7 is generated to a depth of two.

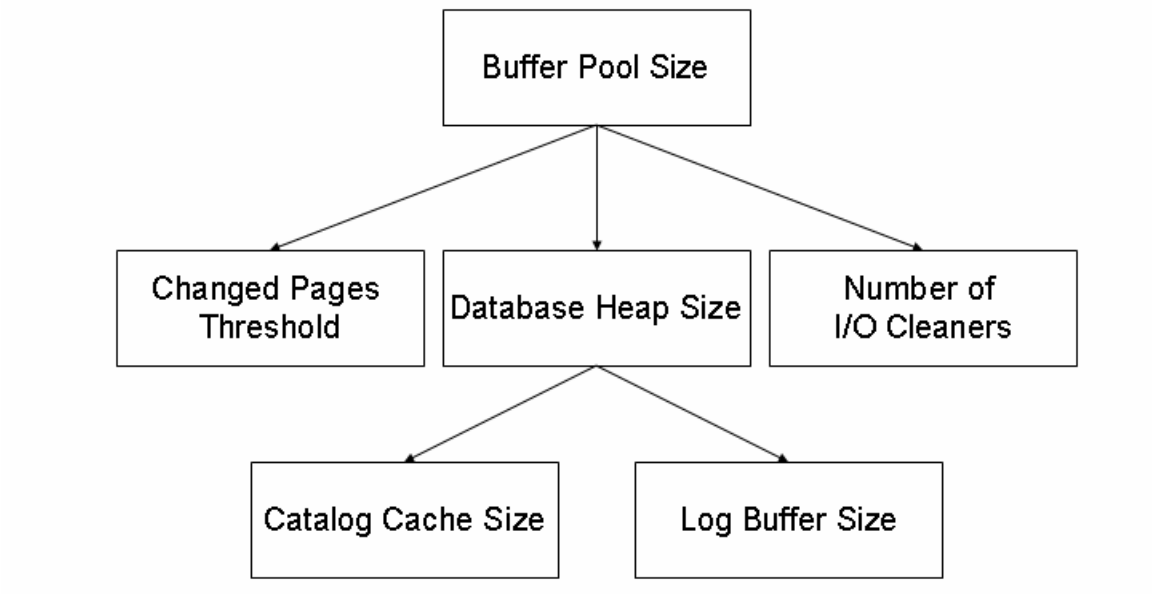


Figure 7 - An example of a generated forward resource tree.

Although the forward resource trees are useful for determining the effects of adjusting a particular resource, they do not help in determining which resource settings may cause another resource to perform poorly. For example, assume that the buffer pool resource is chosen for adjustment by a diagnosis algorithm. It may be beneficial for both the diagnosis and tuning algorithms to know which resource adjustments may have caused the buffer pool resource to perform poorly. To determine which resources may have affected the buffer pool resource, we generate a **reverse resource tree**. A reverse resource tree has the selected resource as the root. Each node one edge away from the root is a resource that, if adjusted, will have an effect on the root resource. By generating a reverse resource tree, it is possible to determine if the root cause of the performance problem is actually the resource at the root node or another resource that is causing the resource at the root node to behave poorly. Reverse resource trees are also effective in that if we have multiple resources that are being considered for tuning, it is possible to

compare reverse resource trees to see if the resources have a resource in common that may be affecting the resources considered for tuning. Figure 8 is an example of a reverse resource tree with the buffer pool resource as the root node.

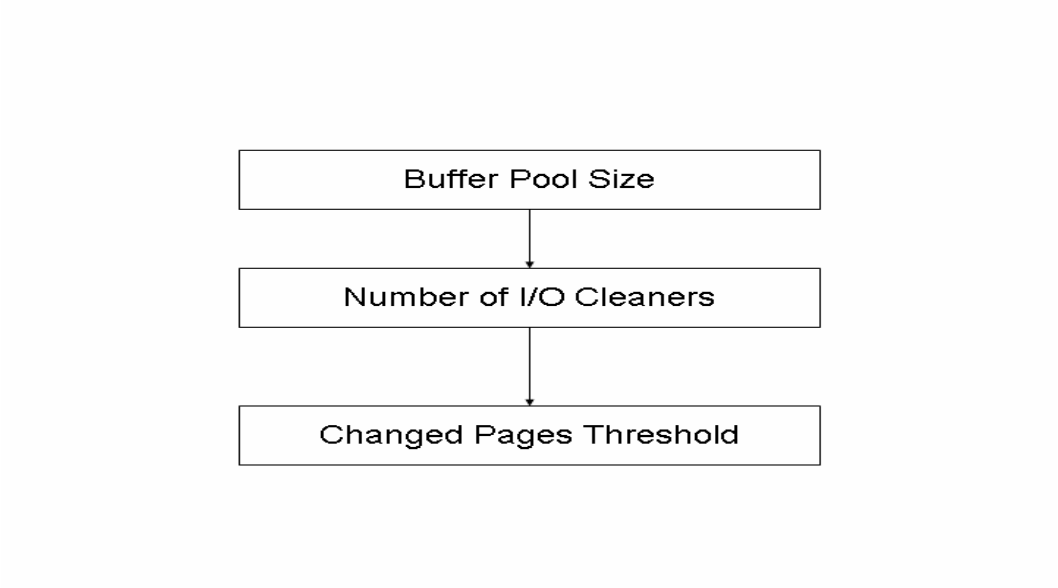


Figure 8 - An example of a generated reverse resource tree.

3.3 Workload Model

Knowledge about the system workload is vital to the proper diagnosis of the DBMS. The diagnosis tree will differ for each type of DBMS workload. Workloads are typically categorized into three different types – On-Line Analytical Processing (OLAP), On-Line Transaction Processing (OLTP), or a mixture of the two. An OLAP workload consists mainly of ad-hoc decision-support queries. These queries are CPU and time intensive. An OLTP workload consists of many short, transaction-oriented queries. OLTP workloads are characterized by the large volume of queries that are processed. A mixed workload can consist of transactions from both workload types. A workload model is needed to

provide information about the DBMS workload for diagnosis purposes. Allowing the diagnosis system to access information from the workload model separates the workload from the rest of the diagnosis system, allowing each to be updated individually.

DBMS performance information is used to create the workload model. The workload model is a collection of data concerning how the DBMS performs and reacts to a given workload. The workload model can be represented as the set

$$W = \{N, H, I\}$$

where in each tuple <N, H, I>, $N = \{N_1, N_2, \ldots N_m\}$; N_i is a resource associated with one or more sets of thresholds and indicators; $H = \{H_1, H_2, \ldots H_n\}$; H_i is a threshold in the form of <hName, hValue>; $I = \{I_1, I_2, \ldots I_n\}$; I_j is an indicator value in the form of <iName, iValue>. An indicator value is a calculated or measured performance value. Indicator values are compared to threshold values to determine if a particular resource is performing well. An example of such a tuple is for the number of I/O cleaners resource. In this case, the resource N is the number of I/O cleaners. The indicator value I is the percentage of asynchronous writes. The threshold value, H, is 95%, indicating our desire that more then 95% of the writes are asynchronous. To determine if the I/O cleaners are performing effectively, we compare the measured number of asynchronous writes to the threshold value. The diagnosis tree will determine a problem with the number of I/O cleaners if the measured number of asynchronous writes does not meet the specified indicator threshold. The ER diagram for the workload model is shown in Figure 9. The ER diagram is the basis of our relational implementation of the workload model.

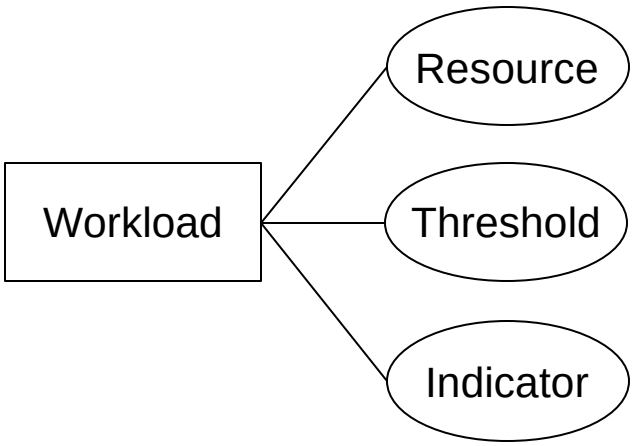


Figure 9 - ER diagram for the workload model.

Diagnosing poor performance depends on knowledge of how each resource should perform given a particular workload type. Threshold values can be determined based on the hardware and workload configurations. The performance level of each resource in the workload model can be measured while a workload is running. For a given workload, acceptable threshold values can be used to determine if a resource is performing well or not. Acceptable threshold values are determined by consulting the DBMS documentation and by observing similar workloads where the performance is known to be good.

Workload models may differ significantly for different workloads. For example, consider OLTP and OLAP workloads. Generic OLTP workloads typically do not have large sort queries, so the workload model may not contain any information about the performance of sort-related resources. If the workload were to change to include OLAP-type queries that performed large sorts, the workload model would now need to contain information

about resource performance for large sorts. The resulting workload model would differ from the original depending on the workload.

Performance information can be collected automatically from a well-tuned database and a given workload. Extracting these performance threshold values allow them to be used for similar workloads at a later point in time. Several different workload models will have to exist for the different workload types. Our initial results indicate that workload models are fairly robust and can be used for different workloads, indicating that only a limited number of workload models would need to be produced to handle most all workloads.

3.4 Diagnosis Rules

Many approaches may be used to diagnose a DBMS performance problem. Diagnosis can begin with a check on high-impact resources, a check on low-impact resources, a check on memory related resources, etc. Diagnosing and adjusting high-impact resources first will result in an aggressive tuning approach while diagnosing and adjusting low-impact resources first will result in a more conservative tuning approach. The approach used in this dissertation is neither aggressive nor conservative with no one set of resources given any preference over another.

Diagnosis rules are also used to determine which resources should be considered for diagnosis. A single performance value may be an indicator that a number of resources can be avoided for diagnosis. By reducing the number of resources checked during the diagnosis process, we can increase the speed of the diagnosis process. The process of

using a single performance value to circumvent the diagnosis of other resources is known as pruning.

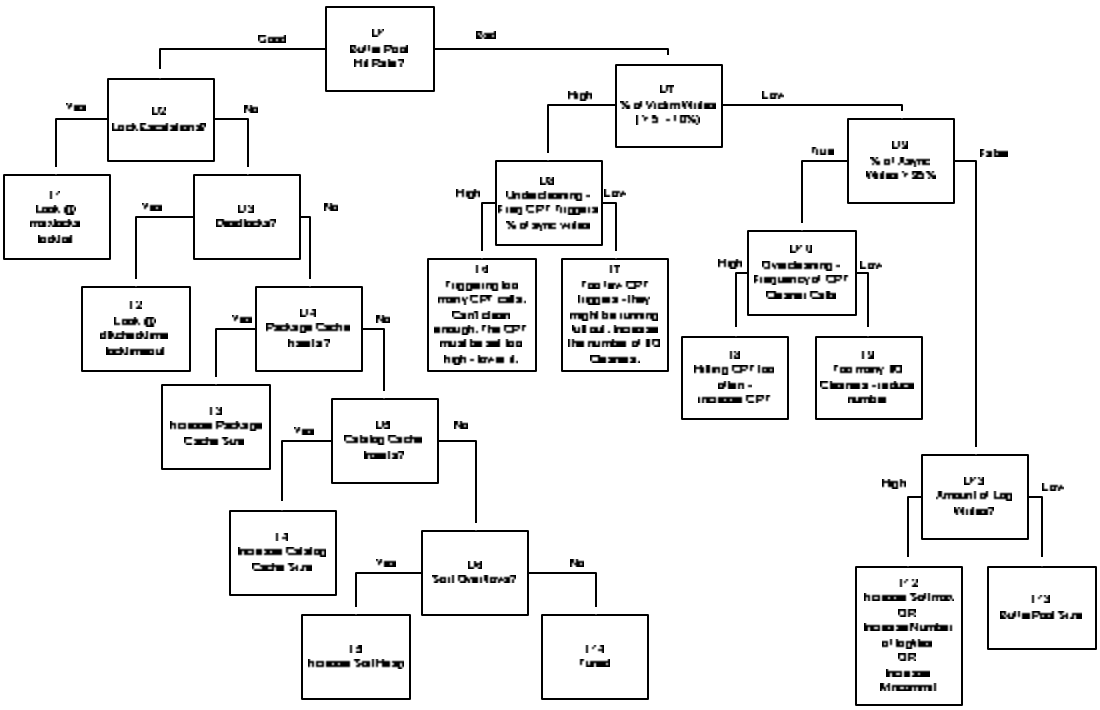
3.5 Diagnosis Tree

The diagnosis tree combines the information from the workload model and the diagnosis rules, an example of which is found in Figure 10. The combination of the diagnosis rules and the workload model into a diagnosis tree provides an easily-understood method of determining the possible resource problem. In Figure 10, non-leaf nodes are decision nodes and are labeled as D_i where *i* is a unique number for each node. Leaf nodes are tuning nodes and are likewise labeled as T_i. These labels are used to identify the nodes throughout the remainder of this dissertation.

The diagnosis tree and the resource model differ in several ways. The edges in the diagnosis tree do not hold the same meaning as the edges in the resource mode. Edges in the resource model indicate a relationship between the resources, while edges in a diagnosis tree only indicate which node should be evaluated next. The resource model stores as much information as possible about the resources while the diagnosis tree stores very little resource information, concentrating on diagnosis information. It is this extra information that is stored in the diagnosis tree that requires a separate structure to be used for diagnosis.

The diagnosis tree is an ordering of the nodes from the workload model based on the diagnosis rules. The workload model defines several resources and the threshold and

indicator values associated with those resources, while the diagnosis rules determine the order in which the resources should be evaluated. The diagnosis tree is traversed from the root node, with an evaluation occurring at each node. The result of the node evaluation determines the direction the tree traversal algorithm takes. Each decision at a node results in pruning. Pruning is the process of ignoring some portion of the diagnosis tree in favour of the further traversal of another portion of the tree. Pruning allows us to focus on key resources which appear to be causing the resource problem while ignoring others that do not seem to be causing the resource problem. Pruning does not alter the structure of the diagnosis tree – it merely eliminates some portions of the tree from the search in order to keep the search space manageable.



3.6 The Diagnosis System

The automatic diagnosis system can be represented by the tuple

$$S = <R, D>$$

where R is a resource model to represent the resources and D is a diagnosis tree used in the diagnosis process. Each component of the automatic diagnosis system, such as the resource model or the diagnosis tree, is needed to properly diagnose the DBMS. Components can be individually modified for different DBMSs.

The diagnosis system is first activated when the Quartermaster architecture determines that some performance level has not been achieved. The diagnosis tree is traversed, returning a list of one or more resources that should be considered for tuning. Information from resource trees generated by the resource model are used to either expand or possibly reduced the number of resources to be considered for tuning. Tuning algorithms are used to determine which resource will achieve the largest performance increase. Resource adjustments can be made and the Quartermaster system will continue to monitor performance to determine if the diagnosis system is further needed.

Chapter 4

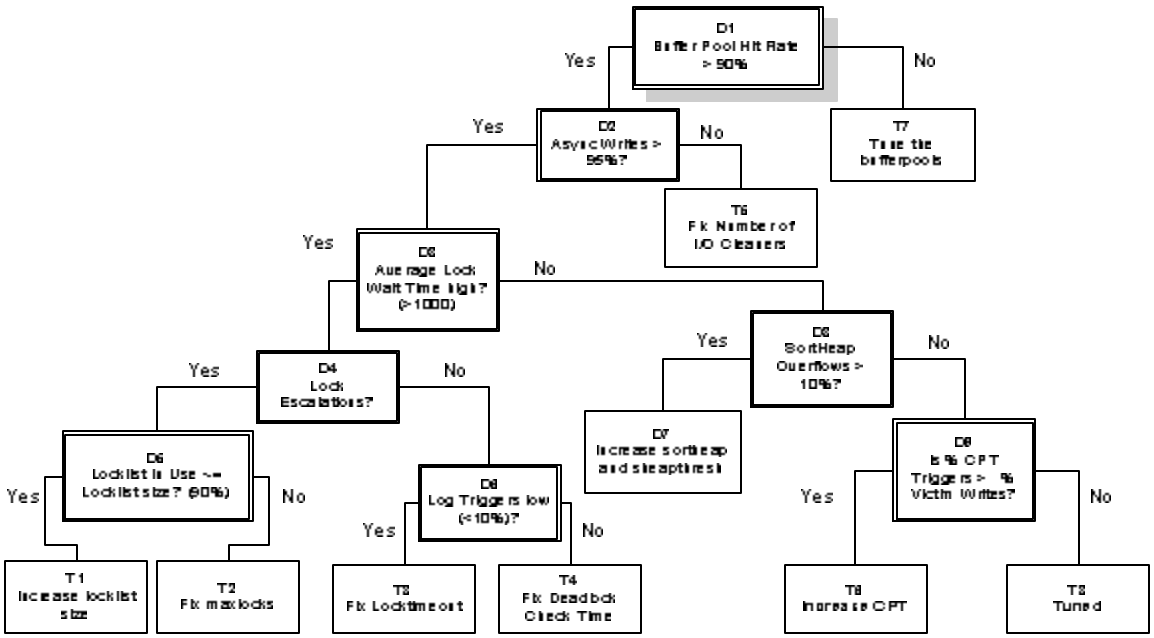
Building the Diagnosis Tree

The success of automatic DBMS diagnosis depends on the applicability of the diagnosis tree for a specific situation. The diagnosis tree is used with resource model and performance information to diagnose performance problems, as seen in Figure 2. The positioning of the decision nodes in the decision tree and the threshold values used to assess DBMS performance are key to correctly diagnosing performance problems. Creation and tuning of the diagnosis tree can only be completed after gathering information about the interactions between the DBMS and the workload. This chapter describes the process for an example diagnosis tree for an OLTP workload on IBM's DB2 DBMS.

4.1 The Initial Diagnosis Tree

The initial diagnosis tree, shown in Figure 11, is constructed based on DB2 documentation [IBM00] and information from experts. Non-leaf nodes are decision

nodes and are labeled as D_i where *i* is a unique number for each node. Each decision node is a set of one or more resources N_i, each with a list of one or more threshold values T_i and indicator values I_i. Threshold and indicator values are used to determine the performance of the resources, affecting the tree traversal decision made at decision node D_i. Leaf nodes are tuning nodes and are labeled as T_i. Tuning nodes contain a list of resources, N_i, that should be further examined by tuning algorithms for adjustment. The initial diagnosis tree was constructed from general heuristics used by a DBA when tuning a DBMS. Node D1, which refers to the buffer pool hit rate in the initial diagnosis tree, is one that many DBAs would ask when beginning to diagnose a DBMS. Following the same logic, node D2 is modeled after the next logical question a DBA would ask given the answer to the question in D1. We continued building the tree in this way with the assumption that it would model the tuning actions of a DBA.



Each DBMS is tuned differently for different types of workloads. The initial tree is designed for a general OLTP workload. The choice of resources to include in the tree is based on the impact of the resource on OLTP workload performance. High and medium impact resources are selected due to their effect on the system. High impact nodes are placed closer to the root of the tree to ensure that they are diagnosed first, while resources with lower impacts are positioned further down the tree. For example, root node D1 examines the buffer pool hit rate to determine if the buffer pool size needs to be adjusted since the size of the buffer pools has a significant effect on performance.

The ability of the diagnosis tree to diagnose the DBMS' performance effectively depends on the accuracy of the threshold values. For example, node D_1 in Figure 11 uses a threshold value of 90% to determine whether or not the buffer pool hit rate is acceptable. A hit rate of 90% is not unreasonable depending on the OLTP workload. Poor threshold values can result in performance problems being improperly diagnosed as correct or they can direct the diagnosis process towards resources that may not need to be adjusted. Threshold values vary based on the DBMS workload.

We perform a series of tests to determine appropriate threshold values for each workload. These tests are described in detail in Section 4.2. We collect DB2 performance data along with the number of transactions completed for each test. Individual tests are done for multiple allocations of each resource in the diagnosis tree. Each series of tests provides a snapshot of how the workload reacts to various settings for a particular resource.

Performance data are also analyzed to determine the best indicator variables for each resource. Indicator variables are observed or calculated values that are either directly or inversely related to the performance of the resource in question. The value of the indicator variable changes significantly when the performance of the system increases or drops. Threshold values are adjusted based on changes in indicator variables to determine if the resource is performing well. For example, in Figure 11 the indicator used to determine how well system I/O is working is the buffer pool hit rate. A low buffer pool hit rate indicates that there is a problem with system I/O. A high buffer pool hit rate indicates that system I/O is working well. We set a threshold value to distinguish between good and poor performance for the buffer pool.

4.2 Tuning the Tree

The effect of DBMS resources on the workload and the level of performance expected from the DBMS are needed to tune the diagnosis tree. We determine the effect of specific DB2 resources on performance through a series of experiments. Each resource is modified while all others are held constant and performance data is collected.

