Policy-Based Tuning in Autonomic Database Management Systems Using Economic Models

by

Harley Boughton

A thesis submitted to the
School of Computing
in conformity with the requirements for the
degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
January, 2006

Abstract

A key advantage of Autonomic Computing Systems will be the ability to manage according to business policies. Implementing this ability is not a trivial problem as there is little similarity in the metrics used for measuring database performance and business performance. These translations can be simplified, however, by having a configuration model that reflects the business policies. Economic models allow for a system that mirrors the types of policies used to define performance in a business.

One such business policy comes from using Value Based Management [31], in which a manager is able to define the business units that are most important when it comes to allocation of capital resources. This concept can be applied to a Database Management System (DBMS) running multiple workloads corresponding to different business units. Importance information can be utilized in making resource allocation decisions, such as allocating buffer space.

In this dissertation, we utilize an economic model to address the buffer pool sizing problem in DBMSs. We use this context to implement importance as a parameter for resource allocation. We investigate a number of meanings for importance and identify how this additional information can best be used in the allocation of main memory.

Acknowledgments

I would like to extend my sincerest thanks to my supervisor, Patrick Martin, for providing me with this opportunity. I would like to thank him for all his guidance and advice over the years I have pursued my education and research at Queen's University.

I would also like to thank Wendy Powley for her support and commitment to the database research group. She has been a wonderful source of advice and a great sounding board for the countless challenges encountered throughout the course of this research.

I would like to extend my gratitude to the School of Computing at Queen's University for their support. I would also like to acknowledge IBM Canada Ltd. and CITO for the gracious financial support they have provided.

I would like to acknowledge my lab mates and fellow students who have provided endless inspiration during my stay at Queen's University. Similarly, I would like to thank the students from around the world that I met at the Toronto IBM Center for Advanced Studies (CAS).

I would like to thank my family and friends for their unwavering support, even when I moved halfway across the country.

Finally, I would like to express my sincerest appreciation and love to my wife, Megan, for all her help during these past few years. She was always supportive of my pursuit of this work, even though it meant a significant delay to the start of our happily married life together.

Table of Contents

List of Tables

List of Figures

List of Equations

Glossary of Acronyms

DBA	Database Administrator
DBMS	Database Management System
DSS	Decision Support System
OLTP	Online Transaction Processing
TPC-C	Transaction Processing Performance Council Benchmark C

Chapter 1

Introduction

1.1 Motivation

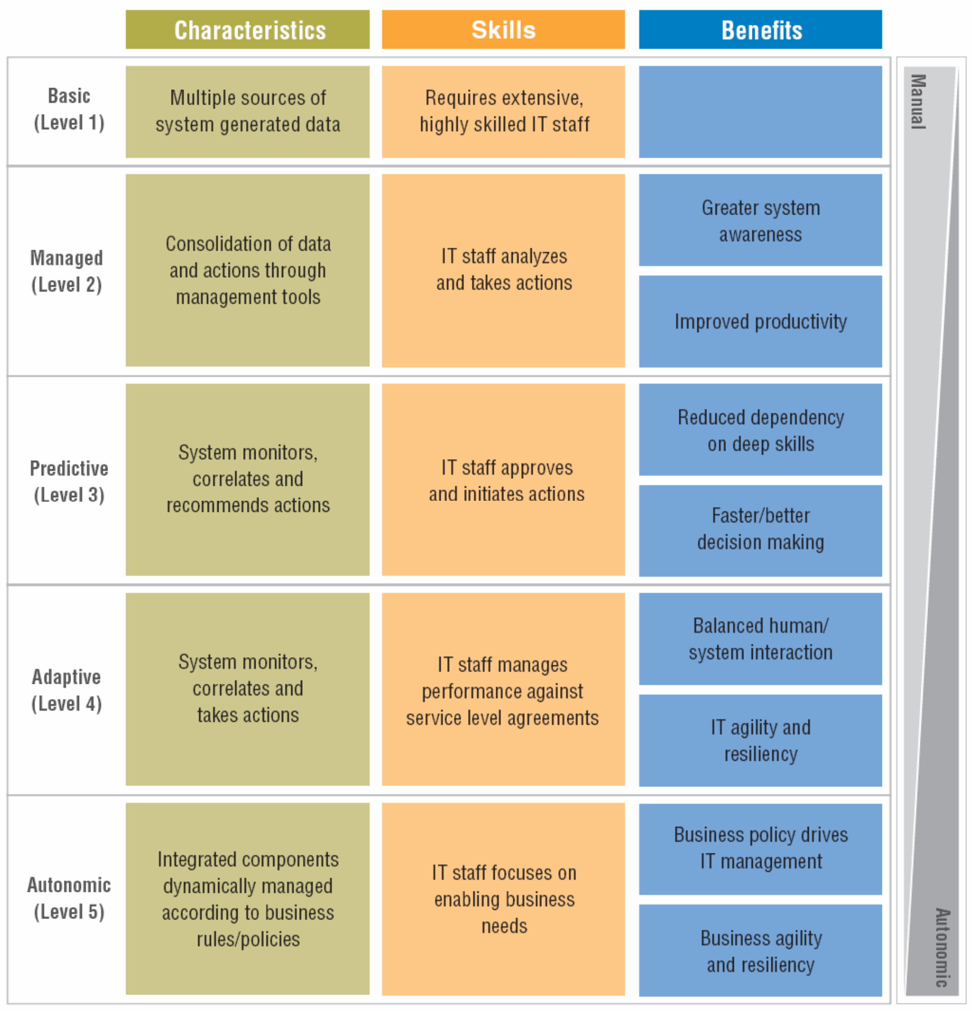
Computing systems have become increasingly complex over the last few decades. This complexity is approaching a point where system administrators and highly skilled IT professionals, let alone managers with corporate policy decision making approval, are unable to comprehend all aspects of the system's day to day performance [19]. This crisis has been brought to the attention of the computing world through initiatives such as IBM's Autonomic Computing. IBM put forth the challenge for all elements of computing, hardware and software, to become self managed in a method reminiscent of the human autonomic system [17]. These Autonomic Computing systems should be self-configuring, self-tuning, self-protecting, and self-healing. The goal is to enable the system to be managed more directly by business policies. This will allow for those with decision making authority to have direct control over the computing systems that are a central part of their business [22].

In particular, Database Management Systems (DBMSs) have become a core component in most organization's computing systems. DBMSs are so complex that many require specialized database administrators (DBAs) to be kept on staff for the day to day management of the system. According to the US Department of Labor, in 2002, there were 110,000 DBAs in the United States alone [8]. One of their key roles is tuning the DBMS so that the system can meet the required IT goals, such as throughput and response time goals. Determining these IT goals from the typical high-level business policies that govern most organizations is no easy task.

Most management policies are not written in terms of response time and throughput, but instead they are concerned with measures like revenue and return on investment (ROI). Additionally, database administrators are typically not involved in corporate policy decision making and must attempt to translate business policy into low-level technical requirements for the DBMS. This is a non-trivial exercise as there is little similarity in the metrics used for measuring database performance and business performance [2].

1.2 Problem

IBM has proposed a 5 level progression in the development of Autonomic systems (Figure 1.1) [18]. This progression serves as a guide to developers and details the milestone steps required to attain Autonomic capabilities. Computing at the Basic level offers no help to IT administrators who must obtain system data through independent sources, collate, analyze and decide the proper system administration tasks on their own. The Managed level introduces system management tools that simplify the acquisition of system data and provide consolidated reports to ease the analysis and execution of administrative tasks. Computing systems at the Predictive level introduce system initiated guidance for IT administrators. These systems are able to self-monitor and suggest future courses of action. However, the IT administrator is still responsible for initiating the actions. The final two levels involve systems that are not only self-monitoring, but self-managing as well.



	Characteristics	Skills	Benefits
Basic (Level 1)	Multiple sources of system generated data	Requires extensive, highly skilled IT staff	
Managed (Level 2)	Consolidation of data and actions through management tools	IT staff analyzes and takes actions	Greater system awareness Improved productivity
Predictive (Level 3)	System monitors, correlates and recommends actions	IT staff approves and initiates actions	Reduced dependency on deep skills Faster/better decision making
Adaptive (Level 4)	System monitors, correlates and takes actions	IT staff manages performance against service level agreements	Balanced human/ system interaction IT agility and resiliency
Autonomic (Level 5)	Integrated components dynamically managed according to business rules/policies	IT staff focuses on enabling business needs	Business policy drives IT management Business agility and resiliency

Figure 1.1: IBM's 5 Level Autonomic Computing Progression [18]

Much of the work that has been accomplished so far within Autonomic Computing fits into the Adaptive level, which involves creating systems that manage themselves with respect to some IT-oriented performance goal [11][34][36]. Further research is concerned with trying to move towards the Autonomic level, which involves implementing policies that mirror those used by business organizations. This allows for easy to understand system management policies that do not require specialized IT knowledge. For example, a high-level business policy may describe the working hours of the organization, such as 9-5, Monday to Friday or 24/7. The difficulty comes in the implementation and what this policy means to the system. It is a non-trivial exercise to translate high-level policies into low level implementations. For example, if a policy specifying business hours is defined, this may signal that the workload after hours is significantly different and, thus, the system should be tuned accordingly at closing time. This requires that the system be able to detect and characterize the new workload, determine how the system tuning parameters need to be changed, and finally be able to effect the change.

One type of policy that has a great deal of impact on the way in which the DBA makes decisions is the "importance policy". This type of policy allows managers to differentiate the importance of work being done on the DBMS. This becomes more critical as businesses consolidate workloads of different business units onto a single DBMS. The importance policy endows the DBA with additional information that can be used in making configuration decisions. As multiple workloads are consolidated to a single DBMS, inevitably, these workloads will begin to compete for physical resources, such as memory and CPU. The importance policy indicates how resources should be divided among competing workloads.

The buffer pool, for example, is an area of main memory reserved for buffering data to reduce disk accesses and is a critical factor in database performance [4]. As disk accesses are significantly slower than memory accesses, the DBA wants to retain as much data in the buffer pool as possible. This must be accomplished while balancing the need for additional main memory for other system needs, such as working memory for sorts and table joins. Determining the size of the buffer pool is typically done by making an initial estimate based on workload and database characterizations and is then refined by monitoring a number of buffer pool parameters, most notably buffer hit rate. Buffer hit rate is a key metric in measuring the success of a buffer sizing as it reflects the number of times a page of data is found in memory as opposed to needing to be retrieved from disk. This problem is further complicated when multiple buffer pools are involved. There has been much research done in attempting to automate this process [28].

When confronted with a finite resource that is not sufficient to meet the demands of all workload classes, the DBA must make a decision to compromise performance for some work. The additional importance information can assist in making a proper decision on how to make this compromise. However, determining the proper sizes of each buffer pool and taking into account multiple importance levels is a complex problem.

A key problem in implementing an importance policy is determining what it means to say that one workload class is more important than another. When utilized to make tuning decisions, different interpretations of importance are possible. For example, a more important class gets access to the resources it needs before a less important class; an important class can hold resources not in use in anticipation of work to come and an important class can appropriate resources from less important classes when needed. Additionally, the degree to which one class is more important than another affects tuning decisions and is another problem that needs to be addressed when implementing an importance policy.

One technique that has been used to address this disparity between high-level and low-level metrics is to introduce an economic model into the low-level system.

Economic models have been used in a number of resource allocation problems in computing with great success [9][12][15][26][29][35].

An economic model for system tuning typically involves a pricing system for resources. The rules of the system can vary to create the desired system behaviour. Since the model has an inherent sense of pricing and cost, business policies that express these ideas can be more easily implemented. The models are easily understood by many of the business policy makers and do not require the specialized IT knowledge that other models require. This eases the translation as the low-level system now functions in a similar manner as many high-level business policies. A system can therefore be managed directly according to business policies, which is the goal of Autonomic Computing.

1.3 Research Statement

The goal of this research is to investigate how a Workload Class Importance business policy may be implemented in an Autonomic Database Management System. The challenge in implementing this policy comes from translating this importance information into the DBMS. To ease this translation, we utilize an economic model for resource allocation. Specifically, we investigate this policy in the context of the buffer pool sizing problem.

We create an economic model representation of the buffer pool sizing problem and implement the model as an offline simulation. The model is utilized to investigate a number of possible meanings for importance and identify how this information can be used to allocate buffer pool memory between competing workloads. These various importance policies are implemented by adjusting various parameters and rules in the model.

We present experimental results where we look at a three class workload in which the classes are competing for buffer pool memory. We use the simulation to make memory allocation decisions for these workloads according to their level of importance and estimated need for buffer pool space.

The rationale for this project is that it helps make the case for the move towards “business policy based” Autonomic tuning. Importance policies are one of the most common examples of a business policy that could be used for tuning. With level 5 of IBM’s Autonomic Progression explicitly mentioning “business policy” as the driving force behind decisions, this is a key step.

1.4 Thesis Organization

The remainder of the dissertation is organized as follows. Chapter 2 outlines the related research conducted in the area of autonomic computing, resource allocation, economic models, and database workload priority. Chapter 3 describes the economic model used for allocation of resources. Chapter 4 describes the Workload Class Importance Policy and its syntax and semantics. Chapter 5 describes the simulator and presents a set of experiments to verify our approach. The thesis is summarized and future work is discussed in Chapter 6.

Chapter 2

Background and Related Work

This research draws from a number of areas in order to address the problem of implementing business policy based self-tuning in a Database Management System. This chapter provides some background information and references previous research in each of the four main areas addressed by this work. In Section 2.1, we present some previous work in the area of developing Autonomic Database Management Systems, while addressing how our work looks to further the progress towards Autonomic Computing. Section 2.2 looks at work in the field of Goal-Oriented Resource Allocation. It presents the problem of buffer pool sizing and some previously proposed solutions. Section 2.3 discusses some of the work that has been done using economic models in computing. Finally, Section 2.4 examines the issue of implementing Priority in computing systems.

2.1 Autonomic Database Management Systems

Since 2001, when IBM introduced their Autonomic Computing Manifesto [17], there has been great interest in Autonomic Computing within the scientific community. As computational power has become ever present and cheaper, software designers have harnessed this power to create ever more feature rich, yet complex, environments. Additionally, IBM pointed out that the new difficulty in managing computing systems is that they no longer involve single systems or software environments [22]. Computing systems are increasingly including a number of heterogeneous systems and software environments, connected both locally and over the Internet.

These monolithic computing environments are approaching the limits of human capability to understand and manage effectively. Not only is there a crisis of understanding, but there is also a huge problem with the amount of available skilled IT staff to manage these systems. Some estimates put the required number of IT staff required to maintain computer systems globally as high as 200 million [8].

The only viable solution to this crisis requires computing systems to manage themselves, that is, to take high-level objectives from administrators and handle the low-level maintenance themselves. Autonomic Computing is a term coined by IBM to describe technologies for computing systems that are able to *self-configure, self-optimize, self-heal,* and *self-protect*. These are systems that are able to seamlessly install and configure new hardware and software, strive to continually improve their own performance, diagnose and repair software and hardware problems, and detect and protect against malicious attacks [22]. Great strides have been made on a number of issues key to developing these types of systems. Research on self-tuning DBMSs has included such topics as index selection [30], materialized view selection [1] and memory management [7][36].

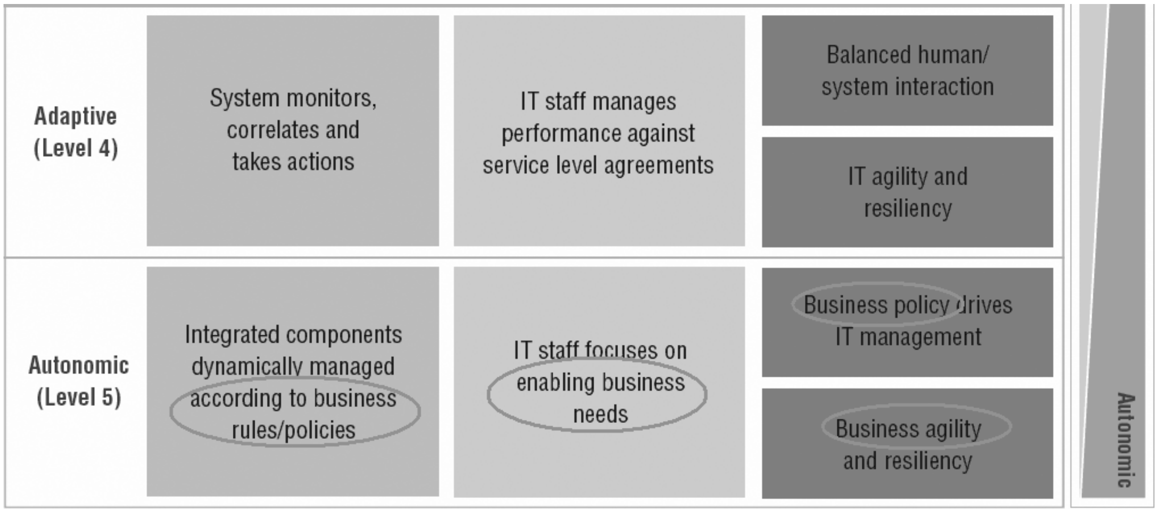

Adaptive
(Level 4)
System monitors, correlates and takes actions
IT staff manages performance against service level agreements
Balanced human/ system interaction
IT agility and resiliency
Autonomic
(Level 5)
Integrated components dynamically managed according to business rules/policies
IT staff focuses on enabling business needs
Business policy drives IT management
Business agility and resiliency
Autonomic

Figure 2.1: Highlighting the difference between Adaptive and Autonomic computing [16]

Referring to Ganek's and Corbi's evolution towards Autonomic operation (Figure 2.1) [16], however, much of this work fits into the Adaptive computing level. The key feature that differentiates Autonomic computing from Adaptive computing is the ability to manage according to business policies. Whereas the goals used in much previous work involve IT metrics such as response time or throughput [7][28], there is little work that directly addresses the issue of managing IT systems according to high-level business policies [2]. The ability to be managed according to high-level business policies would allow those business executives who typically form business policy to have more direct control over computing systems as opposed to requiring IT workers to translate the business objectives into IT objectives.

Our work is directly concerned with the implementation of business policies in an attempt to achieve the Autonomic computing level. We augment an automated resource allocation method to implement the concept of priority to reflect an importance policy based on Value Based Management.

2.2 Goal-Oriented Multi-Class Resource Allocation

Resource allocation is a key challenge in computing. How to determine allocations of resources to various applications on a computing system can be accomplished in a number of ways. Historically, much of the work done with regards to DBMS resource allocation involved optimizing system wide performance, while more recent work separates the workload into classes that share some commonality [5].

