Sizing Multiple Buffer Pools for DBMSs

by

Wenhu Tian

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 2003

Abstract

The buffer area is a key resource in database management systems (DBMSs) and the performance of a DBMS is greatly influenced by the effective use of the buffer area. Current DBMSs such as DB2 Universal Database (DB2/UDB), logically divide the buffer area into a number of independent buffer pools. Each database object (table or index) is assigned to a specific buffer pool. The task of configuring the buffer pools, which defines the mapping of database objects to buffer pools and the setting the size for each of the buffer pools, is crucial for achieving optimal performance.

In this thesis, we focus on the buffer pool sizing problem. The goal of our research is to support the DBMS automatically determining an optimal buffer pool sizes for a given workload. This problem has been shown to be a complex constrained optimization problem. Currently this task is performed manually by the database administrators (DBAs).

We present a cost model based on data access time and use a greedy algorithm to solve the optimization problem. The approach is implemented and verified against the TPC-C benchmark database using DB2/UDB. Experimental results show the cost model is accurate, and that the greedy algorithm is fast and sufficient in finding an optimal buffer pool sizes.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Pat Martin, for his excellent guidance, precious advice and endless support during my graduate study and research at Queen's University. Without his help, this thesis would never have got finished.

I would also like to thank Wendy Powley, our wonderful database expert, for the help in setting up the experimental environments and the suggestions about writing this thesis. My thanks also go to my friendly labmates, whose help and advice is very helpful to this thesis.

Special thanks are given to the School of Computing at Queen’s University for providing me the opportunity to pursue graduate studies and their support. I also thank IBM Canada Ltd. NSERC, and CITO for the financial support.

Finally, I would like to thank my parents, and my beautiful wife, Jie Lu, for their love, support, and encouragement in these years.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Motivation

Among the numerous directions of database research, database performance tuning has long been an intriguing subject. It is a black art of adjusting a database management system's (DBMS's) operational parameters for a specific workload in order to achieve the best possible performance [1]. Today's commercial database systems provide up to several hundred "tuning knobs" for this purpose. Most of the tuning work is currently performed by the database administrators (DBAs). However, little help is provided by DBMSs themselves in choosing the settings of these parameters in an intelligent way. Certainly, there are some "rules of thumb" [2, 3, 4], but these rules must by no means be applied blindly as their validity may depend critically on workload and system properties. Therefore, the task of tuning a DBMS is complex and time-consuming. Only an expert can effectively understand and carry out such tasks, which raises substantial barriers to an average customer's ability to own and operate a DBMS.

On the other hand, with the rapid deployment of new computer technologies over the last few years, more novel DBMS applications are in use, including OLTP [5] (On-line Transactions Processing), OLAP [6, 7] (On-line Analytical Processing), data mining [8], web applications and applications involving multimedia applications data [9]. These applications have a wide range of resource demands. The DBMS default settings are often meaningless to the users and haphazard resource allocations may lead to severe resource contention, which results in severe performance degradation. Generally, for optimal tuning in such varying and dynamic environments, a staff of knowledgeable system administrators and performance experts is required. However, with the general drop of hardware costs, the expense of human administration and tuning staff may dominate the overall costs of ownership for a database system, making in-house DBAs infeasible. Simplifying the task of DBMS tuning will allow more widespread use of DBMSs. Moreover, another significant growth potential for the database system lies in embedded applications such as the information services in cellular phones, cars, palm-pilots, and so on. These applications face much tighter resource constraints in their settings. Careful resource tuning is required, but these small gadgets do not typically provide an interface for tuning. Self-tuning is therefore a key issue for these lightweight embedded applications.

The manageability of the DBMS is one of the most important challenges facing the database industry today. To simplify the tuning task, experts agree that shifting some or all of the tuning and configuring responsibility from DBAs to the DBMS itself will be required in the next generation of DBMSs [1, 10, 11]. These auto-

configuring and self-tuning DBMSs will help not only large database owners to reduce their human resource costs, but also small database owners to manage their performance problems without recruiting a staff or increasing their computer system costs in some other way. As a result, the topic has received attention from both the academic and the commercial worlds [1, 12, 13, 15, 16, 17, 18, 19].

Automatic buffer tuning [1, 16, 17, 18, 19] is an important first step towards achieving a self-managing DBMS. The buffer area, a key resource for DBMSs, is a block of memory for caching tables and indices between applications and disks. A page replacement algorithm and some background processes manage the transfer of data between the buffer and the disk. The effective use of the buffer area greatly influences the performance of a DBMS [1, 2, 3, 4, 19]. In previous studies, people have focused on improving the buffer hit rate. If the probability of finding a requested tuple in the buffer pool is high, then the transaction response time decreases because of lower I/O delays. High buffer hit probability also lowers disk contention, which allows the disk I/Os to be serviced more quickly. Moreover, high buffer hit probability reduces CPU overhead due to I/O path length and interrupt processing. All these together effectively decrease the path length per transaction and increases the system transaction throughput for a given processor configuration.

Current DBMSs, such as IBM DB2 Universal Database (DB2/UDB)[20] and Oracle 8i [21], support multiple buffer pools, which means the whole buffer area, according to some pre-defined policies [3, 22], is partitioned into sets of logical buffer pools, and different database objects are assigned to individual buffer pools. The feature of multiple buffer pools is an attempt to provide better control over buffer

cache utilization for sophisticated users. In DB2/UDB, the number of buffer pools and the size of each buffer pool are set manually by configuration parameters. Hence, technically speaking, there are two major issues relevant to efficient utilization of the multiple buffer pools. The first one is the buffer pool configuration problem, which involves choosing the proper number of pools and finding a policy of assigning database objects to buffer pools. The second issue is buffer pool sizing problem, which involves selecting a suitable size for each buffer pool. Xiaoyi Xu proposed a solution to the first problem. His algorithm, a data mining-based clustering algorithm [22], separates database objects into different buffer pools based on the similarities of database objects. However, the sizing problem is not mentioned in his thesis [22].

Buffer pool sizing issue is also important since a poor size selection may result in unacceptable system performance. To obtain a good buffer pool size, DBAs typically tune little by little based on buffer pool hit rates until the system reaches acceptable performance. Obviously this way is time consuming and shifting this tuning task to DBMSs becomes crucial to ensuring continuous good performance. The buffer pool sizing problem has been shown to be a complex constrained optimization problem [17]. Chung et al. [17] and Martin et al. [19] once attempted to tackle this problem. Their approaches are both goal-oriented and use iterative algorithms to find a reallocation that benefits a target transaction class. A potential problem with the goal-oriented approach, as noted in [19], is that it requires DBAs to pre-define reasonable class-specific goals, which is a very difficult task. Another problem with the goal-oriented approach has to do with its prolonged time to find the suitable buffer allocation because of the iterative use of the sizing algorithm.

1.2 Objective

In this thesis we study the buffer pool sizing problem, that is, how to properly allocate the DBMS's buffers area among multiple buffer pools. Previous studies, such as Dynamic Tuning [17] and Dynamic Reconfiguration Algorithm (DRF) [19] are both goal-oriented buffer sizing algorithms and use an iterative approach to find a suitable sizing scheme. Different from Dynamic Tuning and DRF, this thesis investigates the possibility of DBMSs automatically finding the optimal buffer pool sizes without setting any transaction class goals and without using the iterative approach. The techniques developed here are applicable to any DBMS with multiple buffer pool configuration capabilities, such as DB2/UDB [20] and Oracle [21], and these techniques can be easily incorporated into the DBMS software. This work represents an important step forward towards autonomic DBMSs. Two main benefits can be derived by automatically tuning buffer pool sizes: simplified performance management and improved performance compared to manual tuning.

The objectives of this research are the following:

- To investigate the research issues of automatically sizing the multiple buffer pools in DBMSs and to formally define the problem.
- To study how the buffer pools are used by the database transactions and the dependency relationship among the multiple buffer pools.
- To establish cost models to evaluate the effectiveness of a specific selection of buffer pool sizes.
- To find an efficient way to solve the constrained buffer pool size optimization problem.

- To implement the approach and evaluate it with experiments.

1.3 Thesis organization

The rest of the thesis is organized as follows. Chapter 2 presents the background and previous work on DBMS buffer management. Chapter 3 formalizes the buffer pool sizing problem, presents the cost models for the problem, and discusses the complexity of the problem. Chapter 4 proposes a greedy sizing algorithm and discusses the considerations in implementing it. Chapter 5 demonstrates the experimental results. Finally, Chapter 6 summarizes the thesis and proposes ideas for future work.

Chapter 2 Background and Related Work

Our work in this thesis is part of a large set of research on automatic resource management for DBMSs. In Section 2.1, we present the background information in this area; specifically the AutoDBA project at Queen's University. The buffer area model is introduced in Section 2.2. In Section 2.3, we review the previous work on buffer pool management. Buffer pool sizing problem for DB2/UDB is presented in Section 2.4.

2.1 Automatic Resource Management in DBMSs

Traditionally, the database workload is well-known and predictable. The database resources are manually tuned to achieve peak performance. As DBMSs continue to expand into new application areas, for instance, On-Line Transaction Processing [5], On-Line Analytical Processing [6, 7], web e-Commerce [23] and so on, the complexity and the diversity of database workloads are increasing, which makes it often impossible to predict the workload that a DBMS will have to handle.

Furthermore, these nontraditional workloads require more complex query processing capacities. It is also impracticable to provide universally good performance by solely having a well-engineered product [11]. There is a clear need for a DBMS that can tune itself to ensure acceptable performance.

Today's commercial database systems all require database administrators (DBAs) to manually adjust "tuning knobs" to tune the system. Little help is provided by the DBMSs in choosing the settings of these parameters in an intelligent way, which means that DBAs must have significant understanding of DBMS components and relationships among many of the "tuning knobs" that are exposed to the application or user. With the increasing complexity of the DBMSs and their workloads, manually tuning the performance of a DBMS via direct adjustment of low-level system parameters is becoming an arduous task for DBAs [1, 11, 19]. Automating the tuning process is vitally important.

To achieve the goal of a self-managing DBMS, "auto-tuning" capabilities and "zero-admin" systems have been put on the research and development agenda as high priority topics [11]. Academic research has been conducted on related aspects, such as query optimizer hints [24], physical database design [25, 26], memory management [1, 16, 17, 18, 19], transaction routing [27], multiprogramming level [14, 15], and so on. Meanwhile, the world famous database providers, such as IBM, Microsoft, Sybase and Oracle, to name a few, have also been taking the initiative to help reduce complexity and improve quality of service through the advancement of self-managing capabilities within a database environment. Typically, Microsoft's

AutoAdmin [28] and IBM's SMART (Self Managing and Resource Tuning) [29] are projects deployed towards the self-administering DBMSs.

The Database System Research group at Queen's University has carried out research in the past few years to automate the process of configuring and tuning IBM DB2/UDB [20]. We have specifically investigated the problem of providing "no knobs operation" for DBMSs. A "no knobs" DBMS requires *self-tuning and self-configuring algorithms* for its resources, which analyze the performance of the system and suggest new resource allocations to improve the system's performance. AutoDBA is our current project for automatic resource management in DBMSs [30]. The project vision is to have a self-tuning DBMS that can automatically adapt to its environment in order to consistently deliver high performance.

Our research focuses on automating the process of configuring and tuning one of the most important resources in DBMS, namely the buffer pools. More specifically, we have been working on the goal-oriented buffer pool sizing [19] and the autonomic buffer pool clustering (that is, mapping database objects into buffer pools) [22]. However, until now we have not addressed the problem of optimally sizing buffer pools in a non-goal-oriented environment, such as the TPC-C.