The OLTP workload used for our tests is based on the TPC-C benchmark specification [TPC]. The benchmark is discussed in Appendix B. Several portions of the benchmark specification were not implemented for this dissertation thus the results presented are not representative of published TPC-C benchmark results.

We verified the impact of each resource on DB2 by running the workload several times while varying the value of the resource. The default resource settings are determined by the DB2 Performance Wizard [IBM00]. The DB2 Performance Wizard is a tool shipped with DB2 that can be used for the initial allocation of resources. During each test all resource values are set at their "Wizard" default values with the exception of the resource being studied. Performance data is collected for each run and stored for further analysis. Table 1 includes information about the default resource values and the settings of each resource used in the testing.

Resource Name	Default Value (unit)	Values Tested
Changed Pages Threshold	80 (percent)	10, 20, 30, 40, 50, 60, 70, 80, 90
Database Heap	1215 (4k pages)	1215, 2049, 3072, 4096, 5120, 6144
Deadlock Check Time	10,000 (msec)	10,000, 20,000, 30,000, 40,000, 50,000, 75,000, 100,000
Locklist Size	295 (4k pages)	200, 300, 400, 500, 750, 1000, 5000, 10,000
Lock Timeout	Infinity (lock will never time out)	9, 10, 11, 15, 20, 30, 50, 100, infinity
Log Buffer Size	48 (4k pages)	48, 500, 1000, 1500, 2000, 2500, 3000, 5000, 10,000, 20,000
Minimum Commit	1 (count)	1, 2, 3, 4, 5, 10, 15, 20, 25
Number of I/O Cleaners	5 (process)	5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Sortheap Threshold	10,000 (4k pages)	250, 500, 1000, 2000, 3000, 5000, 10,000
Sortheap Size	256 (4k pages)	20, 25, 30, 35, 40, 50, 100, 150, 200, 250, 500, 1000
Softmax	70 (percent)	50, 75, 100, 150, 200, 250, 300, 400

Table 1 – Example resources for diagnosis tree tuning.

The DBMS workload is run for a period of 20 minutes for each individual resource setting. We consider the first three minutes of each run to be a warmup period that allows the workload to stabilize. We collect throughput data every five seconds during the entire run but ignore the warmup period when calculating average throughput values. We determined that this sample size ensures that the throughput measurements are accurate to within 25 **tpmC** 95% of the time. tpmC is the term used to specify the **t**ransactions **p**er **m**inute "**C**" for the Transaction Processing Performance Council's (TPC's) "C" benchmark. tpmC is the average number of New Order transactions processed by the DBMS each minute.

DB2 performance information was collected during each workload run. Performance information included internal DBMS counters and watermarks that are accessed through performance monitoring tools included with the DB2 software. This performance data are vital to diagnosis as it is used to determine how the internal DBMS components are working. Data is collected for two five-minute periods during each 20 minute run. The first collection period begins at the one minute mark while the second begins at the 10 minute mark. Performance data are collected at the beginning of the run to allow observation of DBMS performance during ramp-up time. It is important to collect data early in the run as poor resource allocations can cause the database to stop executing transactions during an early part of the run. Early data collection allows us to determine how the DBMS is performing in the early portion of the run for those situations where no data are collected at a later time in the run. Failure is possible later in a run if enough deadlocks occur. Performance data are collected during the middle of the run to provide

observation during a stable period in the workload. A list of the data collected is located in Appendix D.

4.2.1 Interpreting the Data

Several examples of collected performance data are presented below along with an interpretation of the data.

Number of I/O Cleaners

I/O cleaners are asynchronous processes that remove dirty (modified) pages from the buffer pool in order to make room for new pages. Without the I/O cleaners, transaction processes are forced to pause execution in order to synchronously write a dirty page back to disk. Synchronously writing dirty pages back to disk is known as a dirty page steal. Dirty page steals reduce throughput performance because the CPU is forced to suspend the execution of a transaction to make a synchronous I/O to disk. The number of I/O cleaners is a high-impact resource for non-read-only workloads. Figure 12 shows the impact on performance when the number of I/O cleaners is adjusted for the test workload. In Figure 12 the left y-axis shows the percentage of asynchronous writes made during the collection period. The right y-axis shows the throughput of the system in tpmC. The x-axis shows the number of I/O cleaners that are allocated for a run. The point shown is an average throughput value and represents the particular test run for the number of I/O cleaners.

The performance of the DBMS and the throughput of the workload are affected by the allocation of I/O cleaners. As we reduce the number of I/O cleaners below 30, the throughput drops. As the number of I/O cleaners increases over 60, the tpmC value begins to decline slightly. Performance is level between 30 and 60 I/O cleaners. The main negative performance impact occurs when too few I/O cleaners are allocated in a system. The peak in tpmC in Figure 12 shows that there is an optimal I/O setting for this configuration, as either too many or too few I/O cleaners result in degraded performance.

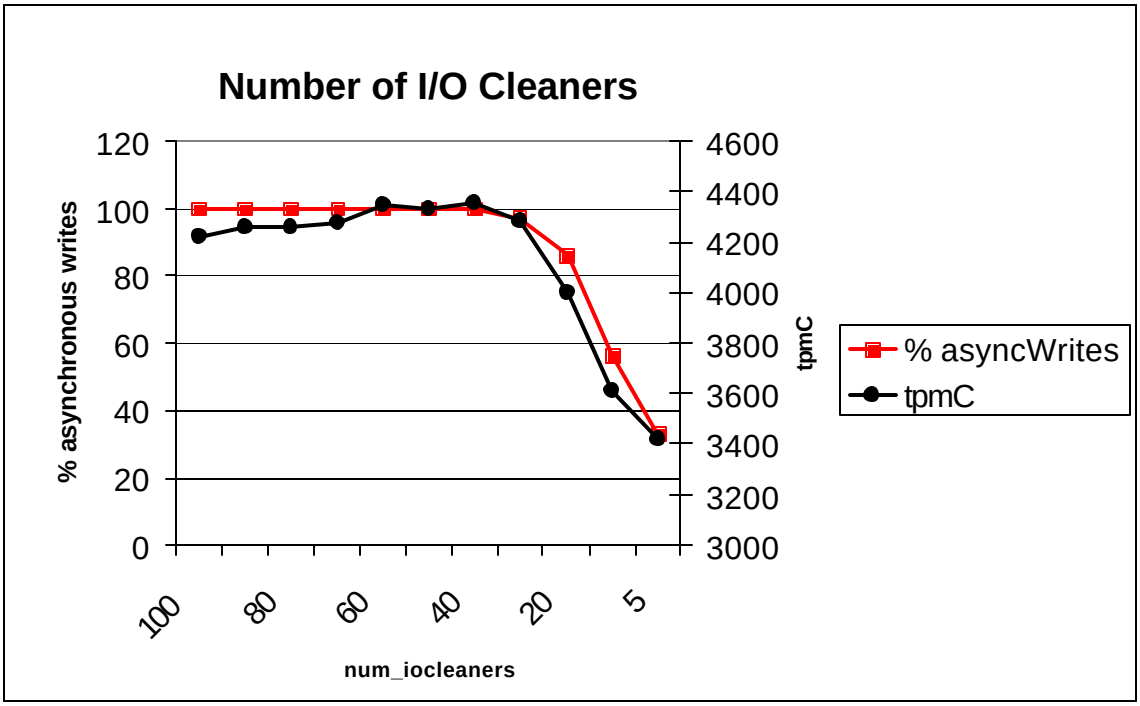


Figure 12 - The impact of I/O cleaners on performance.

We can also see that the percentage of asynchronous writes is a good indicator value for the I/O cleaners resource. The total number of writes that occur are a combination of asynchronous and synchronous writes. Increasing asynchronous writes results in a

decrease in synchronous writes. Our tests show that the peak throughput for our workload occurs where the percentage of asynchronous writes reaches 100%. The curve mapping the throughput and number of I/O cleaners matches the curve mapping the percentage of asynchronous writes and number of I/O cleaners. These matching curves show the relation between an increase in the number of I/O cleaners and an increase in throughput. Our test results show that asynchronous write levels of less than 95% are detrimental to DBMS throughput. It can be concluded that a threshold value of 95% for asynchronous writes is appropriate for this workload. An observed percentage of asynchronous writes less than 95% signifies a performance issue and indicates than the number of I/O cleaners should be adjusted.

Deadlock Check Time

The deadlock check time resource allows the DBA to set, in milliseconds, the amount of time that the DBMS waits before it checks for deadlocks. If the deadlock check time is set too low, then deadlock checking is performed frequently, which consumes CPU resources. If the deadlock check time is set too high, then deadlocked processes may wait for a long period of time before they are either resolved by the system or they time out. The effect of different deadlock check time settings on the test workload is demonstrated in Figure 13. The left y-axis shows the number of deadlocks that occur per 10,000 transactions for our workload. The right y-axis illustrates both the tpmC and the average lock wait time for the workload. The x-axis shows the values for the deadlock check time resource. We define the deadlock rate to be the number of deadlocks that occur for every 10,000 transactions. The deadlock rate is affected by several different workload and

system factors such as they amount of data contention, the number of concurrent transactions, the size of the lock list, the positioning of data within the tables and the throughput of the application. Average lock wait time is calculated from the amount of time spent waiting for locks and the number of locks issued. Calculations are made using performance information collected from DB2.

Decreasing the deadlock check time will result in decreases of the deadlock rate and the average lock wait time while increasing throughput. Deadlocks are resolved more quickly when the deadlock check time is set low. Resolving deadlocks more quickly results in a lower average lock wait time. Reducing the lock wait time has a direct effect on the throughput of the system, as time spent waiting for locks is wasted time. The CPU overhead of checking for deadlocks more frequently is offset by the time saved by resolving deadlocks. It should be noted that data contention will almost always occur in an OLTP workload with updates. It is therefore not uncommon to observe some number of deadlocks occurring in the system. For this reason, completely eliminating deadlocks from the system is not a practical option.

The presence of unresolved deadlocked transactions in the DBMS is an invitation for more deadlocks to occur since deadlocked transactions retain their locks on data objects. Figure 13 shows an increase in the deadlock rate as the deadlock check time increases. Although we measured an increase in the deadlock rate with the increase in the deadlock check time, we do not choose the deadlock rate as an indicator value.

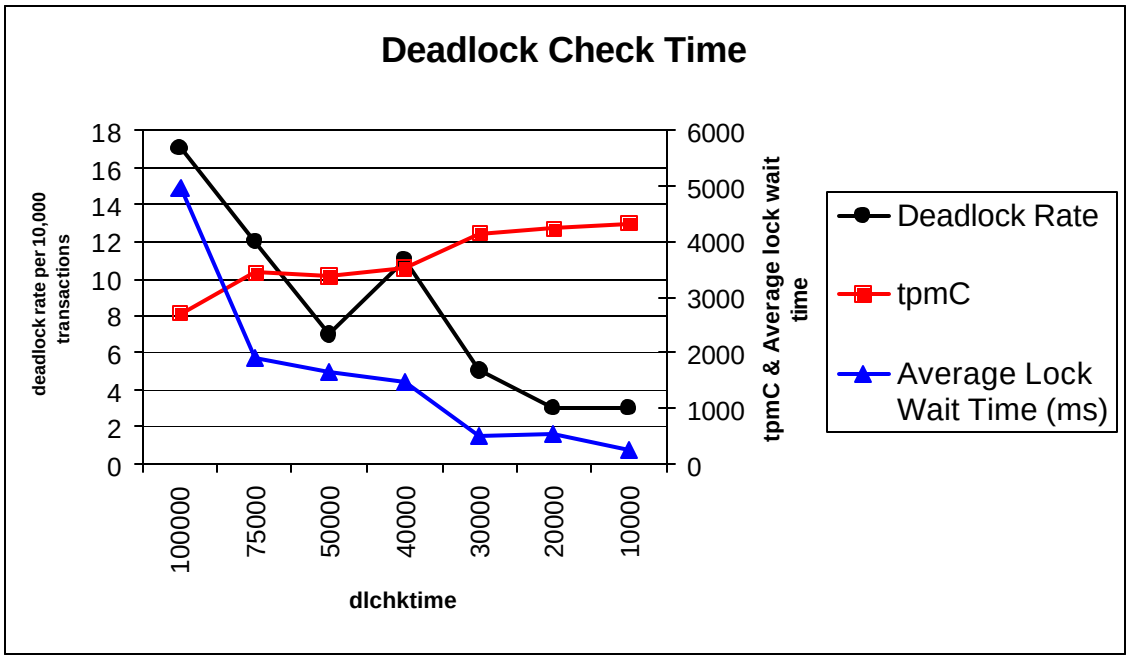


Figure 13 – The impact of deadlock check time on performance.

Deadlock rate is not chosen as an indicator variable due to the variance in the acceptable deadlock rate for different workloads. Instead, we choose the average lock wait time as an indicator variable for the deadlock check time resource. System throughput decreases as the average lock wait time increases. The best performance occurs when the average lock wait time is low. Figure 13 shows acceptable throughput when the average lock wait time is below 1000 msec. We chose a threshold value of 1000 msec for the test workload.

Changed Pages Threshold

Changed pages threshold is a value set by the user to determine when the I/O cleaners should begin to clean dirty (modified) pages out of the buffer pool. It is expressed as the percentage of dirty pages that must exist in the buffer pool before the I/O cleaners are triggered. Legal values for the changed pages threshold setting range from 5 to 99. A low

changed pages threshold may result in overcleaning the buffer pool, i.e., modified pages are written to disk and then modified again while still in the buffer pool. Overcleaning results in data pages being written back to disk multiple times, creating extra disk I/O and wasting CPU cycles. If the changed pages threshold value is set too high, then agents may be forced to execute dirty page steals because no clean pages are available. The changed pages threshold setting should be set based on the number of I/O cleaners and the workload type.

The effects of altering the changed pages threshold resource on the performance of the test workload can be observed in Figure 14. The left y-axis shows the percentage of victim writes and changed pages threshold triggers. The right y-axis shows the throughput in tpmC. The x-axis indicates the changed pages threshold setting for each data point.

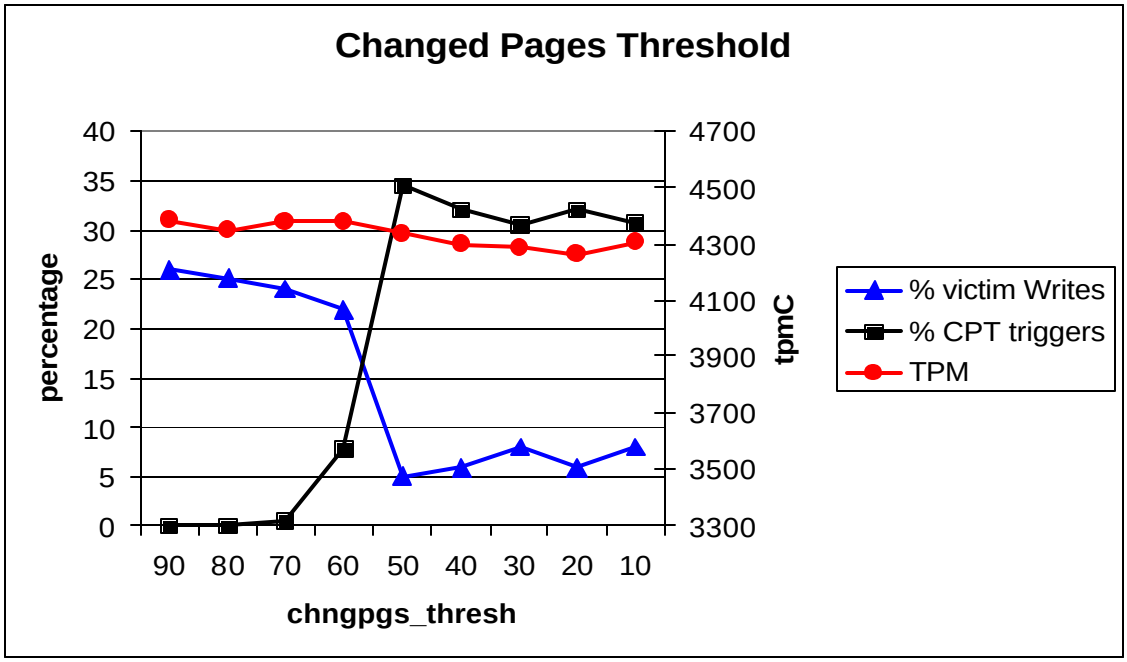


Figure 14 - The impact of the changed pages threshold resource on performance.

The impact of the changed pages threshold on the performance of the DBMS is less obvious than with the number of I/O cleaners and the deadlock check time. Figure 14 presents a case where the workload performance does not vary significantly regardless of the value of the changed pages threshold. The best throughput occurs when the number of changed pages threshold triggers is low and the number of victim writes (or dirty page steals) is between 20% and 25%. There is a decrease in performance when the percentage of changed pages threshold triggers is larger than the percentage of victim writes. These observations result in a node in the diagnosis tree that compares the percentage of victim writes to the percentage of changed pages threshold triggers. If the percentage of changed pages threshold triggers is greater than the percentage of victim writes, the diagnosis algorithm suggests an increase in the changed pages threshold value.

4.3 Modifying the Diagnosis Tree

The initial diagnosis tree in Figure 11 embodies all of the aspects of the DBMS documentation and tuning experience available during construction. The tuned tree based on our experiments is shown in Figure 15. The first difference between the tuned tree and the initial tree from Figure 11 is in the structure of the diagnosis tree. The root node in the initial diagnosis tree examines the buffer pool hit rate. Although checking the buffer pool hit rate is a logical first step in diagnosing DBMS performance, the buffer pool hit rate is only a valid measure of performance for this workload if other conditions are met. For example, the buffer pool hit rate cannot be used as a valid measure of performance if lock escalations are occurring in the DBMS. Early experiments showed that lock escalations can cause a well-configured buffer pool to have a low hit rate. Increasing the size of the

buffer pool will not increase the hit rate, as the size of the buffer pool is not the source of the hit rate problem. It is necessary to determine if lock escalations are occurring before the buffer pool hit rate can be used for diagnosis. The tuned diagnosis tree evaluates lock escalations before the buffer pool hit rate.

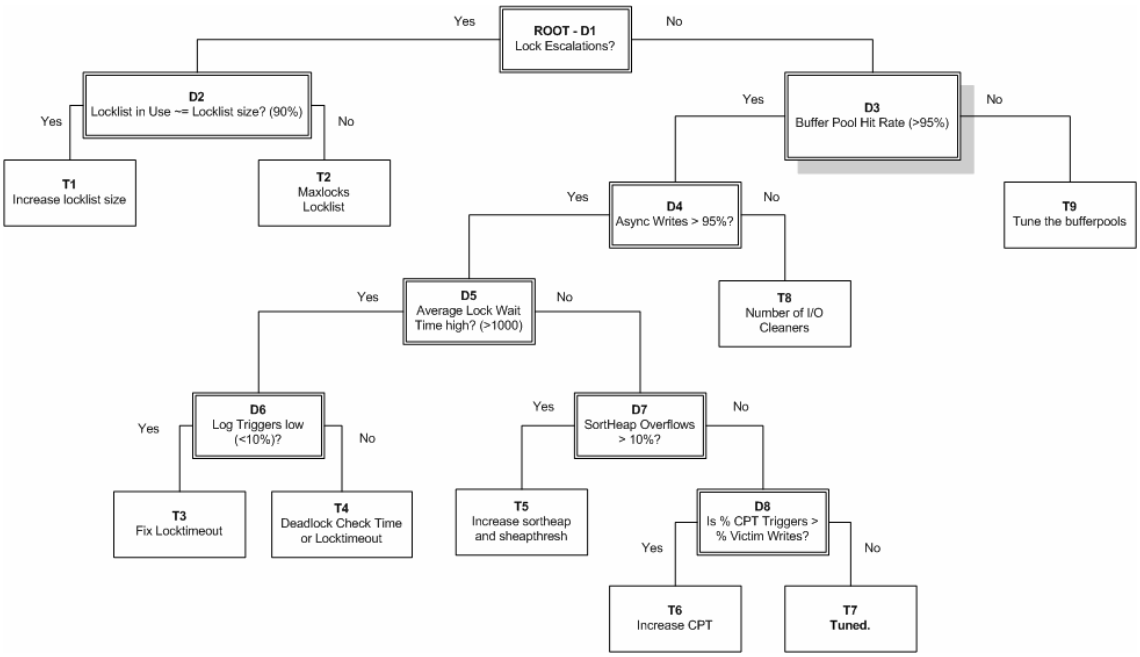


Figure 15 - The tuned diagnosis tree.

The second difference is that the tuned diagnosis tree contains more specific threshold values than the initial diagnosis tree. Initial threshold values are estimates based on little performance information. Evaluating the workload performance on the DBMS provides reasonable threshold values for the diagnosis tree. Threshold values partially depend on the tuning policy desired, as threshold values for a conservative system will differ from those for an aggressive system. A conservative system will forsake some performance for stability and will be more restrained in the modification of resource values. An aggressive

system will forsake some stability for performance and will be less restrained in the allocation of resources to gain performance. An automated system should provide the ability for individual DBAs to adjust threshold values. The tree shown in Figure 15 is somewhat aggressive in that it considers only resources with a high or medium impact on the DBMS.