Traditionally, database workloads are categorized into one of two categories: Online Transaction Processing (OLTP) or Decision Support Systems (DSS). DBAs would use tuning strategies specific to how similar their workload was to an OLTP or DSS workload. However, as more enterprises consolidate their databases to a single DBMS, the resulting workload becomes a mix of the two types [6]. The combination of short running transactions and long running decision support queries creates a workload whose resource consumption and execution time is hard to predict [6]. Thus, many solutions involve segregating portions of the workload and allocating resources to each class as the resource requirements for a class can be better understood. Instead of trying to optimize overall system allocations, a DBA can optimize the allocation of each class and achieve an overall system optimization [7].

One of the most important resources in determining system performance is memory management [4]. There are a number of parameters that a DBA must tune when optimizing memory management, but one of the most common is determining the size of the buffer area. The buffer area is a portion of memory that the DBMS reserves for caching pages read from disk. By having the appropriate pages in the buffer area, the DBMS can greatly reduce the data access time for executing queries. The easiest way to ensure that the proper pages are in the buffer area is to have a very large buffer area so that an increased number of disk pages are kept in memory. However, main memory is a finite resource and allocating more memory for the buffer area results in reduced space for other DBMS needs such as working space for sorting and joining data.

Another method to optimize the buffer area is to recognize that different database objects are accessed according to different patterns. For example, indexes and data tables that are read sequentially are typically accessed in different patterns [36]. One way to deal with this is to logically separate the buffer area into separate buffer pools where page replacement is local to each buffer pool, and assign database objects with similar access patterns to the same buffer pool.

There are a number of rules-of-thumb that DBAs traditionally utilize in defining buffer pools [36]. These rules-of-thumb are used by DBAs to address two problems, the *buffer pool configuration problem* and the *buffer pool sizing problem.* The buffer pool configuration problem involves determining how many buffer pools are necessary and which database objects belong in each. For this work, we look at the buffer pool sizing problem. The buffer pool sizing problem involves finding the optimal allocations of memory for a set of buffer pools so that the best possible database performance is achieved [28]. However, previous work examines this problem from the perspective of the database objects as opposed to multi-class workloads with separate buffer pools assigned to workload classes.

This work examines the buffer pool sizing problem in the case where multiple workload classes run concurrently on the same DBMS, with a separate buffer pool assigned to each class. Our research further builds upon the class model by introducing importance information into the resource allocation problem. As opposed to attempting to optimize each class to an equal degree, we use class importance to give priority to optimizing some classes more than others.

2.3 Economies for Computer Resource Allocation

Human economies are designed to handle the large scale distribution of a near infinite number of goods among just as many agents. Their scale far exceeds what is likely to be encountered in dealing with resource allocation in a computing system. However, the benefit for computing is that, along with developing over centuries, they have been studied just as long. There is a wealth of knowledge, theories, equations, algorithms, and models that have been used to explain the actions within human economies that can be reused or adapted to resource allocation in computing.

Many researchers have found a number of key tools and benefits that economic models can bring to the resource allocation problem in computing systems [12][15][26]. These include utility functions to describe consumer allocation preferences in a concise mathematical formula [32], decentralization through the use of multiple brokers and agents, and largely scalable resource allocation solutions [15].

Utility, as used in economics, describes an amount of happiness or satisfaction gained from consumption of an allocation of a commodity. A utility function is a mapping of this satisfaction to various allocations of the commodity. In terms of computing resources, utility can be thought of as a measure of usefulness. When an agent is given an allocation of a resource, such as disk space, main memory, or CPU time, there is a certain amount of usefulness that is obtained. A utility function (Figure 2.2) would then provide a mapping between all allocations of that resource and the usefulness the agent would achieve.

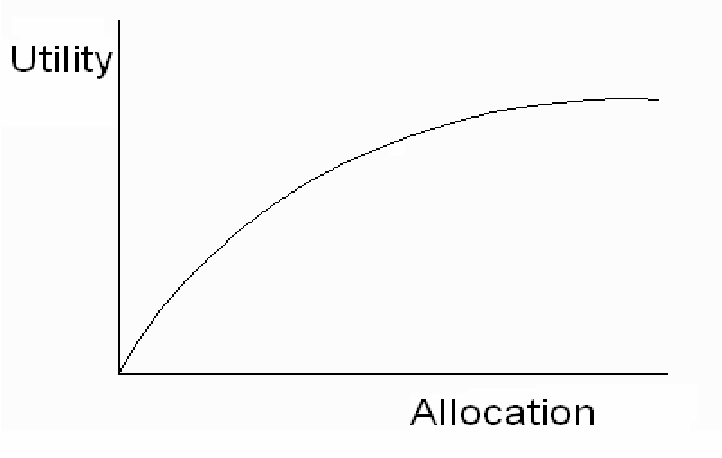


Figure 2.2: Sample utility curve

These functions can usually be expressed as mathematical formulas and combining these functions into multi-dimensional utility functions for a number of commodities allows one to find an allocation of a number of resources to best satisfy an agent [32]. Even with this very understandable application to computer resource allocation problems, the study of the practical applications of utility functions in computing is still fairly new [32]. Our work examines a utility function used by a database workload class to determine the usefulness of an allocation of buffer pool space.

Economic models typically contain suppliers and consumers that exchange goods. There are many different ways to implement this, with each implementation providing certain features to accomplish a specific goal [9]. Using wealth and auctions creates a transparent decision making process for allocations [26] and, if certain conditions are met provide an efficient convergence to a Pareto optimal resource allocation [35]. Pareto optimality implies that no individual's utility can be improved without diminishing the utility of others. This is a very desirable quality for the solutions provided by an economic model.

Through the use of brokers and agents, decentralization of the resource allocation decision is achieved. Brokers and agents need not have any knowledge of the overall system state. Brokers look only to maximize their profit for selling resources, while consumer agents are only concerned with maximizing their own utility [15]. Additionally, if the utility functions are monotone, smooth, and convex, this competitive system will lead to a Pareto optimal solution.

We implement a basic economic model in our work to represent resource allocation in a Database Management System. We use this simulator as a solution to the buffer pool sizing problem. We then utilize the model to implement an importance policy and view its effects on the buffer pool allocations suggested by the model.

2.4 Business Policy Management and Priority

Although researchers have been pursuing Autonomic Computing for a number of years, there is very little research that examines self-tuning systems guided by business policies [2]. Most research is concerned with optimizing traditional IT metrics. However, as stated previously, it is very difficult to directly translate many business policies into traditional IT measures of performance. By using an economic model for resource allocation, we attempt to bridge the gap between IT measures and business metrics such as profit and return on investment.

One business policy of particular interest is that of an importance policy. Such a policy would be useful in organizations utilizing management ideas such as Strategic Business Unit (SBU) Valuation that comes from Value Based Management [31]. By determining a valuation for an SBU, managers utilizing tools such as the McKinsey matrix [33] are able to determine the appropriate SBU to which new capital should be assigned. In a situation where computing infrastructure is shared among these business units, one can readily see how an importance policy could translate to some sort of priority for computing resources. We attempt to implement such an importance policy within our economic model framework.

Trying to incorporate importance/priority information into the resource allocation decision has been a topic of research for sometime now. In relation to DBMSs, much of the work on using priority information has focused on scheduling queries [10]. Previous research on using priority to manage physical resources in a DBMS has attempted to transfer ideas of priority from other computing resources such as CPU scheduling and buffer management [10]. However, there are a number of issues not addressed, such as the degree of difference in importance between various levels of priority and what priority should mean in resource allocation.

Recent work in utilizing priority information has focused on "real-time database systems" (RDBMS) [13]. Priority information is typically used as additional information for making different decisions, such as CPU scheduling or concurrency conflict resolution. However, RDBMSs are typically interested in allocating resources for individual queries to meet specific deadlines as opposed to adjusting for class-based performance goals. Additionally, most priority schemes are based on adjusting schedules through some sort of admission control policy.

In our work, we look at the implementation of importance in a DBMS in the context of the buffer pool sizing problem. We examine a number of definitions of what importance information should mean in terms of resource allocation. We also experiment to determine the degree to which a high-priority class should be more important than a low-priority class. Using an economic model, this is accomplished through simple rule changes such as how wealth is allocated and how auctions are conducted.

Chapter 3

Economic Model

Economic models used for resource allocation vary significantly from implementation to implementation, usually based on differing goals. They range from simple models designed for low overhead costs to much more sophisticated models designed to emulate an intelligent system [9][15][26][29][35].

All economic models for resource allocation consist of four basic elements, namely a supplier, consumers, resources and a mechanism for trade, as shown in Figure 3.1. Typically, economic models are concerned with the case where there are a limited number of suppliers and many consumers. This competitive model adapts itself well to resource allocation problems where resources are limited. In every economic model resources are supplied to consumers through some sort of trading mechanism. Most often, this mechanism involves representative money used by consumers to obtain the supplied resource [12]. Using auctions as the mechanism for trade provides a well understood decision making process with easy to trace results.

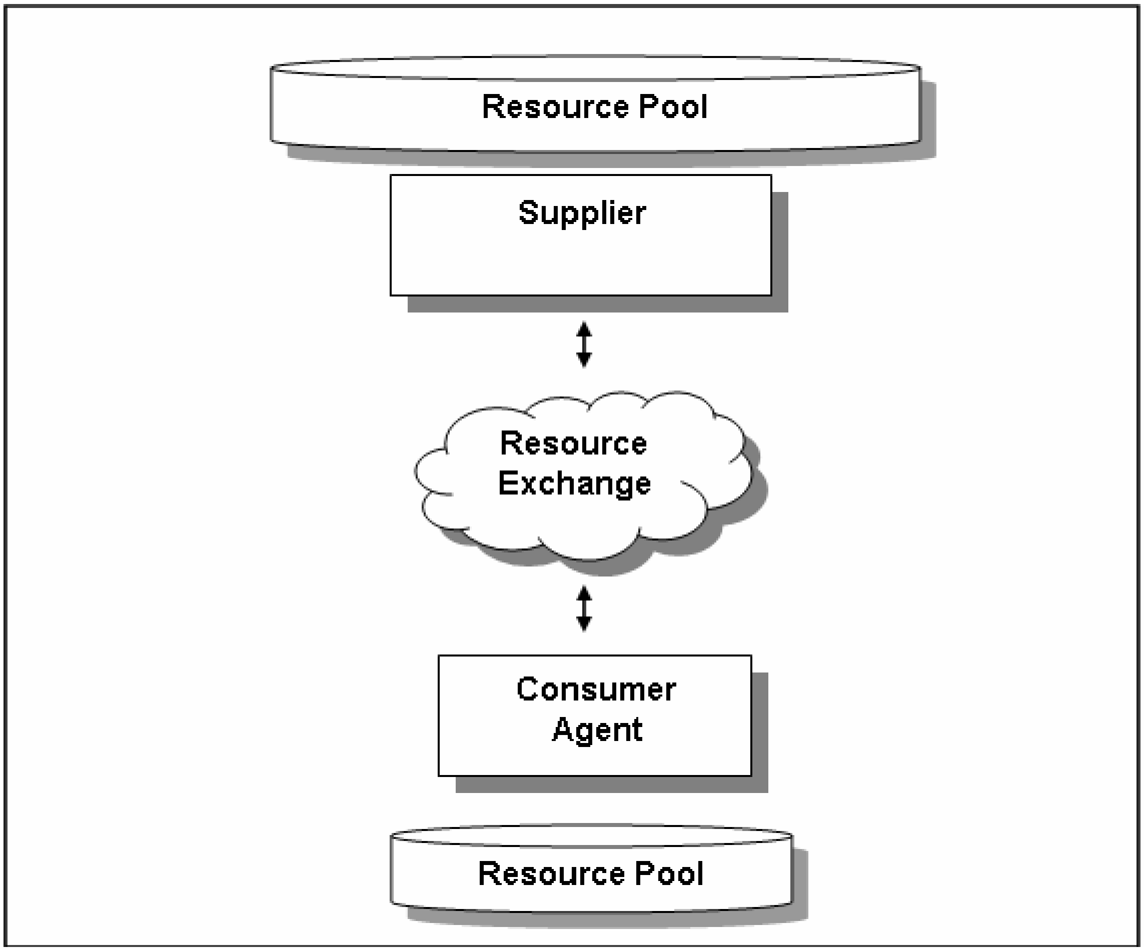


Figure 3.1: Components of an economic model

3.1 Our Economic Model

For the purposes of this dissertation, we develop an economic model to represent the buffer pool sizing problem. We utilize an instance of the model consisting of three consumer agents and a single broker. The resource being supplied is a finite amount of buffer pool memory pages. The broker is responsible for allocating buffer pool space to the consumer agents. The agents represent three OLTP workloads running simultaneously on the DBMS. The consumer agents are each assigned wealth based on the estimate of the work they must complete. The resource broker conducts auctions of memory page blocks to sell them to the consumer agents. The consumer agents each have an associated utility function used to determine the maximum amount of wealth they are willing to spend for the block of memory pages currently for auction. The agents submit this value as a sealed-bid to the resource broker who selects the highest bid as the winner and assigns the resource accordingly. This continues until all resources have been allocated or no consumer agent desires more resources.

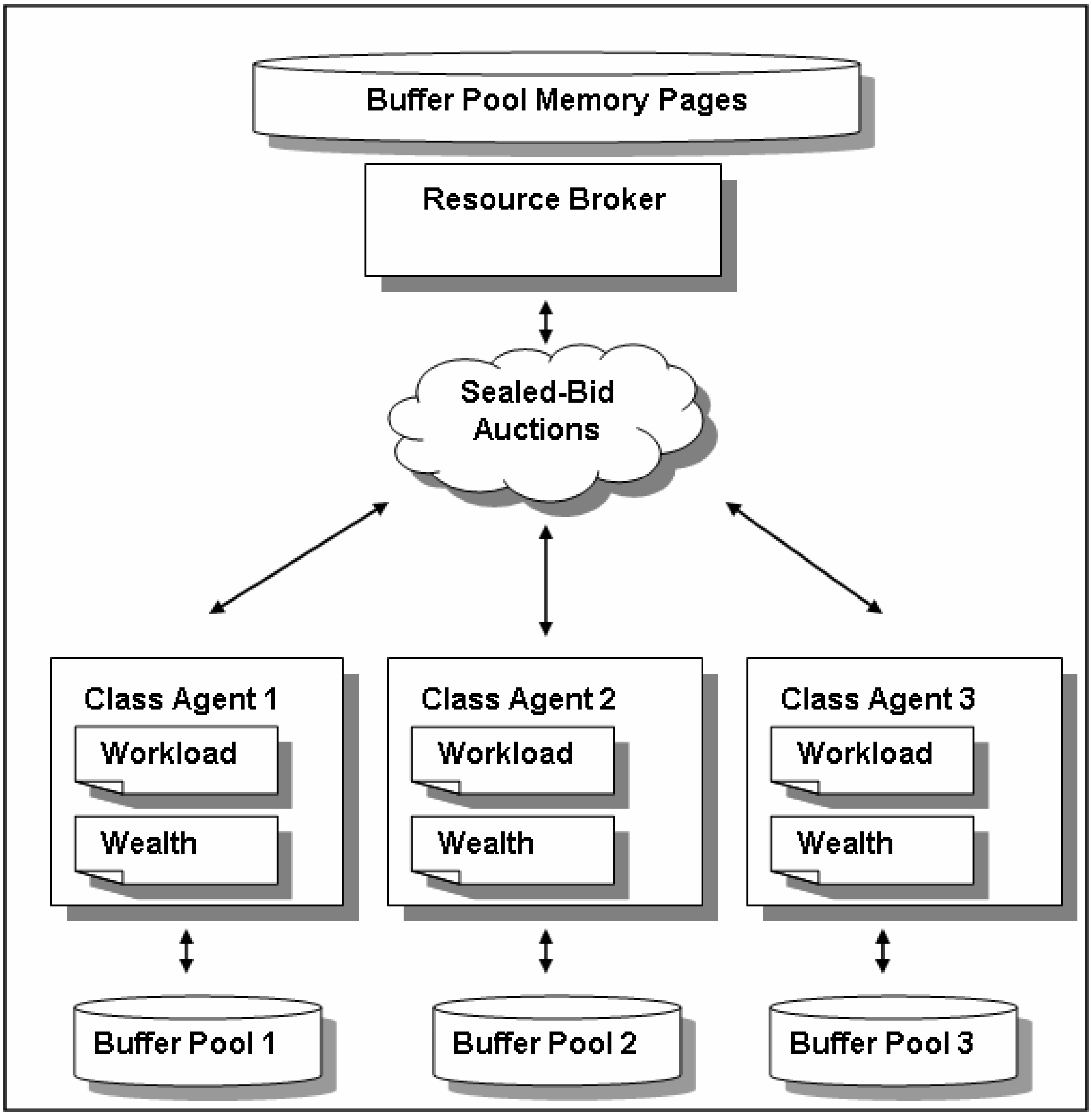


Figure 3.2: Economic model used for simulation

This economic model in Figure 3.2 represents a slightly different version of the buffer pool sizing problem. Typically, the buffer pools are associated with the database itself, not the user workloads. The various database objects, such as tables and indices are divided into buffer pools depending on access patterns and sizes and appropriate buffer pool sizes must be determined for these sets of objects. Our model, however, takes a more user-centric approach. We assume that each class of work or workload on the DBMS is assigned its own amount of buffer pool space. Currently, an object to buffer pool mapping is a many-to-one relationship (a buffer pool may contain many objects while an object may only be contained in a single buffer pool) [36]. With a move towards a service oriented architecture that involves the DBMS being used by various users as a service, being able to define an amount of buffer memory for each user will be useful. Thus, we use this alternative model of the buffer pool sizing problem to determine the appropriate amount of buffer pool memory for each consumer class agent.

This basic economic model is used for evaluating the appropriateness of an economic model as a solution for the buffer pool sizing problem. The minimal feature set of this model allows for implementing the importance policies desired, while allowing transparent decisions that can easily be interpreted [35].

3.2 Class Agents

For the purposes of this work, a workload class is a subset of the workload running on a DBMS that shares some user-defined commonality. A class could be defined as all work originating from a specific user id, such as that of a CEO, or it could be defined based on the type of work it represents, such as after hours reporting jobs. These classes of work are defined by users so that they can receive some sort of differentiated treatment. In the context of this model, classes are defined according to their importance level; all work in a given class has the same level of importance. In our economic model, we use three example classes. Each of these classes has an associated workload that runs on the DBMS.

Consumer agents are defined and represent each class running on the DBMS in the model. The agents are assigned an amount of wealth based on an estimate of the amount of work they are representing. The wealth of a class agent, in this economic model, is a metaphor for monetary currency. This estimate is obtained using the estimated number of I/O operations for each query as given by the DB2 EXPLAIN utility. For a workload *W* that contains *n* queries q_i per resource allocation interval, each with an I/O estimate $est(q_i)$, the initial amount of wealth assigned to the class agent *c* would be:

$$Wealth(c) = \sum_{i=1}^{n} est(q_i) \qquad \textbf{Equation 3.1}$$

The agent uses this wealth to "purchase" the resources it needs to complete the work for which it is responsible.