2.2 DBMS Buffer Area Model

DBMSs use a buffer area to manage data blocks in memory. The buffer area (shown in Figure 2-1) is located in the System Global Area and is shared by all processes connected to a DBMS instance. The buffer area is used to hold copies of data blocks read from the data files comprising the database. The advantages of using a buffer area are that: it eliminates physical I/O on frequently accessed blocks; it

provides a fast access path to find block(s) in memory; and it helps to maintain concurrency control and multi-version consistency on blocks. To better describe and understand the importance of the DBMS buffer area and its management, we first introduce some basic DBMS concepts. We then present a buffer area model that can be used for any DBMS with the capability of multiple buffer pools.

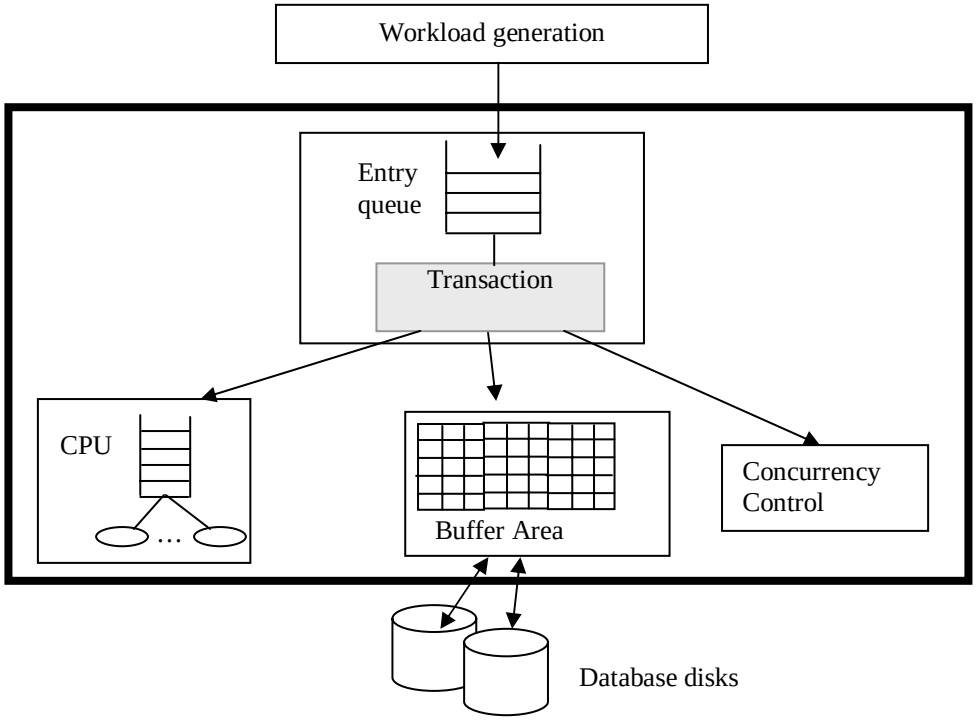

Figure 2- 1 A Transaction Processing Node

2.2.1 Terms and Definitions

The section briefly explains the entities and terms used in this thesis.

- **Container:** a container is general term used to describe the allocation of a physical space. It can be identified by a directory name, a device name, or a file name.

- **Database Objects**: database objects are used to denote the data tables and indices in a database system. They include user-controlled tables and indices as well as system-managed tables and indices. In our case, we do not consider system-managed objects, such as catalog tables or temporary tables.
- **Tablespace:** a database is organized into parts called tablespaces. Essentially, a tablespace is a physical location where tables and indices are stored. A database can contain multiple tablespaces. Once a tablespace is created, you can then create tables within the tablespace. A single tablespace can span many containers, but each container can belong to only one tablespace.
- **Disk Pages**: data in a database is stored and retrieved in units called disk pages or blocks. The disk page size is a parameter and is set up when a tablespace is created. Usually the page size is at 4KB, 8KB, 16KB or 32KB.
- **Buffer Pages**: the DBMS buffer area is organized into a sequence of pages or frames. The page size is normally the same as the disk page size.
- **Buffer Pools**: the buffer area in a DBMS is structured into buffer pool(s). A buffer pool is an array of fixed-size buffer frames, each of which is either empty or contains a disk page. The number of frames in the buffer pool is determined by the "user" at the initialization stage. Buffer pools are used to cache database disk pages. A buffer pool can be assigned to cache multiple database objects, but one database object can only be cached by single buffer pool.

2.2.2 Multiple Buffer Pool Model

The DBMS buffer area is organized into buffer pool(s). Each buffer pool consists of buffer pages of the same size as the disk page size. The database buffer manager is responsible for managing the I/O between the buffer pools and the disks. A data access issued by a transaction is called a *logical read*. For each logical read, the buffer manager first searches the buffer pool to see if the requested data page is in it. If it is then the data is returned to the transaction from the buffer pool immediately. However, if the required page is not in the buffer pool (called a page miss) the buffer manager must retrieve it from the disk, which is called a *physical read*. If there are no free pages (namely, empty buffer pages) to hold the new page then a "victim" must be selected for replacement. The portion of logical reads that require a physical read is called the buffer pool *miss rate*. The Buffer pool is an important resource for a DBMS since a physical access to a data page on disk is much more expensive than an access to database page in the buffer. Its size is also a key parameter that will affect database performance.

Traditionally the buffer in a DBMS is used as a single area (namely, one buffer pool), where all database objects (tables and indices) are cached. However, to better utilize the buffer area, vendors have provided the ability to split the buffer area into multiple buffer pools [31]. Instead of a single buffer pool, a sophisticated user can now configure the whole space as multiple buffer pools and assign database objects to the pools based on the pattern of usage.

In one sense, each buffer pool is like a private club. The DBA should try to organize a number of buffer pool "clubs" within the database so that the buffer pools

will be more effective. For example, in a database buffer pool, random data accesses conflict with sequential data accesses. By assigning conflicting objects to different buffer pools, we may greatly reduce the conflicts and hence improve the buffer area's efficiency. Furthermore, with multiple buffer pools, it would also be beneficial to create special purpose buffer pools that have very specific and well-defined objectives. More scalability would be achieved if you reduce the number of physical I/Os by caching frequently accessed data (such as the important indices) in a separate buffer pool. For example, suppose we have a database system with two transaction classes T1 and T2, sharing one buffer pool [17]. If we assume that class T1 has a high arrival rate and poor locality of reference, and assume that T2 has a lower arrival rate and smaller locality, then class T1 may monopolize the buffer space, which will make class T2 perform badly. If however we split the buffer space into two buffer pools, one dedicated to class T1, another to class T2, we may greatly improve class T2's performance with minimal impact on class T1.

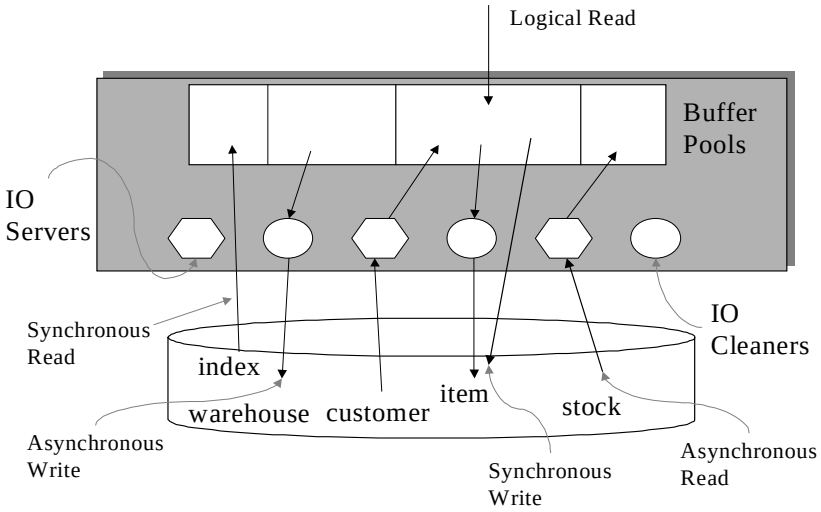

Figure 2- 2 Multiple Buffer Pool Model

The buffer area model used in our work is shown in Figure 2-2. It is similar to the structure of the buffer area in DB2/UDB [20]. The buffer area is partitioned into a number of independent buffer pools. Database objects (tables and indices) are assigned to specific buffer pools when the system is configured. For example, in Figure 2-2, indexes are assigned to the first buffer pool, the "warehouse" table is assigned to the second buffer pool, the "customer" and "item" tables are assigned to the third buffer pool and the "stock" table is assigned to the fourth buffer pool. An object's pages are moved between disk and its designated buffer pool. The size of each buffer pool is set by configuration parameters and page replacement is local to each buffer pool.

The performance of the buffer pool is enhanced by performing *asynchronous I/O*, which is system-initiated data transfer between disk and the buffer pools. *I/O servers* can perform prefetching where appropriate (called asynchronous read). *I/O cleaners* can asynchronously write dirty buffer pages back to disk (called asynchronous write).

The job of tuning multiple buffer pools includes the following tasks:

1. Determine the number of buffer pools and map database objects to the buffer pools.
2. Determine the optimal size of each buffer pool.
3. Determine the number of I/O servers and I/O cleaners.

This thesis focuses on the second task, which we call the *buffer pool sizing* problem. The number of buffer pools and the size of the buffer area remain fixed but

the buffer pool sizes are allowed to vary with the change of workload in order to maintain optimal system overall performance.

2.3 Related Work in Buffer Pool Management

In relational database management systems, the buffer manager is responsible for all the operations on buffers, including buffer assignment to queries, replacement decisions and buffer reads and writes in the event of page misses. Because the difference in speed between disk and memory accesses is large, the effectiveness of the buffer pool management algorithm is very important to the performance of the DBMS. Although the memory capacity of systems is constantly growing, memory management issues continue to be the focus of the database research community. Numerous studies have been devoted to static memory allocation and buffer management [24, 33, 34, 35, 36, 37]. A few recent papers address techniques that adapt to a dynamic environment [14, 15, 16, 17, 18, 19]

The global least recently used (LRU) buffer management algorithm [32] assumes no knowledge of the data access patterns. The hot set model [34] defines the notion of a hot set for nested-loop join operations and tries to pre-allocate a join's hot-set prior to its execution. This approach is culminated in the work of Chou and DeWitt [35], in which the concept of query locality set is introduced and the DBMIN algorithm is proposed. DBMIN is further extended in the Marginal Gain allocation [36] and later it is employed to perform predictive load control based on disk utilization [37]. A page Faulting Characteristic Model (FCM) is proposed by Chen et al. [38], and an adaptive database buffer allocation algorithm based on this model is implemented as well.

While each of these algorithms was designed to manage a multiclass workload, they suffer from focusing solely on *intra-query buffering*, that is, these algorithms consider the buffer pool as being divided up among several concurrently executing queries. Other queries are required to wait until enough memory is available. *Inter-query* buffering was investigated in [14, 15, 18] for a multiclass workload. Brown et al. [15, 16] proposed the fragment fencing algorithm for managing a multiclass workload to meet response-time goals of each specific class and investigated its performance. Fragment fencing's goal is to determine the minimum number of pages required for each fragment to ensure that a class meets its response time goal, which is called the fragment's *target residency*. Later on, to overcome the problems with the fragment fencing approach (as noted in [16]), Brown et al. proposed another improved goal-oriented algorithm, namely class fencing [18], which uses a more general hit rate prediction technique based on a notion that they call *hit rate concavity*.

So far, all the above buffer management algorithms, from the basic LRU algorithms to the more complex adaptive or dynamic buffer allocation algorithms work on a single buffer pool and are aimed at improving the efficiency of the single pool. The single buffer pool is shared among the concurrent queries and space is allocated when each new query arrives in. We therefore call the algorithms *Query-Oriented* buffer management algorithms [32]. None of the above algorithms deal with buffer management in a multiple buffer pool environment, where the buffer allocation is *Page Type-Oriented* [32], which means that the buffer area is allocated among the database objects, instead of the queries.