A third difference between the initial and tuned diagnosis trees is that DBMS testing provided new information about certain resources. An example of this involves the changed pages threshold. Documentation for the changed pages threshold implies that it is an important resource with a high impact on the performance of the DBMS. DBMS documentation also implies that changed pages threshold triggers have less overhead than dirty page steal triggers. The results in Figure 14 show this to be false for the hardware, software and workload used and, in fact, dirty page steals are preferred to changed pages threshold triggers for this workload. This information is included in the modified diagnosis tree.

A fourth difference is the removal of some resources from the diagnosis tree. One example is the log buffer size. Although it was assumed that the size of the log buffer would have an effect on the throughput of the DBMS, the results show this is not the case for our test workload. Figure 16 shows that adjusting the size of the log buffer has very little effect on the throughput of the workload. Little or no change can be seen in the throughput, the percentage of dirty page steals, the percentage of log triggers or the percentage of asynchronous writes.

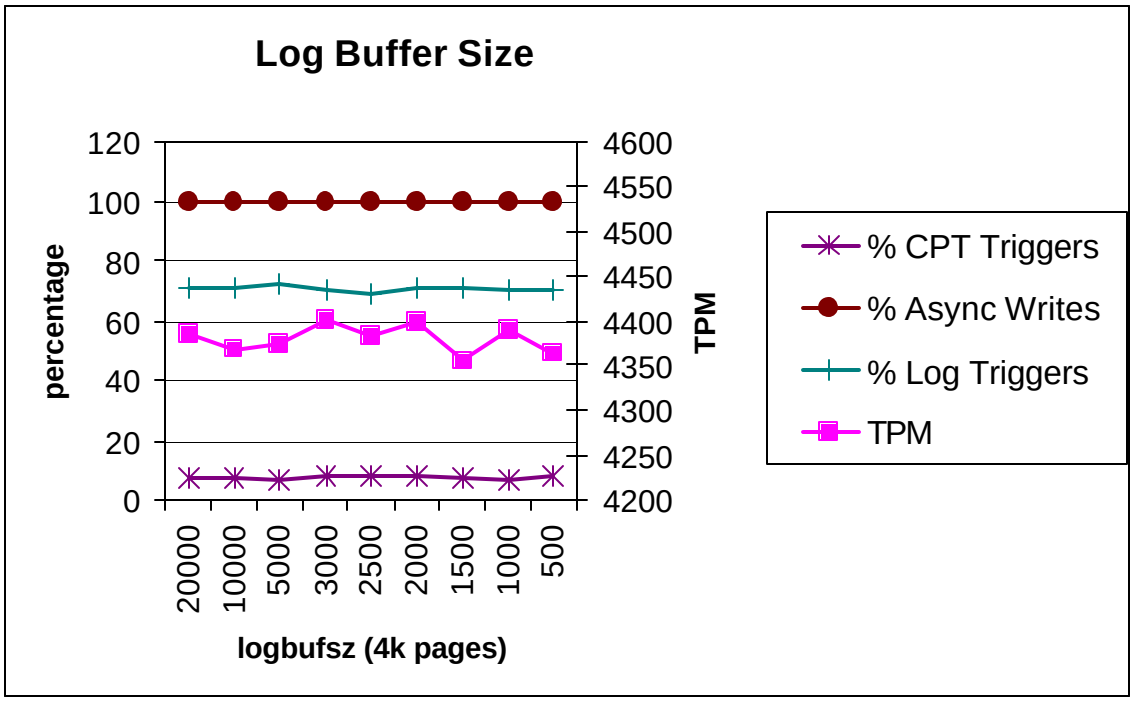


Figure 16 - The effects of the Log Buffer Size on performance.

4.4 A Generic Tuning Tree

The diagnosis tree in Figure 15 presents a good foundation for the creation of a generic diagnosis tree for OLTP workloads. Additional diagnosis nodes have to be added to provide the ability to diagnose all resource allocation problems. A more generic diagnosis tree would have the ability to diagnose the database heap size, the catalog cache size and the package cache size as well as other DB2 resources. The diagnosis tree proposed in Figure 17 is a more general diagnosis tree than the one used in this dissertation. As the generic diagnosis tree was created as a result of our testing, it was not used for any of our experiments. Further testing is needed to determine the appropriate threshold values and the tree structure is open to the insertion of new diagnosis nodes. Verification of this tree is beyond the scope of this thesis.

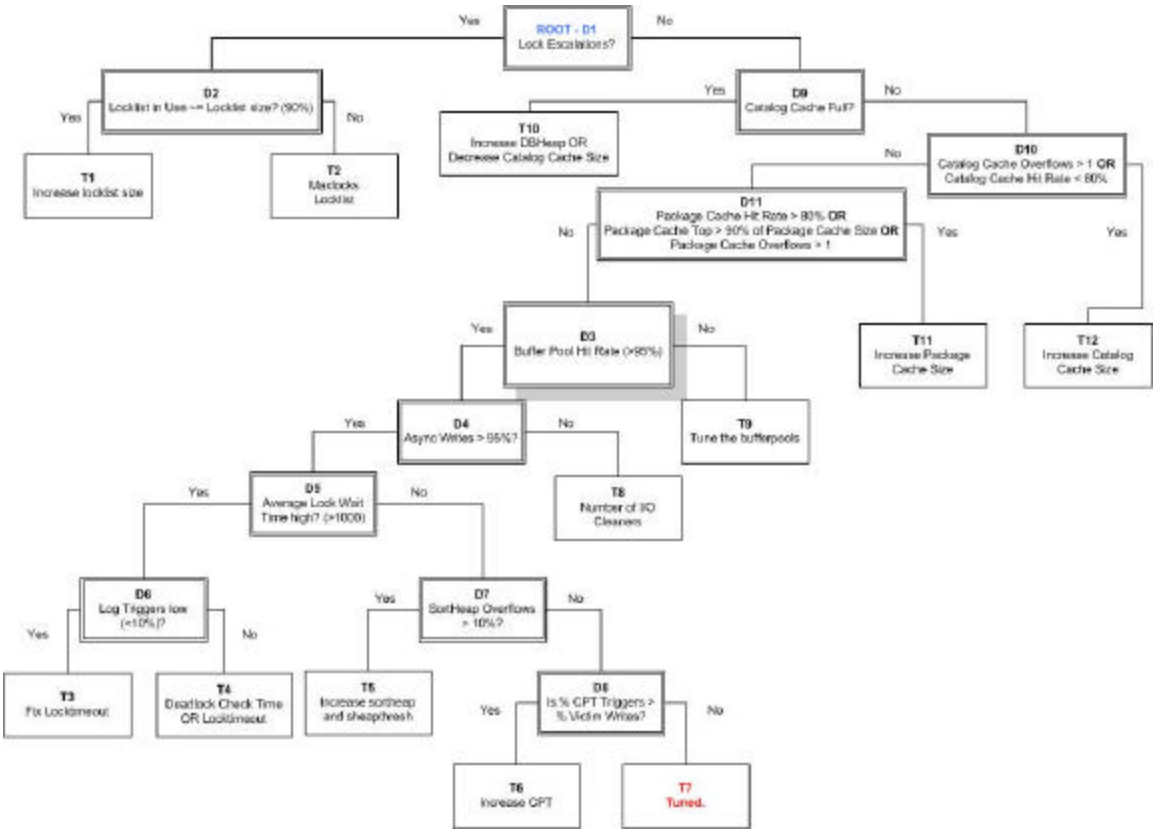


Figure 17 - Proposed generic diagnosis tree.

The new nodes proposed for the generic diagnosis tree are meant to diagnose the database heap, catalog cache and package cache memory sizes. The database heap memory is memory allocated when the first connection is made to the database. Control block information for tables, indexes, table spaces and buffer pools are all kept in this memory space. The catalog cache is a reserved piece of memory available for caching table descriptors for tables and views. The catalog cache is used to store these descriptors in memory in order to avoid disk accesses. The descriptors are needed when compiling an SQL statement. The package cache is a reserved piece of memory used to cache package information when executing static and dynamic SQL statements.

Node D9 is proposed in order to determine if the catalog cache is full. A catalog cache that is not full indicates wasted memory, and the catalog cache size should be reduced accordingly. Node D10, using the knowledge that the catalog cache is full, is proposed to determine the activity in the catalog cache. Frequent catalog cache overflows or a low catalog cache hit rate indicates that the catalog cache size may be too small. Tuning node T12 also suggests the database heap as a possible tuning parameter, as the catalog cache in contained within the amount of memory allocated to the database heap. If the catalog cache seems to be performing well with no overflows and a good hit rate, we move on to proposed node D11. D11 checks the performance of several parameters associated with the package cache, including the package cache hit rate, the package cache size top and the number of package cache overflows. If any of these performance values indicate poor package cache performance, then tuning node T11 will suggest that the package cache and/or the database heap be increased in size. If the package cache performance is good, then we will continue with node D3 which was part of the tuning tree presented in this dissertation.

Chapter 5

Evaluation of Diagnosis Framework

The success of the diagnosis system presented in this dissertation is based on the ability of the system to correctly diagnose performance problems in a working DBMS. In this chapter we present a set of experiments that illustrate the diagnostic ability of a prototype diagnosis system on a working DB2 database system.

Portions of the diagnosis system have been implemented for evaluation purposes. At present, we have constructed a resource model, a workload model and a resulting diagnosis tree for a subset of the resources available for DB2. The implemented resources were chosen based on their significant impact on the workloads used. We have implemented a diagnosis tree and a resource model to deal with the selected resources from DB2. Our algorithm will traverse the resource tree and return tuning suggestions from a tuning node. A forward resource tree is automatically generated by the traversal algorithm. Reverse resource trees are presently generated by hand. Adjusting the

resources suggested by the diagnosis tree is also done manually, as DB2 does not yet support the dynamic adjustment of resources.

5.1 The Test Environment

The test environment consists of an IBM server running the Windows NT Server operating system and DB2 version 7.2 as the DBMS. Detailed information about the test hardware and software can be found in Appendix A. The database workload is based on the Transaction Processing Council OLTP benchmark, TPC-C [TPC]. The transactions are based on an order-entry environment that simulates placing and delivering orders, recording payments, monitoring stock levels and checking the status of orders. Information about the TPC-C benchmark is found in Appendix B.

The test database contains 100 warehouses (approximately 10GB of data in total). Transactions are executed against the database by 60 client processes. Throughput data are collected by the database workload application in five-second intervals throughout the run. DB2 performance information is collected over a five minute period from the 10 minute mark to the 15 minutes mark of the run. The total run time for each test is 20 minutes. The throughput is averaged over the last 17 minutes of each run.

The workload used for testing is a version of the TPC-C database benchmark workload. This workload is used because it exemplifies a generic OLTP workload. The benchmarking workload is able to adequately test the DBMS in a non-deterministic fashion while still producing comparable and repeatable results. The TPC-C workload is cyclic in nature, running five different transactions continuously for the duration of the

tests. Each transaction type will differ only in the data used in the transaction, resulting in a workload that is cyclic in nature. The workload is able to provide, during a single run, multiple data points that can be used for statistical purposes. In order to show that there is no significant difference between the data collected in multiple runs, we have completed a statistical analysis of three workload runs where none of the DBMS parameters are changed. The resulting tpmC for the three runs are very close in value, ranging from 4245.76 tpmC to 4265.72 tpmC. The observed variance is high due to the variability in the tpmC values collected every five seconds. The statistical analysis shows that with two degrees of freedom, we calculated an F value of 0.367566 and an F-critical value of 3.012417. With the value of F smaller than F-critical, we fail to reject the null hypothesis that there is significant difference between the means. In order to reject the null hypothesis with a confidence of 95%, we need a P-value of less than 0.05. Our P-value of 0.692591 is much higher than 0.05, further strengthening the fact that there is no statistical difference between the three runs. Data for the statistical analysis is shown in Table 2. Data points collected for our statistical analysis are found in Appendix J.

Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Run 1	181	770880	4259.006	49691.01
Run 2	181	768483	4245.762	48042.02
Run 3	181	772096	4265.724	54665.81

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	37344.45	2	18672.22	0.367566	0.692591	3.012417
Within Groups	27431790	540	50799.61			
Total	27469134	542				

Table 2 - Statistical analysis of test workload.

Calculating the confidence interval for the workload is important in order to determine the confidence that can be given to the measured workload throughput. Appendix F contains the calculation of the confidence interval for our workload. The resulting confidence interval is 25.3 tpmC, meaning that our calculated throughput results are within 25.3 tpmC of the actual throughput. Combining this information with the statistical analysis showing the repeatability of workload throughputs, it was decided that individual data point collection could be used for our experiments. As a result, the data points shown in our experiments are not averages of multiple runs, but the results of an individual test run.

5.2 The Evaluation Process

Determining the ability of the diagnosis system to diagnose a DBMS correctly involves five distinct steps – initializing the DBMS, restoring the database, running and monitoring the workload, diagnosing the performance data and tuning the DBMS. Figure 18 is a diagram of the evaluation process. DBMS initialization occurs only once during each series of tests. The remaining steps are repeated until the diagnosis algorithm has determined that the DBMS is tuned. Each diagnosis loop results in the collection of throughput data that we use to graph the performance of the workload. We describe the steps in detail below.

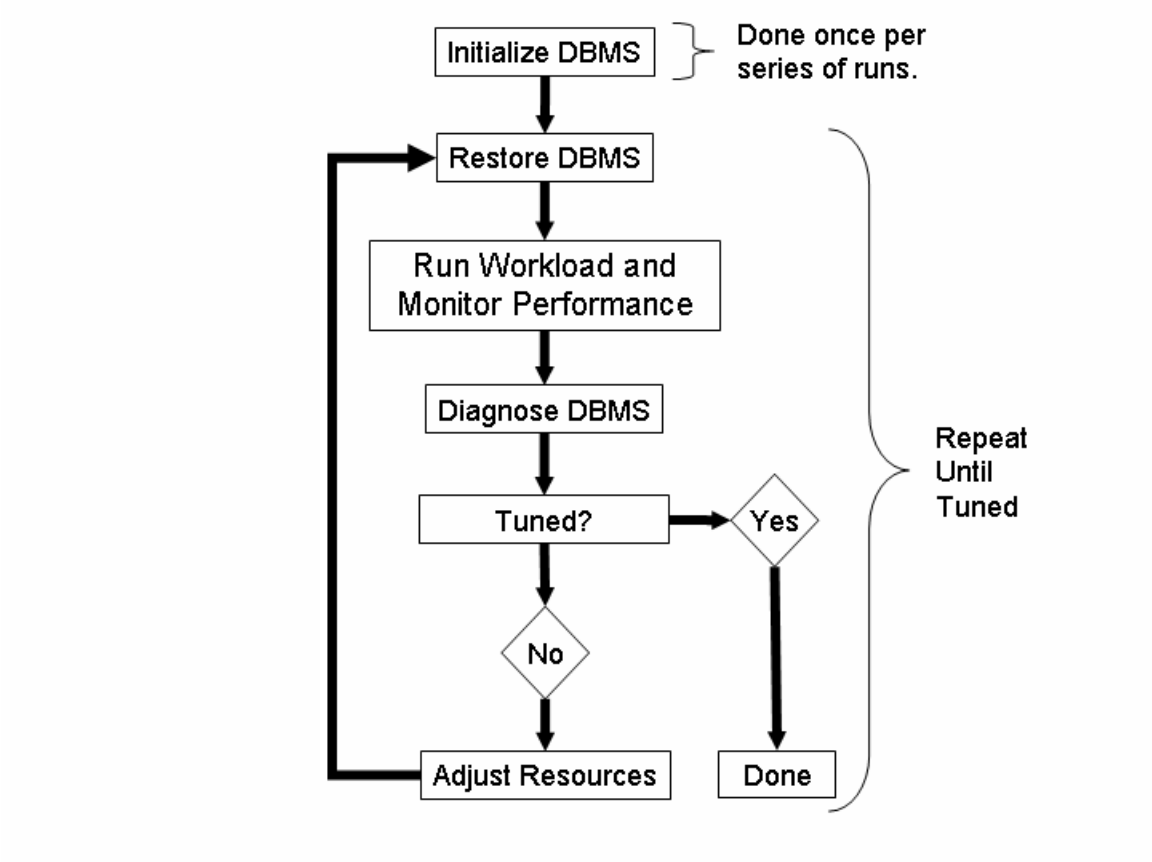


Figure 18 - Evaluation Process.

1) – Initialization

Initialization prepares the database and DBMS by adjusting them to a set of default resource allocations. The values used in the default resource allocation have been shown in our experiment to adversely affect DBMS performance. A list of the resource values used for the untuned DBMS allocation is found in the "Untuned" column of Table 3. Table 3 also lists the values for each resource when tuned by either an expert or by the DBMS Tuning Wizard. The expert and wizard values are used as comparisons when evaluating the diagnosis algorithm.

2) – Restoring the database

The database must be restored after each time the workload is run on the DBMS. Restoring the data ensure that each run is conducted under the same database initial conditions, that is, the same initial database state.

3) – Running and monitoring the workload

While a workload is run, the monitor program collects performance data from the DBMS and the workload applications. Performance data is collected over a five minute period starting 10 minutes into the run. Throughput data is collected every five seconds for the duration of the run. The performance data is stored in a separate database where it can be accessed by the diagnosis algorithm. The schema for the performance data is found in Appendix G. A list of the performance data collected is found in Appendix D.

4) – Diagnosis

Diagnosis begins once the data is collected. The performance data is retrieved from the database where it was stored to aid in the diagnosis of the DBMS. Upon examination of the data, the diagnosis algorithm suggests a list of resources for tuning. The diagnosis algorithm may also return no list, signifying that no resources have been flagged for adjustment.

Resource	Expert Tuned	Untuned	DBMS Wizard
Locklist size	295 4k pages	40 4k pages	295 4k pages
Number of I/O Cleaners	40	1	5
Deadlock Check Time	20,000 msec	100,000 msec	10,000 msec
Lock Timeout	-1 (infinity)	5 seconds	-1 (infinity)
Changed Pages Threshold	60	20	80
Sortheap Size	10,000 4k pages	256 4k pages	256 4k pages
Sortheap Threshold	10,000 4k pages	512 4k pages	10,000 4k pages

Table 3 - DBMS resource values.

5) – Tuning

The DBMS is tuned by adjusting each resource according to the associated strategy in Table 4. The resource adjustment strategy is used to ensure that resource adjustments are consistent throughout all experiments. Resource allocations were modified only when indicated by the diagnosis algorithm. Single adjustments are made to the system in order to determine the impact of the resource on performance. Adjustments are made using the values shown in Table 4. The initial value of the resource is indicated in the table, as are the values to be used if the resource is diagnosed. For example, consider the locklist size resource. The initial value is 40 4k pages. If the locklist size is diagnosed, we will adjust the resource to the value stored in the Step 1 column, 60 4k pages. If the locklist size is diagnosed again, we will further adjust the resource to the value stored in the Step 2 column, 80 4k pages. We will continue with this naïve tuning strategy using the appropriate steps and values shown in Table 4.

In general, we diagnose and tune only one resource per diagnosis tree traversal. There are two exceptions to this, involving two sort-oriented resources and two lock-oriented resources. In the case of the sortheap size and sortheap threshold resources, documentation clearly indicates that these resources must be adjusted as a pair [IBM00]. They are therefore diagnosed and tuned as a pair. In Node T4 of the diagnosis tree in Figure 15, the diagnosis is to adjust either the deadlock check time or the lock timeout resource. A naïve strategy of adjusting the deadlock check time twice, followed by adjusting the lock timeout resource once, was chosen. This pattern is followed whenever Node T4 is the result of a diagnosis.

Resource (unit of measure)	Initial Value	Step 1	Step 2	Step 3	Step 4
Locklist Size (4k pages)	40	60	80	100	120
Number of I/O Cleaners	1	10	20	30	40
Deadlock Check Time (msec)	100,000	50,000	10,000	5,000	2,000
Changed Pages Threshold (%)	20	30	40	50	60
Locktimeout (seconds)	5	10	20	50	100
Sortheap Size (4k pages)	256	512	1024	2048	4096
Sortheap Threshold (4k pages)	512	1024	2048	4096	8192

Table 4 – Tuning strategy.

Using the results of the diagnosis, one or more resources are chosen for adjustment. Resource adjustments made will remain for the rest of the runs in this test. Steps 2-5 are then repeated until the diagnosis algorithm determines that the database is tuned.

5.3 Typical Tuning Scenarios

Tuning is needed throughout the life of a database in order to maintain peak performance. Several common situations occur during the lifetime of a database that cause it to need retuning, such as the addition of new hardware, the addition of new transactions, or an increase in size of the data stored. Three such situations were chosen as a representative sample and these scenarios are outlined in Table 5.

Scenario	Explanation
Size Increase	An increase in the database size can cause the DBMS to perform poorly. We simulate doubling the size of the database by reducing the buffer pool size by half.
Workload Shift	Altering the occurrence of transactions in the workload can cause poor performance. Adjusting transaction percentages in the workload simulates a workload shift.
Transaction Variation	Varying the transactions that are executed against the database can reduce DBMS performance. In the transaction variation scenario, a new type of transaction is added to the workload.