These agents are also given the ability to evaluate the utility of a resource allocation using an assigned utility function. They use the marginal utility (the difference in utility between two allocations) calculated using the utility function to determine their maximum bid. The maximum bid is the marginal utility multiplied by the agent's current wealth. Thus, if a class agent c has a current resource allocation of x buffer pool pages and y additional pages were available for bid, the bid submitted by class agent c, *Bid(c)*, would be:

$$Bid(c) = (Utility(x+y) - Utility(x)) \times Wealth(c)$$

Equation 3.2

This provides reasonable bids of a percentage of their wealth that matches the estimated percentage increase in performance. For example, if a class agent calculated that winning the current auction would provide a 10% increase in performance, the agent would be willing to spend 10% of its wealth to purchase that resource.

3.2.1 Utility Function

The class agents create a preference curve for their designated resource based on the workload queued to be completed. In this economic model, this utility curve is a representation of the usefulness of an allocation of buffer pool memory. As the key metric in evaluating buffer pool performance is typically hit rate, we use a hit rate estimation curve as the utility function for these class agents. This allows us to use the marginal increase in hit rate as the marginal utility for a given allocation of buffer pool memory.

We can also use the utility function in reverse. This allows us to find an estimated buffer pool size to achieve a desired hit rate. We use this functionality to pre-assign buffer pool space to guarantee a minimum level of performance.

The marginal utility is the difference in utility between two different allocations. In this economic model, since the utility functions represent the relationship between buffer pool size and buffer pool hit rate, the utility will always increase with an increase in buffer pool size. However, due to the diminishing returns of buffer pool size increases on hit rate, the marginal utility will always decrease. This will provide the desired behaviour in the model so that class agents will spend on resources to a point where the marginal benefit does not outweigh the cost.

This monotonic decreasing utility curve in a competitive model will also allow us to achieve a Pareto-optimal allocation as described in Chapter 2.3

3.2.2 Belady's Equation

For our economic model, we have chosen to use Belady's equation [3] as a hit rate estimator. This equation provides a hit rate estimate for a given allocation by using only two sample points. These sample points consist of a buffer pool size and hit rate pair. Thus, by running the workload with two different buffer pool sizes and measuring the hit rate, we can use Belady's equation to estimate the hit rate of alternative sizes. Furthermore, if we assume that the buffer pool access pattern for a given workload does not vary over time, we can use two trial samples, early in a workload's execution, to provide us with a hit rate estimator useful for dynamic resizing of the buffer pool over the course of the remaining workload. If a workload class is defined by something such as a business application using a set of predefined queries or an order entry department, this assumption is valid.

For any buffer pool of size *S*, Belady's equation allows us to estimate the hit rate *HR(S)* as:

$$HR(S) = 1 - a \times S^{b}$$ **Equation 3.3**

The constants *a* and *b* are calculated using the sample points. Once $HR(S_1)$ and $HR(S_2)$ have been collected for buffer pool sizes S_1 and S_2, we can use the following equations to solve for *a* and *b*:

$$b = \frac{\ln(1 - HR(S_2)) - \ln(1 - HR(S_1))}{\ln(S_2) - \ln(S_1)}$$ **Equation 3.4**

$$a = \frac{1 - HR(S_1))}{e^{b \times \ln(S_1)}}$$ **Equation 3.5**

Once we have *a* and *b* for a given workload, we can solve Belady's equation and use it as a hit rate estimator. This provides a curve similar to what is seen in Figure 3.3.

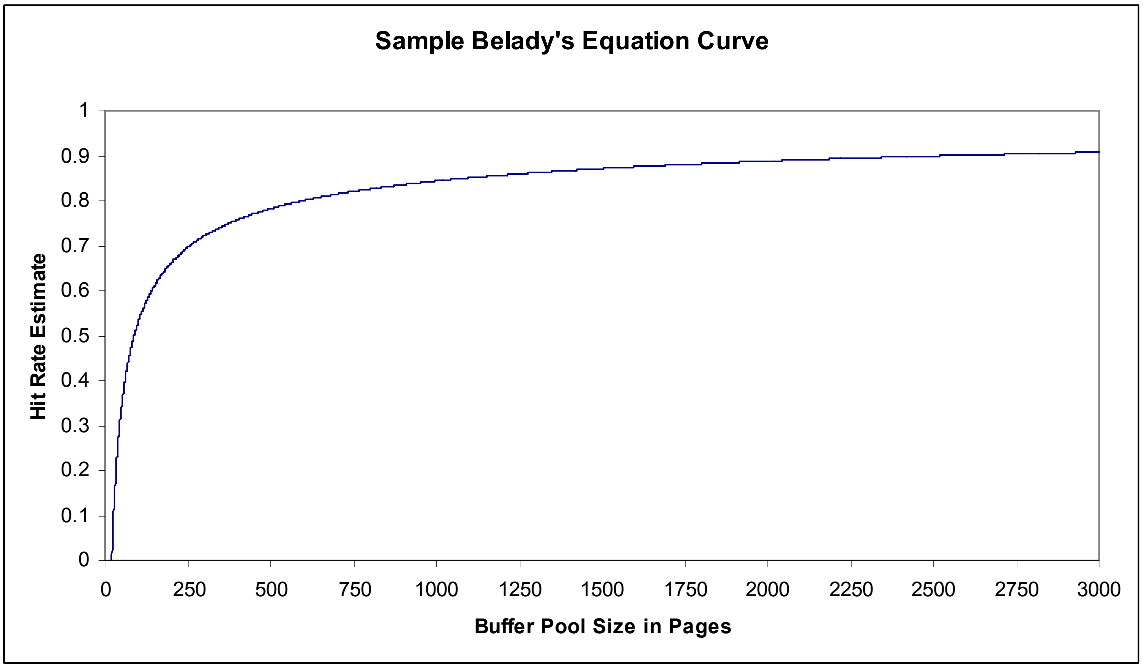


Figure 3.3: Sample curve formed by Belady's equation

Since hit rate is typically the most important factor in determining an appropriate buffer pool size, we use the hit rate as a measure of utility. For example, if increasing the buffer pool by 100 pages of memory would increase the hit rate from 65% to 70%, we would state that the 100 pages of memory had a marginal utility of 5%.

We can also use the inverse of Belady's equation to estimate the necessary buffer pool size to meet a target hit rate. The inverse function to solve for size *S* given *HR(S)* is:

$$S = (\frac{1 - HR(S)}{a})^{-b} \qquad \textbf{Equation 3.6}$$

These simple equations provide the class agents with an efficient way to estimate their utility for a given allocation of buffer pool memory and determine their maximum bids accordingly.

3.3 Resource Broker

The broker is responsible for administering the auctions. The auctions used in this model are sealed-bid auctions, where agents submit their maximum bid and the broker selects the highest bid as the winner. As an agent wins auctions, it gains resources but loses wealth. They are, therefore, less likely to bid on additional resources as they give up wealth and thus allow other classes with less wealth and resources the chance to win resources. The main memory is divided into blocks of memory pages. Agents make bids based on their utility function curves until they have sufficient resources, insufficient wealth, or all resources have been claimed. The specific protocol for how the auctions function affects the method in which resources are allocated. We use this to implement different definitions of importance, which is discussed in the next chapter.

Auctions are held at predefined intervals, where class agents bid on the resources available using their assigned wealth. The resource broker in this economic model simply coordinates these auctions. However, in more complex models, the brokers can, in turn, try to maximize their own desired resource, wealth. These brokers measure demand and try to maximize the price at which they sell resources. In our basic model, though, the way in which the resource broker maximizes profit is by selling all available resources in each auction period.

The process that the broker follows in allocating resources is as follows (Figure 3.4):

```
While unallocated pages > 0 and number of bidders > 0{
     Determine number of memory pages, y, for auction{
          If unallocated memory pages > pageBlockSize
               y = pageBlockSize
          Else y = unallocated memory pages}
     Solicit bids from class agents{
          For each class agent
               Get bid(c)}
     If number of bidders > 0 {
          Determine max bid
               Allocate y to highest bidder
               Charge highest bidder bid(c) for allocation
          Unallocated pages -= y}
}
```

Figure 3.4: Memory Broker auction algorithm pseudo code

This process is repeated at the beginning of each interval to determine the buffer pool allocations to be used for the interval.

3.4 Economic Model Simulator

As implementing an economic model for resource allocation within the IBM DB2 Universal Database (DB2 UDB) [21] engine is outside the scope of this work, we have implemented a simulator to make the resource allocation decisions. The simulator is an implementation of the economic model representing the buffer pool sizing problem. The simulator was created using the Java programming system. The simulator is implemented as shown in the UML diagram presented in Figure 3.4. The Simulator class contains the main method and controls the simulation.

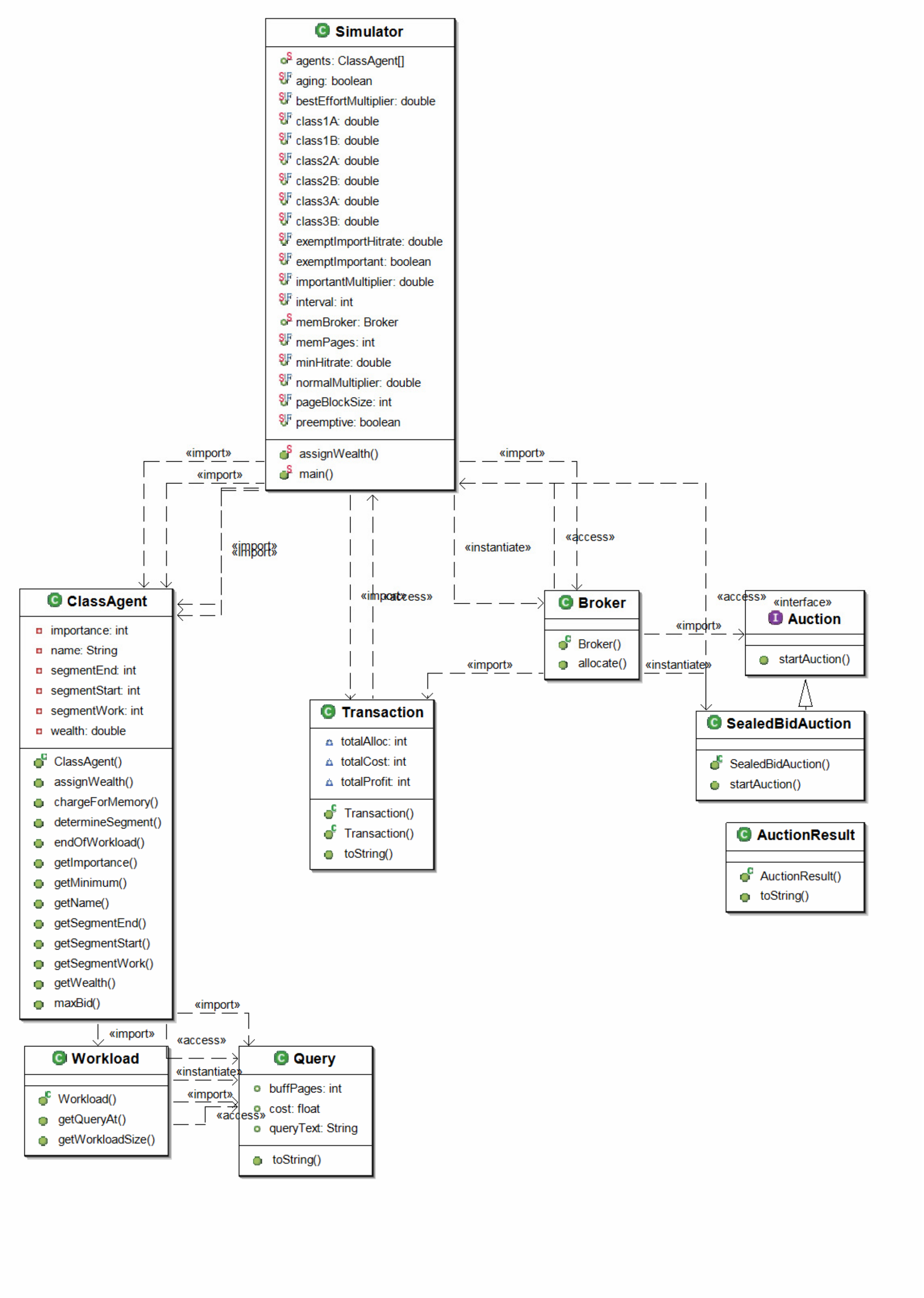


Figure 3.5: UML description of Economic Model Simulator classes

As inputs, the Simulator takes three SQL workload script files. These workload files are augmented with the estimated runtime and the estimated number of I/O operations for each query as obtained using the DB2 EXPLPAIN utility. As output, the Simulator provides a list of buffer pool sizes for each workload at each specified interval. For the purposes of this work, we also output the cost for each class to purchase the allocated resources along with the amount of wealth they retained.

The Simulator creates a ClassAgent object for each workload provided. It also initializes a Broker object to act as the memory broker. The ClassAgents determine the queries that are executed in the first interval as specified by the Simulator. Once they have determined the queries that are applicable, they calculate the sum of the estimated I/O operations and report this to the Simulator. The Simulator uses this value to determine the wealth assigned to each ClassAgent for the interval. Once wealth is assigned to all ClassAgents, the Simulator allows the Broker to begin the allocation process. The broker determines the number of memory pages for auction and elicits bids from each of the ClassAgents. The Broker assigns the memory pages to the ClassAgent that submits the highest-bid and charges them for the amount of the bid. This auction process ends when there are no remaining memory pages, the ClassAgents have no remaining wealth, or the ClassAgents to not desire any more memory pages. At this point, the Broker submits a Transaction containing the AuctionResult to the Simulator and the Simulator advances to the next interval. Once all intervals are completed, the Simulator outputs the resulting schedule of allocations. The parameters used for adjusting the simulation of the economic model are shown in Table 3.1.

Name	Type	Description
interval	int	Specifies the number of queries per interval
memPages	int	Specifies the total number of memory pages for the broker to allocate
pageBlockSize	int	Specifies the number of memory pages available Per auction
class1A	double	The calculated A value for class 1 utility function
class1B	double	The calculated B value for class 1 utility function
class2A	double	The calculated A value for class 2 utility function
class2B	double	The calculated B value for class 2 utility function
class3A	double	The calculated A value for class 3 utility function
class3B	double	The calculated B value for class 3 utility function

Table 3.1: Economic model simulator parameters.

For this work, the values for *a* and *b* needed to solve Belady's equation (used as the utility function) were pre-calculated by running the workload with different buffer pool sizes as we had a pre-specified workload script. However, this could easily be done online by adjusting the buffer pool size for two time intervals and recording the results, or offline by using a number of very accurate offline hit rate estimators that analyze the workload [28].

The remaining parameters are used for implementing the various possible Workload Class Importance Policies described in the next chapter. They are shown in Table 3.2.

Name	Type	Description
preemptive	boolean	Specifies whether the importance scheme is Preemptive or non-preemptive
exemptImportant	boolean	Specifies whether the Exempt Important importance scheme is used
aging	boolean	Specifies whether priority aging is used
minHitrate	double	A parameter used for pre-allocation in the non-preemptive importance schemes
exemptImportantHitrate	double	A parameter used for pre-allocation in the Exempt Important scheme
importantMultiplier	double	Wealth multiplier for High Importance classes
normalMultiplier	double	Wealth multiplier for Normal Importance classes
bestEffortMultiplier	double	Wealth multiplier for Best Effort classes

Table 3.2: Workload Class Importance Policy parameters

In Chapter 4, these parameters, their effect on the execution of the simulation, and their implementation details are described. The settings for these parameters for the experiments are presented in Chapter 5.

3.5 Model Overhead Analysis

In this section, we discuss the impact this model would have when implemented in a modern DBMS. As implementation of this model into IBM's DB2/UDB is beyond the scope of this work, experimentation to determine the overhead introduced into the system is not possible. However, an examination of the complexity of the algorithm provides some indication.

The auction algorithm presented, the sealed-bid auction, is one of least complex trade mechanisms available. For a specified interval, I_k, the Broker initiates *m* auctions where *m* = memPages/pageBlockSize. The Broker must then obtain bids from *n* ClassAgents. To compute these bids, each ClassAgent must calculate the marginal utility for the currently available memory pages using Belady's equation. We denote the time taken for this calculation as T_{util}. Thus, the equation to estimate overhead introduced into the system is in Equation 3.5.

$$\sum_{I=1}^{k} m \times n \times T_{util}$$ **Equation 3.7**

The number of ClassAgents, representing concurrent workloads on the DBMS, should never reach a very large value for *n* so the overhead of the auction mechanism is mostly dependant on *m*. We found however, due to the diminishing returns nature of buffer pool hit rates, having a small pageBlockSize does not benefit the model as the marginal utility of only a few pages of memory is typically very low. Thus, the granularity of the resource allocation process does not benefit from being too fine and *m* should be a small value as well. Finally, since the utility function is based on a single equation, the time to calculate the value of marginal utility, T_{util}, will be small. Thus, this model, used for a single resource and utilizing Belady's equation for a utility function will introduce little overhead to the system. However, if the model were expanded to multiple resources, the ClassAgent calculations would be more intensive and the number of resource auctions would increase.

The most significant overhead will come from the frequency at which this allocation takes place. At each interval, the model will execute *m* resource auctions and *n* resource reallocations. However, the granularity of the model determines its responsiveness as an autonomic system. We use a constant interval that is fairly course-grained (10,000 queries). A finer-grained interval would allow the system to adapt to changing demands more quickly at the cost of increasing the overhead.

Currently, the model runs very quickly compared to the total execution time of the workloads, but without being able to implement this in a DBMS, we are unable to accurately characterize the impact on the system. This model uses a minimal amount of statistics from the DBMS and uses a minimal set of economic functionality. Further research will need to be conducted to determine how an expanded economic model, using multiple resources and allowing consumers to choose between resources, would affect the overhead for the DBMS.

Chapter 4

Workload Class Importance Policy

This research addresses two key problems in implementing an “importance policy” in a self-tuning Database Management System. These two key problems are the degree of importance that a high-importance class has compared to a low-importance class and what being an important class means in terms of how resources are allocated, definition of importance. In this section, we describe the questions these two problems raise and our proposed solutions. We define possible scenarios for solutions and, in the next section, provide experimental data to evaluate the best solutions.