The concept of multiple buffer pools was first introduced in DB2 [31], and later used by Oracle [21]. To help users better use the multiple buffer pools, the product documentation provides some general rules of thumb for configuration of multiple buffer pools. By and large, these documented rules only focus on solving the buffer pool data mapping issue (that is, how many pools and how to assign database objects into pools). According to these rules, DBAs manually determine the number of pools and assign database objects into pools.

To automate the process of buffer pool data mapping, Xu [22] proposed a buffer pool clustering algorithm called *BPCluster*. BPCluster takes a data mining approach to group data objects into a specific buffer pool according to their similarities. Once the database objects have been assigned to buffer pools, an appropriate size must be chosen for each pool. Xu did not address the buffer pool sizing issue in his thesis.

The issue of buffer pool sizing has been tackled by several researchers [17, 19]. All centered on the goal-oriented on-line dynamic buffer pool size tuning for data requests originating from transactions. In the goal-oriented database system, the DBA specifies performance goals for each transaction class and the system decides how these goals should be achieved. The DBMS adjusts the configuration parameters in the correct direction to change the resource allocation in order to achieve the goals.

The Dynamic Tuning algorithm [17] monitors goal satisfaction of each buffer pool and periodically changes buffer pool sizes to improve goal satisfaction. The rationale is that the value of the performance index (*PI*) can be reduced by increasing the buffer pool size and can be increased by decreasing the buffer pool size. Here the buffer pool performance index is defined as

$$PI(SIZE)=RT(SIZE)/GOAL$$

where *RT* is random access response time for a buffer pool with size *SIZE*, and *GOAL* is user-defined goal response time. *RT* is assumed to be proportional to the buffer pool miss rate, and calculated by

$$RT(SIZE) \approx MissRate(SIZE) \times DELAY$$

where *DELAY* is the average time required for moving a page from disk to memory. If the requested page is in the buffer pool the *DELAY* is assumed to be negligible.

The Dynamic Reconfiguration Algorithm (DRF) [19] is a goal-oriented buffer pool self-tuning algorithm developed as a part of our project studying automatic resource management in DBMSs. The aim of the DRF is to provide an allocation of buffer pages to buffer pools such that the performance goals of the transaction classes using the database are met. The goal satisfaction in DRF is measured by the Achievement Index (*AI*), which is given by

$$AI = \frac{Goal\ Average\ Response\ Time}{Actual\ Average\ Response\ Time}$$

To meet the goals, DRF tunes the buffer pool sizes based on the assumption that the average response time for a transaction class is directly proportional to the average data access time for instances of the class, while the average data access time is a function of the buffer pool size, and is given by

$$C_i = \sum_{j=1}^{b} L_i(B_j) \times costLR_j(m)$$

where b is the number of buffer pools, $L_i(B_j)$ is the number of logical reads of buffer pool j by transaction class T_i, $costLR_j(m)$ is the average cost of a logical read from buffer pool B_j of size m.

The Dynamic Tuning Algorithm [17] and the DRF [19] both deal with the buffer pool sizing problem, but they still need manual intervention to set the transaction goals. P. Martin et al. [19] observed that choosing reasonable goals is a difficult task, which led them to wonder about the feasibility of requiring users to provide such goals. Another sizing approach totally without the manual interference is preferred.

Buffer pool sizing is a complex constrained optimization problem [17]. Choosing the correct size for each buffer pool is a complex task because of the number of parameters involved and the complexity of their interactions. It requires DBAs to have a detailed understanding of the nature of activities that compete for the resources related to buffer pool sizing. In reality, buffer sizing is a time consuming and complicated process when done manually. Even if an individual were to effectively solve the optimization problem, it is unlikely that this process would be re-evaluated frequently. The buffer sizes would therefore have to be picked to yield long-term optimal performance. However, the pattern of buffer pool requests exhibits significant variations from the long-term average. Therefore, a set of algorithms that can detect the dynamically changing workload characteristics and automate the process of buffer pool sizing will improve performance relative to a long term choice. In the next chapter, we continue to further talk about the buffer pool sizing problem.

Chapter 3 Problem Statement

The buffer pool sizing optimization problem is first introduced. We then present two cost models to formulate the optimization problem in Section 3.2. The complexity of solving the buffer pool sizing problem is finally examined in Section 3.3.

3.1 Formalization of the Buffer Pool Sizing Problem

To better understand the buffer pool sizing problem, we now define a model to formally describe it. Suppose a DBMS has a buffer area of size *M* pages and the area is divided into *N (N>1)* buffer pools, BP_1, BP_2, …, BP_N.

DEFINITION 1. (**Buffer Pool State**) We denote vector $V = < S_1, S_2, \ldots, S_N >$ to be a buffer pool state, where S_i is the size of BP_i in unit of buffer pages, $0 < S_i < M$ *and* $M = \sum_{i=1}^{N} Si$.

DEFINITION 2. (**Buffer Pool State Neighbour**) A neighbour of a buffer pool state is another buffer pool state. The neighbour of state $V = < S_1, S_2, \ldots S_i, \ldots, S_j, \ldots, S_N >$ is defined as $V' = < S_1, S_2, \ldots, S_i + \Delta, \ldots, S_j - \Delta, \ldots, S_N >$, where $1 \le i, j \le N$, $i \ne j$, and Δ is the user-defined minimum transferable number of buffer pages between two buffer

pools, such as 1, 100, or 1000 buffer pages. All the neighbours of buffer pool state V compose V's neighbour set N, which is defined as $N_V = \{V' | V' \text{ is a neighbor of buffer pool state } V\}$.

DEFINITION 2. (**Buffer Pool State Space**) All the buffer pool states comprise the **B**uffer **P**ool **S**tate **S**pace (*BPSS*). *BPSS={V| V is a buffer pool state}.*

DEFINITION 3. (**Buffer Pool State Cost**) Each state V has an associated cost *c(V)*. The cost is used to evaluate the effectiveness of a specific buffer pool state. The metric used for cost could be the number of page replacements, the number of disk I/Os, the average data access time, and so on.

DEFINITION 4. (**Optimal Buffer Pool State**) For a given workload, there exists a buffer pool state $\overline{V}$ ($\overline{V} \in BPSS$), such that

$$c(\overline{V}) = \min_{V \in BPSS} c(V)$$

we call $\overline{V}$ the optimal buffer pool state.

In other words, the optimal buffer pool state is the best buffer sizing scheme for a fixed-sized buffer area and a fixed number of buffer pools under a stable workload. It is expected that the optimal buffer pool state produces the best database performance (for example, maximum transaction throughput) for a given workload. Our goal in this work is to find the optimal buffer pool state $\overline{V} = < \overline{S}_1, \overline{S}_2, \ldots, \overline{S}_N >$ from the *BPSS*. This is a typical searching optimization problem.

To further formulate this optimization problem, a suitable cost function is required. The cost function is applied to calculate the cost of a specific buffer pool

state. Searching techniques then attempt to optimize the value of this cost function to locate the optimal buffer pool state.

3.2 Cost Functions

In DBMS buffer management, numerous approaches to evaluating the buffer allocation scheme have been proposed in the literature [16, 17, 18, 19]. Basically these approaches can be classified into two kinds. One is the page fault based approach for example, the approaches in [16, 17, 36], which use the number of page faults to evaluate a buffer allocation scheme, the other is the time based approach, which uses the data access time to evaluate a buffer allocation scheme. DRF [19] is a typical example of this kind.

3.2.1 Page Fault-Based Cost Function

A Page fault occurs when the requested data page is not found in the buffer pool. The frequency of occurrence of a page fault is represented by the buffer pool miss rate. The page fault is the main contributor to the data access time on a buffer pool. This is shown in the following equation. We denote buffer access time as *B*; miss rate as *MR*; disk access time as *D*. Due to the fact that accessing memory is thousands of times faster than accessing physical disks, we set $B=0.001*D$. Then, the effective buffer pool access time is calculated as:

$$E= (1-MR)*B+MR*D=0.999*MR*D \approx MR*D$$

Apparently, by lowering the buffer pool miss rate, we can decrease the effective buffer pool data access time. Previous research [32, 33, 34, 35] therefore has focused on minimizing the page faults (minimizing the miss rate) as the main goal of buffer

management. To achieve this goal, researchers paid much attention to analyzing the data reference patterns [35] and the buffer replacement algorithms [32, 33, 34]. Precisely speaking, these techniques are required to work in one buffer pool.

In a DBMS supporting multiple buffer pools, for example, DB2/UDB [20], the previous ideas [32, 33, 34, 35] are still applicable to each of the multiple buffer pools for reducing the number of page faults. However, none of these ideas was proposed for sizing multiple buffer pools to lower the system's overall page fault rate. Therefore, more work on the DBMS buffer management, especially the work on setting the relative buffer pool sizes appropriately to further reduce the number of page faults, is required.

For a given workload, the frequency of a page fault in a multiple buffer pool system can be calculated by averaging the miss rates of all the buffer pools. For a system with N buffer pools, the system's frequency of a page fault is denoted as weighted miss rate (*WMR*), which is defined by:

$$WMR = \sum_{i=1}^{N} (Wi \times MRi)$$

where W_i is the access weight of buffer pool i, which means how much of the data accesses goes to buffer pool i, and MR_i is the miss rate of BP_i, which is dependant on the buffer pool size S_i.

The above equation calculates the expected frequency of an occurrence of a page fault in the whole system. Similar to the case of single buffer pool, we assume the lower the *WMR*, the less the buffer data access time, which yields a better buffer pool sizing scheme. So for our aforementioned optimization problem, *WMR* is one option for measuring the cost of a buffer pool state.

3.2.2 Data Access Time (DAT) based Cost Function

For a transaction processing system, the largest component of transaction response time is typically the data access time (*DAT*). Lower data access time means faster transaction response. For the case of single buffer pool, the data access time for a transaction instance depends on the number of logical reads issued by that transaction. So an estimate of the average data access time of a transaction instance *T* is given by:

$$DAT_T = costLR_T \times noLR_T$$

where *costLR* is the average cost of a logical read from the buffer pool, and $noLR_T$ is the number of logical reads issued by transaction *T*, which is relatively fixed for a given workload.

In the page fault-based cost function, it was assumed that the data access time is caused by the page faults and is proportional to the number of page faults. This is true for a fixed-sized single buffer pool because all the data accesses go to the single buffer pool and the average cost of a logical read is fixed. For multiple buffer pools, data accesses distribute among different buffer pools. The different access patterns may cause the buffer pool *costLR* to vary among the multiple buffer pools even when the buffer pools have the same hit rate. The possible reasons are:

- The data objects cached by the buffer pools are from different containers and the multiple containers may have different speeds, which result in different elapsed times for a physical read.

- Buffer pools have different replacement frequencies. The buffer manager or the I/O cleaner(s) take different times to flush out the dirty pages, which cause the average read time to vary for different buffer pools.

If a multiple buffer pool configuration is employed in the transaction processing system and the application-related data objects are suitably mapped to the buffer pools, for the same workload and the same transaction instance *T*, $noLR_T$ is not changed, but it is distributed among these multiple buffer pools according to the buffer pool access weights. So for a multiple buffer pool configuration, the above data access time equation is translated into:

$$DAT_T = \sum_{i=1}^{N}((noLR_T \times Wi) \times costLRi)$$
$$= noLR_T \times \sum_{i=1}^{N}(W_i \times costLR_i)$$

where *N* is the number of buffer pools used in by the application, and $costLR_i$ is the average cost of a logical read from buffer pool BP_i. Note that $\sum_{i=1}^{N}(W_i \times costLR_i)$ gives the expected cost of a logical read from all buffer pools. We write it as "weighted *costLR*" (*WcostLR)*:

$$WcostLR = \sum_{i=1}^{N}(W_i \times costLR_i)$$

Noticeably, a lower *WcostLR* indicates less *DAT*, thus yielding faster transaction response. Therefore, WcostLR becomes another option for measuring the cost of a multiple buffer pool state.