Table 5 - DBMS tuning scenarios.

The database size scenario is a significant problem for DBAs. DBMS tuning values become outdated as the database grows or shrinks in size. Database growth depends on the number and frequency of updates. The percentage of data held in the buffer pools decreases as the amount of data accessed from the tables increases, which has a negative effect on the buffer pool hit rate. A buffer pool is an area of memory that is used to cache data and index information from the DBMS disks. Increasing the amount of data may

also have an effect on the number of I/O cleaners needed to move data, the number of I/O servers needed to retrieve the data from the disks, and various other DBMS resources.

The workload shift scenario simulates a change in the way that a database is used. Each transaction presents a different load on the DBMS. As the percentage of each transaction changes in the workload, the workload requirements on the DBMS shifts. Table 6 lists the original percentages and the modified percentages of each transaction in the workload.

Transaction Name	**Original workload**	**Workload Shift**
New Order	45%	24%
Payment	43%	24%
Order Status	4%	22%
Delivery	4%	4%
Stock Level	4%	26%

Table 6 – Transaction frequencies for the original and modified workloads.

The transaction variation scenario involves the addition of a new transaction to the workload that uses the same data as the original transactions and so interferes with the original workload to some extent. The transaction added to the workload is a sort query that is embedded in a loop. It is not required that the sort query be run a particular percentage of times during each test, but that the query is run as many times as possible during each test period.

5.4 Scenario 1 – Size Increase

Original Configuration

We ran the original OLTP workload with the relative frequencies specified in Table 6. The buffer pool was 100,000 4k pages in size, which is approximately 400MB. This allows us to simulate a relatively small database situation where sufficient memory is available for good system performance. Table 7 presents the data from the tuning process for the original workload. For each run, the "Changed Resource" column indicates the resource value that was altered from the previous run, the "Diagnosis" column provides the diagnosis made by the algorithm for this particular run and the "tpmC" column shows the resulting throughput expressed in **T**ransactions **P**er **M**inute **C** (**tpmC**) for the run.

The evaluation process in Figure 18 shows that the diagnosis process will iterate until the "tuned" condition is met. The diagnosis tree in Figure 15 contains T7, the tuning leaf that signifies when the system is tuned. If a tree traversal results in the return of node T7, then the diagnosis tree was unable to diagnose any performance problems and the database is considered tuned. The results in Table 7 for Run 1 show that during the first run the resource allocations are so poor that the throughput (tpmC) is zero. The zero throughput is a result of poor lock resource allocations, resulting in a deadlocked system. The diagnosis tree suggests tuning the locklist size parameter resulting in an adjustment according to the tuning strategy outlined in Table 4. The locklist parameter is adjusted from 40 pages to 60 pages and the workload is run again, which results in the diagnosis found in Run 2 of Table 7. Increasing the size of the locklist alleviates the deadlock problem and results in increased throughput.

Run	Changed Resource	Diagnosis	tpmC
1	Starting Configuration	Locklist	0.00
2	locklist = 60	Number of I/O Cleaners	3897.18
3	num_iocleaners = 10	Number of I/O Cleaners	3150.18
4	num_iocleaners = 20	Deadlock check time and/or Lock timeout	4254.41
5	dlchktime = 50000	Deadlock check time and/or Lock timeout	2984.18
6	dlchktime = 10000	Number of I/O Cleaners	4593.00
7	num_iocleaners = 30	Deadlock check time and/or Lock timeout	4258.88
8	locktimeout = 10	Deadlock check time and/or Lock timeout	1418.29
9	dlchktime = 5000	Number of I/O Cleaners	5986.94
10	num_iocleaners = 40	Done	6270.76

Table 7 - Diagnosis of the original workload on a small database.

Although the change from Run 1 to Run 2 is a significant performance increase, the diagnosis algorithm suggests that the number of I/O cleaners should be adjusted. The number of I/O cleaners is increased from one to 10. Increasing the number of I/O cleaners results in a decrease in throughput. This decrease in throughput is caused by an increase in the average lock wait time. Further diagnosis suggests another increase in the number of I/O cleaners. This diagnosis may seem counter-productive, as the previous drop in performance was caused by an increased average lock wait time triggered by an increase in the number of I/O cleaners. It should be noted that both I/O cleaners and lock wait time have poor measured performance at this point in time – the number of I/O cleaners is diagnosed first due to its position in the diagnosis tree. Adjusting the number of I/O

cleaners for Run 4 results in a performance increase that more than compensates for the decrease noticed from Run 2 to Run 3.

Diagnosis of the performance data from Run 4 results in the tuning suggestion of adjusting either the deadlock check time or lock timeout resources. This diagnosis results from an average lock wait time that is above the threshold value. In the case where two resources are diagnosed for tuning, a tuning policy must be followed to determine the order in which the resources are tuned. In the case of the deadlock check time and lock timeout resources, the tuning policy used requires that the deadlock check time resource be adjusted twice for every adjustment of the lock timeout resource. As a result, the first two times that this tuning combination is suggested by the diagnosis algorithm, the deadlock check time will be adjusted. When the combination is suggested a third time, the lock timeout will be adjusted. The pattern of tuning the deadlock check time twice for each adjustment of the lock timeout continues from there. As a result of this tuning pattern, the deadlock check time resource is adjusted from 100,000 msec to 50,000 msec, resulting in a decrease in tpmC from 4254.41 to 2984.18. A further diagnosis is made to adjust again the deadlock check time resource from 50,000 msec to 10,000 msec, resulting in an increase in tpmC from 2984.18 to 4593.00. The temporary decrease in performance when the deadlock check time resource was set at 50,000 msec was caused by an increase in the average lock wait time performance measurement. When the deadlock check time was set at 100,000 msec, deadlocks were given enough time to resolve themselves, resulting in the recorded performance. When the deadlock check time was reduced to 50,000 msec, the deadlocks did not have enough time to resolve, resulting

in many lost transactions and a high lock wait time, greatly reducing performance. The continued reduction in the deadlock check time resource to 10,000 msec results in the deadlocks being resolved faster, resulting in less lock wait time and higher throughput.

Run 6 diagnosis results in adjusting the number of I/O cleaners from 20 to 30 due to a low percentage of asynchronous writes. The resulting decrease in tpmC is due to increased lock wait time due to increased data contention. The increased lock wait time results in the diagnosis in Run 7 requiring the adjustment of either the deadlock check time resource or the lock timeout resource. The deadlock check time resource has already been adjusted twice so the lock timeout resource is now adjusted. The resulting resource change produces a significant drop in tpmC from 4258.88 to 1418.29. This drop in performance is directly related to a significant increase in the lock wait time due to the increase in the amount of time a lock must wait before timing out. The increase in lock wait time results in the Run 8 diagnosis of deadlock check time and lock timeout. We modify the deadlock check time resource (based on our naïve tuning strategy), resulting in an increase in performance from 1418.29 to 5986.94 while reducing our average lock wait time. The diagnosis for Run 9 involves adjusting the number of I/O cleaners due to a low percentage of asynchronous writes. The adjustment results in increasing the percentage of asynchronous writes and increasing performance. Diagnosis of the performance data results in the "tuned" node being returned. The diagnosis algorithm has no further suggestions and the diagnosis and tuning process is complete.

We evaluate the success of the diagnosis algorithm by comparing the final throughput of the DBMS with tpmC values achieved by our expertly tuned system and a system tuned by the DB2 Tuning Wizard. The expertly tuned system represents the best possible throughput achieved when tuning the system by hand. The expert who tuned the DBMS for these experiments is a DBA and research associate for the Database System Laboratory at Queen's University with over 10 years of experience tuning DB2. Table 3 shows the resource allocations used for both the expert and tuning wizard configurations.

The final measured throughput for the diagnosed configuration is 6270.86 tpmC. The throughput for the expert configuration is 6355.65 tpmC while the throughput for the wizard configuration is 4966.00 tpmC. The diagnosis system is able to tune the system to 98.67% of the expert throughput and 126.28% of the tuning wizard throughput. Throughput values are summarized in Figure 19.

Figure 19 shows that each step taken during the diagnostic process does not always increase system throughput. The changes made in several of the iterations actually cause a decrease in database throughput. The diagnosis algorithm does not consider database throughput during the diagnosis process, but instead examines the values of the indicator variables. Resource adjustments are made if an adjustment will increase or decrease the indicator variable appropriately. Improving the value of indicator variables will result in improved database performance.

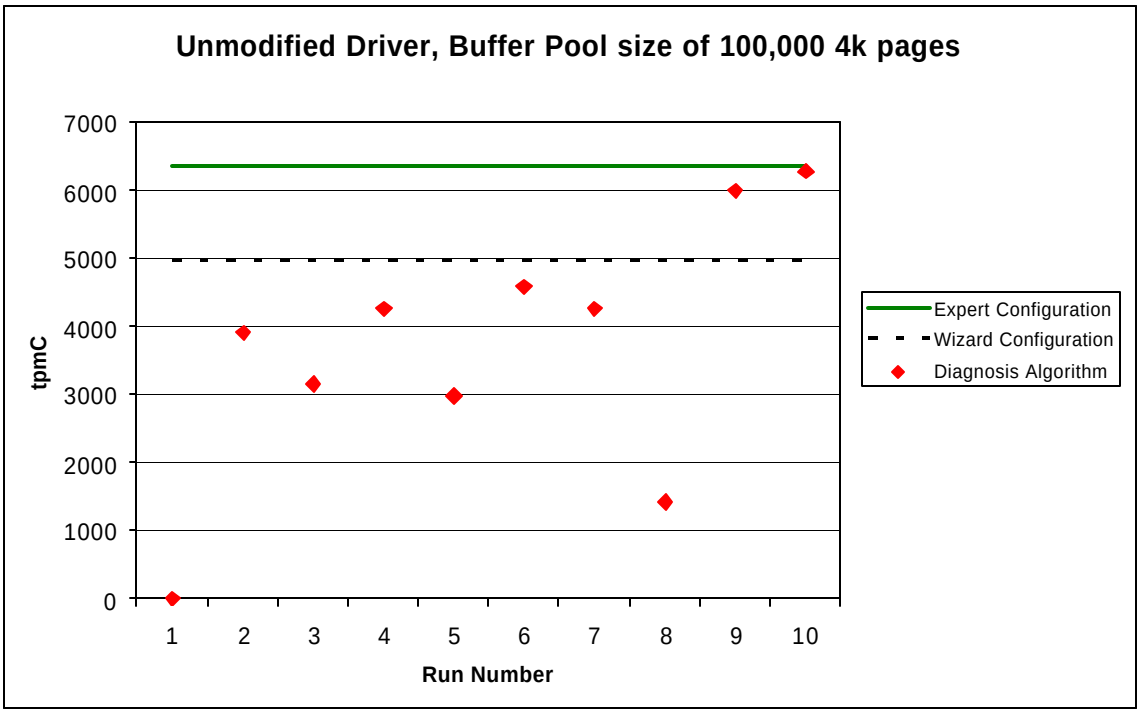


Figure 19- Throughput results for the original workload on a small database.

A resource adjustment is considered valuable if the performance of the resource or indicator variable increases. For example, in Run 6 shown in Table 7, the throughput of the workload is 4593.00 tpmC. The diagnosis algorithm suggests increasing the number of I/O cleaners from 20 to 30. The diagnosis is based on performance data which indicates that the percentage of asynchronous writes is 85.0%. The diagnosis tree recognizes that the percentage of asynchronous writes is below the threshold of 95% and suggests an increase in the number of I/O cleaners to fix the problem. Increasing the number of I/O cleaners to 30 results in 99.7% asynchronous writes for Run 7, a value above the threshold. The throughput for Run 7 is 4258.88 tpmC, a decrease of 334.12 tpmC from the previous run. Although the performance of the system decreased slightly, increasing the number of I/O cleaners as suggested by Run 6 is the appropriate action. The throughput decrease can be attributed to a shift in the performance bottleneck. In

Run 6 the performance bottleneck is due to a lack of I/O cleaners and a low percentage of asynchronous writes. Increasing the number of I/O cleaners results in an increase in the percentage of asynchronous writes, allowing more data to be processed by the database. Processing the extra data increases the percentage of log triggers, an indication that problems exist with the deadlock check time and lock timeout variables. The increased data contention in Run 7 results in a lower throughput than recorded in Run 6. Although the resource adjustment suggested in Run 6 is beneficial to the overall performance of the database, the resulting throughput recorded for Run 7 is lower than in Run 6. This type of resource adjustment is only detrimental to system performance if tuning is stopped at Run 7. By continuing with the diagnosis and tuning process, the new bottleneck can be diagnosed and throughput improved.

Diagnosis time is an important factor for an automated diagnosis system. Diagnosis time is measured by the amount of time needed from one diagnosis to the next. Each iteration of the diagnosis process involves running the workload, waiting for the workload to stabilize, collecting the performance and throughput data, running the diagnosis algorithm and then making the resource adjustment. Reducing the time required per iteration will reduce the amount of time needed to completely diagnose the system. In our present setup, the workload is run for a period of 20 minutes per iteration. The 20 minutes includes a warm-up period for the workload to stabilize, data throughput collection and performance data collection. The average run time for the diagnosis algorithm is between 10 and 20 seconds and the time needed to adjust the diagnosed DBMS resource is less

than 30 seconds. The total amount of time needed per iteration is therefore approximately 21 minutes.

The amount of time currently needed per iteration is restricted by the inability to adjust DBMS resources while the database is running. This restriction requires that the DBMS be shut down and restarted before changes take effect, resulting in longer iterations times. As database technology advances, dynamically adjustable database tuning parameters will greatly reduce iteration time.

We can also reduce the number of iterations to further reduce the total amount of time needed to diagnose a DBMS. In Table 7, the number of iterations needed to tune the workload is 10. Several of the tuning steps in Table 7 require the same resource to be tuned twice in a row, such as diagnosing the number of I/O cleaners in Step 2 and Step 3. Step 2 suggests adjusting the number of I/O cleaners from 1 to 10 while Step 3 suggests adjusting the I/O cleaners from 10 to 20. The use of a more intelligent tuning algorithm during the diagnosis in Step 2 may result in increasing the number of I/O cleaners from 1 to 20, effectively eliminating Step 3. The use of intelligent tuning algorithms instead of the naïve tuning process from Table 4 can help reduce the number of iterations needed to tune the DBMS. In the case outlined in Table 7 the number of iterations could be reduced from 10 to 7.

Another method to reduce the number of iterations needed to tune the DBMS is the use of resource trees. A reverse resource tree enables us to expand the diagnosis space once a

problem resource is identified. Expanding the diagnosis space to include related resources provides information that can be beneficial to the tuning process. In Step 1 of Table 7, the locklist resource is selected for tuning. The reverse resource tree generated in Figure 20 indicates that the resources directly related to locklist are deadlock check time, lock timeout, maximum number of locks, and maximum number of applications. The maximum number of agents and the average number of applications are indirectly related through the maximum number of applications resource. In this case, both the deadlock check time and lock timeout resources are later diagnosed by the diagnosis tree. Using the resource tree identifies both of these resources as potential tuning candidates early in the diagnosis process, allowing us to run tuning algorithms for these resources. Intelligent tuning algorithms may suggest earlier adjustments for both deadlock check time and lock timeout, reducing the number of iterations needed to tune the DBMS.

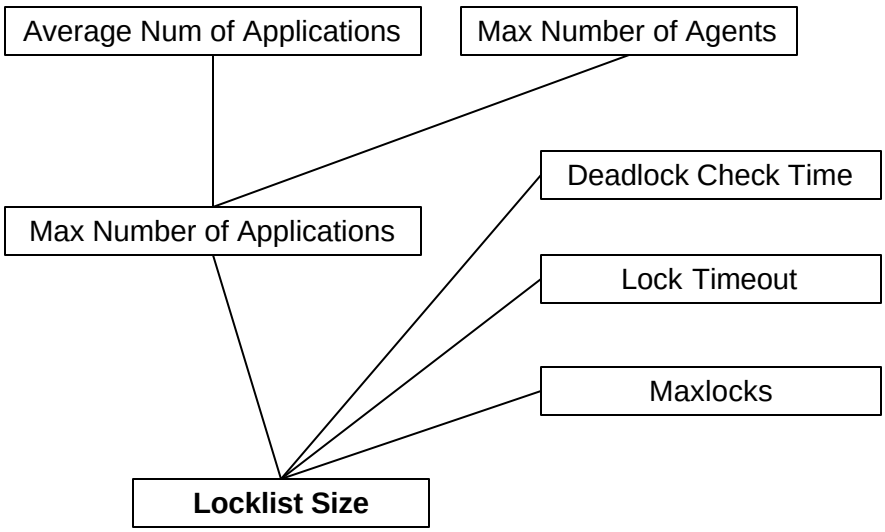


Figure 20 - Reverse resource tree with Locklist Size as root.

Step 2 and Step 3 in Table 7 both diagnose the number of I/O cleaners resource as the cause of performance problems. The reverse resource tree shown in Figure 21 identifies the buffer pool and changed pages threshold resources as related to the number of I/O cleaners and possible causes of the performance problem. Although identified as resources to be considered for tuning, neither related resource is adjusted during this run. This also applies for Steps 6 and 9 where the number of I/O cleaners is diagnosed. The forward and reverse resource trees used can be found in Appendix I.

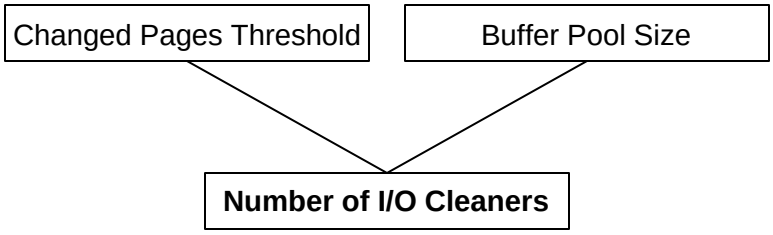


Figure 21 - Reverse resource tree with I/O Cleaners as root.

Modified Configuration

We simulate database growth by reducing the size of the buffer pool to 50,000 4k pages. We simulated database growth as opposed to building a larger database in order to avoid hardware complications such as lack of disks. We simulate growth by decreasing the memory to data ratio by decreasing the amount of available memory. The original OLTP workload is used and the DBMS resources are set to the untuned values (see Table 3). The results of the diagnosis process are found in Table 8. Figure 22 shows the workload performance at each step of the diagnosis.

The diagnosis system is able to tune the system to 98.91% of the expert throughput, and achieves a throughput of 121.95% of the tuning wizard configuration. The diagnostic and tuning process requires 10 iterations to meet the stop condition. Figure 22 shows that even though some individual tuning steps actually decrease the workload throughput, the overall resulting throughput is high.

The number of iterations to tune the DBMS in this case could be reduced to as little as four iterations with improved tuning algorithms. Step 2 and Step 3 in Table 8 both suggest changes to the number of I/O cleaners. An intelligent tuning algorithm may be able to reduce the pair of iterations to a single iteration by suggesting an immediate jump in I/O cleaners from 1 to 20. Step 4 through Step 7 could also be condensed into one or two steps while Step 8 and Step 9 could be condensed to one step. Tuning algorithms that would allow larger steps to be taken during the tuning process would reduce the number of iterations needed to tune a workload.

Once again the first resource diagnosed by the diagnosis tree is the locklist size. The reverse resource tree in Figure 20 shows that the resource is directly affected by deadlock check time, lock timeout, the maximum number of locks and the maximum number of applications. Table 8 shows that both deadlock check time and lock timeout are adjusted during the tuning process. Tuning algorithms could have predicted these changes earlier, thereby reducing the number of iterations needed during the tuning process.

Run	Changed Resource	Diagnosis (End)	tpmC
1	Starting Configuration	Locklist	0.00
2	locklist = 60	Number of I/O Cleaners	3222.76
3	num_iocleaners = 10	Number of I/O Cleaners	3449.41
4	num_iocleaners = 20	Deadlock Check Time and/or Lock Timeout	3188.88
5	dlchktime = 50000	Deadlock Check Time and/or Lock Timeout	2834.59
6	dlchktime = 10000	Deadlock Check Time and/or Lock Timeout	3819.82
7	locktimeout = 10	Deadlock Check Time and/or Lock Timeout	2587.59
8	dlchktime = 5000	Number of I/O Cleaners	3868.65
9	num_iocleaners = 30	Number of I/O Cleaners	4097.65
10	num_iocleaners = 40	Done.	4233.65

Table 8 - Diagnosis of the original workload on a large database.