4.1 Importance Policy

One important aspect of Value Based Management is the ability to quantify the value of business units. Using tools such as the McKinsey or GE matrix [31], an enterprise is able to measure the value of various strategic business units. This valuation can be used to, for example, determine where newly acquired capital should be utilized or, in other words, which are the most important business units in terms of resource allocation. This directly relates to the resource model developed in the previous chapter. In the economic model previously discussed, multiple workload classes are competing for limited resources, just as business units compete for capital in an enterprise. Using the valuation ideas of Value Based Management, one could determine importance labels to be applied to the various workload classes in the economic model in the case where the class agents each represent a business unit. A combination of these labels defines an importance policy for the system.

The importance policy examined in this work involves assigning one of three importance levels to each workload class running on the DBMS. We use the labels “High Importance”, “Normal Importance”, and “Best Effort”.

A class that is “High Importance” should demonstrate some priority over classes that have “Normal Importance” or “Best Effort” importance. In a consolidated enterprise system, this “High Importance” label could, for example, be applied to a class of work representing an OLTP-like order entry department as this work is directly revenue generating or a class of work corresponding to queries entered by the company CEO and considered most important. The “Normal Importance” label would likely apply to all other business related workloads such as an HR department that will run reports that they need during business hours, but have a lower priority than business units directly affecting revenue. Finally, the “Best Effort” label would be applied to classes of transactions that do not have any strict deadline, such as background DBMS maintenance work or after hours reporting queries.

These three levels, we believe, provide a reasonable scenario for implementing importance in the DBMS. These three labels allow, in addition to normal workloads, a way to both raise and lower the importance of workloads on the DBMS. A discrete labeling system is also preferential to a continuous value describing importance as it is much easier to make decisions choosing the appropriate level for a class from a small number of well defined importance levels. A larger enterprise may require more levels of discretion for various classes of work. Additional levels of importance could be added to the system, however, some experimentation to determine their degree of importance, as defined below, would be necessary.

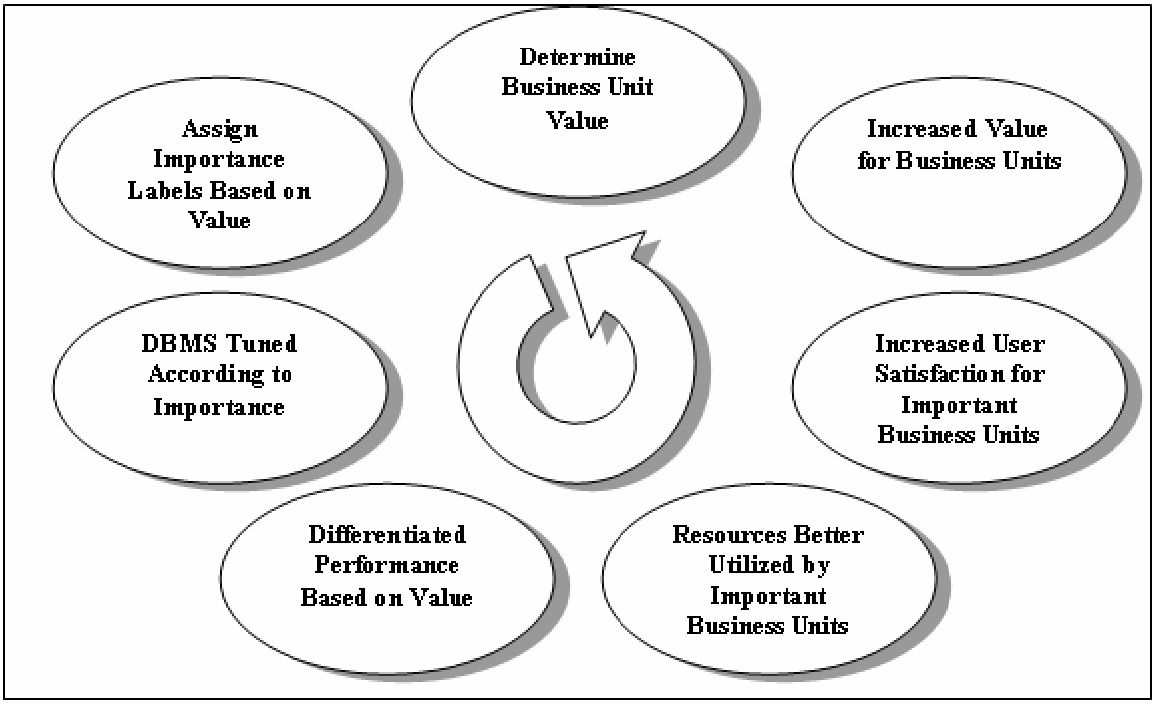

Determine Business Unit Value
Increased Value for Business Units
Increased User Satisfaction for Important Business Units
Resources Better Utilized by Important Business Units
Differentiated Performance Based on Value
DBMS Tuned According to Importance
Assign Importance Labels Based on Value

Figure 4.1: Value Based Management Value Added Cycle

Using this Workload Class Importance Policy to tune the DBMS according to the importance of the various business units can help organizations to increase their value according to principles of Value Based Management [31]. By improving the performance of important business units, the enterprise will improve the user satisfaction for users of the important workloads without additional investment in resources. This will help to further increase the value of these strategic business units. This leads to the cycle shown in figure 4.1.

Using Value Based Management tools, an enterprise is able to determine importance labels for different workloads on the DBMS. The DBMS will tune resources according to these labels using the economic model presented in chapter 3. This will result in the important workload classes being able achieve an increase in performance, leading to increased user satisfaction. Increasing user satisfaction is one way in which an enterprise can add value [31]. This increase in value is then taken into account when re-evaluating the value of strategic business units. This iterative cycle will help enterprises refine their critical IT operations without the need for specifying low-level system requirements.

4.2 Degree of Importance

A key problem in implementing different levels of importance is how to differentiate between the levels. This involves determining how much more important one level is than another. The values are especially important when using the importance levels in low-level resource allocation decisions. In this work, we determine appropriate weights for the importance levels through experimentation with the economic model.

Each consumer agent in the economic model is responsible for a workload. An estimate of the amount of work to complete (obtained by using the sum of the estimated number of I/O operations per query using the DB2 Explain utility) is used to determine the wealth awarded to the agent for the current allocation interval, where all work is considered equal. This is then multiplied by an "importance multiplier" to enforce a degree of importance. For example, an agent with a "High" degree of importance may have a multiplier of 3.0, while an agent with a "Best Effort" degree of importance may have a multiplier of only 1.0. By adjusting these multipliers, we affect the amount that one class is more important than another by affecting their ability to outbid other classes for resources. In our experiments, we look at the impact of a range of values for these multipliers.

The numbers selected for experimentation consist of a number of combinations of weights to represent different sets of degrees. In all experiments, the Best Effort class is given a multiplier of 1.0. We experiment with weights for the High Importance class that are similar to the Best Effort class (such as 3.0) and weights that represent a significant increase (such as 10.0), as well as middle values (5.0 and 6.0). For the Normal class, we try values that range between the Best Effort multiplier and the High Importance multiplier (2.0 and 5.0). These combinations of degrees allow us to examine how the relative importance of the different classes affects both individual class resource allocations and overall system performance.

4.3 Definition of Importance

Paramount in implementing importance in the DBMS is defining what importance means. This definition describes the differences in entitlements and abilities between a higher-priority and lower-priority class.

For this work, we are concerned with defining what importance means with regard to resource allocation. We therefore base our definition on the entitlement one class has to resources compared to other classes. We present three definitions of importance that we experiment with using our economic model simulation. They represent two classical definitions of priority, namely non-preemptive and preemptive priority [10] as well as a variation on the preemptive model where High Importance classes are exempt from preemption.

The first definition of importance states that all classes are entitled to their minimum necessary resource allocation. Additional resources available in the system are more likely to be assigned to important classes. This represents a non-preemptive model. All classes are able to complete some work during an interval, with any additional resources being used to improve the performance of important work. This allows us to provide a minimum guaranteed level of performance. In our economic model, this is accomplished by pre-allocating the minimum required resources to each class before the auctions begin. However, if the total resources available are less than the sum of the minimum allocations, we then scale the allocations according to their ratio of the total resources requested. To calculate the minimum required resources, we pick a target buffer pool performance level and use the class's utility curve to calculate the necessary resource allocation to meet that hit rate. Thus, for a given workload class *c*, where *Guarantee(c)* is the calculated minimum resources pre-allocated, then:

$$\mathit{Importance(c)} \equiv \mathit{Allocation}(c) \geq \mathit{Guarantee}(c) \tag{Equation 4.1}$$

In our experiments, we try both a 50% and 75% target buffer pool hit rate to determine the minimum resource requirement and guaranteed minimum hit rate.

The second definition of importance states that the requirements of important classes should be satisfied before those of less important classes. Important classes may be allocated all resources in the system such that less important classes must wait until resources are made available. This definition represents a preemptive priority model.

For the economic model simulation, this means that all resources are auctioned to the highest bidding class through the competitive mechanism. For a given workload class *c*, then:

$$\mathit{Importance(c)} \equiv \mathit{Allocation}(c) \geq 0 \tag{Equation 4.2}$$

The final definition of importance guarantees a high level of performance to High Importance classes and allows other classes to compete for remaining resources. The purpose of this importance scheme is to address the possibility of a lower-importance class preempting the High Importance class. Similar to the non-preemptive scheme, we implement this in the economic model by a pre-allocation of resources to the High Importance class. The pre-allocated resource amount is based on a high level of performance for the class buffer pool. For a given workload class c, with a minimum hit rate guaranteed by a pre-allocated amount of buffer pool pages Guarantee(c), then:

$$\textit{Importance(c)} \equiv Allocation(c) \geq Guarantee(c) \forall c \text{ where } \textit{Importance(c)=High Importance}$$

$$\textit{Importance(c)} \equiv Allocation(c) \geq 0 \forall c \text{ where } \textit{Importance(c)=\{Normal, Best Effort\}}$$

Equation 4.3

In our experiments, we try a target hit rate of 80%, 90%, and 95% and use the class's utility curve to determine the necessary resource allocation to meet those targets.

4.4 Aging Importance

Allowing priorities to age is another interesting aspect of an importance policy that we chose to examine in this work. Aging involves gradually incrementing the priority of objects over time. This is commonly used to prevent starvation in CPU-scheduling so that lower-priority processes are not blocked indefinitely by a steady stream of higher-priority jobs [25]. This is particularly relevant to our preemptive-style importance policy definition. This allows a lower importance class to eventually overcome the preemption of the High Importance class.

In this work, aging involves allowing a class to better compete for resources the longer the workload has been running. To implement this in the economic model, we allow classes to accumulate the wealth they do not spend in previous allocation periods. Thus, if a class is preempted, it will have twice the wealth in the next allocation period and be better able to compete against other classes that have likely expended all their wealth in the previous period. For the non-aging schemes, classes do not carry over any wealth that is unspent.

In our experiments, we utilize the economic model simulation to implement each of the various schemes described. We experiment with a number of sets of degrees of importance. We also implement each of the described definitions of importance. Additionally, we use each of these with and without aging. The economic model allows us to implement each of these through simple rule and parameter changes. We also experiment to determine which of these schemes provides the best Workload Class Importance Policy. The policy should provide a clear benefit to High Importance classes while mitigating the overall impact on the system.

Chapter 5

Experimental Analysis

We describe the experimental environment in Section 5.1. We present experiments to show the validity of our economic model simulation as a method for buffer pool allocation in Section 5.2. Section 5.3 discusses degree of importance and presents experimental results supporting the degrees chosen for experimentation. In Section 5.4, we experiment with different definitions of importance to determine the most useful definition for buffer pool memory allocation.

5.1 Experimental Environment

For our experiments, we use IBM's DB2 Universal Database version 8.2 [21] running on an IBM xSeries 240 PC server with the Windows XP operating system [23]. The server is equipped with two 1 GHz Pentium 3 processors, 2 GB of RAM and an array of 22 disks.

We use a single instance of the DBMS with three identical databases. The databases are TPC-C like and each contains 10 GB of data. As an object to buffer pool mapping is a many-to-one relationship (a buffer pool may contain many objects while an object may only be contained in a single buffer pool) [31], we use the three separate databases to allow each workload to have its own buffer pool while still having access to all database objects.

The economic model simulation was created using the Eclipse IDE [14] and Java 1.4.2 [27]. It was implemented as described in Chapter 3. It is run on a standard desktop computer.

5.1.1 Methodology

We generated three OLTP type workload scripts, each consisting of 120,000 queries based on the 5 different transactions of the TPC-C benchmark (see Appendix A). The workloads are similar, however, the order and proportions of the transactions vary slightly. This is done to provide workloads with slightly varying resource needs. The workloads are divided into 12 segments of 10,000 queries each to provide us with uniform points at which to resize the buffer pools. This provides us with a consolidated workload reminiscent of a single organization with three business units running separate OLTP type workloads simultaneously against different sets of tables within the same database.

The workloads are entered as input to the economic model simulator, which produces a list of allocations at each of the checkpoints. At each segment checkpoint in the workload script, the buffer pool size is modified using the ALTER BUFFERPOOL statement with the IMMEDIATE keyword [20].

The IMMEDIATE keyword, when used in the ALTER BUFFERPOOL statement, allows for dynamic resizing of the buffer pools without starting and stopping the DBMS. When the command is used to increase the size of a buffer pool, additional memory pages are simply added from the database shared memory. If the size is decreased, pages are released from the LRU (least recently used) queue [24], which is a list of pages that can be made available for reading in new data from disk. The dynamic resizing of the buffer pool selects the best candidate pages to be released first in order to minimize disk accesses. We assume that DB2 dynamically resizes the buffer pools with the least possible impact.

Once the allocations have been determined by the simulation, they are entered in the workload scripts, the databases are restored to their initial state. The workloads are then run concurrently while we monitor the buffer pools throughout execution and collect statistics at each segment checkpoint. We record the number of logical and physical reads for data and indexes for each of the three buffer pools. This data is used to calculate the hit rates for each segment of the workloads.

5.2 Economic Model Simulator Validation

The first set of experiments was conducted to determine the validity of the economic model as a method for resource allocation. We begin by determining an appropriate maximum amount of available buffer pool memory for the three buffer pools. We also determine the granularity for the allocation of memory pages. The values of *a* and *b* are determined for each of the workload class agent's utility functions. We then determine a manual allocation of buffer pool memory to each of the three buffer pools. Due to the time consuming nature of calculating the manual configuration, it results in a static configuration based on the entire duration of the workloads. The simulator, however, is able to produce a dynamic allocation schedule. We examine the model's ability to allocate resources both with and without Workload Class Importance Policy labels. Finally, we compare the simulation's results for the model not using importance information to the manual configuration to determine its effectiveness at producing allocations. This comparison uses the total number of physical reads generated by all of the workloads. A physical read occurs when the workload requires a page of information, either an index or data page, and that page is not available in the buffer cache. Thus, the system is forced to read that data from the hard disk, which is a typically time consuming operation. Therefore, the lower the number of physical reads the system must make, the better overall system performance will be achieved.

5.2.1 Economic Model Simulator Experimental Results

We begin by determining two key parameters for the simulation, memPages and pageBlockSize. To determine the total amount of memory available, the workloads were run concurrently with a minimal amount of buffer pool space (500 4KB pages for each buffer pool) and again at a much higher size (20,000 buffer pool pages each). With the smaller size, the average hit rate for the three workloads was 78.82% and with the larger size the average was 97.46%. Using these results, we estimated the requirements to achieve a 95% average hit rate using Belady’s equation for all three workloads. We then rounded this number up to 32,768 4KB memory pages for a total of 128 MB of available buffer pool memory. With this amount of resources, an even allocation to all workloads should give a 95% hit rate and if a single class agent appropriates all resources, it would be able to achieve a hit rate near 97.5%.

We then tested the effects of using different sizes for pageBlockSize. This parameter is the number of memory pages available in each auction. This affects the level of fine-tuning to which class agents can obtain resources. Using a block size of 50 pages not only increases the number of auctions that are held for each allocation period, but the marginal utility calculated by the class agents was so low that they quit bidding very early. Page block sizes of 100 pages and 500 pages were also tried. We found that using a course granularity of 500 pages or greater did not allow classes to approach the maximum performance level for their wealth because they could be up to 499 memory pages below their ideal allocation. We settled on using a page block size of 100 pages as it provided a balance of refinement for the allocations and a large enough marginal utility for class agents, using the utility functions we've defined, to make accurate bidding decisions.

The values of *a* and *b* were determined by using the buffer pool performance results for a buffer pool of 500 pages and of 20,000 pages from the previous experiments. Using the equations presented in Chapter 3, values for *a* and *b* were obtained for each class agent.

For the all of the remaining experiments presented in this work, we set the memPages parameter to 32, 768 and the pageBlockSize parameter to 100. Similarly, the values of *a* and *b* for each class agent are constant for all of the experiments presented in this work. Only the parameters affecting the Workload Class Importance Policy are altered and these settings are discussed for each experiment.

To show the validity of this approach to resource allocation, we first manually determine an allocation. This manual allocation does not take into account importance, but tries to maximize the hit rate for each buffer pool while minimizing the overall number of physical disk accesses. This allocation was arrived at by using an initial estimate based on the workload characteristics and the comparative needs of the three workloads. The DBMS was then configured to use this initial allocation. The buffer pool performance was monitored and the allocations were refined. This refinement continued until all workloads exhibited similar performance and the total number of physical reads reached a minimal value. This resulted in the allocation shown in Table 5.1.

Allocations	int 1	int 2	int 3	int 4	int 5	int 6	int 7	int 8	int 9	int 10	int 11	int 12
Manual												
Class 1	9980	9980	9980	9980	9980	9980	9980	9980	9980	9980	9980	9980
Class 2	11389	11389	11389	11389	11389	11389	11389	11389	11389	11389	11389	11389
Class 3	11399	11399	11399	11399	11399	11399	11399	11399	11399	11399	11399	11399
Preemptive w/ Aging												
Class 1	11368	11300	11200	11200	11168	11400	11300	11268	11200	11200	11300	11368
Class 2	10800	10868	10800	10668	10600	10500	10800	10600	10600	10700	10868	10700
Class 3	10600	10600	10768	10900	11000	10868	10668	10900	10968	10868	10600	10700
Preemptive w/o Aging												
Class 1	11368	11200	11100	11100	11068	11400	11200	11168	11100	11100	11268	11300
Class 2	10800	10900	10800	10600	10600	10500	10968	10600	10668	10800	10900	10768
Class 3	10600	10668	10868	11068	11100	10868	10600	11000	11000	10868	10600	10700

Table 5.1: Comparing manual buffer pool allocations to results of economic model

We then use the economic model simulator to provide a set of allocations with all classes at an equal level of priority. We ran the simulation using the preemptive model (all resources allocated through competition) since it is the purest form of the economic model. We tested both with and without aging to see if there were any significant differences at this point. For these tests, all classes were labeled as Normal Importance.