3.3 The Search Space

By definition, a buffer pool state space (*BPSS*) contains all the possible buffer pool states. We form the buffer pool state search space by linking the buffer pool states with their neighbours. Figure 3-2 shows an example of a search space where the bold lines show a path from a given initial buffer pool state to an optimal buffer pool state. The graph looks like a spider web, where each node represents a buffer pool state and it is associated with a presumed cost of this buffer pool state. The size of the search space can be calculated trivially by:

$$SIZE_{SearchSpace} = \frac{(M/\Delta - 1)!}{(M/\Delta - N)!*(N-1)!}$$

where M is the size of the buffer area, for example, N is the number of buffer pools, and Δ is the user-defined transferable number of buffer pages between two buffer pools in order to form a new buffer pool state neighbour.

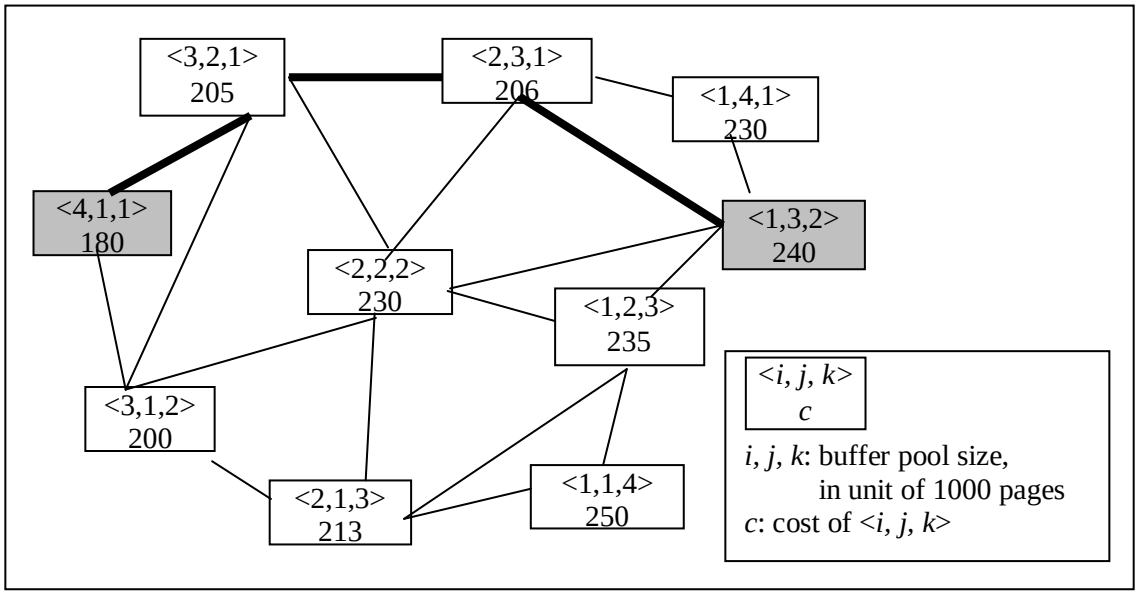

Figure 3- 1 Search Space (3BPs, Buffer Area = 6,000 Pages, Δ = 1,000 Pages)

The search space rapidly grows when the number of buffer pools is increased or if the buffer memory is expanded. Figure 3-2 and Figure 3-3 demonstrate the growth respectively. Figure 3-2 shows the relationship between the space size (in logarithm) and the size of buffer area when the number of buffer pools is fixed, Figure 3-3 shows the search space size (in logarithm) versus the number of buffer pools when the size of buffer area is fixed.

Currently, supporting multiple buffer pools is a new feature of modern DBMSs. More and more complex database applications are starting to attempt this improved feature. Moreover, a DBMS configured with several gigabytes of buffer area is commonplace. Therefore, the buffer pool state search space is really huge in reality. We do believe that the optimal buffer pool state lies somewhere in this search space. However, it is also an extremely difficult task to find the optimal buffer pool state from such a large search space. Exhaustive search, which checks every possible buffer pool state, is certainly impractical. This could result in a long interruption for the normal system operation or a worse system performance, which is intolerable for the on-line transaction processing applications. Therefore, a suitable heuristic approach is required to speed up the search. In the next chapter, we investigate two sizing heuristics for this purpose: a simulated annealing algorithm and a greedy algorithm.

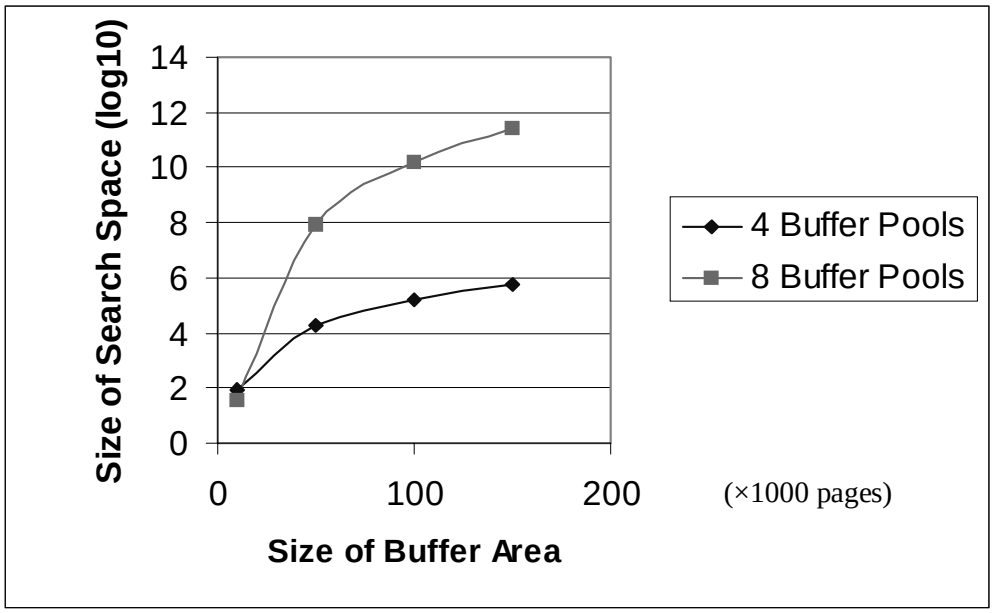

Figure 3- 2 Size of Search Space vs. Size of Buffer Area

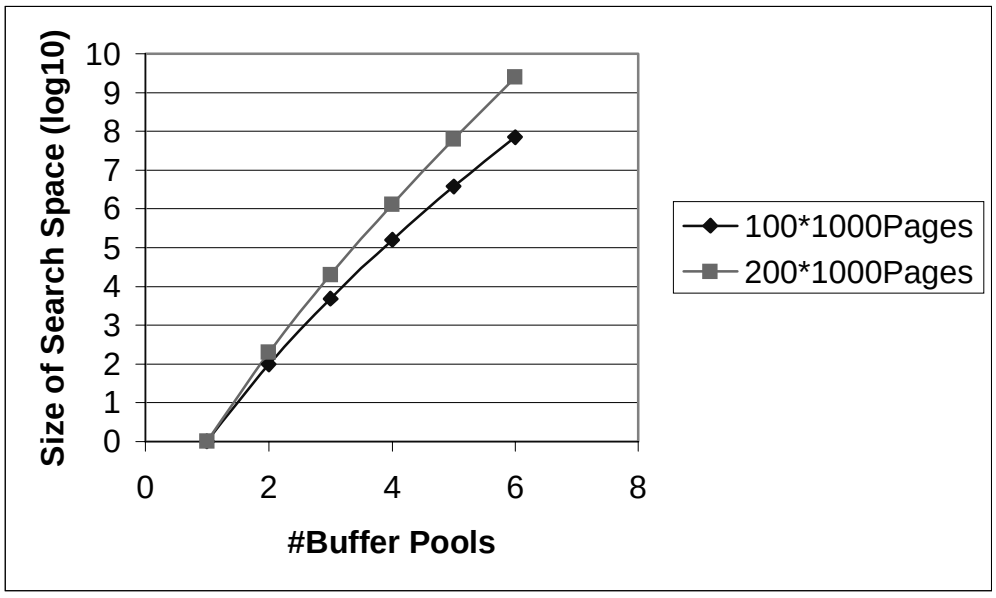

Figure 3- 3 Size of Search Space vs. Number of Buffer Pools

Chapter 4 Buffer Pool Sizing Algorithm

A greedy sizing algorithm is presented in Section 4.1 for solving the buffer pool sizing optimization problem. The buffer pool hit rate estimator and the DAT estimator, the components of the cost functions, are presented in Section 4.2. Finally, in Section 4.3 we introduce a DB2/UDB Buffer Pool Snapshot, which helps us implement the estimators.

4.1 A Greedy Algorithm for the Buffer Pool Sizing Problem

A greedy algorithm is a "single-minded" algorithm in the sense that it gobbles up all of its favorites first. It assumes that the path to the best optimum is a series of locally optimal steps and produces a good solution quickly by making the choice that looks best at the moment until a feasible set is constructed. Greedy algorithms have been used in many classic optimization problems such as the Traveling Salesman Problem (TSP) [40] and the Minimum Spanning Tree Problem [41, 42]. Figure 4.2 shows the algorithm for the buffer pool sizing optimization problem.

```
s:=initial;
found =false;
repeat
        get s's neigborSet  N_s;
         choose  s̄  from N_s, such that cost(s̄ )<cost(v), ∀v ∈ N_s
         if  cost(s̄ ) < cost(s)
                  s:= s̄ ;
         else
                  found:=true;
 until (found==true);
optimum:=s;
 return optimum;
```

Figure 4- 1 A Greedy Sizing Algorithm

The advantage of the greedy algorithm is its simplicity and straightforwardness. However, as a local optimization technique, it does not guarantee the optimal solution to all problems. We claim the following proposition in order to prove that the greedy algorithm in Figure 4-2 is sufficient to find an optimum for the buffer pool sizing optimization problem. A known feature of the buffer pool is the following assumption.

Assumption 4.1: *Adding pages to a buffer pool never degrades the performance of the buffer pool.*

Proposition 4.1: *Given a stable workload, if a buffer pool state is not an optimal state, there must exist at least one neighbour more optimal than it.*

Proof. We will do a proof by induction on the number of buffer pools.

Consider the case of N ($N>1$) buffer pools, BP_1, BP_2, …, BP_N. S_i represents the size of buffer pool BP_i ($1 \le i \le N$) and the total amount of buffer area $M = \sum_{i=1}^{N} S_i$.

Step 1: Let $N=2$

In this case, there are only two buffer pools, BP_1 and BP_2 in the system. If the current buffer pool state $V_2=<S_1, S_2>$ is not an optimal state, then a neighbour of V_2, which may be $V_2'=<S_1+\Delta, S_2-\Delta>$ or $V_2'=<S_1-\Delta, S_2+\Delta>$, must have a lower cost than V_2. Therefore, this justifies the above proposition when $N=2$.

Step 2: Let $N=k$ $(k>2)$,

In this case, the system is configured with k buffer pools, BP_1, BP_2,…, BP_k.. We assume the above proposition is true for $N=k$. This means if the current buffer pool state $V_k=<S_1, S_2, \ldots, S_k>$ is not an optimum then at least one neighbour state of V_k: $V_k'=<S_1, S_2,\ldots, S_i+\Delta,\ldots,S_j-\Delta,\ldots, S_k>$ $(1\leq i, j\leq k, i\neq j)$ must exist such that V_k' has a lower cost than V_k.