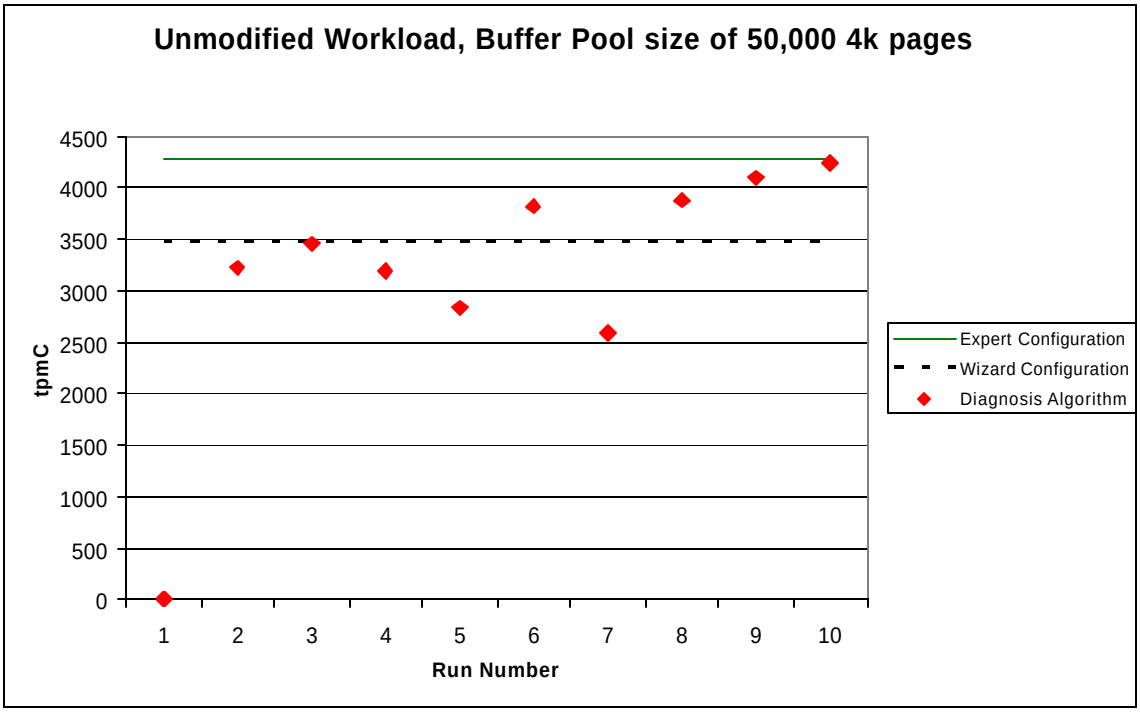


Figure 22 - Throughput results for the diagnosis of the original workload on a large database.

5.5 Scenario 2 – Modified Workload

The modified workload scenario consists of the same transactions as the original workload with different relative frequencies. The frequencies for the modified workload are specified in Table 6. The modified workload is intended to simulate a shift in workload over time. The modified workload is tested on both the large and small database configurations.

Small Database Configuration

When the small database is used with the modified workload, the diagnosis system is able to tune the system to 97.1% of the expert throughput and 185.4% of the tuning wizard throughput. The diagnostic and tuning process, located in Table 9, requires 10 iterations to finish. A graph of the resulting throughput for each configuration step along with the wizard and expert configuration throughputs are presented in Figure 23.

As discussed in Section 5.4, the end throughput of the diagnosis process is high despite the fact that several individual steps in the tuning process cause decreases in throughput. The number of iterations needed to tune the DBMS is 10, but that number can be reduced by using more intelligent tuning algorithms. Better tuning algorithms may condense steps two and three, steps four through seven and steps eight and nine, reducing the number of needed iterations from 10 to five.

Run	Changed Resource	Diagnosis	tpmC
1	Starting Configuration	Locklist Size	376.76
2	locklist = 60	Number of I/O Cleaners	3388.00
3	num_iocleaners = 10	Number of I/O Cleaners	3785.12
4	num_iocleaners = 20	Deadlock Checktime and/or Lock Timeout	4124.00
5	dlchktime = 50000	Deadlock Checktime and/or Lock Timeout	2289.12
6	dlchktime = 10000	Deadlock Checktime and/or Lock Timeout	2763.24
7	locktimeout = 10	Deadlock Checktime and/or Lock Timeout	1931.65
8	dlchktime = 5000	Number of I/O Cleaners	5649.24
9	num_iocleaners = 30	Number of I/O Cleaners	5984.00
10	num_iocleaners = 40	Done	6241.29

Table 9 - Diagnosis of the modified workload on a small database.

As discussed in Section 5.4, the diagnosis of the locklist size in Step 1 result in the resource tree in Figure 20 that predicts the adjustment of deadlock check time and lock timeout.

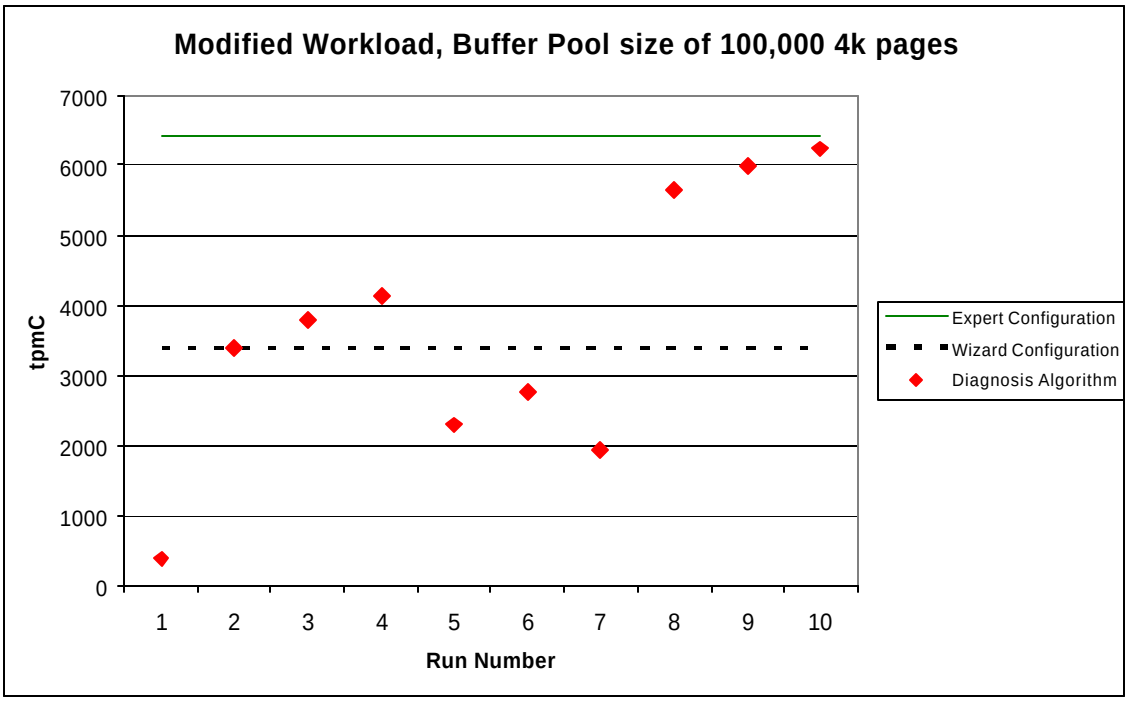


Figure 23 - Throughput results for the diagnosis of the modified workload on a small database.

Large Database Configuration

When the large database is used with the modified workload, the diagnosis system is able to tune the system to 96.01% of the expert throughput and 180.74% of the tuning wizard throughput. The diagnostic and tuning process is completed in eight iterations. A graph of the resulting throughput of the configuration steps along with the wizard and expert configurations are presented in Figure 24.

As discussed in Section 5.4, the decrease in throughput due to individual tuning steps does not hinder the ability of the diagnosis tree to effectively diagnose the DBMS. Diagnosis requires eight iterations in this case. More effective tuning algorithms could reduce the process to four iterations by combining steps one and two and steps four

through seven. This would reduce the total number of iterations needed to diagnose this case from eight to four.

Run	Changed Resource	Diagnosis	tpmC
1	Starting Configuration	Number of I/O Cleaners	972.24
2	num_iocleaners = 10	Number of I/O Cleaners	0.00
3	num_iocleaners = 20	Locklist Size	634.00
4	locklist = 60	Deadlock Checktime and/or Lock Timeout	1465.12
5	dlchktime = 50000	Deadlock Checktime and/or Lock Timeout	2056.00
6	dlchktime = 10000	Deadlock Checktime and/or Lock Timeout	2004.53
7	locktimeout = 10	Deadlock Checktime and/or Lock Timeout	1558.06
8	dlchktime = 5000	Done	2175.29

Table 10 - Diagnosis of the modified workload on a large database.

The initial diagnosis in Table 10 is the number of I/O cleaners. The reverse resource tree in Figure 21 recommends considering both the buffer pool size and the changed pages threshold for tuning. Neither of these resources is later tuned by the diagnosis tree, as the buffer pool hit rate threshold does not drop below 95%. The diagnosis in Step 3 is for locklist size. As described in Section 5.4 and Figure 20, deadlock check time, lock timeout, maximum number of locks and maximum number of applications are suggested for tuning consideration. Both deadlock check time and lock timeout are later diagnosed by the diagnosis tree and adjusted.

Changes in a DBMS workload can alter the delicate balance of a tuned system. The ability to adjust to changes in transaction frequency allows a DBMS to perform well with a modified workload. Results presented in Table 9 and Table 10 show the ability of the diagnosis tree to correctly diagnose performance problems for a modified workload. Although the resulting throughput is lower for the large database configuration as compared to the small database configuration, the throughput in both cases is higher than the corresponding expert and wizard configurations. These runs show that the effectiveness of the diagnosis tree is not dependent on a specific workload.

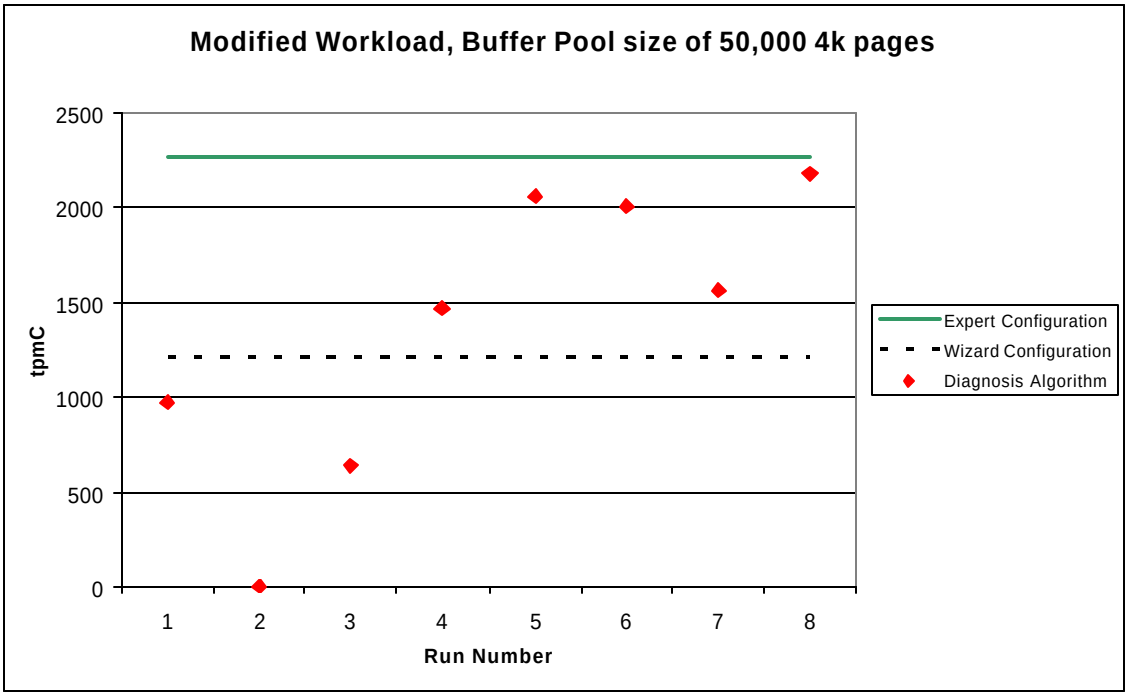


Figure 24 - Throughput results for the diagnosis of the modified workload on a large database.

5.6 Scenario 3 – Workload Change

A new transaction is added to the original database workload in order to simulate a workload change. The new transaction issues a sort query that sorts data found in the item

table of the TPCC database. The sort transaction interacts with the same data as the original queries. The addition of the sort transaction tests the ability of the diagnosis algorithm to correctly diagnose a new workload. The new workload is tested on the simulated small and the large database configurations.

The sort transaction differs significantly from the other transactions that comprise the workload. The sort query is not related to the other queries – it runs independently of the other queries. The sort query makes use of the sortheap memory, resulting in our being able to diagnose sort overflows and sortheap usage. The sort query is a select of the Item Name and Item ID fields from a table and sorting them based on the Item ID field. Each query sorts 100,000 records.

Small Database Configuration

When the small database configuration is used in the first test with the changed workload, the diagnosis system is able to tune the system to 100.09% of the expert throughput and to 126.31% of the tuning wizard throughput. The diagnostic and tuning process takes 13 iterations to finish. A graph of the resulting throughput of the configuration steps along with the wizard and expert configurations can be found in Figure 26.

With intelligent tuning algorithms, Step 2 to Step 3, Step 4 to Step 5, Step 7 to Step 8 and Step 10 to Step 12 can all be reduced to one step each allowing the workload to be tuned in only eight iterations as opposed to 13. It should be noted that the last three steps in the tuning process (Step 10 to Step 13) do not seem to have an impact on the throughput of

the workload (from 6134.00 to 6202.65 tpmC). The tuning that occurred from Step 10 to Step 13 involves improving the sort query. The sort query does not have a direct effect on the resulting throughput, resulting in only a small increase in tpmC. The performance data for runs 10 through 12 indicates sort overflows occurring in the system. Sort overflows occur when there is not enough memory allocated in the sortheap for the sort to occur in memory. A sort overflow indicates that temporary space on the hard drive will be used to aid in the sort. These sort overflows do not occur in step 13 because of the increased sort heap and sort heap threshold. Reducing the number of sort overflows is beneficial to the performance of the underlying database engine. Separate testing of the sort query, shown in Figure 25, shows that sort overflows reduce the response time of the sort query.

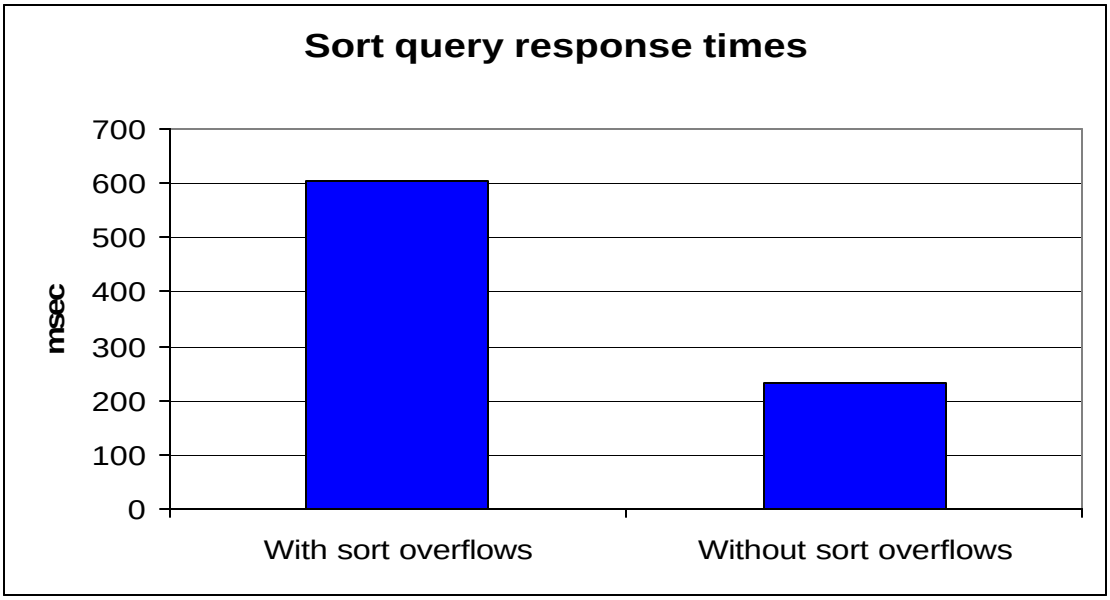


Figure 25 - The effect of sort overflows on sort query response time.

Testing shows that for the sort query in question, the sort time when sort overflows happen is 605 msec. Increasing the sort heap and sort heap thresholds removes the sort overflow and results in a sort time of 235 msec.

Run	Changed Resource	Diagnosis	tpmC
1	Starting Configuration	Locklist Size	1.35
2	locklist = 60	Number of I/O Cleaners	2854.18
3	Num_iocleaners = 10	Number of I/O Cleaners	5198.88
4	Num_iocleaners = 20	Deadlock Checktime and/or Lock Timeout	3415.06
5	dlchktime = 50000	Deadlock Checktime and/or Lock Timeout	4019.94
6	dlchktime = 10000	Number of I/O Cleaners	4545.53
7	Num_iocleaners = 30	Deadlock Checktime and/or Lock Timeout	3770.76
8	Locktimeout = 10	Deadlock Checktime and/or Lock Timeout	1833.29
9	dlchktime = 5000	Number of I/O Cleaners	5889.00
10	Num_iocleaners = 40	Sortheap Size and Sortheap Threshold	6134.00
11	sortheap = 512 sheapthresh = 1024	Sortheap Size and Sortheap Threshold	6155.29
12	sortheap = 1024 sheapthresh = 2048	Sortheap Size and Sortheap Threshold	6187.06
13	sortheap = 2048 sheapthresh = 4096	Done	6202.65

Table 11 - Diagnosis of the changed workload on a small database.

The locklist size is diagnosed in Step 1 of Table 11. As discussed in Section 5.4, the adjustment of deadlock check time and lock timeout are accurately predicted. Use of

intelligent tuning algorithms could result in both of these resources being tuned earlier, reducing the number of iterations needed for tuning.

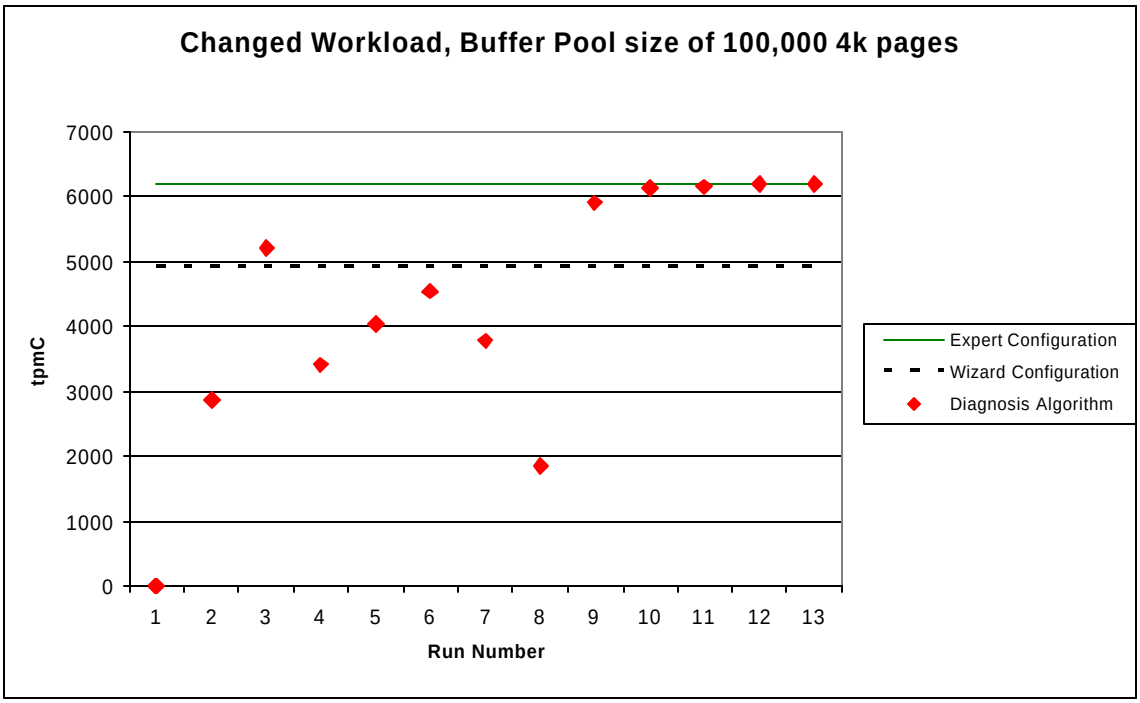


Figure 26 - Throughput results for the changed workload on a small database.

Large Database Configuration

When the large database is used the diagnosis system is able to tune the system to 101.2% of the expert throughput and 123.60% of the tuning wizard throughput. The diagnostic process requires 13 iterations to finish. Diagnosis results are found in Table 12. A graph of the resulting throughput for each configuration step along with the wizard and expert configurations is represented by Figure 27.

The large database changed workload test takes 13 iterations to properly diagnose and tune, which can be reduced with intelligent tuning algorithms. Step 2 through Step 5,

Step 7 though Step 8, and Step 9 through Step 11 can all be condensed to one step each, reducing the number of iterations needed to tune the DBMS to seven.