However, running the simulation with all classes at the same level of importance should provide the same resulting allocations regardless of whether they are all at High Importance, Normal, or Best Effort.

The resulting allocations are different from those of the manual configuration as they demonstrate two different methods for reaching a similar goal. Whereas the manual configuration gave Class 1 the lowest allocation, the economic model consistently allocates the largest amount of buffer pool space to Class 1. This is due to the initial estimate, the starting point, used by the manual configuration. Although, through the refinement phase, the amount of memory assigned to Class 1 increased, the goal hit rates were reached before the allocation to Class 1 exceeded the other class allocations. In the economic model simulation, all classes start at the same point, there is no initial estimate.

We ran each of the resulting workload scripts on the test system using the allocations in Table 5.1. The configurations were each run three times and the results presented below are an average of the three runs. The resulting number of physical reads from the three runs showed a variation of less than 0.3% from the mean. This low variation is expected as the workload scripts are identical for each test run. We compare the resulting allocations of the simulator and the manual allocation.

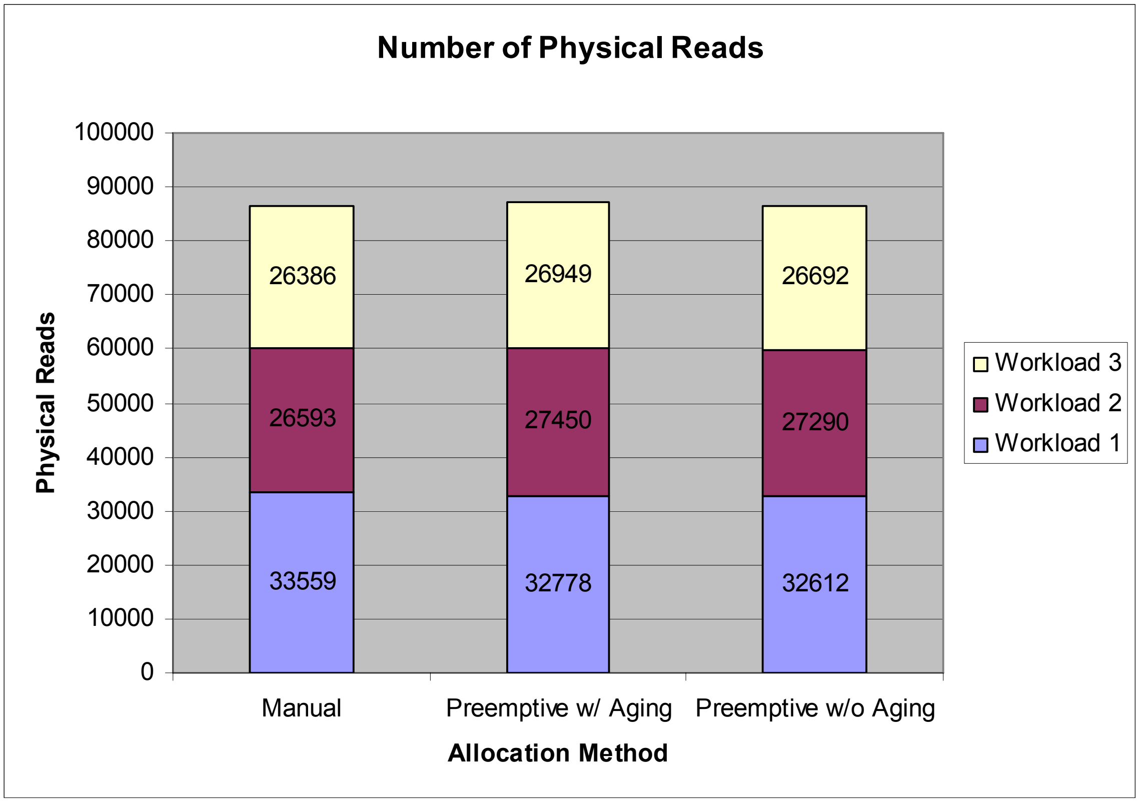


Figure 5.1: Total number of physical reads for configurations suggested by manual configuration and simulation

The two simulator allocations resulted in performance very similar to that of the manual static allocation. As seen in Figure 5.1, the total number of physical reads is similar for each of the configurations.

With no importance levels set, it appears that the economic model simulation provides configurations resulting in similar performance. Thus, we can assume that the economic model simulation will provide us with an adequate buffer pool space allocation system.

The next experiments involved running the economic model simulation with a number of different combinations of importance. As the workloads all have similar resource needs, when run with different combinations of importance labels, a correct allocation will show a High Importance class to receive a large portion of the allocation, while a Best Effort class will receive a low allocation. Each of these runs is done using the Preemptive model without Aging (by setting parameters preemptive = true and aging = false) so as to provide the purest example of the economic model at work. This results in all allocations being done through competition and the results of the previous interval does not affect future intervals. Some sample results are presented in Table 5.2.

Allocations	int 1	Int 2	int 3	int 4	int 5	int 6	int 7	int 8	int 9	int 10	int 11	int 12
Class 1 – High	15400	15300	15268	15300	15300	15500	15368	15368	15368	15268	15300	15400
Class 2 - Normal	13068	13100	13100	12968	12900	12768	13000	12900	12900	13000	13100	12968
Class 3 – Best	4300	4368	4400	4500	4568	4500	4400	4500	4500	4500	4368	4400
Class 1 - High	14668	14568	14468	14568	14568	14800	14600	14600	14600	14500	14600	14668
Class 2 - High	14000	14100	14100	13900	13900	13700	14000	13900	13868	14000	14068	13900
Class 3 – Best	4100	4100	4200	4300	4300	4268	4168	4268	4300	4268	4100	4200
Class 1 – Best	5000	5000	5000	5000	4968	5100	5000	5000	5000	5000	5068	5100
Class 2 - Normal	14000	14068	13900	13700	13700	13600	13968	13768	13668	13800	14000	13868
Class 3 - Normal	13768	13700	13868	14068	14100	14068	13800	14000	14100	13968	13700	13800

Table 5.2: Allocation results from different combinations of importance labels using simulator

As can be seen in these allocations, workloads with similar importance levels receive similar allocations. Additionally, as there are fewer High Importance classes, the Best Effort classes receive higher allocations. All of the allocations are as we expected, thus confirming that the model simulator is able to allocate buffer pages appropriately according to importance levels. However, these results are with the (1.0, 2.0, 3.0) multiplier set (by setting the following parameter values: importantMultiplier = 3.0, normalMultiplier = 2.0, and bestEffortMultiplier = 1.0), so we must still experiment to determine that these are correct degrees of importance for each importance level. This is shown in Section 5.3.

5.3 Degree of Importance

When implementing an importance policy, the degree to which one class is more important than another is a key problem. We experiment, using our economic model simulation, to determine the proper degree of difference for the various priority levels. The model examines three different importance levels: High Importance, Normal, and Best Effort.

The degrees of importance are implemented through multipliers for the wealth assigned to a class; a higher importance class will have a higher multiplier, giving the class more total wealth with which to purchase resources and be able to outbid competitors. In all experiments, the Best Effort class is given a multiplier of 1.0. This is reasonable, since a Best Effort class should only be able to acquire resources if there is no other higher importance class that desires them. Thus, a Best Effort class will only be able to pay a minimum price for resources. We try a number of other multipliers for the High and Normal importance classes.

5.3.1 Evaluation Criteria

A proper Importance scheme within an economic model will provide the highest price difference between what the high priority class is paying for resources compared to the lower priority classes. This will demonstrate that the high priority classes are able to purchase the resources needed before the lower priority classes. However, this is mitigated by the need to cause the lowest increase in the number of physical reads for the lower priority classes so that overall system performance is impacted as little as possible. Since hit rate is proportional to buffer space allocation, we look for the highest average allocation for the normal priority and best effort classes. Thus, we look for the best combination of these two metrics:

- Price Differential: the difference in average price paid for resources between high priority and best effort classes, a higher value is better.
- Average Allocation: The average number of buffer pool pages allocated to the normal priority and best effort classes, a higher value is better.

The results for each metric are normalized with respect to the mean for that metric and then combined into a single metric to show the best combination of the two.

5.3.2 Experimental Results

We tried five different sets of multipliers. In order of Best Effort, Normal, and High Importance, the sets were: (1.0, 2.0, 3.0), (1.0, 2.0, 5.0), (1.0, 2.0, 10.0), (1.0, 5.0, 6.0), and (1.0, 5.0, 10.0). These represent a number of different relative differences in importance, such as High Importance being similar to Best Effort or very much preferred, and Normal Importance spanning the range from Best Effort to High Importance. We test these parameter settings using the Preemptive model of importance with and without Aging.

The results of the simulation for these different sets of multipliers provided two sets that performed better than the others according to our criteria of price differential and average allocation (Figures 5.2 and 5.3).

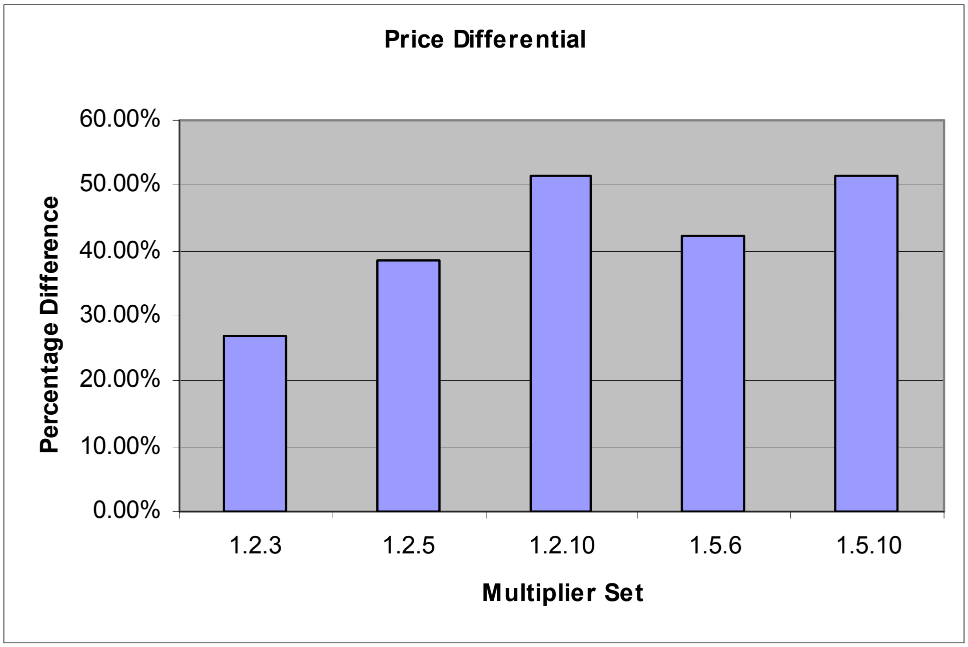


Figure 5.2: Price Differential between High Importance and Best Effort classes for different multiplier sets

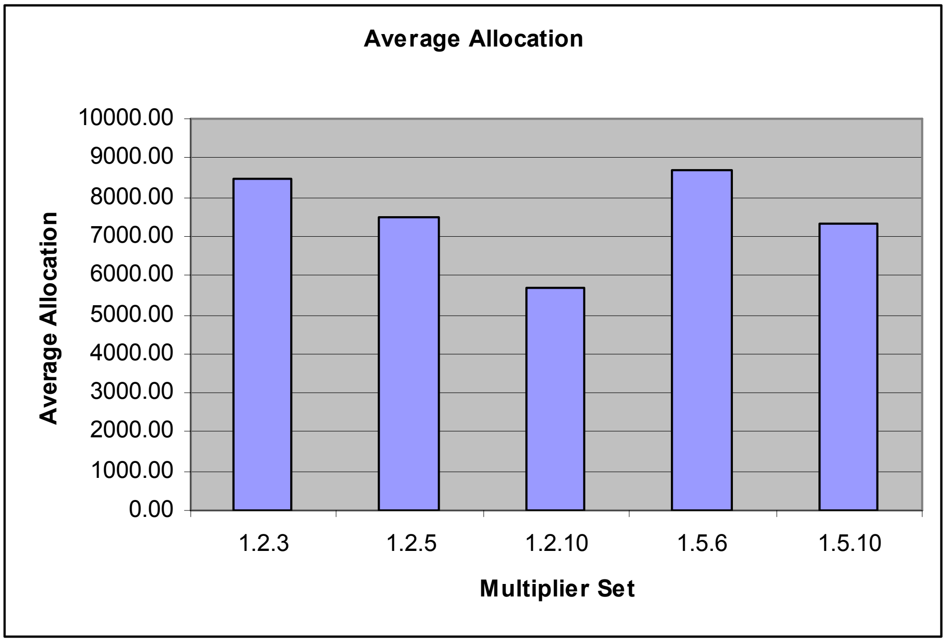


Figure 5.3: Average Allocation awarded to Normal and Best Effort classes using different multiplier sets

From the Figures, we see that the fourth set (1.0, 5.0, 6.0) scored best in average allocation while the fifth set (1.0, 5.0, 10.0) provided the best price differential. When we normalize the multiplier set metrics and combine them into a single score we see a clear leader as seen in Figure 5.4.

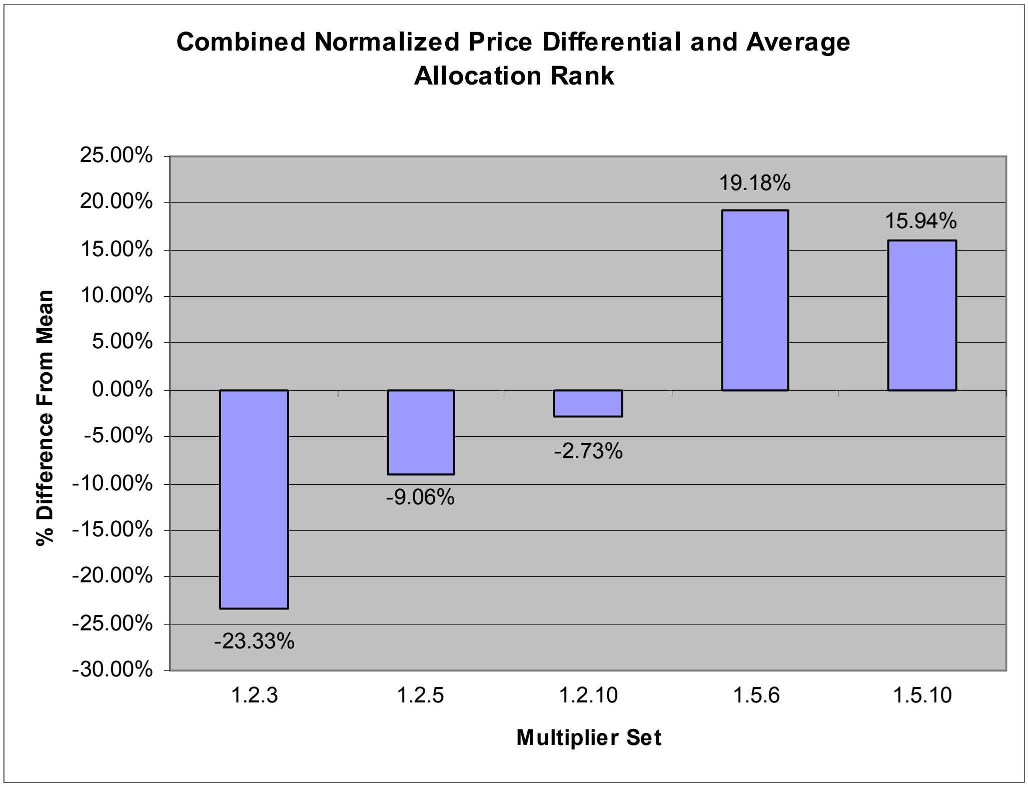


Figure 5.4: Combined normalized price differential and average allocation rank

The fourth set (1.0, 5.0, 6.0) provides the best combination of our two desired metrics. We further experimented to compare these two candidates (the fourth and fifth sets) and ran the different allocations (Table 5.3) on our test system to find their resulting total number of physical reads, with the lowest number indicating the best multiplier set (Figure 5.5).

Allocations	int 1	int 2	int 3	int 4	int 5	int 6	int 7	int 8	int 9	int 10	int 11	int 12
1.5.6												
High Importance	15400	15300	15268	15300	15300	15500	15368	15368	15368	15268	15300	15400
Normal Importance	13068	13100	13100	12968	12900	12768	13000	12900	12900	13000	13100	12968
Best Effort	4300	4368	4400	4500	4568	4500	4400	4500	4500	4500	4368	4400
1.5.10												
High Importance	18200	18100	18068	18068	18068	18300	18100	18100	18100	18068	18100	18200
Normal Importance	10968	11000	11000	10900	10900	10700	10968	10868	10868	10900	10968	10868
Best Effort	3600	3668	3700	3800	3800	3768	3700	3800	3800	3800	3700	3700

Table 5.3: Allocations used for determining multiplier set

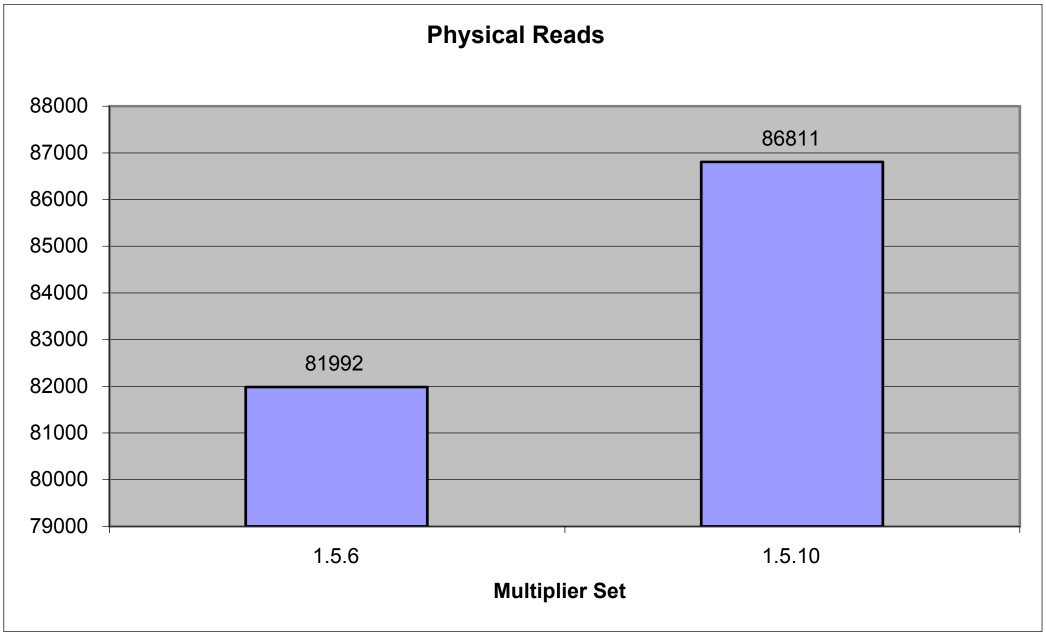


Figure 5.5: Total physical reads on test system using best candidate multiplier sets

We found that when run using our test set up, the set of (1.0, 5.0, 6.0) resulted in 5.55% fewer physical reads than (1.0, 5.0, 10.0). Thus, we selected (1.0, 5.0, 6.0) as our multiplier set for our further experiments in implementing an importance policy. Although these specific multipliers may be specific to these experiments, there are rules-of-thumb that can be observed from these results.