Step 3: Let $N=k+1$ $(k>2)$,

In this case, the system has $k+1$ buffer pools: BP_1, BP_2, …, BP_k, BP_{k+1}, and the current buffer pool state $V_{k+1}=<S_1, S_2, \ldots, S_k, S_{k+1}>$ is not an optimal state. By fixing one buffer pool size, say S_t $(1\leq t\leq k+1)$, the remaining buffer pool sizes, $S_1, S_2, \ldots,S_{t-1}, S_{t+1},\ldots, S_k, S_{k+1}$ consist in a buffer pool state $V_k=< S_1, S_2, \ldots,S_{t-1}, S_{t+1},\ldots, S_k, S_{k+1}>$ for k buffer pools: BP_1, BP_2, …, BP_{t-1}, BP_{t+1},…, BP_k, BP_{k+1}.

- If S_t is exactly an optimal size for BP_t, then

  $V_k=<S_1, S_2, \ldots, S_{t-1}, S_{t+1},\ldots, S_k, S_{k+1}>$ must be a non-optimal state if $V_{k+1}=<S_1, S_2, \ldots, S_k, S_{k+1}>$ is not an optimal state. According to the assumption in Step 2, there exists at least one neighbour of V_k, say $V_k'=< S_1, S_2, \ldots,S_i+\Delta, \ldots, S_{t-1}, S_{t+1},\ldots, S_j-\Delta, \ldots, S_k, S_{k+1}>$ $(1\leq i, j\leq k+1, i\neq j)$ more

optimal than V_k. Therefore, combined with S_t, buffer pool state V_{k+1}'=< S_1, S_2, …,$S_i+\Delta$, …, S_{t-1}, S_t, S_{t+1},…, $S_j-\Delta$, …, S_k, S_{k+1} > must exist and more optimal than the current state V_{k+1}=<S_1, S_2, …, S_k, S_{k+1}>.

- If S_t is not an optimal size for BP_t, then

  At least one buffer pool size in S_1, S_2, …, S_{t-1}, S_{t+1},…, S_k, S_{k+1}, say S_r ($1 \le r \le k+1$, $r \neq t$) is also not at its optimal size. In other words, buffer pool state V_2=<S_r, S_t> is not an optimal buffer pool state. According to the Step 1, a neighbour of V_2, that is V_2'=<$S_r+\Delta$, $S_t-\Delta$> *or* V_2'=<$S_r-\Delta$, $S_t+\Delta$> must be existing and more precise than V_2=<S_r, S_t>. Therefore, we say that buffer pool state V_{k+1}'=<S_1, S_2, …,$S_r+\Delta$, …,S_{t-1}, $S_t-\Delta$, S_{t+1},…, S_k, S_{k+1}> or V_{k+1}'=<S_1, S_2, …,$S_r-\Delta$, …,S_{t-1}, $S_t+\Delta$, S_{t+1},…, S_k, S_{k+1}>, which incorporates S_1, S_2, …, S_{t-1}, S_{t+1},…, S_k, S_{k+1}, must be existing and more precise than the current buffer pool state V_{k+1}=<S_1, S_2, …, S_k, S_{k+1}>.

By summing up the above two points, we see that Proposition 4.1 still holds for $k+1$ buffer pools.

Based on the discussion in Step 1, 2 and 3, we can finally conclude that Proposition 4.1 is true.

■

Corollary 4.1: *From any given initial buffer pool state, an optimal buffer pool state can be reached by applying the above greedy algorithm.*

If the initial buffer pool state is not an optimal buffer pool state, according to Proposition 4.1, a neighbour more optimal than it must exist. The link to that optimal neighbour is followed and then this neighbour becomes the new initial state. This

process is repeated until a state is reached for which no more optimal neighbour exists. The final state is an optimal buffer pool state. An example is shown in Figure 3-1, where we set <1, 3, 2> as the initial state, for example, and reach an optimal state <4, 1, 1> finally by following the bold lines between neighbours.

4.2 Cost Estimators

To realize the above sizing algorithm, we need to implement the cost function first in order to evaluate each specific buffer pool state. In the following sections, we discuss the methods to compute the aforementioned two cost functions: the page fault-based cost function and the DAT-based cost function. The symbols used in our cost estimation equations are summarized in the following table:

Symbol	Meaning
BP_i	The *i*-th buffer pool
S_i	Size of BP_i
N	The number of buffer pools
$noLR_i$	Number of logical reads on buffer pool *i*
$noPR_i$	Number of physical reads on buffer pool *i*
$noDataLR_i$	Number of data logical reads on buffer pool *i*
$noIndexLR_i$	Number of index logical reads on buffer pool *i*
$noDataPR_i$	Number of data physical reads on buffer pool *i*
$noIndexPR_i$	Number of index physical reads on buffer pool *i*
$readTime_i$	Data read time on buffer pool *i*
$HR_i(Si)$	Hit rate of buffer pool *i* with size S_i
$MR_i(Si)$	Miss rate of buffer pool *i* with size S_i
W_i	The access weight of buffer pool *i*
$costLR_i(Si)$	Cost of a logical read of buffer pool *i* with size S_i
$WHR(V)$	Weighted system hit rate of buffer pool state *V*
$WcostLR(V)$	Weighted System cost of a logical read of buffer pool state *V*

Table 4- 1 Symbols used in Cost Estimators

4.2.1 Hit Rate (HR) Estimator

A hit rate function is used to formulate the relationship between the different buffer pool sizes and the corresponding buffer pool hit rates. This is the foundation of realizing the page fault based cost function. Buffer hit rate prediction is a complex process and another topic in database research. Dan et al., [43] and Xi [44] propose analytical methods to estimate buffer hit rate. By characterizing the data access pattern and using stochastic theory, they predict the buffer hit probabilities for different buffer sizes. A curve-fitting method was employed in DRF [19], in which the curve is approached by a second-order polynomial using the least square approximation. In this thesis we are interested in on-line dynamic buffer pool tuning, so we choose a simple approach to hit rate prediction based on Belady's equation [45], which is from Belady's virtual memory study and has been used in Dynamic Tuning [17]. Belady's equation is simpler than the methods used by Dan et al. [43] and Xi [44] because the time-consuming trace analysis is unnecessary.

For buffer pool *BP* with size *S*, Belady models buffer pool hit rate function *HR(S)* as:

$$HR(S) = 1 - a \times S^{b} \tag{4.1}$$

where constants *a* and *b* are specific to a particular combination of workload and buffer page replacement policy. To compute *a* and *b*, hit rates from two different buffer allocations are collected and plugged into equation 4.1. For example, if hit rate $HR(S_1)$ and $HR(S_2)$ are collected for two different buffer sizes S_1 and S_2 for the same buffer pool, we can derive the equations for *a* and *b* by applying the logarithmic function to the previous equation 4.1. We have

$$b = \frac{\ln(1-h(S_2)) - \ln(1-h(S_1))}{\ln(S_2) - \ln(S_1)}$$

$$a = \frac{1-h(S_1)}{e^{b \times \ln(S_1)}}$$

With a specific *a* and *b*, we can then use equation 4.1 to predict the hit rate for any given buffer size.

4.2.2 Data Access Time (DAT) Estimator

An approach to estimating the cost of a logical read for a buffer pool is presented in DRF [17]. It assumes that the cost of a logical read is composed of several components:

- The CPU processing cost associated with a logical read (*cpuLR*).
- The delay added by IO servers performing asynchronous reads (*costAR*).
- The delay caused by IO cleaners performing asynchronous writes (*costAW*).
- The cost from logical reads that result in a physical read.
- The cost from logical reads that involve a physical write.

To compute the above components (except *cpuLR*), the least squares approximation curve fitting technique is used. For example, a first-order polynomial and least squares approximation is used to curve-fit the proportion of asynchronous reads and the proportion of synchronous writes to a buffer pool, a second-order polynomial and least squares approximation is used in buffer pool miss rate estimation, and so on. We observed that the more approximation is used, the more error will be accumulated in the final *costLR* estimation and the higher the probability

of deriving a non-optimal buffer pool state. As a result, we now present a simple approach that requires less approximation.

As we know, memory access is much faster that disk access, which makes it reasonable to assume that the data access time is caused solely by disk accesses. Therefore, the following equation should hold:

$$DAT = costPR * noPR = costLR * noLR \tag{4.2}$$

by converting (4.2), we obtain

$$costLR = \frac{noPR}{noLR} * costPR \tag{4.3}$$

where $\frac{noPR}{noLR}$ stands for the miss rate of the buffer pool, which is also equivalent to $1 - HR$. Replacing $\frac{noPR}{noLR}$ with (*1-HR*), we derive the following equation from Equation (4.3):

$$costLR = (1 - HR) * costPR \tag{4.4}$$

where *costPR* is the average time required to perform a physical read. For a specific database container (device), we can assume the cost to perform such a physical access is fixed (the same assumption was used in Dynamic Tuning [17]), namely, *costPR* is a constant. Therefore, theoretically speaking, Equation 4.4 represents a linear relationship (shown in Figure. 4-3) between buffer pool hit rate and buffer pool *costLR* of the form $f(x) = kx+c$, where *costLR* is *f*, *HR* is *x*, *-costPR* is *k* and *costPR* is *c*.

According to Equation 4.4, values of one *costLR* and one *HR* are required in order to derive the constant *costPR*,. By collecting the *costLR* and the *HR* from a buffer allocation, say S_1, and then plugging them back into Equation 4.4, we obtain

$$costPR = -k = c = \frac{costLR(S_1)}{HR(S_1)}$$

Therefore, the cost of a logical read for the buffer pool with size *S* can be estimated by:

$$costLR(S) = -\frac{costLR(S_1)}{HR(S_1)} \times HR(S) + costPR$$

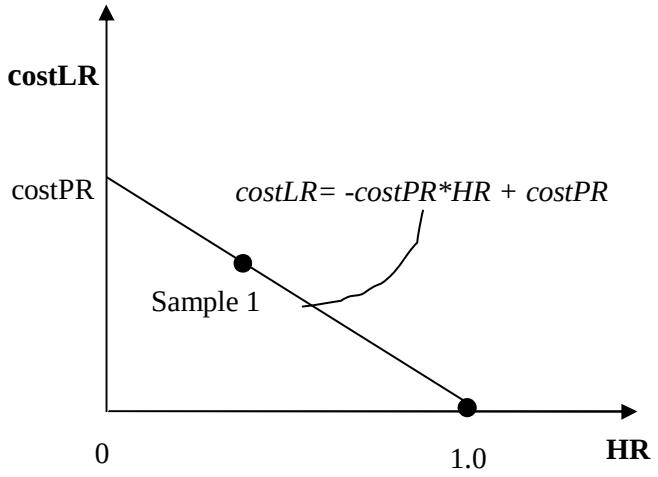

Figure 4- 2 Ideal Relationship of *costLR* vs. *HR*

An assumption is hidden in Equation 4.4; that is, the highest attainable buffer pool hit rate is 1.0. However in most cases, buffer pool hit rate barely reaches the highest hit rate of 1.0 unless the size is enlarged big enough to hold all associated database objects. Shown in the hit rate concavity theorem [18], the slope of the hit rate curve never increases as more memory is added. In other words, the hit rate curve

becomes flattened gradually with the increase of the buffer size. As Brown said, the slope of the hit rate curve represents the marginal increase in hit rate obtained by adding an additional page of memory. Thus it is useless to continue increasing buffer pool size after the curve has already flattened.

For this reason, we can not assume the buffer hit rate can reach 1.0 with increasing the buffer size. In other words, we can not assume the data access time will go down to *ZERO* with the increase of the buffer size. Moreover, some data reads involve physical writes (as shown in [17]), which make the read time definitely greater than *ZERO*. As a result, it will be inaccurate to use Coordinate (1.0, 0) to derive the linear line (the dot line in Figure 4-4). Alternatively, to get the accurate linear line, we can take one more sampling at another buffer size S_2. Combining with the one at S_1, we get

$$k = \frac{costLR(S_1) - costLR(S_2)}{HR(S_1) - HR(S_2)}$$

$$c = costLR(S_1) - k * HR(S_1)$$

The line drawn using Coordinate ($h(S_1)$, $costLR(S_1)$) and Coordinate($h(S_2)$, $costLR(S_2)$) is more accurate than the line drawn using Coordinate($h(S_1)$, $costLR(S_1)$) and Coordinate(1.0, 0) since Coordinate ($h(S_1)$, $costLR(S_1)$) and Coordinate ($h(S_2)$, $costLR(S_2)$) are both on the linear part of the curve.