Run	Changed Resource	Diagnosis	tpmC
1	Starting Configuration	Locklist Size	51.06
2	locklist = 60	Number of I/O Cleaners	3111.53
3	Number of I/O Cleaners = 10	Number of I/O Cleaners	3516.18
4	Number of I/O Cleaners = 20	Number of I/O Cleaners	3300.71
5	Number of I/O Cleaners = 30	Number of I/O Cleaners	4010.59
6	Number of I/O Cleaners = 40	Deadlock Checktime and/or Lock Timeout	3083.24
7	Deadlock Check Time = 50000	Sortheap Size and Sortheap Threshold	4019.29
8	Sortheap = 512 Sortheap Threshold = 1024	Sortheap Size and Sortheap Threshold	3763.12
9	Sortheap = 1024 Sortheap Threshold = 2048	Deadlock Checktime and/or Lock Timeout	3292.12
10	Deadlock Check Time = 10000	Deadlock Checktime and/or Lock Timeout	2666.71
11	Lock Timeout = 10	Deadlock Checktime and/or Lock Timeout	2390.12
12	Deadlock Check Time = 5000	Sortheap Size and Sortheap Threshold	4172.59
13	Sortheap = 2048 Sortheap Threshold = 4096	Done	4214.88

Table 12 - Diagnosis of the changed workload on a large database.

The reverse resource tree generated for Step 1 accurately predicts the adjustment of the deadlock check time and the lock timeout resources. The use of the resource tree and tuning algorithms could result in both of these resources being tuned much earlier in the diagnosis process.

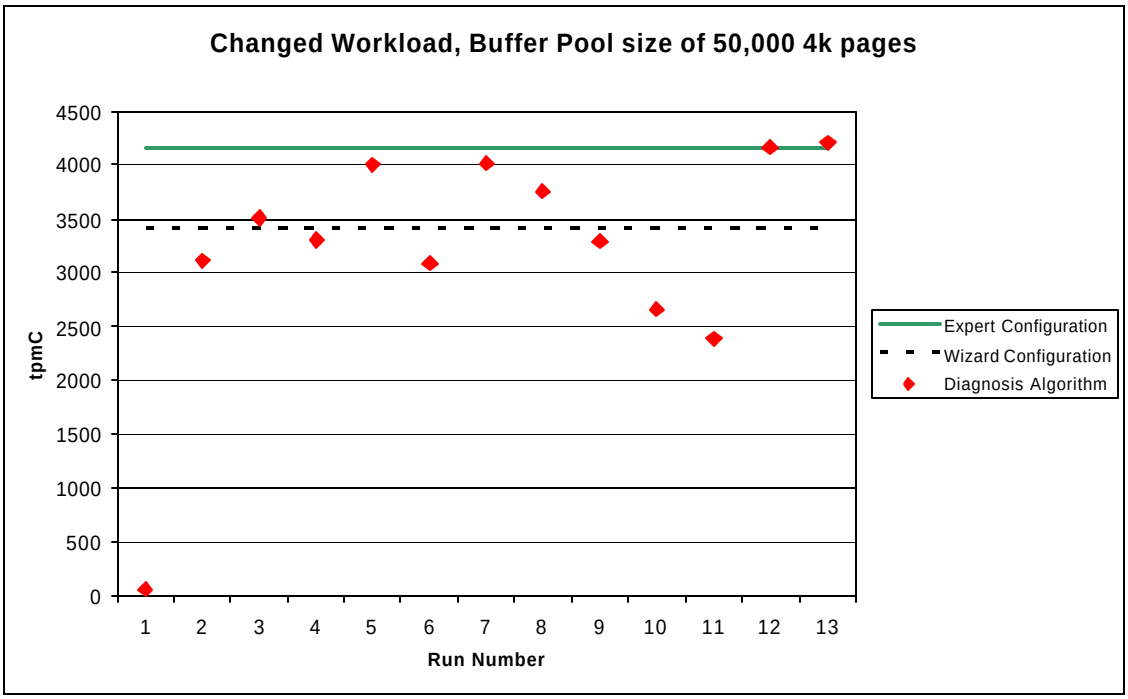


Figure 27 - Throughput results for the changed workload on a large database.

5.7 Diagnosis Tree Lifespan

The diagnosis tree must be able to adapt to different workload situations. Results from Section 5.4 through Section 0 show that the diagnosis tree is able to diagnose a DBMS workload given a change in workload or a change in database size. Although some changes in workload can be handled by the diagnosis tree, a significant change in the workload may render the diagnosis tree unable to diagnose a performance problem. The ability of the diagnosis tree to correctly diagnose performance problems related to new transactions is a function of the diversity in the transactions used when creating the initial diagnosis tree. For example, the sample diagnosis tree in Figure 10 contains the node D_4 to detect package cache inserts. In the tuned diagnosis tree (Figure 15) this node is eliminated because the workload does not generate any package cache inserts beyond the warmup period, making it impossible to collect data on how to correctly tune the DBMS

when they happen. As a result, a transaction that causes package cache inserts cannot be properly diagnosed by the diagnosis tree in Figure 15, requiring that the diagnosis tree be retuned or regenerated if such a transaction is added to the workload. If a transaction causing package cache inserts had been included in the initial workload when the diagnosis tree was created, the diagnosis tree would be able to diagnose package cache insert problems. If new transactions generate performance problems beyond the scope of the diagnosis tree, it will be unable to diagnose these problems.

A significant change in the DBMS workload will also cause the diagnosis tree to need to be retuned or rebuilt. The experiments presented in this dissertation demonstrate that the addition of new OLTP transactions, changing transaction frequencies and increasing the size of the database do not alter the effectiveness of the diagnosis tree. The addition of queries that are not handled by the diagnosis tree, such as queries that cause significant package cache inserts, will reduce the effectiveness of the diagnosis tree.

The lifespan of the diagnosis tree is also based on the hardware configuration. Changes to the hardware configuration such as adding more memory or disks will alter the effectiveness of the diagnosis tree. Resources will react differently in the DBMS depending on the underlying hardware, rendering the data collected to tune the diagnosis tree obsolete. A change in hardware, such as modifying the number of disks or amount of memory, may result in the need to retune the diagnosis tree.

5.8 Summary

The results over the six test cases have shown to be very positive in diagnosing DBMS resource problems and helping to improve overall database performance. Using only a naïve tuning strategy, we are able to correctly diagnose and tune each test case to within a few percentage points of the expert throughput, and in all cases we are able to outperform the wizard throughput. Our measure of success was to achieve at least the performance of the Tuning Wizard, something that we have surpassed. By matching the performance of the expert configuration, we deem this method successful for the diagnosis of performance problems in our test DBMS.

Building and tuning the various models and diagnosis trees have provided several opportunities for the acquisition of knowledge. Several lessons were learned over the course of model construction and testing, including the following:

- The relationships between resources are complex and poorly understood.
- The performance impact of some resources is complex and poorly understood.
- The relationships between resources can have a significant impact on the diagnosis process.
- Resource performance can only be used for diagnosis and tuning purposes if the state and performance of related resources are also taken into consideration.
- Workload throughput may not be the best measuring unit during the tuning process as some useful tuning steps may result in lower throughput values. Further tuning steps may result in performance increases that compensate for temporary drops in throughput.

- Using diagnosis and relationship models results in a well-defined strategy for diagnosis and tuning. This provides a clear diagnosis and tuning path that can be easily studied to derive new tuning strategies.

Our initial attempt to create a diagnosis system has resulted in a robust diagnosis tree and resource model that is presently able to diagnosis performance problems in several different scenarios. These scenarios include changing the size of the database, changing the frequency of transactions within the workload, adding new transactions to the workload, and combinations of both types of workload modification with database size changes. The accuracy of the diagnosis system given these six situations leads us to conclude that the diagnosis tree is able to handle workload changes that may be seen by a database over time. The robustness of the workload lends strength to belief that the creation of a single diagnosis tree for a given workload type is possible, bringing us closer to a completely automated DBMS.

Chapter 6

Conclusions

Achieving a high level of performance from a DBMS is a difficult task. The inherent competition that exists between DBMS tasks places a strain on various hardware and software resources. A delicate balance must be achieved when allocating resources in order to obtain high levels of performance. The issue of performance is further complicated by the fact that DBMSs do not stay in tune forever – as the data in the database changes and the application workload changes, database performance may drop. A DBMS must be tuned regularly in order to maintain high levels of performance. The automation of DBMS resource management removes the need for human interaction in order to maintain performance levels. Automation involves two steps: diagnosing the offending resource and then adjusting that resource to increase performance. With hundreds of possible DBMS resources to adjust, diagnosing which resource to adjust is the starting point for self-managing DBMSs. This dissertation focuses on the diagnosis process, outlining a framework for automating the diagnosis process.

6.1 Contributions

Designing an automated DBMS diagnosis system is an important research issue. As the complexity of DBMSs and workloads increase, the need for an automated system to configure and maintain the DBMS increases. DBMS diagnosis and tuning is time-consuming, repetitive and expensive. The ability to automatically diagnose performance problems will reduce the need for expensive DBAs and lower the operating costs associated with a DBMS. A reduction in operating costs coupled with a decrease in the complexity of database administration will encourage a more widespread use of DBMSs.

This dissertation demonstrates that a diagnosis framework can be constructed to automatically diagnose at least a subset of DBMS performance problems. The diagnosis model and the resource model are designed so that they can adapt to a specific DBMS environment. The models are designed to fit within the Quartermaster framework to provide a fully automated diagnosis and tuning system. The research contributions of this work include:

- A formal definition the DBMS diagnosis problem and analysis of the complexity of the problem. We show heuristic methods are required to solve the problem.
- The development of the diagnosis model demonstrates that the diagnosis process can be successfully automated. The results presented in Chapter 5 confirm the ability of a sample diagnosis tree to correctly identify system bottlenecks for a generic OLTP workload. The diagnosis model provides a basis for the creation of other diagnosis trees for different database workloads and different DBMSs.

- Chapter 4 presents generalized methods used to create diagnosis trees. These methods can be applied to different workloads in order to determine how that particular database should be tuned. The methods presented generate specific information for the workload, DBMS and hardware configuration at hand. This information can be used to generate specific diagnosis and tuning rules while avoiding generalized tuning rules that may not apply to the given setup. The presentation of these methods sets the stage for further research into automatic generation of the diagnosis tree.
- The models used demonstrate that the diagnosis system is consistently able to correctly diagnose performance problems on a working DBMS. The diagnosis system is able to perform when the database size changes, the workload frequencies are changed or new transactions are added. The results presented in Chapter 5 show that the diagnosis system is able to adapt to a changing workload, further exemplifying the versatility of the model.
- The models presented in Chapter 3 provide the basis for a generic tuning model. The models presented in this dissertation can be applied to other software systems where resource allocation is an issue.

6.2 Future Work

The research in this dissertation has many possible extensions. Relevant research topics include the automatic generation and modification of diagnosis trees, the creation of a generic workload, the convergence of the diagnosis process, increasing the number of resources that can be diagnosed by the system, the creation of intelligent tuning

algorithms and integrating the diagnosis tree into a DBMS application. The creation of the diagnosis tree is presently done by hand using performance data collected while running the workload. The diagnosis tree is then modified by hand when adjustments are needed. Further research is needed into automating the construction and modification of the diagnosis tree. The ability to automate the construction of the diagnosis tree would almost completely remove the need for a DBA to tune the DBMS, further reducing the operating costs for the DBMS. A learning and self-modifying diagnosis tree would be able to alter itself as the workload changed, adjusting threshold values and node positioning in order to maintain peak DBMS performance regardless of the workload.

Further research is needed into the creation of a generic test workload. Presently the workload used to create and tune the diagnosis tree is the same workload that would be run on the target system. The creation of a generic OLAP or OLTP workload would allow the creation of a diagnosis tree based on the resource interactions of a generic workload. This would result in a more generic diagnosis tree able to handle all new transactions added to the workload, eliminating the problem of a diagnosis tree becoming outdated when new types of transactions are added to the workload.

The convergence of the diagnostic process to insure optimal performance also deserves consideration. There is presently no method to determine if the diagnosis tree will eventually tune the DBMS to a state of optimal performance. Research into the optimal performance of a DBMS and the ability of the diagnosis tree to tune the DBMS correctly is needed.

Significant work is needed to increase the size of the diagnosis tree. At present, the number of resources diagnosed by the diagnosis tree is only a portion of the resources available for adjustment. An increase in the number of resources diagnosed is needed if a completely self-diagnosing system is to be achieved.

An increase in the number of automatic tuning algorithms must coincide with a larger diagnosis tree. An increase in the size of the diagnosis tree is not beneficial for an automated system unless the appropriate tuning algorithms exist for the resources diagnosed. Replacing the present naïve tuning strategy with intelligent tuning algorithms will greatly reduce the number of iterations needed to tune the DBMS.

The present diagnostic system is intended to run on performance data that was collected from a running DBMS. The diagnosis system is run separately from the DBMS and the results of the diagnosis are then applied to the DBMS. Research into the integration of the diagnosis system into a DBMS application is needed. Integration would supply the diagnosis system with more performance data, allowing the diagnosis system to better determine how the DBMS is working and suggest more appropriate resource allocations.

Additional research is needed to determine how the diagnosis algorithms and the Quartermaster framework can be applied to other software applications, such as web servers, operating systems and other resource-intensive applications.

Finally, work is needed to integrate the diagnosis system into a working DBMS. The present diagnosis system is designed and constructed to work outside of a DBMS. Integrating the diagnosis system into a DBMS may result in changes to both the DBMS and the diagnosis system. Integration of the diagnosis system should begin on a small scale with the implementation of a diagnosis tree and tuning algorithms that can manage some of the DBMS resources. The initial integration of a diagnosis system for well-understood resources with well-defined tuning algorithms will serve as a well-defined enhancement to a DBMS that can be further tested for design and implementation issues as well as an easily understood addition for customers to use. By slowly introducing a well-defined automatic diagnosis to the customer base, a DBMS company will more easily be able to convince customers of the merits of an automated diagnosis and tuning system. As automated diagnosis becomes better understood at the developer and customer levels, the scope of the diagnosis algorithms can be widened to include more resources until all DBMS resources are managed automatically.

The integration of the automated diagnosis system will require some changes to be made at the DBMS level. One of the most significant changes that must be made to the DBMS is the ability to modify resource allocations without having to stop the DBMS engine. Dynamically adjusting resources is key to fully automating the resource diagnosis and tuning process. It is also expected that the diagnosis system will have to be modified to allow DBAs to override various DBMS resource settings. The present diagnosis system assumes that all of the resources in the diagnosis tree are available for tuning. It is possible that a DBA may decide, for some reason unknown to the diagnosis process, that

a resource should not be adjusted. The diagnosis algorithm will have to be adjusted to handle such situations where only a subset of the resources can be adjusted.

The automatic diagnosis system presented in this dissertation is designed to run parallel to any DBMS. It is expected that integration of the diagnosis system into a DBMS will result in significant changes to the implementation of the diagnosis system. It is believed that the underlying principles of our automated diagnosis system will remain regardless of how it is implemented.

References

[AAM94] Agnar Aamodt and Enric Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches, *Artificial Intelligence Communications*, IOS Press, Vol. 7 No. 1, pp. 39- 59, 1994.

[AGR00] Sanjay Agrawal, Surajit Chaudhuri and Vivek Narasayya. Automated Selection of Materialized Views and Indexes for SQL Databases, *Proceedings of the 26th International Conference on Very Large Databases*, September 10-14, Cairo, Egypt, pp. 496-505, Morgan Kaufmann Publishers, 2000.

[BAC84] J. Bachant and J. McDermott. R1 revisited: Four years in the trenches, *AI Magazine* Vol. 5, No. 3, 1984.

[BEN99] Darcy Benoit, Wendy Powley and Patrick Martin. Quartermaster: A Framework for Automatic Tuning of Database Management Systems, *Department of Computing and Information Science*, Queen's University, June 1999.

[BER98] Phil Bernstein *et al*. The Asilomar Report on Database Research, *SIGMOD Record*, Vol. 27, No. 4, pp. 74-80, December 1998.

[BEL57] Richard Bellman. *Dynamic Programming*, Princeton University Press, 1957.

[BIG00] J.P. Bigus, J.L. Hellerstein, T.S. Jayram, and M.S. Squillante. AutoTune: A Generic Agent for Automated Performance Tuning, *Practical Application of Intelligent Agents and Multi Agent Technology*, 2000.

[BIR93] R.S. Bird and O. de Moor. From Dynamic Programming to Greedy Algorithms. In *Formal Program Development*, Volume 755 of *Lecture Notes in Computer Science, C. Moller, H. Partsch and S. Schuman, Editors, pages 43-61, 1993.*

[BRO92] Carla E. Brodley and Paul E. Utgoff. *Multivariate Decision Trees*, COINS Technical Report 92-82, University of Massachusetts, December 1992.

[BRO93] Kurt P. Brown, Michael J. Carey and Miron Livny. Towards an Autopilot in the DBMS Performance Cockpit, *High Performance Transaction Systems Workshop*, 1993.

[BRO94] Kurt P. Brown, Manish Mehta, Michael J. Carey and Miron Livny. Towards Automated Performance Tuning For Complex Workloads, *Proceedings of the 20th International VLDB Conference*, pp. 72-84, Santiago, Chile, 1994.

[BRO95] Kurt P. Brown. Goal-Oriented Memory Allocation in Database Management Systems, *PhD Thesis*, University of Wisconsin-Madison, 1995.

[CHA99] Surajit Chaudhuri. Letter from the Special Issue Editor, *Bulletin of the Technical Committee on Data Engineering*, Vol. 22, No. 2, page 2, June 1999.

[CHA00] Surajit Chaudhuri and Gerhard Weikum. Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database System, *Proceedings of the 26th International Conference on Very Large Databases*, pp. 1-10, Cairo, Egypt, 2000.

[CHA00-2] Surajit Chaudhuri and Vivek Narasayya. Automating Statistics Management for Query Optimizers. *Proceedings of 16th International Conference on Data Engineering*, pp. 339-348, San Diego, USA 2000

[CHU95] Jen-Yao Chung, Donald Ferguson, George Wang, Christos Nikolaou and Jim Teng. Goal oriented dynamic buffer pool management for data base systems, *IBM Technical Report TR94-0125*, 1994.

[CHV83] Vašek Chvátal. *Linear Programming*, W.H. Freeman and Company, 1983.

[COL00] *The Columbia Encyclopedia*, Sixth Edition, http://www.bartleby.com/65/li/linearpr.html, 2001.

[CUR96] Sharon Curtis. *A Relational Approach to Optimization Problem*, PhD Thesis, Somerville College, University of Oxford, April 1996.

[CUR97] Sharon Curtis. Dynamic Programming: A Different Perspective, In *Algorithmic Languages and Calculi*, R. Bird and L. Meertens, Editors, pp. 1-23, Chapman & Hall, London, U.K., 1997.

[DAV92] Randall Davis and Walter Hamscher. Model-based reasoning: Troubleshooting, in *Reading in Model-based Diagnosis*, pp. 3-24, Morgan Kaufmann Publishers, 1992.

[deK92] Johan deKleer and Brian C. Williams. Diagnosing Multiple Faults, in *Readings in Model-based Diagnosis*, pp. 100-117, Morgan Kaufmann Publishers, 1992.

[deK89] Johan deKleer and Brian C. Williams. Diagnosis with Behavioral Modes, *Proceedings IJCAI-89*, Detroit, Mi, pp104-109, 1989.

[EOP02] http://www.e-optimization.com/resources

[GAS75] Saul Gass. *Linear Programming: Methods and Applications, Fourth Edition*, The Kingsport Press, 1975.

[HAR99] David G. Hart, Joseph L. Hellerstein and Po C. Yue. Automated Drill Down: An Approach To Automated Problem Isolation For Performance Management, *In Proceedings of the Computer Measurement Group*, pp. 376-384, 1999.

[HEL97] Joseph L. Hellerstein. Automated Tuning Systems: Beyond Decision Support. *In the Proceedings of the 1997 Computer Measurement Group*, 1997.

[HOR88] Eric Horvitz, John Breese and Max Henrion. Decision Theory in Expert Systems and Artificial Intelligence, *International Journal of Approximate Reasoning*, pp. 247-302, 1988.

[IBM00] IBM DB2 Universal Database Administration Guide Version 7.1, 2000.

[JOH92] Robert Johnson. *Elementary Statistics*, Sixth Edition, PWS-Kent Publishing Company, 1992.

[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, New Jersey, 1984.

[LEA96] David Leake. CBR in Context: The Present and Future, in *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, D. Leake, Editor, pp. 3-30, AAAI Press/MIT Press, 1996.

[LEU93] Scott T. Leutenegger and Daniel Dias. A Modeling Study of the TPC-C Benchmark, In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 22-31, Washington, D.C., May 1993.

[LOM99] David Lomet. Letter from the Editor-in-Chief, *Bulletin of the Technical Committee on Data Engineering,* Vol. 22, No. 2, page 1, June 1999.

[LUG93] George F. Luger and William A. Stubblefield. *Artificial Intelligence – Structures and Strategies for Complex Problem Solving,* Second Edition, 1993.