- The Best Effort class should have a multiplier of one to ensure that it is not able to be allocated resources at anything but the minimum price.
- The High Importance class should have a significantly higher multiplier than the Best Effort class to ensure that it has the wealth to purchase resources and reach its desired allocation before other classes.
- The Normal Importance class should have a multiplier similar to that of the High Importance class so that its overall impact on the system is mitigated by a higher average allocation, resulting in a lower total number of physical reads.

5.4 Definition of Importance

The second key problem in implementing a Workload Class Importance Policy is defining what importance means. Traditionally, in computing, there are two implementations of priority: preemptive and non-preemptive. In a preemptive scheme, one agent can commandeer all resources, preventing others from executing. A non-preemptive scheme, on the other hand, does not allow one class to prevent others from executing. Additionally, these schemes have been augmented with Aging. That is, as an agent is forced to wait for the resources it needs, its ability to compete for buffer pool memory is increased.

We investigate a number of definitions of importance, as explained in Chapter 4. These include:

- Preemptive: In a preemptive scheme, one class may appropriate all resources preempting the execution of others.
- Non-Preemptive: In the non-preemptive scheme, a minimum amount of resources are guaranteed to all classes.
- Preemptive exempting High Importance: In this scheme, we pre-allocate the resources the high priority class needs and use a preemptive model for the remaining resources. This guarantees a level of performance only for important classes.

The two importance schemes that offer an initial allocation of memory also required additional experimentation to determine the appropriate initial allocation. For the Non-Preemptive schemes, we looked for a minimum value that would allow every class agent to achieve a modest buffer pool hit rate. We try initial allocations targeting a 50% and a 75% buffer pool hit rate for each class agent. The allocation is determined using the inverse of Belady's equation for each class agent. This is meant to act as a guaranteed minimum level of performance for all workload classes.

For the High Importance exempt Preemptive scheme, we try to provide a high level of performance to the High Importance classes while still allowing some resources to be available to less important classes. We select initial allocations targeting 80%, 90% and 95% buffer pool hit rates for the High Importance classes while the less important classes receive no initial allocation. This ensures that the High Importance classes do not have their performance impacted by a lower importance class with a large amount of wealth.

In both of these schemes, the class agents are charged appropriately for these allocations according to their amount of wealth. As stated in Chapter 3, a class agent pays a corresponding percentage of its wealth for a corresponding hit rate. Thus, an initial allocation targeting a 50% buffer pool hit rate would cost 50% of a class agent's wealth. This allows for classes to still participate in the economy, however, they have less wealth and will likely be able to win fewer auctions as a penalty for their being granted a minimum performance. This removes the competitive advantage of having a pre-allocated amount of memory. If the classes were not charged and they were pre-allocated a significant amount of memory, they would be able to still make large bids for resources they normally could not afford.

We examine each of these importance schemes with and without Aging. Aging is implemented in the economic model simulation by allowing class agents to accumulate wealth. In the non-aging schemes, class agents do not carry over any unspent wealth from segment to segment.

5.4.1 Evaluation Criteria

To evaluate the effectiveness of the importance schemes, we examine two criteria. First, we want an importance scheme that provides the highest benefit to the classes designated as High Importance. This will be evident by the hit rate that is achieved from the allocations provided by the simulation. The hit rate is the percentage of the time when a requested page of information, either an index page or data page, is found in the buffer cache. This is a common measure of the performance of a single buffer. A higher hit rate is better.

Secondly, we want to provide this benefit to important classes with as little effect on the rest of the system as possible. Thus, we also look at the total number of physical reads recorded by the system for each importance scheme as in Section 5.2 and 5.3. Again, a lower score is better for total physical reads. We claim that the scheme that can provide the best combination of these two criteria provides us the most beneficial definition of importance when looking at allocating buffer pool space.

5.4.2 Experimental Results

For the following experiments, we continue to use memPages = 32,768 and pageBlockSize = 100. The values of *a* and *b* for each class agent are the same as determined in Section 5.2. Every result showing buffer pool performance statistics, such as hit rate and physical reads, presents of an average of three runs on the test setup.

We first experiment to determine the initial allocations in the Non-Preemptive and High Importance Exempt schemes. We begin with the Non-Preemptive scheme (set parameter preemptive = false), with and without Aging, at an initial allocation targeting 50% and 75% buffer pool hit rates for each class agent to determine the proper value for the minHitrate parameter. The results are as shown in Figures 5.6 and 5.7.

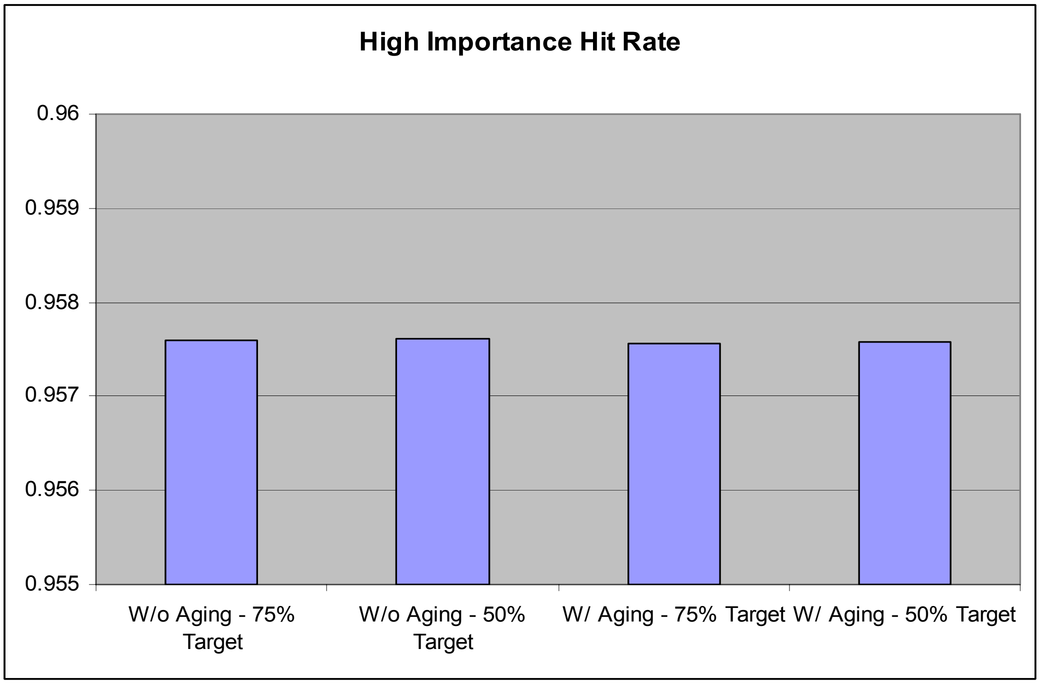


Figure 5.6: Hit rate of High Importance class using target 50% and 75% buffer pool hit rate initial allocations for Non-preemptive scheme

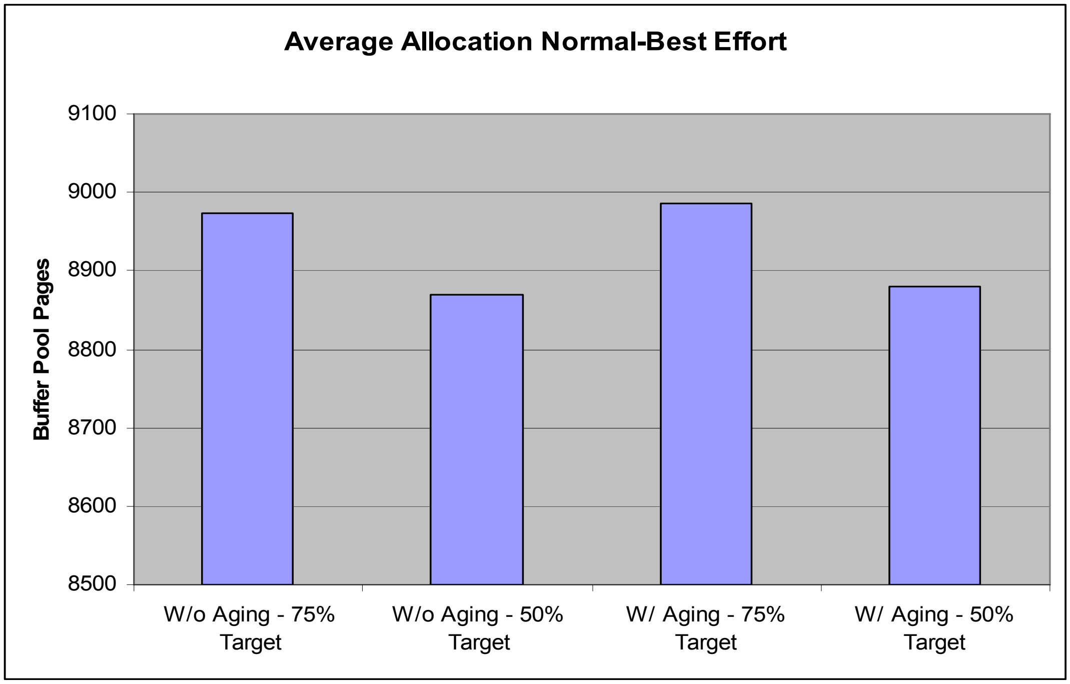


Figure 5.7: Average allocation of Normal and Best Effort classes using target 50% and 75% buffer pool hit rate initial allocations for Non-preemptive scheme

As shown in Figure 5.6, the initial allocation has little effect on the hit rate achieved by the High Importance class. This is due to the High Importance class agent's ability to acquire additional resources. However, there is a noticeable difference in the average allocations of the Normal and Best Effort classes. Using an initial allocation targeting 75% hit rate results in a higher average allocation for the two lower importance classes. This will result in a lower number of total physical reads due to the diminishing returns nature of the relationship between buffer pool size and hit rate. By reallocating memory from the High Importance class to the less important classes, the increase in physical reads for the High Importance class is more than offset by the decrease in physical reads for the lower importance classes. Thus, we use a 75% target for the initial allocation (set parameter minHitrate = 0.75) in the following experiments.

Similarly, we examine three initial allocations for the High Importance Exempt scheme to determine the proper setting for the exemptImportantHitrate parameter. They are initial allocations targeting 80%, 90%, and 95% buffer pool hit rates. Again, these initial allocations are calculated using the inverse of Belady's equation. As shown in Figure 5.8, we see the best hit rates are achieved by the 80% and 95% allocations.

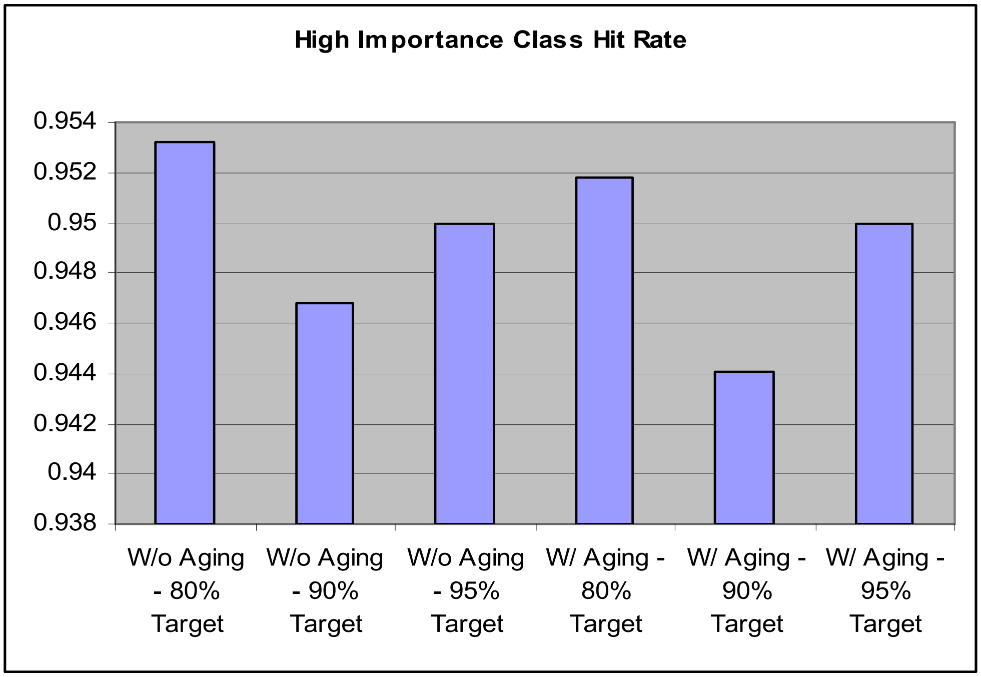


Figure 5.8: High Importance class hit rates achieved using target 80%, 90%, and 95% buffer pool hit rate initial allocations for Preemptive exempt High Importance scheme

However, when looking at the allocations received by the High Importance class under each of these schemes, we see that, under the 80% scheme the class must still compete for most of its allocation (Figure 5.9) to achieve this high hit rate.

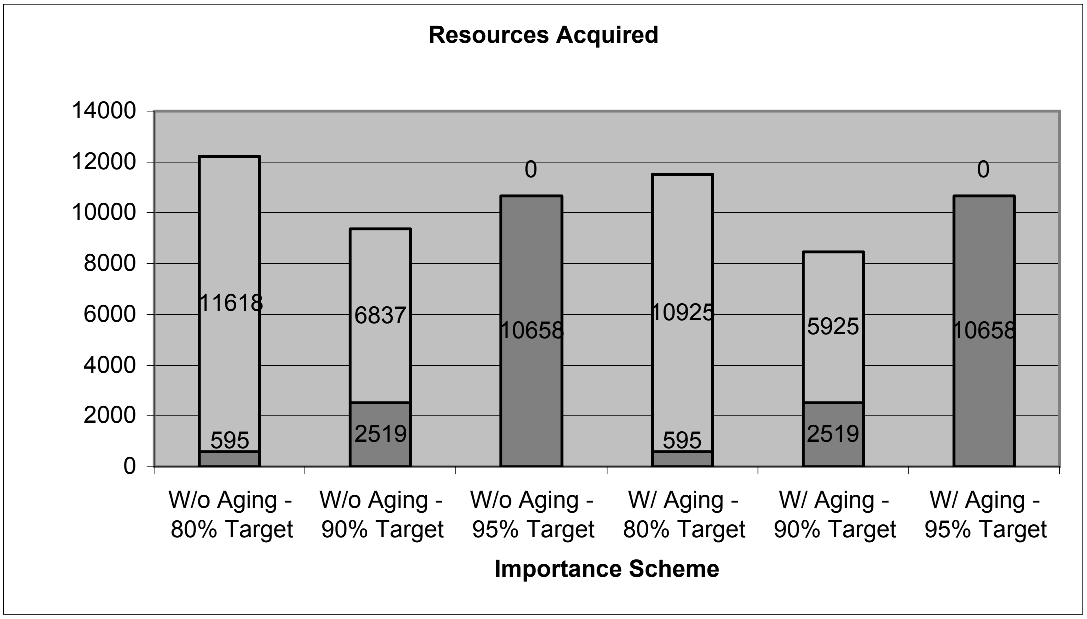


Figure 5.9: Initial allocation versus total allocation under each scheme (darker portion is initial allocation, lighter portion represents resources won through competition)

As is shown, with the target 80% hit rate initial allocation, the High Importance class is very dependant on winning resources through competition for its performance. This is not a desired quality as another class could acquire those resources. The target 95% buffer pool hit rate initial allocation scheme gives a High Importance class the resources it needs so that it can maintain a very high hit rate without needing to compete for resources. This the property that we desire for the initial allocation, so the target 95% buffer pool hit rate initial allocation for the High Importance class (set parameter exemptImportantHitrate = 0.95) is used in all further experiments. This allows us to guarantee a high-level of performance without being dependant on the ability to compete for resources. This will help to prevent the High Importance classes from being preempted.

We began the experiments by running each importance scheme in simulation. We then ran the resulting allocations (Table 5.4) on our test set up three times, recording both hit rates and physical reads. Figures 5.10 and 5.11 present the average results.