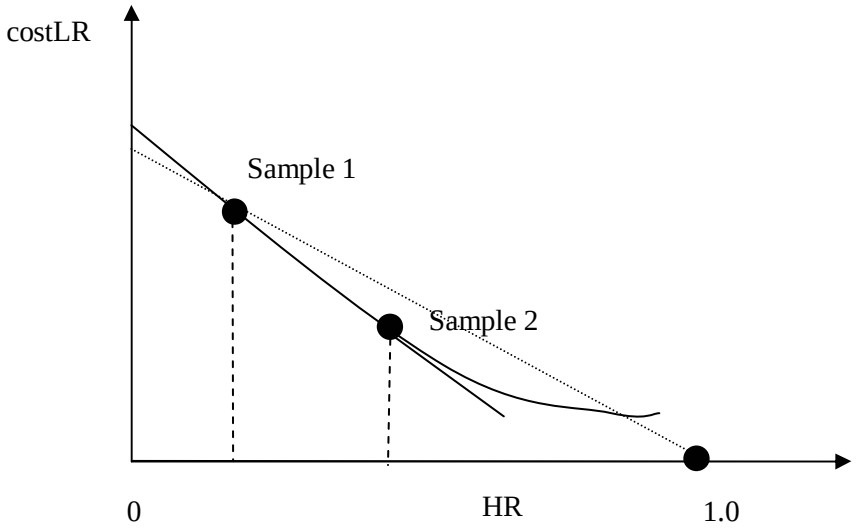

Figure 4- 3 Actual Relationship of *costLR* vs. *HR*

So far, we have discussed the implementations of two cost functions. If we integrate them into the sizing algorithm shown in Figure 4-1, we obtain two buffer sizing algorithms. One is uses the page fault-based cost function, which tries to minimize the system wide page faults, the other uses the DAT based cost function, which tries to minimize the time of a logical read. The comparison between these algorithms, that is, which one is more accurate in finding an optimal buffer pool state, is carried out in the next chapter.

Before testing the greedy algorithms in a real database system, we first look at the buffer pool information that can be acquired from the system and that is helpful to calculate buffer pool hit rate and buffer pool *costLR*.

4.3 A Buffer Pool Snapshot

In order to realize the above cost estimators, we need information to acquire buffer pool *HR* and *costLR* for the sampling buffer sizes and the information can be

extracted from the database system. DB2/UDB [20] is used as our testing system for buffer pool sizing, so information in this section is specific to DB2/UDB [20], however other systems, such as Oracle [21] also provide equivalent information.

DB2/UDB [20] provides a snapshot facility to get the I/O statistics called a buffer pool snapshot. The buffer pool snapshot is a picture of the current buffer pool state and aimed to help DBAs tune the buffer pools for an application. Figure 4-5 is an example of DB2/UDB buffer pool snapshot. This snapshot was taken on the DTPCC database, which was configured with 3 buffer pool, BP_D1, BP_D2 and BP_X.

```
             Bufferpool Snapshot
Bufferpool name                            = BP_D1
Database name                              = DTPCC
Database path                              = S:\DB2\NODE
Input database alias                       =
Buffer pool data logical reads             = 124048
Buffer pool data physical reads            = 0
Buffer pool data writes                    = 861
Buffer pool index logical reads            = 144870
Buffer pool index physical reads           = 0
Total buffer pool read time (ms)           = 0
Total buffer pool write time (ms)          = 15196
Asynchronous pool data page reads          = 0
Asynchronous pool data page writes         = 861
Buffer pool index writes                   = 0
Asynchronous pool index page reads         = 0
Asynchronous pool index page writes        = 0
Total elapsed asynchronous read time       = 0
Total elapsed asynchronous write time      = 15196
Asynchronous read requests                 = 0

             Bufferpool Snapshot
Bufferpool name                            = BP_D2
Database name                              = DTPCC
Database path                              = S:\DB2\NODEInput
database alias                             =
Buffer pool data logical reads             = 1531043
Buffer pool data physical reads            = 289991
Buffer pool data writes                    = 255094
Buffer pool index logical reads            = 0
Buffer pool index physical reads           = 0
Total buffer pool read time (ms)           = 13640401
Total buffer pool write time (ms)          = 11096640
Asynchronous pool data page reads          = 0
Asynchronous pool data page writes         = 251996
Buffer pool index writes                   = 0
Asynchronous pool index page reads         = 0
Asynchronous pool index page writes        = 0
Total elapsed asynchronous read time       = 0
Total elapsed asynchronous write time      = 10960130
Asynchronous read requests                 = 0

             Bufferpool Snapshot
Bufferpool name                            = BP_X
Database name                              = DTPCC
Database path                              = S:\DB2\NODE
Input database alias                       =
Buffer pool data logical reads             = 79
Buffer pool data physical reads            = 1
Buffer pool data writes                    = 4
Buffer pool index logical reads            = 4390775
Buffer pool index physical reads           = 97790
Total buffer pool read time (ms)           = 1033737
Total buffer pool write time (ms)          = 856396

......   ......

Asynchronous read requests                 = 0
```

Figure 4- 4 A DB2/UDB Buffer Pool Snapshot

By using information in the buffer pool snapshot, several useful values can be derived:

- **Buffer Pool Access Weight**

  Buffer pool access weight W_i denotes the percentage of system logical reads that go to buffer pool BP_i. It reflects how heavily BP_i is used by the transactions. For a stable workload, buffer pool weights are constant, even when buffer pool sizes vary. W_i can be calculated by the following equation:

$$Wi = \frac{noDataLR_i + noIndexLR_i}{\sum_{j=1}^{N}(noDataLR_j + noIndexLR_j)}$$

  where N is the number of buffer pools, $1 \leq i \leq N$.

- **Buffer Pool Hit Rate (HR)**

  Buffer pool hit rate HR_i is the percentage of all accesses on BP_i that are satisfied by the data in BP_i. A greater HR_i means a lower frequency of disk I/O issued from BP_i. Buffer pool hit rate for BP_i can be calculated as follows:

$$HR_i = 1 - \frac{noDataPR_i + noIndexPR_i}{noDataLR_i + noIndexLR_i}$$

  where N is the number of buffer pools, $1 \leq i \leq N$.

  Correspondingly, buffer pool miss rate of BP_i is obtained by

$$MR_i = 1 - HR_i$$

  By use of buffer pool access weights, the system weighted hit rate is given by:

$$WHR = \sum_{i=1}^{N}(W_i * HR_i)$$

where N is the number of buffer pools, $1 \leq i \leq N$.

WHR, taking into account all buffer pools, is the system's frequency of disk I/O. It reflects the effectiveness of the whole buffer area. A higher *WHR* means a lower probability that a DBMS need to access disks to bring pages into buffer pools.

- **Cost of a Logical Read (*costLR*)**

The cost of a logical read to buffer pool BP_i is calculated as:

$$costLR_i = readTime_i / (noDataLR_i + noIndexLR_i)$$

A lower *costLR* means less average time is spent on a logical read. *costLR* is dependant on buffer pool hit rate, which is shown in Section 4.4. As the hit rate increases, the page fault rate decreases, which yields a smaller elapsed time for a logical read. *costLR* is useful in identifying buffer pools with heavy physical I/O.

By taking all buffer pools into account, we can get the weighted system *costLR* (*WcostLR*) as follows:

$$WcostLR = \sum_{i=1}^{N} (W_i * costLR_i)$$

where N is the number of buffer pools, $1 \leq i \leq N$. *WcostLR* represents the expected cost (time) of a logical read across all buffer pools. A lower *WcostLR* means less time is spent by the transactions in data accesses.

Chapter 5 Experimental Evaluation

We introduce the experiment environment in Section 5.1 and the evaluation criteria in Section 5.2. In Section 5.3, we present the experiment scheme and results. Finally we discuss the problems in Section 5.4.

5.1 Experiment Environment

We used DB2/UDB Version 7.1 running on an IBM xSeries 240 PC server with the Windows NT 4.0 operating system in our experiments. The machine was equipped with two 1 GHZ PIII processors, 2GB of RAM and 22 disks over which the data was spread. The TPC-C OLTP benchmark [5] was used as the database transaction workload generator.

The database schema from the TPC-C benchmark is composed of nine relations. TPC-C simulates the activities of a wholesale supplier and includes 5 order-entry type transactions. The transactions include entering new orders (*new Order*), delivering orders (*Deliver*), checking the status of an order (*Order Status*), recording payments (*Payment*) and monitoring the level of stock at the warehouses (*Stock*). Clients, or simulated “terminal operators” issue the transactions against the database. The database schema and the relative transaction frequencies are presented in Appendix

A. Each TPC-C table or index was placed in its own table space in order to allow us to map each object to a buffer pool. Table 5-1 shows the tablespace names. The tablespaces for the data tables are ended with _d, and the tablespaces for the indices are ended with _i. The whole database was approximately 10GB in size (100 warehouses).

5.2 Evaluation Criteria

Since an optimal set of buffer pool sizes should lead to best overall performance of a DBMS, we choose to compare the impact of the configuration with the following system wide performance metrics:

- **Transaction Per Minute (TPM)**, is the number of New Order transactions completed per minute during the testing time. This is the ultimate throughput performance metric used in the TPC-C benchmark.
- **System Hit Rate,** the weighed buffer pool hit rate. The metric shows the percentage of page accesses that do not involve a disk access. Higher system hit rate means higher effective disk utilization.

5.3 Experimental Schemes and Evaluation

5.3.1 Experimental Schemes

Before starting buffer pool sizing, we have to decide on the number of buffer pools and the policy of mapping database object to buffer pools, which is referred to as the “buffer pool configuration problem”. *BPCluster* [22], an automated approach to solving the buffer configuration problem, has been proven better than a naïve configuration (random, default and distributed) and comparable with an expert

configuration. For the TPC-C workload, *BPCluster* suggests the buffer pool configurations shown in Table 5-1.

#BP	BP_NAME	TABLESPACE NAME
2 Buffer Pools	BP_D	tbs_wd, tbs_stock_d, tbs_item_d, tbs_cust_d, tbs_orders_d, tbs_neworder_d, tbs_hist_d, tbs_orline_d
	BP_X	tbs_stock_i, tbs_item_i, tbs_cust_i, tbs_orders_i, tbs_neworder_i, tbs_hist_i, tbs_orline_i
3 Buffer Pools	BP_D	tbs_wd, tbs_stock_d, tbs_item_d, tbs_cust_d, tbs_orders_d, tbs_neworder_d, tbs_hist_d, tbs_orline_d
	BP_X1	tbs_item_i, tbs_cust_i, tbs_orders_i, tbs_neworder_i, tbs_hist_i, tbs_orline_i,
	BP_X2	tbs_stock_i
4 Buffer Pools	BP_D1	tbs_wd, tbs_stock_d
	BP_D2	tbs_item_d, tbs_cust_d, tbs_orders_d, tbs_neworder_d, tbs_hist_d, tbs_orline_d
	BP_X1	tbs_stock_i, tbs_item_i, tbs_orline_i
	BP_X2	tbs_cust_i, tbs_orders_i, tbs_neworder_i, tbs_hist_i

Table 5- 1 Buffer Pool Configuration Suggested by BPCluster

We first validate the aforementioned linear relationship between buffer pool *costLR* and buffer pool hit rate. This is the theoretic foundation of our DAT-Based sizing algorithms. For this purpose, we carry out experiments with the case of one, two and three buffer pools respectively to verify its validity.