[MAR00] Patrick Martin, Min Zheng, Hoi-Ying Li, Keri Romanufa and Wendy Powley. Dynamic Reconfiguration: Dynamically Tuning Multiple Buffer Pools, *Proceedings of the International Conference on Database and Expert System Applications* (DEXA'2000), pp. 92-101, September, 2000.

[MOL89] Michael K. Molloy. *Fundamentals of Performance Modeling,* Macmillan Publishing Company, New York, 1989.

[MOZ91] Igor Mozetic. Hierarchical Model-Based Diagnosis, *International Journal of Man-Machine Studies,* Vol. 35 No. 3, pp 329-362, 1991.

[MYL95] John Mylopoulos, Vinay Chaudhri, Dimitris Plexousakis, Adel Shrufi and Thodoros Topaloglou. Building Knowledge Base Management Systems, *Very Large Databases Journal,* Vol. 5, No. 4, pp. 238-263, October 1996.

[PAR01] Sujay Parekh, Neha Gandhi, Joseph Hellerstein, Dawn Tilbury, T.S. Jayram, and Joe Bigus. Using Control Theory to Achieve Service Level Objectives In Performance Management, in *Real-Time Systems,* Vol. 23, No. 1-2, pp. 127-141, 2002.

[PAU98] Daniel Paul, Sudhakak Yalamanchili, Karsten Schwan, and Rakesh Jha. *Decision Models for Adaptive Resource Management in Multiprocessor Systems*, http://www.htc.honeywell.com/projects/arm/, 1998.

[RYM92] Ron Rymon. More On Goal-Directed Diagnosis, in *Proceedings of the Third International Workshop on Principles of Diagnosis*, 1992.

[SEV81] Kenneth Sevcik. Data base System Performance Prediction Using an Analytical Model, In *Proceedings of the 7th International Conference on Very Large Data Bases*, Cannes, France, pp. 182-197, September, 1981.

[SHI00] Tetsuya Shirai *et al*. *DB2 UDB V7.1 Performance Tuning Guide*, IBM Redbooks, 2000.

[SWE99] Steven Sweet. Think About It: Artificial Intelligence & Expert Systems, *How Computers Work*, Vol. 3, Issue 4, November 1999.

[TPC] *TPC-C Benchmark Specification*, http://www.tpc.org

[TPC2] TPC Website, http://www.tpc.org/

[WEI94] Gerhard Weikum, Christof Hasse, Axel Mönkeberg and Peter Zabback. The Comfort Automatic Tuning Project, *Information Systems*, Vol. 19, No. 5, pages 381-432, 1994.

[WEI99] Gerhard Weikum, Arnd Christian König, Achim Kraiss and Markus Sinnwell. Towards Self-Tuning Memory Management for Data Servers, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, pp. 3-11, 1999.

[XU02] Xiaoyi Xu, Patrick Martin and Wendy Powley. Configuring Buffer Pools in DB2 UDB, *IBM Centre for Advanced Studies Conference (CASCON),* Toronto, 2002.

[ZUZ95] Alenka Zuzek, Franc Novak, Anton Biasizzo, Iztok Savnik, and Bojan Cestnik. Sequential Diagnosis Tool for System Maintenance and Repair, *Electronical Review*, Ljubljana, Vol. 62, No. 3-4, pp. 224-231, 1995.

Appendix A

Test Environment

The test hardware consists of an IBM eServer xSeries 240 with two Intel Pentium-III 1GHz processors and 2 GB of RAM. The system contains 22 SCSI 7200rpm disks spread across five disk controllers. Two of the disks are located in the main computer case on a single drive controller and are used for the operating system, the DBMS software and log files. The other 20 disks are located in two EXP300 external storage expansion units. Each of four disk controllers run five disks in the expansion units. The data is spread across all 20 of the disks. The operating system used is Windows NT 4.0 Server (Service Pack 6). The DBMS software is IBM's DB2/UDB version 7.1.

Appendix B

TPC-C Benchmark

The Transaction Processing Performance Council (TPC) is a non-profit corporation founded to define transaction processing, create standardized database benchmarks, and be the distribution point for benchmark results. The goal of the TPC is to create benchmarks that can be run on any hardware, operating system and DBMS combination [TPC2]. The third benchmark produced by the TPC is known as the "C" benchmark and is directed toward On-Line Transaction Processing (OLTP) systems. The "C" benchmark, commonly referred to as "TPC-C", is modeled after real production systems and is intended to simulate an order-entry environment. The environment includes entering orders, delivering orders, recording payments, checking order status and stock level monitoring.

The logical database design of a TPC-C database is based on the nine relations found in Table 13 [LEU93]. The schema for the TPC-C database is found in Figure 28 [TPC]. In Figure 28, the numbers in the entity blocks represent the cardinality of the tables and the numbers next to the arrows represent the cardinality of the relationships. The plus symbol is used to denote that a number is subject to small variations [TPC].

Relation Name	Cardinality	Tuple Length (bytes)	Tuples Per 4K page
Warehouse	W	89	46
District	W*10	95	43
Customer	W*30K	655	6
Stock	W*100K	306	13
Item	100K	82	49
Order		24	170
New-Order		8	512
Order-Line		54	75
History		46	89

Table 13 - TPC-C data relations.

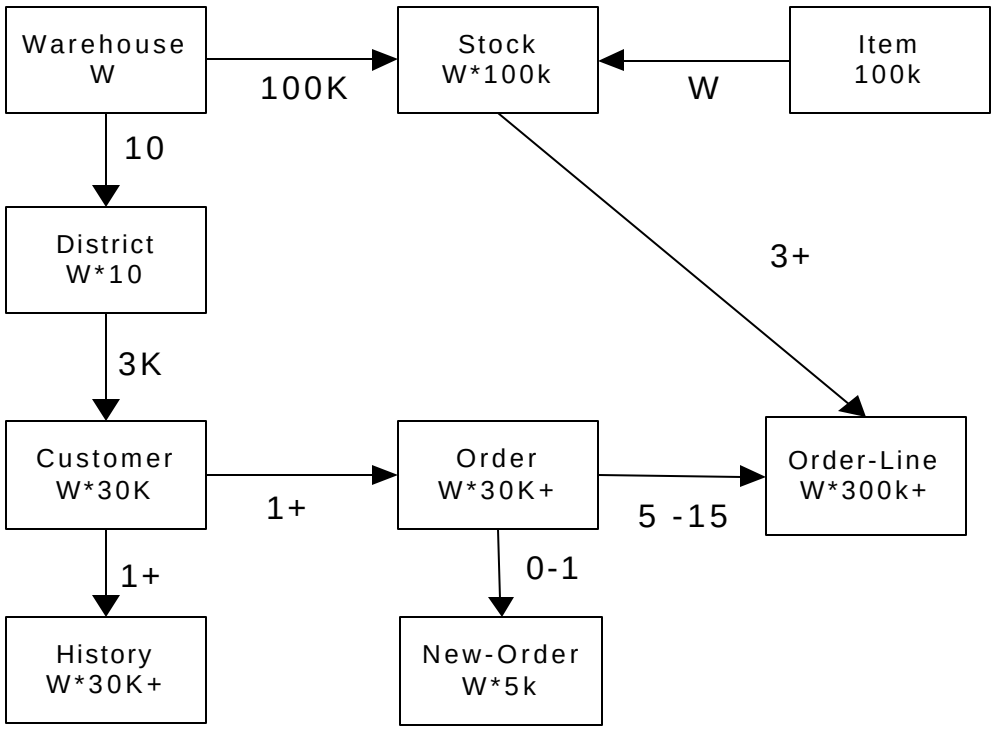


Figure 28 - TPC-C table schema.

TPC-C consists of five transactions. Each transaction is required to execute a specific percentage of all transactions executed by the benchmark. The transaction requirements are listed in Table 14. The transactions are as follows: [TPC]

- **New Order** is a read-write transaction that places an order for items in the warehouse. New Order has a high frequency of execution.
- **Payment** updates the balance of the customer accounts and propagates this information to the district and warehouse sales statistics. Payment has a high frequency of execution.
- **Order-Status** is a read-only transaction that queries the status of customer orders. It has a low frequency of execution.
- **Delivery** is a transaction based on processing batches of new orders. Orders are processed in batches of 10. Delivery has a low frequency of execution.
- **Stock Level** checks stocks levels to determine which stock is below a specified threshold. Stock level is a read-only transaction with a low frequency of execution.

Transaction Name	**Percentage of workload**
New Order	45%
Payment	43%
Order Status	4%
Delivery	4%
Stock Level	4%

Table 14 - Transaction requirements.

Appendix C

DBMS Resources

Asynchronous Page Cleaners – (see I/O Cleaners)

Buffer Pool Size – A buffer pool is a segmented piece of memory used by the database to cache data. The size of the buffer pool can be adjusted by DBAs.

Catalog Cache – Catalog cache is a defined portion of memory used to cache the database catalog.

Changed Pages Threshold (CPT) – The Changed Pages Threshold is the percentage of modified pages allowed in the buffer pool before the I/O cleaner processes are started.

Database Heap – The database heap is the portion of memory allocated per database for use by all applications connected to the database.

Deadlock Check Time – The interval of time the DBMS will wait before checking for deadlocks.

I/O Cleaners – I/O cleaners are processes used by the DBMS to perform asynchronous writes to disk.

Log Buffer – The log buffer is an allocation section of memory used to buffer data that is to be written to the database logs.

Lock Timeout – The amount of time two deadlocked process will wait before they time out and fail.

Log File Size – The size of each log file.

Maxappls – The maximum number of concurrent applications that can be connected to a database.

Maxlocks – The maximum percentage of lock list to be used before lock escalations occur.

Number of commits to group – Specifies the number of commits that can be buffered before being written to disk.

Softmax – The percentage of the log file that would need to be recovered after a crash. This value can be set above 100, allowing multiple files to need restoration after a crash.

Sort Heap Size - The amount of memory allocated for sorts.

Sort Heap Threshold – Sort heap threshold is an instance-wide soft limit on the total amount of memory available for private sorts. Sort heap threshold is also a database-wide hard limit on the memory available for shared sorts.

Appendix D

Performance Data Collected

Note: All values are measured for the full data collection period unless otherwise specified.

Data collected	Description
Number of Transactions	The total number of transactions completed.
Dirty Page Steals	The total number of times a dirty page was synchronously written to disk by a transaction process.
Log Triggers	The total number of times the log threshold was triggered.
Threshold Triggers	The total number of times the changed pages threshold was reached.
Logical Reads	The total number of logical reads.
Physical Reads	The total number of reads that required disk access.
Data Writes	The total number of data pages written to disk.
Index Writes	The total number of index pages written to disk.
Asynchronous Data Writes	The total number of data writes made to disk asynchronously.
Asynchronous Index Writes	The total number of index writes made to disk asynchronously.
Asynchronous Reads	The total number of disk reads that were made asynchronously.
Asynchronous Read Requests	The total number of asynchronous read requests.
Physical Read Time	The total amount of time spent doing physical reads.

Physical Write Time	The total amount of time spent doing physical writes.
Catalog Cache Inserts	The total number of inserts made into the catalog cache.
Package Cache Inserts	The total number of inserts made into the package cache.
Lock Waits	The total number of times processes had to wait for locks.
Lock Wait Time	The total amount of time spent waiting for locks.
Lock List in use	The amount of space in the lock list being used at the time of the snapshot.
Deadlocks	The total number of deadlocks that have occurred.
Lock Escalations	The number of times that locks had to be escalated in order to reduce the total number of locks.
Sort Heap Size	The total amount of memory allocated for sorts.
Sort Overflows	The total number of sorts that have overflowed the sort heap.
Post-threshold Sorts	The total number of sorts that have requested space after the sort threshold has been reached.
Number of Sorts	The number of sorts that have occurred.
Sort Time	The total amount of time spent performing sorts.

Appendix E

Glossary of Terms

Asynchronous write – An asynchronous write is when data is written back to disk by a background I/O cleaner process. Asynchronous writes are beneficial as groups of data can be written at a time.

Buffer Pool – A buffer pool is a segmented piece of memory used by the database to cache data. All data is read and written through the buffer pool.

Buffer pool hit rate – The hit rate of a buffer pool is defined as the percentage of times data is found in the memory as opposed to disk. The formula used for the buffer pool hit rate is:

$$\text{Hit rate} = \frac{\text{Number of logical reads} - \text{Number of physical reads}}{\text{Number of logical reads}}$$

Catalog Cache – Catalog cache is a defined portion of memory used to cache the database catalog.

Changed Pages Threshold (CPT) – The Changed Pages Threshold is the percentage of modified pages allowed in the buffer pool before the I/O cleaner processes are started.

Database Heap – The database heap is the portion of memory allocated per database for use by all applications connected to the database. The database heap contains

control block information and reserves space for the catalog cache and the log buffer.

DB2 Performance Wizard – The performance wizard is an application included with DB2. The application gathers information about the hardware and workload and suggests resource adjustments to improve performance.

DBMS – Database Management System

DBA – Database Administrator

Deadlock Check Time – The interval of time the DBMS will wait before checking for deadlocks.

Decision Node – A decision node is a non-leaf node of the decision tree. Each decision node contains a list of questions related to the performance of the DBMS. The results of the questions will determine the path the tree traversal algorithm will follow.

Decision Tree – A binary tree structure containing decision nodes and tuning nodes.

Dirty page steals – A dirty page steal occurs when a database agent is unable to find a clean page in memory. A modified (i.e. "dirty") page is selected and written to disk by the database agent. A dirty page steal is a synchronous write to disk and should be avoided as the database agent must pause execution of a transaction in order to perform the write to disk.

Indicator Values – An indicator value is a measured or calculated value used to judge the performance of a DBMS resource.

I/O Cleaners – I/O cleaners are background processes used by the DBMS to perform asynchronous writes to disk.

Lock Escalation – Lock escalation occurs when there is not enough room in the allocated locklist memory for all of the locks needed. Escalation is when locks are upgraded in order to save memory space, such as upgrading several row-level locks for one table-level lock.

Log Buffer – The log buffer is a section of memory used to buffer data that is to be written to the database logs.

OLAP Workload – OLAP stands for "On-Line Analytical Processing". An OLAP workload consists of decision-support queries of high complexity and low volume.

OLTP Workload – OLTP stands for "On-Line Transaction Processing". An OLTP workload consists of low complexity queries in high volume.

Resource – A resource is a piece of hardware or software that is in limited supply and can be regulated in usage.

Synchronous Writes – A synchronous write occurs when a database agent is forced to write data to disk. The agent must pause transaction processing in order to make write the data to disk.

Threshold Values – Threshold values are numerical values used as the basis of comparison to determine how well the DBMS is performing. Indicator values are compared to threshold values to determine which path will be followed through the diagnosis tree.

TPC-C – TPC-C is the Transaction Processing Performance Council benchmark for OLTP workloads.

tpmC – Transactions per Minute "C" for the TPC-C benchmark.

Tuning Node – A tuning node is a leaf node in the diagnosis tree used to store tuning suggestions.

Appendix F

Confidence Intervals

Data collected every five seconds over the final 17 minutes of a 20 minute run was used to calculate the standard deviation of the set. The throughput values used are listed in Table 15. The standard deviation for this data is 185.0. The mean is 3414.97.

Using the standard deviation of 185.0, a set size of 205 elements, a Z value of 1.96 and a significance level of 5%, the confidence interval is calculated to be 25.3 tpmC. In other words, the throughput is within 25.3 tpmC 95% of the time. Figure 29 contains the equation used to calculate the confidence interval [JOH92].

$$n = \left(\frac{z\ (\alpha/2) * sd}{E} \right)^2$$

Figure 29 - Confidence interval equation.

Several assumptions where made for the calculation of the confidence interval. The first assumption is that the workload is stable. This assumption is based on the fact that the TPC-C benchmarking workload is well-known as a stable benchmarking. A second assumption is that the various measurements taken are representative of the actual performance of the workload. Given that the TPC-C workload is a repetition of five well-defined transactions, the measurement at each interval is the calculation of the workload throughput for the transactions performed during that given interval. The transaction mix remains constant for the duration of the run, resulting in comparable five second measurement intervals. It is from these sample measurements that we calculate the confidence interval for collected data.

3372	3708	3288	3360	3600	3540	3384	3876	3612
3216	3336	3336	3432	3408	3360	3288	3420	3612
3168	3288	3708	3384	3300	2976	3192	3468	3552
3204	3048	3456	3204	3468	3408	3180	3528	3444
3072	3096	3528	3192	3600	3456	3240	3480	3336
3492	3528	3312	3588	3192	3636	3576	3432	
3228	3504	3360	3588	3504	3312	3348	3264	
3456	3312	3780	3348	3600	3276	3696	3468	
3528	3108	3324	3396	3396	3528	3588	3408	
3072	3684	3564	3504	3492	3204	3624	3216	
3360	2928	3864	3516	3396	3768	3096	3528	
3264	3264	3480	3420	3372	3600	3120	3264	
3480	3444	3480	3732	3504	3132	3408	3672	
3600	3528	3372	3096	3360	3444	3660	3648	
3516	3168	3312	3528	3312	3912	3312	3216	
3552	3492	3276	3240	3156	3384	3564	3444	
3396	3756	3252	3552	3216	3780	3480	3384	
3204	3252	3576	3624	3024	3276	3648	3504	
3216	3624	2976	3216	3372	3756	3336	3636	
3240	3564	3636	3588	3492	3540	3276	3420	
3564	3552	3288	3468	3336	3552	3528	3324	
3348	3396	3456	3636	2940	3552	3336	3540	
3240	3324	3408	3504	3408	3312	3360	3252	
3444	3312	3276	3300	3612	3636	3336	3228	
3504	3300	3552	3600	3204	3384	3708	3492	

Table 15 - Data used for standard deviation calculation.

Appendix G

Performance Monitor Database Schema

The following database schema was used to store data collected by the performance data collection program. Information is then retrieved from this database for use by the diagnosis algorithm.

```
cc.TimeStamp(
        timestamp timestamp not null primary key,
        monitorlength integer);

cc.Cache(
        timestamp timestamp not null primary key,
        cataloginserts integer,
        packageinserts integer);

cc.Locks(
        timestamp timestamp not null primary key,
        lockwaits integer,
        lockwaittime integer,
        locklistinuse integer,
        deadlocks integer,
        lockescalations integer);

cc.TransClass(
        Name varchar(20) not null primary key,
        Freq integer,
        Type varchar(15),
        SQL varchar(500),
        NumLogicalReads smallint);
```

```
cc.TCPerfData(
        TCID varchar(20) not null,
        timestamp timestamp not null,
        ResponseTime decimal(7,4),
        SnapNumber integer,
        IntervalLength smallint,
        Deadlocks smallint,
        Rejects smallint,
        primary key(TCID, timestamp));

cc.BPPerfData(
        bpid varchar(20) not null,
        bpsize integer,
        pagesize integer,
        TCName varchar(20),
        timestamp timestamp not null,
        numlogicalreads integer,
        numphysicalreads integer,
        datawrites integer,
        indexwrites integer,
        asyncdatawrites integer,
        asyncindexwrites integer,
        asyncreads integer,
        asyncreadreq integer,
        physicalreadtime integer,
        physicalwritetime integer,
        hitrate decimal(5,2) ,
        snap smallint,
        primary key(bpid, timestamp));

cc.FreqData(
        timestamp timestamp not null,
        NumTransRun integer);
        create table cc.Goal(
        TCID varchar(20) not null primary key,
        PerfMeasure varchar(20),
        Priority smallint,
        Type char(10),
        value decimal(7,4));

cc.UsesBP(
        BPId varchar(20) not null,
        TCID varchar(20) not null,
        weight decimal(7, 4),
        primary key(BPID, TCID));
```

```
cc.SortInfo(
        timestamp timestamp not null primary key,
        heap_allocated integer,
        overflows integer,
        post_threshold integer,
        total_number integer,
        total_time integer,
        active integer,
        piped_requested integer,
        piped_accepted integer);

cc.AsynchIOCleaners(
        timestamp timestamp not null primary key,
        dirtypagesteals integer,
        logtriggers integer,
        thrshtriggers integer);
```