Average Allocations	Non-Preemptive w/ Aging	Non-Preemptive w/o Aging	High Exempt w/ Aging	High Exempt w/o Aging	Preemptive w/ Age	Preemptive w/o Aging
High Importance	14990	15015	10658	10658	15345	15281
Normal Importance	13408	13259	16497	16523	12981	13070
Best Effort	4515	4494	5613	5588	4442	4417

Table 5.4: Average memory allocations for each Importance scheme suggested by Simulation

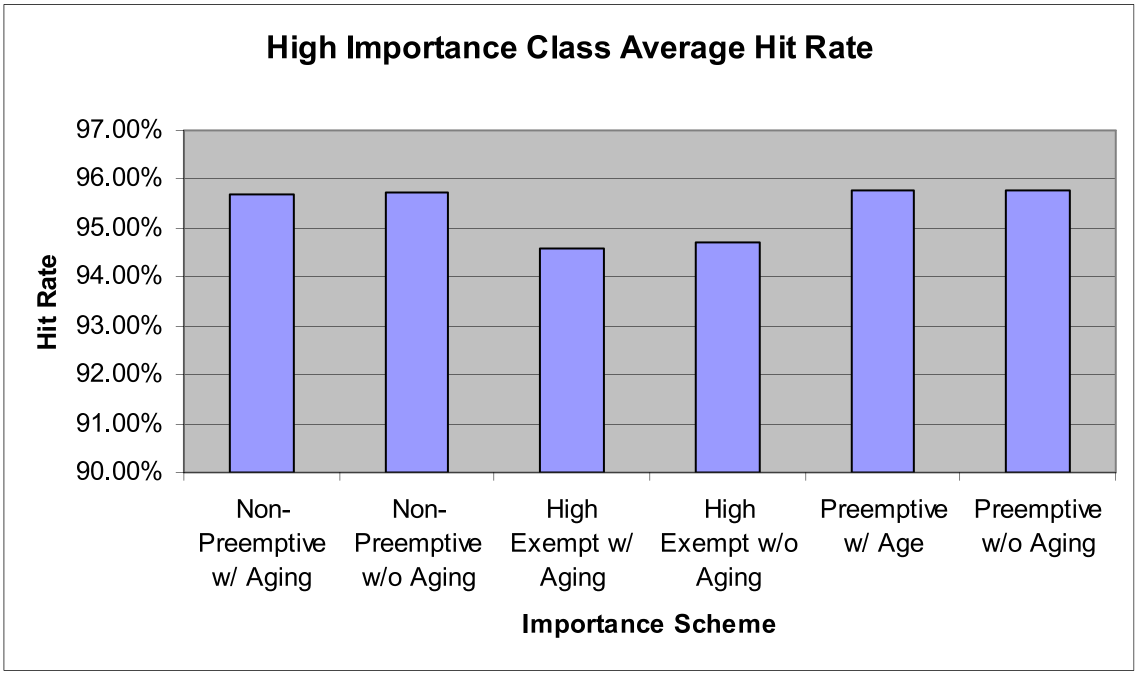


Figure 5.10: Average hit rate for High Importance classes using each importance scheme

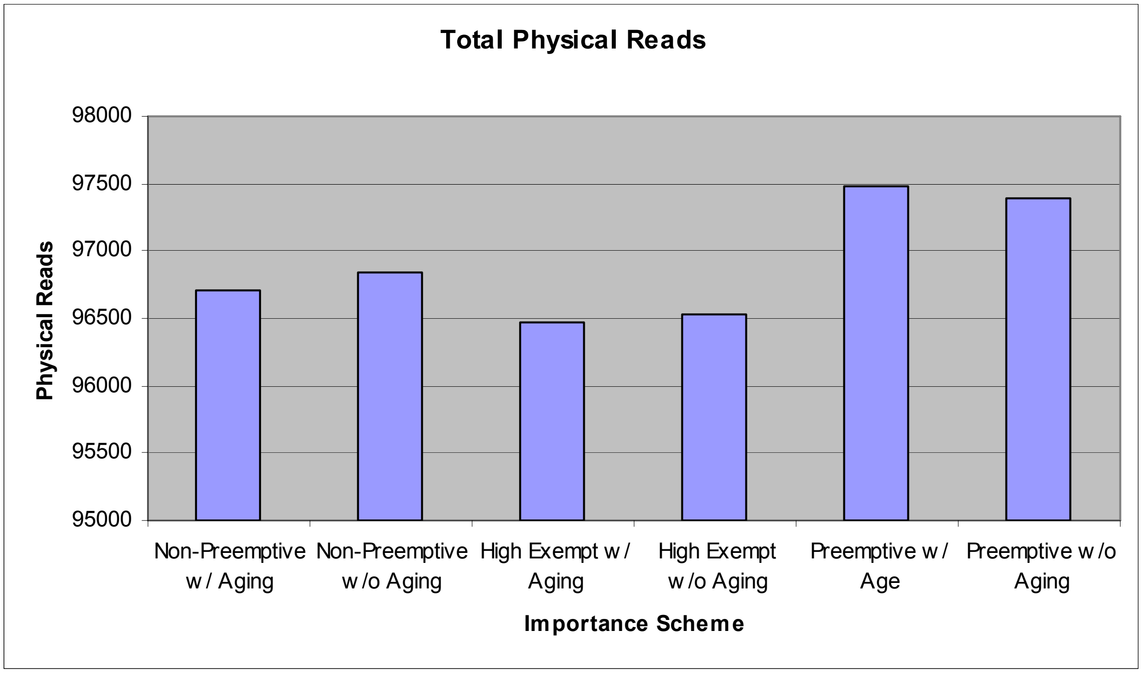


Figure 5.11: Total physical reads recorded using each importance scheme

We first examined the difference between schemes with and without Aging. In general, we see a lower hit rate for the High Importance classes when Aging is involved. We also see a lower total number of physical reads in schemes that include Aging (Figures 5.12 and 5.13).

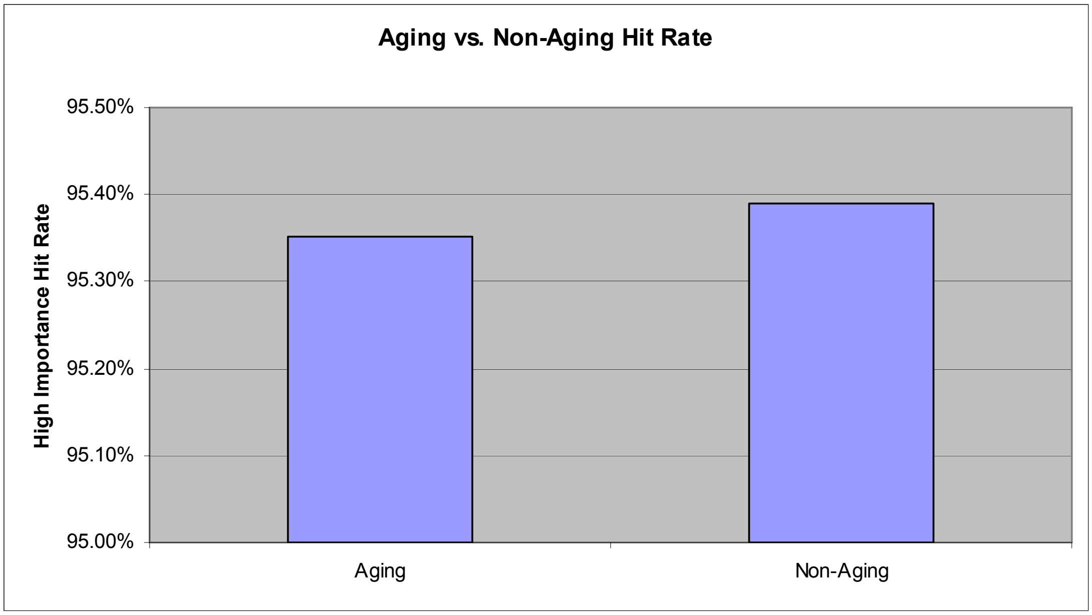


Figure 5.12: Comparing average hit rates for Importance schemes with and without Aging

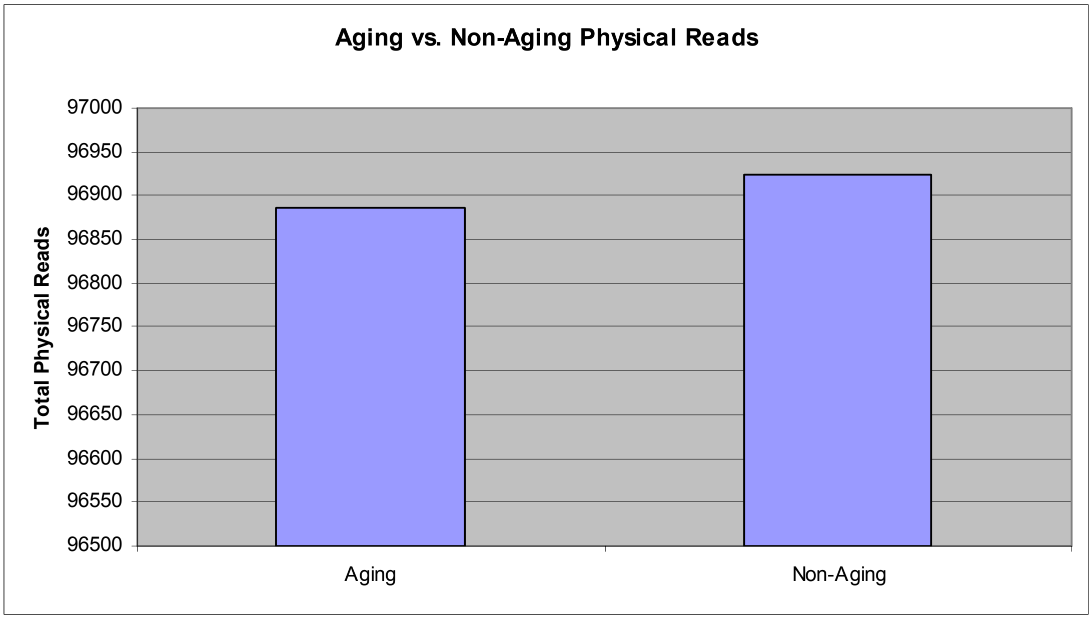


Figure 5.13: Comparing average physical reads for Importance schemes with and without Aging

As can be seen from Figures 5.12 and 5.13, there is no clearly better scheme. However, the results of implementing Aging do seem to create a slight moderating effect, as should be expected. Aging was implemented to allow lower importance classes to eventually be more competitive. In some intervals the lower importance class agents will be winning resources away from the High Importance class. Due to the diminishing returns nature of the relationship between buffer pool size and buffer pool hit rate, this will lower the hit rate of the High Importance class. However, the increase in physical reads incurred by the High Importance class will be more than offset by the decrease in physical reads for the other classes as their hit rates increase.

Figure 5.14, shows each of the different importance schemes normalized and combined scores in the two metrics. Since we want a higher hit rate for the High Importance class and a lower total number of physical reads across all classes, a combined score closest to zero provides the best balance of the criteria.

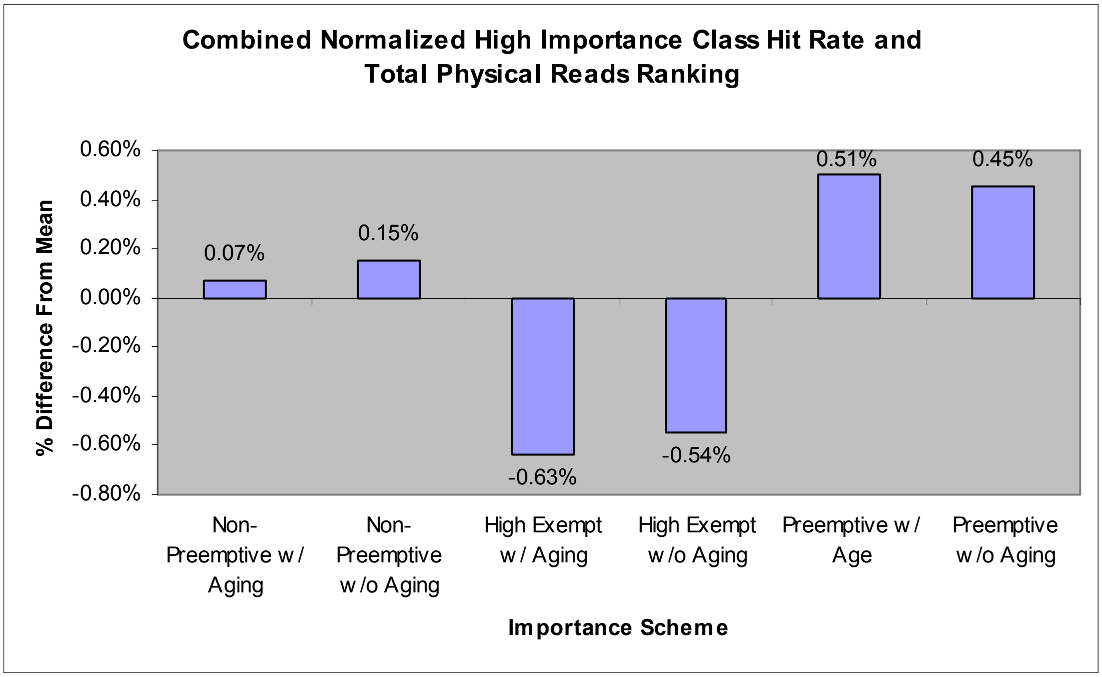


Figure 5.14: Combined normalized High Importance Class hit rate and total physical reads ranking

When looking at each scheme and their performance when we normalize the metrics, we see that the Non-Preemptive schemes provide the best balance of the two criteria. Semantically, this scheme would indicate that all workload classes can be guaranteed a minimum level of performance, while High Importance classes will be able to reach a significantly higher level and overall system performance is impacted as little as possible. Through a basic economic model, this scheme can be achieved in an automated fashion through simple rules of wealth assignment and trade. By assigning wealth based on the estimated number of I/O operations, by pre-allocating resources and charging a minimal amount for them, and by allowing any additional resources to be allocated through a competitive auction system, we were able to implement this importance scheme using our economic model.

Chapter 6

Conclusions and Future Work

To progress to the Autonomic stage, Adaptive computing systems must be guided by business policies. However, many of these business policies do not directly translate to metrics used for computer system performance. In this thesis, we present the use of an economic model to bridge the gap between these different metrics. The model is built to represent the buffer pool memory allocation problem in database management systems. We then utilize this model to investigate the implementation of a Workload Class Importance Policy. The model is implemented as an offline simulation to test the performance of a number of different possible importance policies. Section 6.1 summarizes the contributions of this thesis, while Section 6.2 presents conclusions based on our experiments. Finally, Section 6.3 suggests some direction for future work.

6.1 Thesis Contributions

This thesis examines some of the difficulties in implementing business policy guided Autonomic computing. We identify the translation of typical business policy metrics to computing system performance metrics as a key challenge. As a solution, we suggest and implement an economic model for resource allocation. We use this economic model in simulation to address the buffer pool sizing problem to test the model's validity. Finally, we use this model and simulation to address the difficulties in implementing a Workload Class Importance Policy.

We develop an economic model that contains consumer agents that each represents a different workload running on a single DBMS. These agents are each assigned a utility function that is used to determine the desired allocation of resources. The model also contains a broker that is responsible for allocating the resource through a sealed-bid auction process.

This model is developed into an offline simulation representing the buffer pool sizing problem. A single broker is responsible for distributing buffer pool memory to the various consumer agents representing concurrent workloads on the DBMS. Consumer agents are provided a workload, a utility function, and wealth. Through a series of auctions, the consumers obtain the required buffer pool memory from the broker. The simulation presents the results as a series of memory allocations at checkpoints in the workloads. To validate the model, we compare the resulting allocations of the simulation with those of a manual configuration.

We use the economic model simulator to implement a Workload Class Importance Policy. The model allows us to implement a number of possible importance policies by varying the way in which wealth is allocated to the class agents and the rules of resource allocation followed by the broker. We experiment with each of the different importance schemes and select the one that provides the best performance for the most important classes while still providing a high level of system wide optimization.

6.2 Conclusions

Based on our experimentation, we can conclude:

- The basic economic model used in this work provides an adequate method of allocating buffer pool space. The results achieved are very similar to that of the manual configuration. The added benefits of decentralization, automation, and simple consumer agents make the economic model an interesting approach to utilize for resource allocation.
- The economic model greatly eases the implementation of a Workload Class Importance Policy. Through minor rule and parameter changes, we are able to implement a variety of possible importance policies.
- Non-Preemptive importance schemes provide the best balance of a relatively high hit rate for important classes and a low number of system-wide physical reads mitigating overall system impact. Due to the diminishing returns relationship between buffer pool size and hit rate, giving the initial allocation to the low importance classes provides a greater decrease in the total number of physical reads than allowing the high importance class to win these additional pages and achieve a higher hit rate.
- The degree of importance that provided the best results involved a large differential between the High Importance class and the Best Effort class while having a small differential between the High Importance class and the Normal Importance class.

- Aging importance schemes consistently give a lower average hit rate for the high importance class compared to the same importance scheme without aging, while resulting in a lower number of physical reads system wide. The schemes using aging allow the lower importance classes to appropriate some of the resources from the higher importance classes on occasion. This is again due to the relationship between buffer pool size and hit rate.

6.3 Future Work

There are a number of interesting avenues of future research suggested by this work. Some of the most interesting are:

- Refining the economic model. Although we use Belady's equation as a hit rate estimator for our utility curve, there has been work done suggesting alternative hit rate estimators [28]. Implementing these as utility curves could provide differing results. As well, alternative trade mechanisms, such as reverse auctions could be investigated as they have been used in other economic models with success.
- Extending the economic model. Future work would look at expanding the model to allocate multiple resources by adding additional brokers and to allow classes to trade-off between the various resources by utilizing multiple utility functions. Additionally, more economic features, such as a futures market, could be added to allow for agents to choose when to execute to maximize performance for a given budget. This would be especially useful in a system where the wealth of a class represents a real-world budget.

- Implementing additional business policies. Policies such as an Operating Costs policy could be implemented where the broker can only lend out resources in accordance with the costs it incurs for the resource to run, i.e. if using an additional disk drive costs x dollars and the client is willing to spend y dollars, the broker will only sell the resource if y >= x. This would be useful in guiding systems towards profit maximization.

Economic models provide a very interesting avenue of study for resource allocation problems in Autonomic Computing. Although this research has been underway for some time now [12], it has yet to make much impact in commercial products. However, when taking into consideration the ease with which most business policies translate into economic terms and that business policy guidance is a key feature of Autonomic Computing, these systems should gain additional interest.

References

1. Agrawal, S., Chaudhuri, S. and Narasayya, V. (2000). “Automated Selection of Materialized Views and Indexes,” *Proceedings of the 2000 International Conference on Very Large Databases,* Cairo, Egypt.

2. Aiber, S., Gilat, D., Landau, A., Razinkov, N., Sela, A., and Wasserkrug, S. (2004). “Autonomic Self-optimization According to Business Objectives”. *Proceedings of the International Conference on Autonomic Computing*, 206-213.

3. Belady, L. (1966). “A Study of Replacement Algorithms for Virtual Storage”. Computer, *IBM System Journal*, 5(2), July, 78-101.

4. Brown, K.P, Carey, M.J. and Livny, M. (1993). “Managing Memory to Meet Multiclass Workload Response Time Goals”. *Proceedings of the 1993 International Conference on Very Large Databases*, Dublin, Ireland, 328-341.

5. Brown, K.P., Mehta, M., Carey, M. J., and Livny, M. (1994). “Towards Automated Performance Tuning For Complex Workloads”. *Proceedings of the 1994 International Conference on Very Large Databases*, 72-84.

6. Brown, K. P. (1995). *Goal Oriented Memory Allocation in Database Management Systems*. PhD thesis, University of Wisconsin-Madison.

7. Brown, K.P., Carey, M.J. and Livny, M. (1996). “Goal-Oriented Buffer Management Revisited”, *Proceedings of the 1996 Association for Computing Machinery Special Interest Group on Management of Data*, 353-364.

8. Bureau of Labor Statistics, U.S. Department of Labor. (2005). *Occupational Outlook Handbook, 2004-05 Edition*, Computer Systems Analysts, Database Administrators, and Computer Scientists. Retrieved August 31, 2005 from http://www.bls.gov/oco/ocos042.htm

9. Buyya, R., Abramson, D., Giddy, J. and Stockinger, H. (2002). “Economic Models for Resource Management and Scheduling in Grid Computing”. *Special Issue on Grid Computing Environments, The Journal of Concurrency and Computation: Practice and Experience (CCPE)*, Wiley Press, USA, 14(13-15), 1507-1542.