After verifying the validity of the linear assumption, we continue the experiments to see if the greedy sizing algorithms are capable of finding an optimal set of buffer pool sizes. We also compare the Page Fault-Based sizing algorithm with the DAT-Based sizing algorithm according to the evaluation criteria to see which one is better in finding the size optima. To fulfill these tasks, the following steps are carried out:

1. Run the TPCC driver at 2 different buffer pool states. Buffer pool sizes are randomly created, and the sizes for the same buffer pool in the two buffer pool states are different. The resulting buffer pool hit rates are therefore different and it is possible to derive cost functions. For each run, we collect buffer pool statistical data needed to calculate the *HR* function and the *costLR* function.
2. Compute buffer pool *HR* function and buffer pool *costLR* function for each buffer pool by use of the data collected in Step 1.
3. Run the sizing algorithms starting from a randomly-created initial buffer pool state and using the cost functions in Step 2. Two optimal buffer pool states are obtained, one is from the Page Fault-Based cost function, and the other from the DAT-Based cost function.
4. Run the TPC-C driver using the suggested optimal buffer pool states respectively, and collect the performance data (that is, TPM) and I/O data. Compare the "optimal" buffer pool states according to the TPM data and I/O data to see which is better.
5. Run the TPC-C driver under a variety of buffer pool states. Collect TPM data and the corresponding I/O data (logical read, physical read and total buffer read time). Draw out the performance curves, which are TPM vs. *WHR* and TPM vs. *WcostLR*. According to the curves, we can figure out at which buffer pool state the database gets the best performance.
6. Compare the TPM data that is collected in Step 4 with the curves in Step 5 to see if the "optimal" buffer pool state suggested by the sizing algorithms is really optimal.

In the above steps, each time we run the TPC-C driver for a specified buffer pool state, we let the driver run for 15 minutes so that the system becomes stable and then collect "snapshot" data every 5 minutes for 15 minutes. Each time before a new snapshot, we reset the buffer pool counters, so as to erase the effect of earlier transactions. We calculate buffer pool hit rates, buffer pool *costLR*s and TPM for each snapshot, and take the average of the snapshots to get the average buffer pool hit rates, the average buffer pool *costLR*s and the average TPM for a run to decide the system performance.

A confidence level (refer to Appendix B) of 95% is used in this thesis for our experiments. We can thus say, with 95% confidence, that the results we collected fall in the interval $\pm\varepsilon$ around the actual mean.

5.3.2 costLR vs. Hit Rate

We have shown analytically that the relationship between buffer pool *HR* and buffer pool *costLR* is linear. In this section, we further verify this point using experimental results. Three cases are tested, namely one, two and three buffer pool configurations. Table 5-1 shows the object-buffer pool mappings.

In the one buffer pool case, all database objects share a single buffer pool. The pool size is varied to study the relationship between buffer pool *HR* and buffer pool *costLR*. Table 5-2 shows the sizes and the corresponding *HR*s and *costLR*s. As the hit rate increases the cost of a logical read decreases in a linear fashion as shown in Figure 5-1. From this, we conclude that the linear relationship holds for one buffer pool under TPC-C workload.

Buffer Pool Size (4KB page)	10000	20000	30000	40000	50000	100000
Hit Rate	0.868038	0.8958678	0.912154	0.9238484	0.9323822	0.953883
costLR	3.939482	3.045049	2.48042	2.0668493	1.7594841	1.1256954

Table 5- 2 Hit Rate and *costLR* for One Buffer Pool

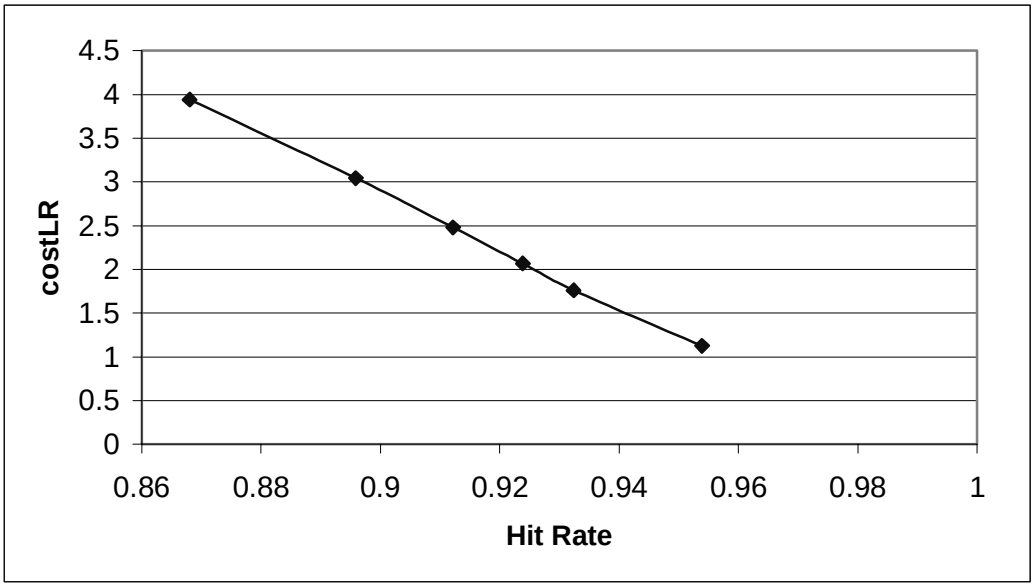

Figure 5- 1 Hit Rate vs. *costLR* (1 BP)

In the other two cases, we let the buffer pools share a buffer area size of 50,000 4KB pages and 100,000 4KB pages respectively. As we get the similar results, we only list the results with 50,000 4KB pages.

In two buffer pool case, data and indices are split into separate buffer pools. Data tables are assigned to buffer pool BP_D and indices are assigned to buffer pool BP_X. The size of BP_D is varied from 1000 4KB pages to 49,000 4KB pages, and the size of BP_X is correspondingly varied from 49,000 4KB pages to 1000 4KB pages to keep the total same.

The relationships between buffer pool *HR* and buffer pool *costLR* for each buffer pool are shown in Figure 5-2. We observe these two curves are close to linear lines.

We also see that neither of the two curves intercepts the x-axis because neither has a maximum hit rate of 1.0. The curve for BP_X is closer to the Coordinate (1.0, 0) because it has a higher reachable maximum hit rate than BP_D. This again tells us we can not assume Coordinate (1.0, 0) is on the linear curves of buffer pool *HR* vs. buffer pool *costLR*.

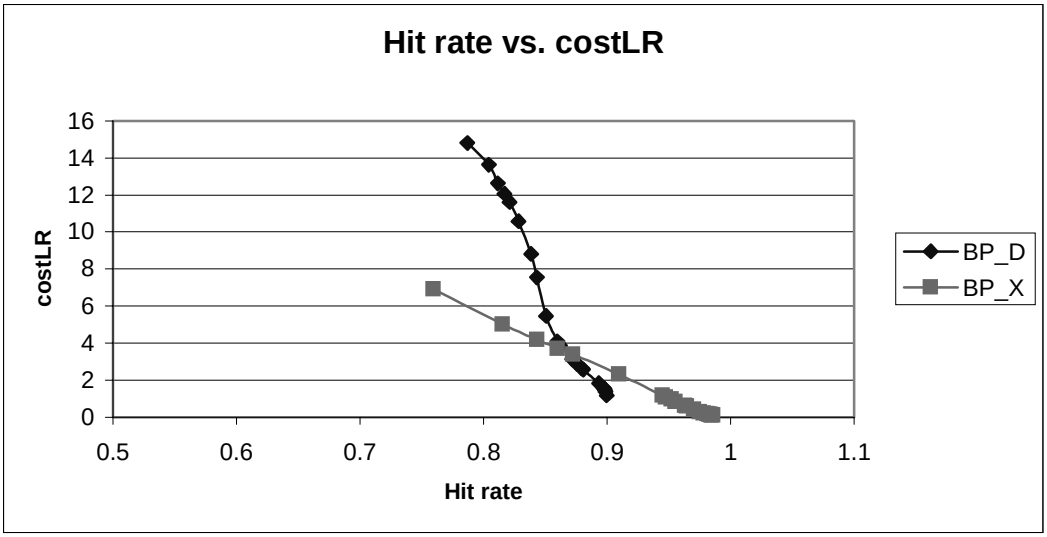

Figure 5- 2 Buffer Pool Hit Rate vs. costLR (2 BPs)

For the three buffer pool case, we fix the size of one buffer pool, and then vary the sizes of the other two buffer pools. Figure 5-3 shows the relationship for BP_D when we fix BP_X1, and Figure 5-4 and Figure 5-5 show the relationship for BP_X1 and BP_X2 respectively when we fix BP_D. Again these curves are basically linear in nature and support our approach using a linear function to estimate buffer pool *costLR*.