Appendix H

Decision Database Schema

```
resource_area (
        name VARCHAR(50) PRIMARY KEY NOT NULL);

resource (
        name VARCHAR(50) PRIMARY KEY NOT NULL);

setting (
        name VARCHAR(50) PRIMARY KEY NOT NULL,
        default_value VARCHAR(50),
        range_value VARCHAR(50),
        unit_of_measure VARCHAR(50),
        present_value VARCHAR(50),
        impact VARCHAR(50));

msetting (
        parent VARCHAR(50) NOT NULL,
        name VARCHAR(50) NOT NULL,
        default_value VARCHAR(50),
        range_value VARCHAR(50),
        unit_of_measure VARCHAR(50),
        present_value VARCHAR(50),
        impact VARCHAR(50),
        PRIMARY KEY (parent, name),
        FOREIGN KEY (parent) REFERENCES setting(name));

marker (
        name VARCHAR(50) PRIMARY KEY NOT NULL,
        type VARCHAR(50),
        unit_of_measure VARCHAR(50),
        value VARCHAR(50));

is_in (
        resource_area_name VARCHAR(50) NOT NULL,
```

```
        resource_name VARCHAR(50) NOT NULL,
        PRIMARY KEY (resource_area_name, resource_name),
        FOREIGN KEY (resource_area_name) REFERENCES resource_area(name),
        FOREIGN KEY (resource_name) REFERENCES resource(name));

has_setting (
        resource_name VARCHAR(50) NOT NULL,
        setting_name VARCHAR(50) NOT NULL,
        FOREIGN KEY (resource_name) REFERENCES resource(name),
        FOREIGN KEY (setting_name) REFERENCES setting(name));

uses_marker (
        resource_name VARCHAR(50) NOT NULL,
        marker_name VARCHAR(50) NOT NULL,
        FOREIGN KEY (marker_name) REFERENCES marker(name),
        FOREIGN KEY (resource_name) REFERENCES resource(name));

related_to (
        setting1 VARCHAR(50) NOT NULL,
        setting2 VARCHAR(50) NOT NULL,
        FOREIGN KEY (setting1) REFERENCES setting(name),
        FOREIGN KEY (setting2) REFERENCES setting(name));

decision (
        name VARCHAR(50) PRIMARY KEY NOT NULL,
        tuning VARCHAR(255));

decision_settings (
        decision VARCHAR(50) NOT NULL,
        setting VARCHAR(50) NOT NULL,
        PRIMARY KEY (decision, setting),
        FOREIGN KEY (decision) REFERENCES decision(name),
        FOREIGN KEY (setting) REFERENCES setting(name));

decision_tree (
        root VARCHAR(50) NOT NULL,
        child VARCHAR(50) NOT NULL,
        switch INT NOT NULL,
        PRIMARY KEY (root, child),
        FOREIGN KEY (root) REFERENCES decision(name),
        FOREIGN KEY (child) REFERENCES decision(name));

operator (
        name VARCHAR(50) PRIMARY KEY NOT NULL,
        symbol VARCHAR(50));
```

```
equation (
        equation_num INT PRIMARY KEY NOT NULL,
        setting1 VARCHAR(50) NOT NULL,
        setting2 VARCHAR(50) NOT NULL,
        issetting INT NOT NULL,
        operator VARCHAR(50) NOT NULL,
        value VARCHAR(50),
        FOREIGN KEY (setting1) REFERENCES setting(name),
        FOREIGN KEY (operator) REFERENCES operator(name));

has_equation (
        decision VARCHAR(50) NOT NULL,
        equation_num INT NOT NULL,
        FOREIGN KEY (decision) REFERENCES decision(name),
        FOREIGN KEY (equation_num) REFERENCES equation(equation_num));

threshold (
        name VARCHAR(50) PRIMARY KEY NOT NULL,
        value VARCHAR(50) NOT NULL,
        unit_of_measure VARCHAR(50) NOT NULL);

transaction (
        name VARCHAR(50) PRIMARY KEY NOT NULL,
        frequency INT NOT NULL,
        readlevel INT NOT NULL,
        priority INT NOT NULL);

trans_has_tables (
        transaction VARCHAR(50) NOT NULL,
        table VARCHAR(50) NOT NULL,
        PRIMARY KEY (transaction, table),
        FOREIGN KEY (transaction) REFERENCES transaction(name));

trans_has_index (
        transaction VARCHAR(50) NOT NULL,
        index VARCHAR(50) NOT NULL,
        PRIMARY KEY (transaction, index),
        FOREIGN KEY (transaction) REFERENCES transaction(name));

transactionclass (
        name VARCHAR(50) PRIMARY KEY NOT NULL);

workload (
        name VARCHAR(50) PRIMARY KEY NOT NULL);

has_transaction (
```

```
        class VARCHAR(50) NOT NULL,
        transaction VARCHAR(50) NOT NULL,
        PRIMARY KEY (class, transaction),
        FOREIGN KEY (class) REFERENCES transactionclass(name),
        FOREIGN KEY (transaction) REFERENCES transaction(name));

has_class (
        workload VARCHAR(50) NOT NULL,
        class VARCHAR(50) NOT NULL,
        PRIMARY KEY (workload, class),
        FOREIGN KEY (workload) REFERENCES workload(name),
        FOREIGN KEY (class) REFERENCES transactionclass(name));

update_history (
        name VARCHAR(50) PRIMARY KEY NOT NULL,
        start_date TIMESTAMP,
        end_date TIMESTAMP,
        setting VARCHAR(50),
        pvalue VARCHAR(50),
        avalue VARCHAR(50),
        pperformance VARCHAR(50),
        aperformance VARCHAR(50),
        feedback INT);

modify (
        num INT NOT NULL,
        name VARCHAR(50) NOT NULL,
        direction VARCHAR(20) NOT NULL,
        comments VARCHAR(255),
        PRIMARY KEY (num, name),
        FOREIGN KEY (name) REFERENCES setting(name));
```

Appendix I

Forward and Reverse Resource Trees

The following is a collection of forward and reverse resources trees used for this dissertation. The root node of each resource tree is noted in bold.

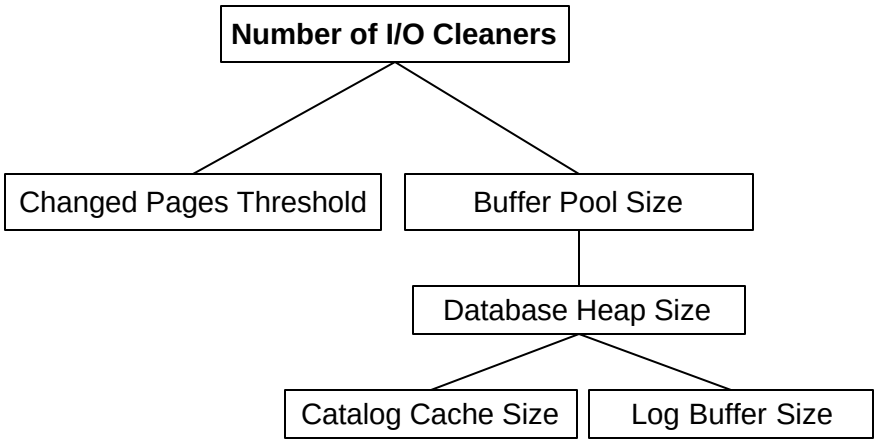


Figure 30 - Forward resource tree for the number of I/O cleaners resource.

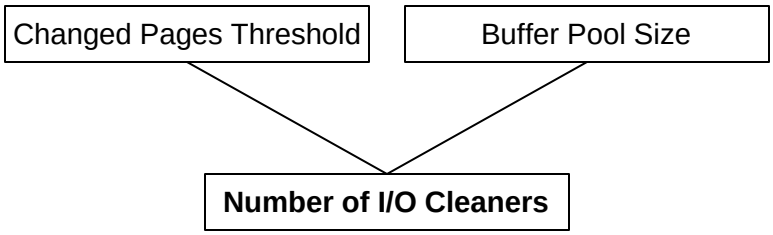


Figure 31 - Reverse resource tree for the number of I/O cleaners resource.

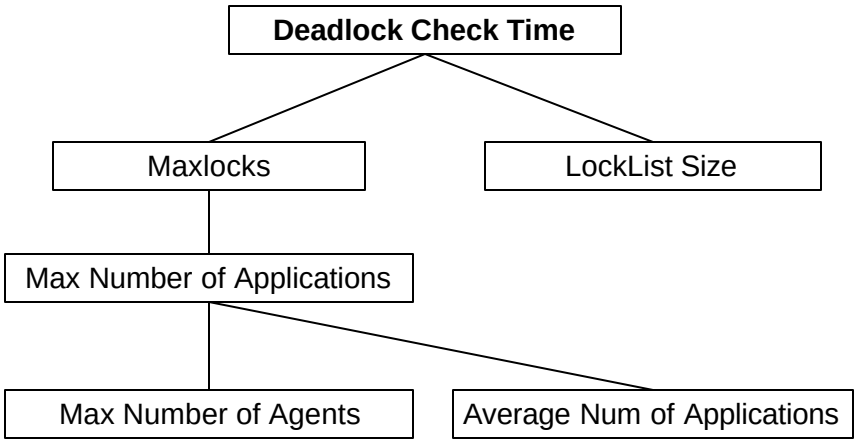


Figure 32 - Forward resource tree for the deadlock check time resource.

Figure 33 - Reverse resource tree for the deadlock check time resource.

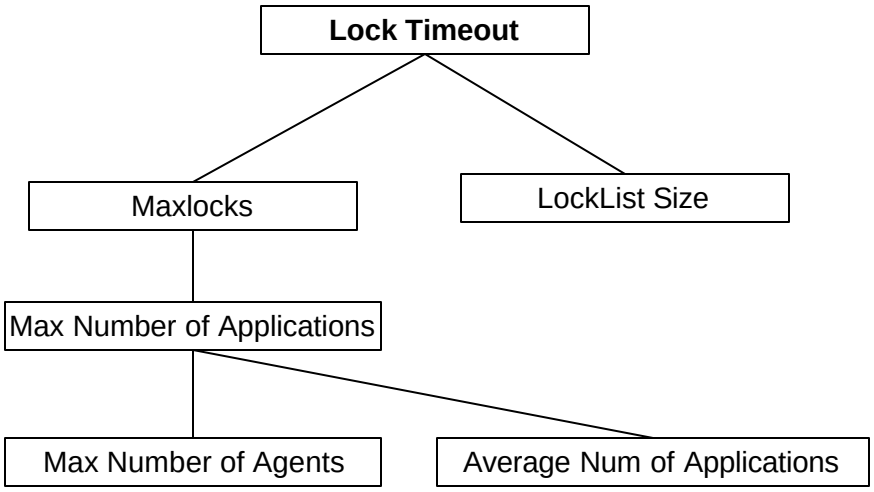


Figure 34 - Forward resource tree for the lock timeout resource.

Figure 35 - Reverse resource tree for the lock timeout resource.

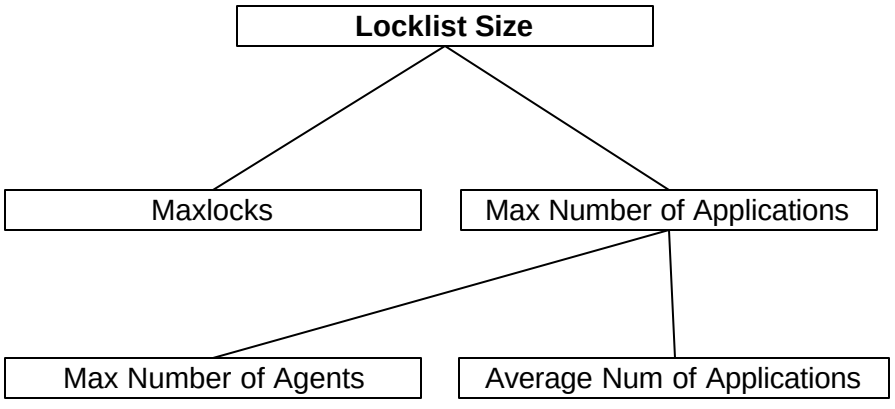


Figure 36 - Forward resource tree for the locklist size resource.

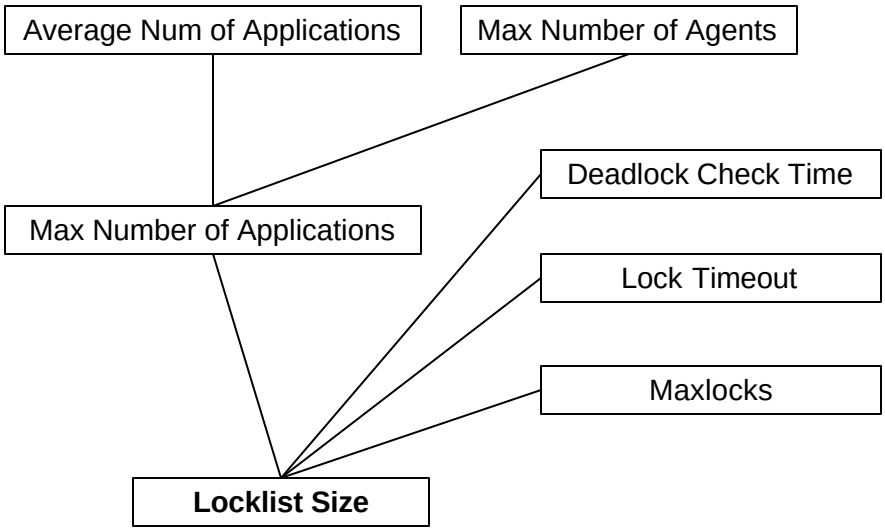


Figure 37 - Reverse resource tree for the locklist size resource.

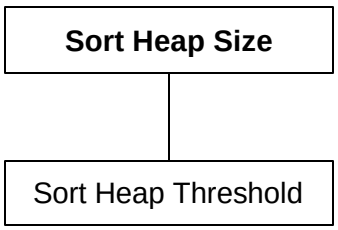


Figure 38 - Forward resource tree for the sort heap size resource.

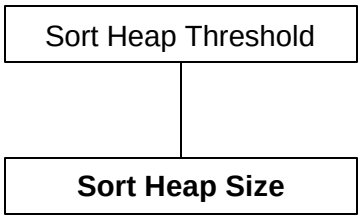


Figure 39 - Reverse resource tree for the sort heap size resource.

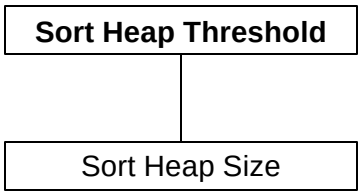


Figure 40 - Forward resource tree for the sort heap threshold resource.

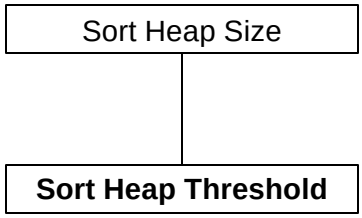


Figure 41 - Reverse resource tree for the sort heap threshold resource.

Appendix J

Statistical Analysis Data

Run #1

4044	4500	4032	4680	4140	4380
4056	4776	4416	4584	4272	4440
4092	4044	4020	3984	4716	4020
4224	4512	4428	4320	4212	4008
4164	4236	4284	4332	4188	4416
4308	4296	3960	4476	4380	4356
4284	3972	4356	3408	4608	4188
4068	4092	3984	4236	3912	3816
4452	4596	4032	4212	4632	4164
4596	4164	4248	4212	4260	4104
4320	4632	4080	4416	4056	4428
3972	4068	4008	4344	4308	4236
4548	3924	4128	4368	4524	4224
4440	4692	4152	4260	4440	4656
4296	4056	4116	4188	4164	4464
4392	4452	4212	4320	4344	4632
4416	4152	3804	4176	4452	3984
4068	4464	4320	4284	4272	4584
4452	3912	4116	4296	4044	4080
4404	4524	4764	4368	3912	4080
4380	4428	4344	4464	4668	3840
4176	4020	3900	4464	4332	3828
4416	4272	4404	4212	4368	3900
4272	4284	4356	4296	3936	4284
4476	4116	4392	3924	4128	4236
4344	4248	4536	4392	4080	4512
4092	4176	4284	4320	4152	
4080	4224	4032	4608	4200	
4536	3744	4320	4092	4284	
4296	4212	4440	4248	4524	
4164	3900	4224	4284	4536	

Run #2

4371	4095	4491	4419	4047	3603
3939	4059	4095	3579	4071	4479
4347	4479	3903	4407	4335	4443
4467	4179	4467	4107	3795	4359
4635	4071	4167	4479	4431	4095
4695	4203	4635	3771	4251	4299
4311	4599	4263	4239	4371	4587
4059	4083	3843	4299	4275	4131
4131	3759	4335	4059	4119	4167
4455	4203	3915	4071	4035	3975
3855	3939	4479	4431	4491	4419
4203	4251	4179	4419	4263	4599
4335	3975	3903	4539	4371	4203
4491	4443	4527	4239	4227	4767
4203	4119	4623	4131	4143	4143
4407	4047	4635	4371	4599	3867
4179	4443	4179	4323	4227	4443
4263	4479	4143	4131	4371	4179
4251	4479	4251	4371	4251	4239
4143	4323	4059	4299	4227	4263
4347	4467	4143	4227	4323	4383
4527	4311	4239	4683	4263	4299
4251	4191	4107	4119	4563	4287
4311	4119	4671	4059	4179	4203
3975	4395	4335	4083	4191	4047
4191	3915	4179	4431	4527	3879
4167	4443	4287	4167	4467	
4275	4107	4467	4419	4035	
3915	4263	3795	4131	4335	
3867	4227	4527	4035	4239	
4407	4263	3807	4455	4059	

Run #3

4060	3964	4084	3880	4120	3880
4336	4636	4144	4504	4132	4072
4396	4504	4252	4492	4072	4336
4648	4648	4384	4156	4804	4600
4312	4060	4240	4060	3904	4204
4060	4288	3988	4432	4432	4060
4228	4444	4252	4324	4216	4036
3952	4780	3940	4108	3856	4300
3796	4240	4024	4276	4360	4480
4540	4360	4312	4480	4204	4396
4336	4360	4240	4252	4132	4312
4252	4396	4156	4288	4144	4720
4456	4012	4564	4828	4048	4024
4144	4096	4528	4264	4132	4756
4060	3808	4312	4012	3736	3892
4780	4588	4588	3904	4228	4624
4120	4048	4720	4600	4552	4036
4108	3928	4288	4504	4072	4324
4120	4252	4120	3940	4672	4360
4204	4108	3700	4396	4444	4216
4288	4192	4636	4336	4540	4480
4108	4420	4564	4060	4132	4180
4576	4408	4504	4288	4036	4348
4408	4624	4168	3952	4588	4144
4336	4204	4720	4300	4312	4252
4168	4264	4432	3988	4384	4432
3916	3952	4120	4300	4168	
4120	4312	4084	4060	4084	
4408	3904	4012	4216	4492	
4120	4456	4264	4468	4300	
4288	4252	4360	4540	4276	

Vita

Name: Darcy Gerard Benoit

Place and Year of Birth: Antigonish, Nova Scotia, Canada, 1973.

Education: St. Francis Xavier University, 1991-1995.

B.Sc. Honours, Department of Mathematics, Computing and Information Systems, 1995.

Queen's University, 1995-1997.

MSc, Department of Computer Science, 1997.

Experience: Lecturer, Jodrey School of Computer Science, Acadia University, 2002-2003.

Research Assistant, Department of Computing and Information Science, Queen's University, 1996-2001.

Teaching Assistant, Department of Computing and Information Science, Queen's University, 1995-2001.

Database Administrator / Software Developer, MediaShell Corporation, Kingston, Ontario, Canada, July 2000 – September 2001.

Java Instructor, Queen's University Mini Enrichment Courses, May 2000, 2001 and 2002. (Taught several 1-week Java courses to high school students).

Research Position, IBM Center for Advanced Studies, Toronto Lab, IBM Canada, summer 1997, 1998.

Research Position, Canada Institute for Scientific and Technical Information, Scientific Numeric Database Section, National Research Council Canada, May-September 1995.

Teaching Assistant, Department of Mathematics, Computing and Information Systems, St. Francis Xavier University, 1994-1995.

Research Assistant, Department of Mathematics, Computing and Information Systems, St. Francis Xavier University, 1994.

Awards:

PhD Fellowship, IBM Canada, 1997-2000.

Graduate Scholarships, Queen's University, 1995 – 2001.

Teaching Assistant of the Year, Queen's University 1999-2000.

Entrance Scholarship, St. Francis Xavier University, 1991.

Publications:

Benoit, Darcy G., Automated Diagnosis and Control of DBMS Resources, Conference on Extending Database Technology (EDBT) PhD Workshop, Konstanz, Germany, March 2000.

Benoit, Darcy G., Dynamic Extensions for Educational Hypermedia, MSc Thesis, Queen's University at Kingston, Canada, 1997.

Scheugraf, Ernst J., and Benoit, Darcy G., "Thesaurus Assisted automatic Indexing of Document Databases:, in Proceedings of the Tenth International symposium on Computer and Information Sciences, Volume II, editors A.E. Harmanci, E. Gelenbe and B. Öencik, pp 239-246, October 1995.

Benoit, Darcy G., "Wordnet and its Uses", APICS Annual Computer Science Conference Proceedings, pp 20-29, October 1994.

Posters:

Diagnosis and Automatic DBMS Tuning, CASCON 2000.

Automatic Diagnosis of Performance Problems in DBMSs, CITO Research Review, 1 May 2001.

Goal-Oriented Resource Management, CASCON 1997.

Diagnosis and Automatic DBMS Tuning, CanDB/IRIS 2000 (Second annual Canadian Database Research Workshop, 12 Nov 2000, Institute for Robotics and Intelligent Systems).

Self-Tuning DBMSs – Automatic Diagnosis Algorithms, CASCON 1999.

Invited Talks:

Benchmarks and Automated Tuning with DB2/UDB, CASCON 2000.