10. Carey, M.J., Jauhari, R. and Livny, M. (1989). “Priority in DBMS Resource Scheduling”. *Proceedings of the 1989 International Conference on Very Large Databases*, 397-410.

11. Chaudhuri, S., & Weikum, G. (2000). “Rethinking Database System Architecture: Towards a Self-tuning RISC-style Database Architecture”. *Proceedings of the 2000 International Conference on Very Large Databases*, Cairo, Egypt.

12. Cheliotis, G. and Kenyon, C. (2003). “Autonomic Economics: Why Self-Managed e-Business Systems Will Talk Money*”. IEEE International Conference on E-Commerce, 2003. CEC 2003*, 120-127.

13. Datta, A., Son, S. H., and Kumar, V. (1999). “Is a Bird in the Hand Worth More than Two in the Bush? Limitations of Priority Cognizance in Conflict Resolution for Firm Real-Time Database Systems*”. IEEE Transactions on Computers* 49(5), 482-502.

14. Eclipse Foundation (2005). “Eclipse.org Main Page”. Retrieved October 30, 2005 from http://www.eclipse.org/

15. Ferguson, D. F., Nikolaou, C., Sairamesh, J., and Yemini, Y. (1996). "Economic Models for Allocating Resources in Computer Systems". In Scott Clearwater, editor, *Market-Based Control: A Paradigm for Distributed Resource Allocation*, Scott Clearwater. World Scientific, Hong Kong

16. Ganek, A.G. and Corbi, T.A. (2003). "The Dawning of the Autonomic Computing Era". *IBM Systems Journal* 42(1), 5 – 18.

17. IBM (2001). "Autonomic Computing: IBM's Perspective on the State of Information Technology". Retrieved January 31, 2005 from http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

18. IBM (2002). "Autonomic computing and IBM". Retrieved March 14, 2005, 2005, from http://www-03.ibm.com/autonomic/pdfs/AC_BrochureFinal.pdf.

19. IBM (2005). "The Problem". *Autonomic Computing Overview*, retrieved March 12, 2005, from http://www.research.ibm.com/autonomic/overview/problem.html.

20. IBM (2005). "Alter Bufferpool Statement". Retrieved October 30, 2005 from http://publib.boulder.ibm.com/infocenter/db2help/index.jsp?topic=/com.ibm.db2.udb.doc /admin/r0000885.htm

21. IBM (2005). "IBM Software – DB2 Universal Database for Linux, UNIX and Windows – Product Overview". Retrieved October 30, 2005 from http://www-306.ibm.com/software/data/db2/udb/

22. Kephart, J.O., Chess, D.M. (2003). "The Vision of Autonomic Computing". *IEEE Computer*, 36(1):41–52.

23. Microsoft (2005). "Windows XP Home Page". Retrieved October 30, 2005 from http://www.microsoft.com/windowsxp/default.mspx

24. Mullins, C.S. (2003). “The Buffer Pool: Dynamic Buffer Changes and Partitioned Access”. *International DB2 Users Group Solutions Journal*, 10(1).

25. Siberschatz, A., Galvin, P.B., and Gagne, G. (2003). *Operating System Concepts. 6th ed.* John Wiley & Sons, Inc., U.S.A.

26. Stratford, N. and Mortier, R. (1999). “An Economic Approach to Adaptive Resource Management”. *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, 142-148.

27. Sun Microsystems, Inc. (2005). “Java 2 Platform, Standard Edition (J2SE) 1.4.2”. Retrieved October 30, 2005 from http://java.sun.com/j2se/1.4.2/index.jsp

28. Tian, W., Martin, P. and Powley, W. (2003). “Techniques for Automatically Sizing Multiple Buffer Pools in DB2”. *Proceedings of Center of Advanced Studies Conference (CASCON)*, Toronto, Canada, 294-302.

29. Tucker, P. (1998). *Market Mechanisms in a Programmed System. Department of Computer Science and Engineering*, University of California, San Diego.

30. Valentin, G., Zuliani, M., Zilio, D., Lohman, G. and Skelly, A.. (2000). “DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes,” *Proceedings of International Conference on Data Engineering*, San Diego, California, 101-110.

31. Value Based Management.net (2005). “What is Value Based Management”. Retrieved September 5, 2005, from http://www.valuebasedmanagement.net.

32. Walsh, W. E., Tesauro, G., Kephart, J. O., and Das, R. (2004). “Utility functions in autonomic systems.*”* *Proceedings of the International Conference on Autonomic Computing*, 70-77.

33. Ward, D. & Rivani, E. (2005). “An Overview of Strategy Development Models and the Ward-Rivani Model”. Retrieved September 5, 2005, from http://ideas.repec.org/p/wpa/wuwpgt/0506002.html.

34. Weikum, G., Mönkeberg, A., Hasse, C., & Zabback, P. (2002). “Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering”. *Proceedings of the 2002 International Conference on Very Large Databases*.

35. Wellman, M. P. (1996). “Market-oriented programming: some early lessons”. *Market-Based Control: A Paradigm for Distributed Resource Allocation*, Scott Clearwater, World Scientific, River Edge, New Jersey.

36. Xu, X., Martin, P., and Powley, W. (2002). “Configuring buffer pools in DB2 UDB”, *Proceedings of Center of Advanced Studies Conference (CASCON).* Toronto, Ontario, Canada.

Appendix A

TPC-C Benchmark

The workloads used in this research are based on the TPC-C benchmark developed by the Transaction Performance Council (TCP). The TPC-C benchmark is used to test the performance of transaction processing systems. It is an online transaction processing (OLTP) benchmark that models an order-entry system.

The TPC-C benchmark is based on an actual business model, that of a wholesale parts supplier. It represents a business that maintains a number of warehouses associated with sales districts. Each warehouse serves ten sales districts and each sales district serves three thousand customers. The benchmark emulates operators in sales districts selecting one of five transactions.

The benchmark utilizes multiple types of transactions to represent typical order-entry behaviours such as entering orders, recording payments, and monitoring stock levels. The most common transaction is entering a new order. Similarly, recording payments for these transactions is also very common. The other transactions represented involve checking order status, recording order delivery, and checking warehouse stock level. The ratios of these transactions are shown in Figure A.1.

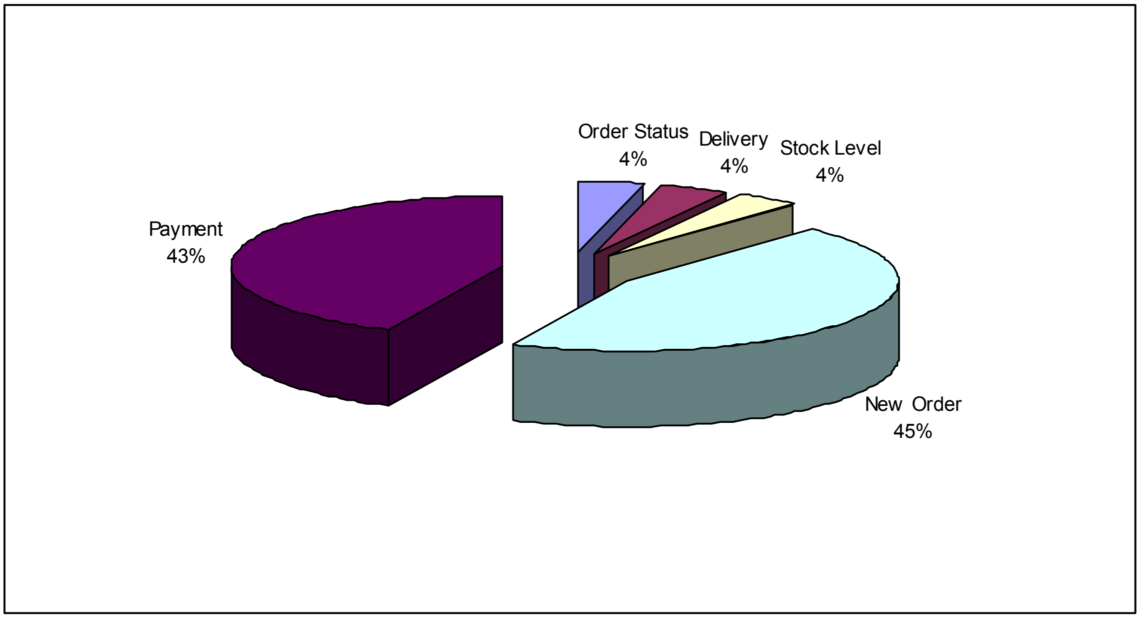


Figure A.1: The percentage frequency of the different transactions in the TPC-C workload.

The database used by the model contains nine relations as shown in Figure A.2. As mentioned, each warehouse serves 10 districts with each district serving 3000 customers. Additionally, each warehouse maintains stock for 100,000 items. The stock for each item at all warehouses is recorded in the Stock relation.

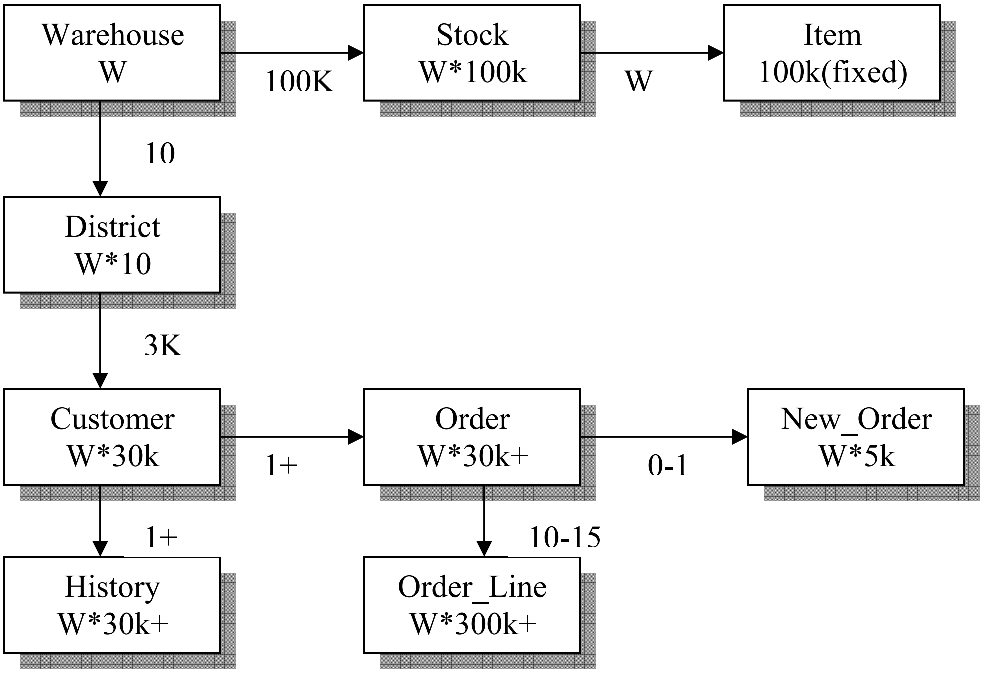


Figure A.2: Database schema for TPC-C benchmark

The Order relation records all orders while the New_Order relation maintains a list of pending orders with the Order_Line relation containing a list of the items for each order. Finally, the History relation keeps records of payments for orders.

The five different transactions function as follows:

- New Order: places an order for, on average, 10 items. The transaction inserts a record in the Order and Order_Line relations and updates the corresponding stock level.
- Payment: updates the Customer relation with new balance and records payment in the History relation.

- Delivery: processes a batch of 10 orders from the New_Order relation, removes them from New_Order and updates the Order relation and Customer relation to indicate the delivery.
- Order Status: checks the current status of a customer's order.
- Stock Level: Checks warehouses for possible supply shortages.

Appendix B

TPC-C Transaction Query Details

This section presents the SQL queries issued by each of the TPC-C transactions. Also presented are the DB2 Explain utility's estimated I/O operations for each query as used in the economic model simulation to estimate the amount of work for a given workload class.

B.1 New Order

```
1. SELECT w_tax, c_discount, c_last, c_credit
        INTO :*s_W_TAX, :*s_C_DISCOUNT, :s_C_LAST, :s_C_CREDIT
        FROM warehouse, customer
        WHERE w_id   = :*s_W_ID
                AND   c_id   = :*s_C_ID
                AND   c_w_id = :*s_W_ID
                AND   c_d_id = :*s_D_ID;
Estimated number of I/O Operations: 3

2. UPDATE district
        SET d_next_o_id = :d_next_o_id
        WHERE d_w_id = :*s_W_ID
                AND d_id = :*s_D_ID;
Estimated number of I/O Operations: 3

3. INSERT INTO orders
        VALUES (:*s_O_ID, :*s_C_ID, :*s_D_ID, :*s_W_ID, :*s_O_ENTRY_D_time,
                        NULL, :*s_O_OL_CNT, :*s_all_local);
Estimated number of I/O Operations: 1

4. INSERT INTO new_order VALUES (:*s_O_ID, :*s_D_ID, :*s_W_ID);
Estimated number of I/O Operations: 1

5. SELECT i_price, i_name, i_data
        INTO :s_I_PRICE, :s_I_NAME, :i_data
        FROM item
        WHERE i_id = :s_OL_I_ID;
```

Estimated number of I/O Operations: 3

```
6. SELECT s_quantity, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dist_05,
                s_dist_06, s_dist_07, s_dist_08, s_dist_09, s_dist_10, s_ytd, s_order_cnt,
                s_remote_cnt, s_data
        INTO :s_S_QUANTITY, :dist_01, :dist_02, :dist_03, :dist_04, :dist_05,
                :dist_06, :dist_07, :dist_08, :dist_09, :dist_10, :s_ytd, :s_order_cnt,
                :s_remote_cnt, :s_data
        FROM stock
        WHERE s_w_id = :s_OL_SUPPLY_W_ID
                AND s_i_id = :s_OL_I_ID;
```

Estimated number of I/O Operations: 4

```
7. UPDATE stock
        SET s_quantity = :s_S_QUANTITY,
                s_order_cnt = :s_order_cnt,
                s_ytd = :s_ytd,
                s_remote_cnt = :s_remote_cnt
        WHERE s_w_id =  :s_OL_SUPPLY_W_ID AND s_i_id = :s_OL_I_ID;
```

Estimated number of I/O Operations: 5

```
8. INSERT INTO order_line
        VALUES ( :*s_O_ID, :*s_D_ID, :*s_W_ID, :i, :s_OL_I_ID,
                        :s_OL_SUPPLY_W_ID, NULL, :s_OL_QUANTITY,
                        :i_OL_AMOUNT, :d_data );
```

Estimated number of I/O Operations: 1

B.2 Payment

```
1. UPDATE customer
        SET     c_data1 = :data1,
                c_data2 = :data2
        WHERE c_id   = :*s_C_ID
                AND   c_w_id = :*s_W_ID
                AND   c_d_id = :*s_D_ID;
```

Estimated number of I/O Operations: 5

```
2. SELECT w_street_1, w_street_2, w_city, w_state, w_zip, w_name, w_ytd
        INTO :s_W_STREET_1, :s_W_STREET_2, :s_W_CITY, :s_W_STATE,
                :s_W_ZIP, :w_name, :w_ytd
        FROM warehouse WHERE w_id = :*s_W_ID;
```

Estimated number of I/O Operations: 2

```
3. SELECT d_street_1, d_street_2, d_city, d_state, d_zip, d_name, d_ytd
        INTO :s_D_STREET_1, :s_D_STREET_2, :s_D_CITY, :s_D_STATE,
                    :s_D_ZIP, :d_name, :d_ytd
          FROM district WHERE d_id = :*s_D_ID AND d_w_id = :*s_C_W_ID;
```

Estimated number of I/O Operations: 2

```
4. INSERT INTO history
          VALUES (:*s_C_ID, :*s_C_D_ID, :*s_C_W_ID, :*s_D_ID, :*s_W_ID,
                              :*s_H_DATE_time, :*s_H_AMOUNT, :hist_data);
```

Estimated number of I/O Operations: 1

B.3 Delivery

```
1. SELECT MIN( no_o_id )
          INTO :o_id :o_id_i
          FROM new_order
          WHERE no_w_id = :*s_W_ID
                    AND   no_d_id = :district;
```

Estimated number of I/O Operations: 6

```
2. DELETE FROM new_order
          WHERE no_w_id = :*s_W_ID
                    AND   no_d_id = :district
                    AND   no_o_id = :o_id;
```

Estimated number of I/O Operations: 4

```
3. UPDATE orders
          SET o_carrier_id = :*s_O_CARRIER_ID
          WHERE o_id   = :o_id
                    AND   o_w_id = :*s_W_ID
                    AND   o_d_id = :district;
```

Estimated number of I/O Operations: 4

```
4. SELECT SUM( ol_amount )
          INTO :ol_amounts
          FROM order_line
          WHERE ol_w_id = :*s_W_ID
                    AND   ol_d_id = :district
                    AND   ol_o_id = :o_id;
```

Estimated number of I/O Operations: 3

```
5. UPDATE ORDER_LINE
          SET ol_delivery_d = :deliveryDate
          WHERE ol_w_id = :*s_W_ID
                    AND   ol_d_id = :district
                    AND   ol_o_id = :o_id;
```

Estimated number of I/O Operations: 14

```
6. SELECT o_c_id, o_ol_cnt
        INTO :c_id, :ol_cnt
        FROM orders
        WHERE o_id   = :o_id
                AND   o_w_id = :*s_W_ID
                AND   o_d_id = :district;
```

Estimated number of I/O Operations: 3

```
7. SELECT c_balance, c_delivery_cnt
        INTO :c_balance, :c_delivery_cnt
        FROM customer
        WHERE c_w_id = :*s_W_ID
                AND c_d_id = :district
                AND c_id = :c_id;
```

Estimated number of I/O Operations: 4

B.4 Order Status

```
1. SELECT c_first, c_middle, c_last, c_balance
        INTO :s_C_FIRST, :s_C_MIDDLE, :s_C_LAST, :*s_C_BALANCE
        FROM customer
        WHERE c_id = :*s_C_ID
                AND c_w_id = :*s_W_ID
                AND c_d_id = :*s_D_ID;
```

Estimated number of I/O Operations: 4

B.5 Stock Level

```
1. SELECT d_next_o_id
        INTO :d_next_o_id
        FROM district
        WHERE d_w_id = :*s_W_ID
                AND d_id = :*s_D_ID;
```

Estimated number of I/O Operations: 2

```
2. SELECT count(distinct S_I_ID)
        INTO :*s_low_stock
        FROM order_line, stock
        WHERE ol_w_id = :*s_W_ID
                AND ol_d_id = :*s_D_ID
                AND ol_o_id < :max_o_id and ol_o_id > :min_o_id
                AND s_i_id = ol_i_id
                AND s_w_id = ol_w_id
                AND s_quantity < :*s_threshold;
```

Estimated number of I/O Operations: 141