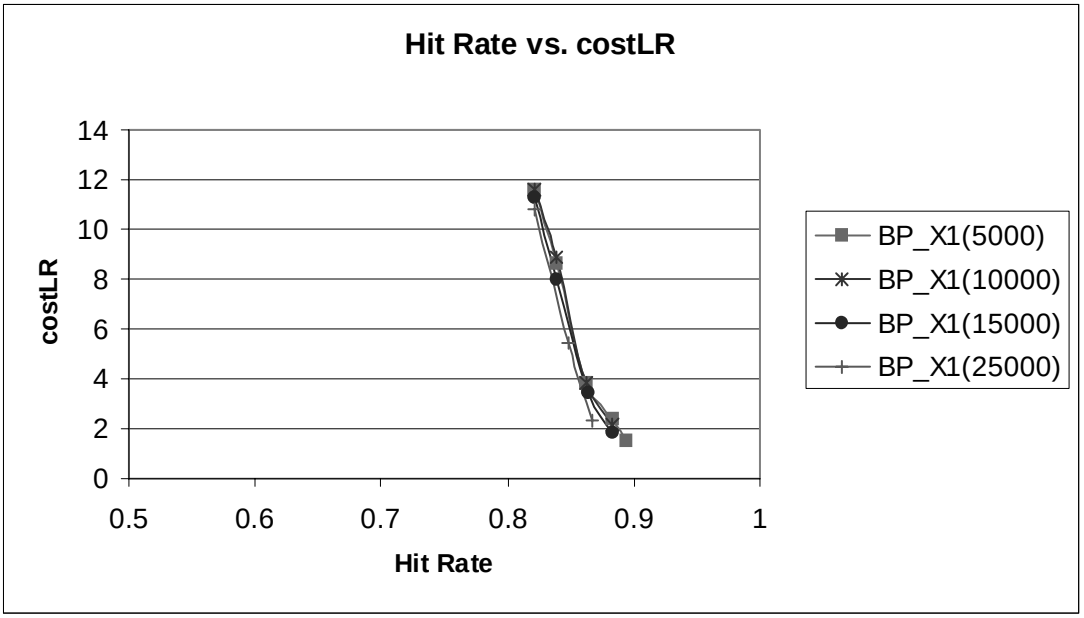

Figure 5- 3 Hit Rate vs. costLR for BP_D, BP_X1 is fixed

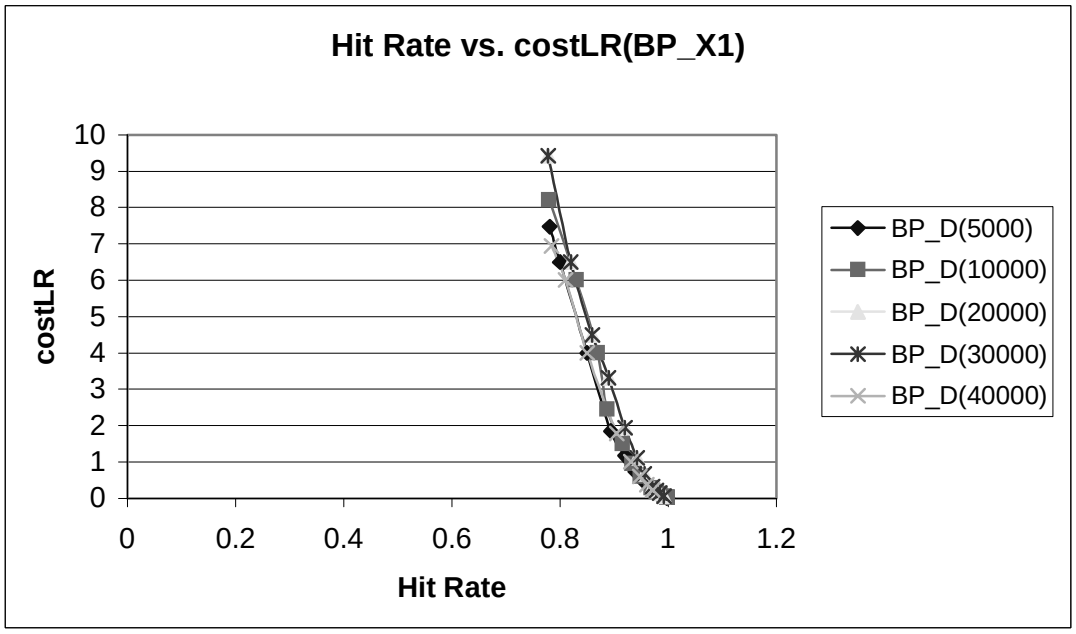

Figure 5- 4 Hit Rate vs. costLR for BP_X1, BP_D is fixed

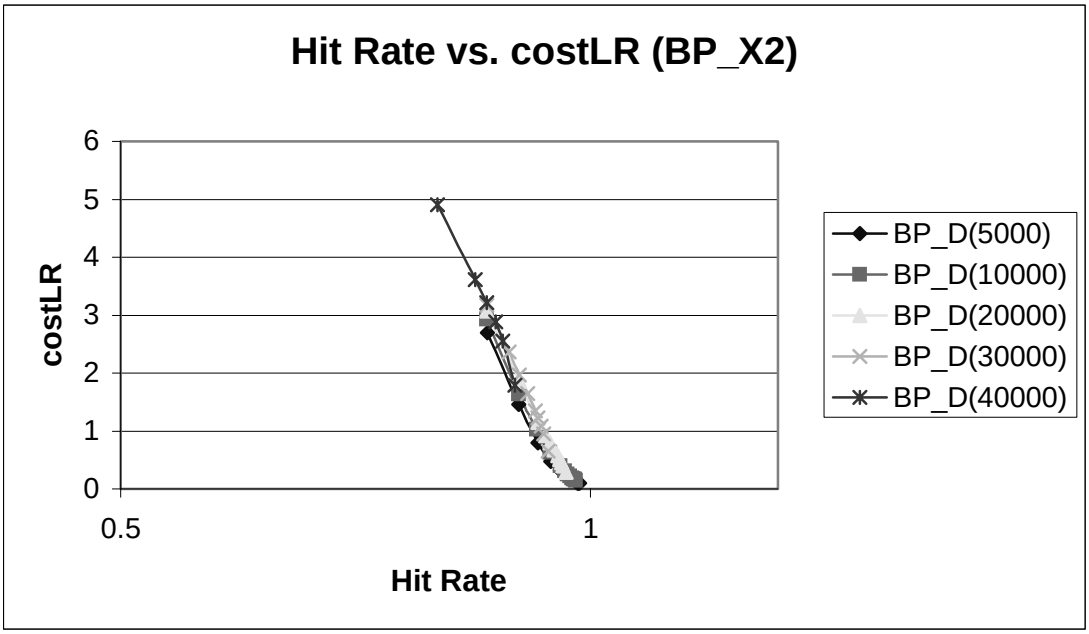

Figure 5- 5 Hit Rate vs. costLR for BP_X2, BP_D is fixed

5.3.3 Evaluation of the Sizing Algorithms

We now evaluate the effectiveness of the buffer sizing algorithms for three cases of multiple buffer pools. Only the results with a buffer area size of 50,000 4KB pages are presented below because we obtained the similar results when the buffer area was 100,000 4KB pages.

5.3.3.1 Two Buffer Pools

We randomly choose <10000, 40000> and <40000, 10000> as the sampling buffer pool states to initially derive the buffer pool hit rate function and the buffer pool *costLR* function. The greedy algorithm suggests the "optimal" buffer pool states shown in Table 5-3. Δ, the number of transferable buffer pages between buffer pools, is chosen at 1000 4KB pages because we found that smaller values do not make a significant performance difference with the buffer pool configurations in Table 5-1.

	DAT-Based	Page Fault-Based
Suggested Buffer Pool State	<29000, 21000>	<19000, 31000>
WHR	0.927361	0.933687
WcostLR	1.551909	1.613233
TPM	4556	4294

Table 5- 3 Suggested Optimal BP Sizes by Sizing Algorithms
(2 BPs, Buffer Area =50,000 4KB Pages, Δ=1000 4KB Pages)

From the above table we observe that DAT-Based greedy algorithm produces more optimal buffer pool sizes than Page Fault-Based greedy algorithm if the measurement is TPM. The suggested buffer pool state by Page Fault-Based greedy algorithm, however, produces higher *WHR*, which improves the effective disk utilization.

We now show that the buffer pool state suggested by DAT-Based algorithm is superior. We first run the system at many buffer pool states and plot the performance curves shown in Figure 5-6 and Figure 5-7. Figure 5-6 demonstrates how the TPM is influenced by the *WHR*. It shows that the maximum system hit rate does not yield the highest TPM. In other words, the system performance is not always growing monotonically with the increase of the *WHR*. On the contrary, shown in Figure 5-7, the TPM grows up gradually when the *WcostLR* decreases, which assures the assumption that the transaction response time is directly proportional to the average data access time [19]. Because the *costLR* of BP_D is more sensitive to the *HR* than that of BP_X (refer to Figure 5-2), a decrease of the *HR* of BP_X, which slightly increases the *HR* of BP_D, can decrease the *WHR*, but it can lower the *WcostLR* to

improve the system TPM. This is why the highest *WHR* does not result in the best TPM.

In Figure 5-6 and 5-7, the highest TPM, 4634, is achieved at buffer pool state <27000, 23000>, and the highest *WHR*, 0.937593, is achieved at buffer pool state<20000, 30000>. Comparing them with the results in Table 5-3, we find they are close. The DAT-Based TPM is 5.8% higher than the Page Fault-Based TPM. This demonstrates that the DAT-Based sizing algorithm produces a superior buffer pool state.

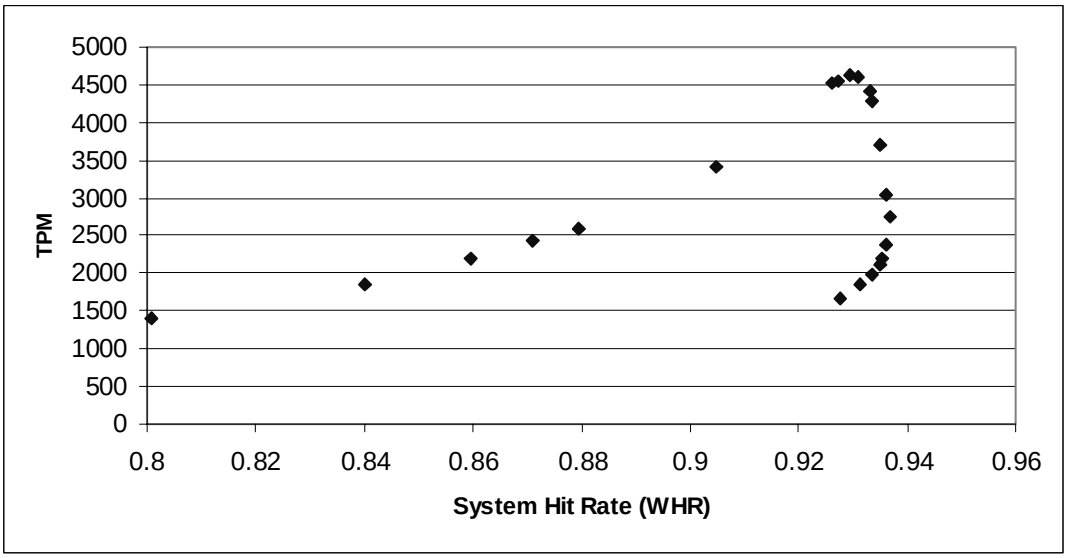

Figure 5- 6 System Hit Rate vs. TPM (2 BPs)

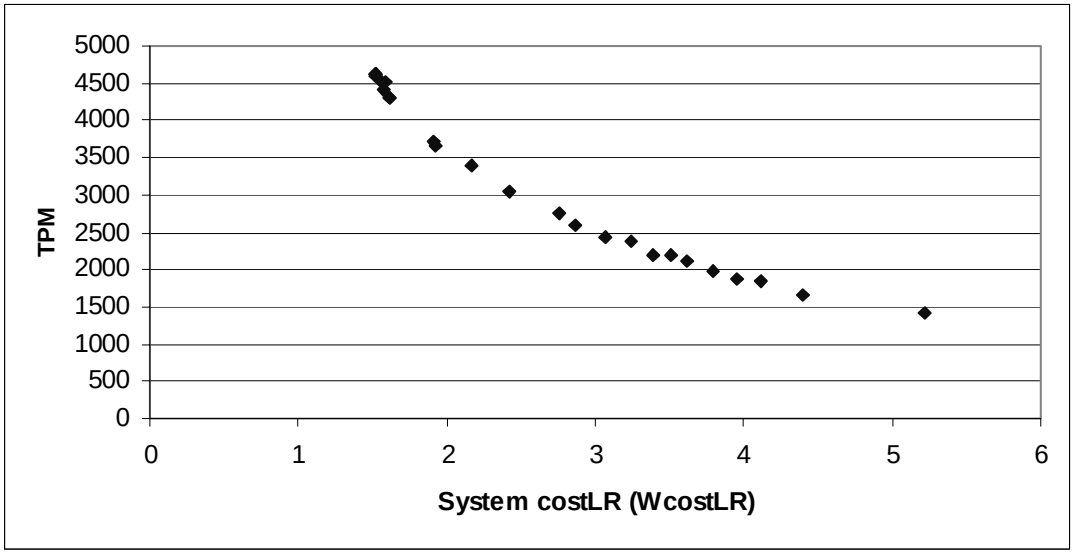

Figure 5- 7 System costLR vs. TPM (2 BPs)

5.3.3.2 Three Buffer Pools

Similar to the two buffer pool case, we start the sizing algorithm with two initial sample buffer pool states that are chosen randomly: <16000, 9000, 25000> and <30000, 5000, 15000>. The "optimal" buffer pool states suggested the greedy sizing algorithms are shown in Table 5-4, where the DAT-Based TPM is 4% higher than the Page Fault-Based TPM.

	DAT-Based	**Page Fault-Based**
Suggested Buffer Pool State	<25000,4000,21000>	<19000,5000,26000>
WHR	0.9307787	0.934229
WcostLR	1.5374819	1.563878
TPM	4493	4318

Table 5- 4 Suggested Optimal BP Sizes by Sizing Algorithms

(3 BPs, Buffer Area =50,000 4KB Pages, Δ=1000 4KB Pages)

To verify the validity of the sizing algorithms, we vary buffer pool sizes and plot the performance curves shown in Figure 5-8 and 5-9. Figure 5-8 shows how the

system performance (TPM) varies with the *WHR*. It is clear that the highest *WHR* does not correlate with the highest TPM. Figure 5-9 however shows that the lowest *WcostLR* is with the highest TPM. This again shows that the DAT-Based cost function is more appropriate for sizing the buffer pools to get better system performance.

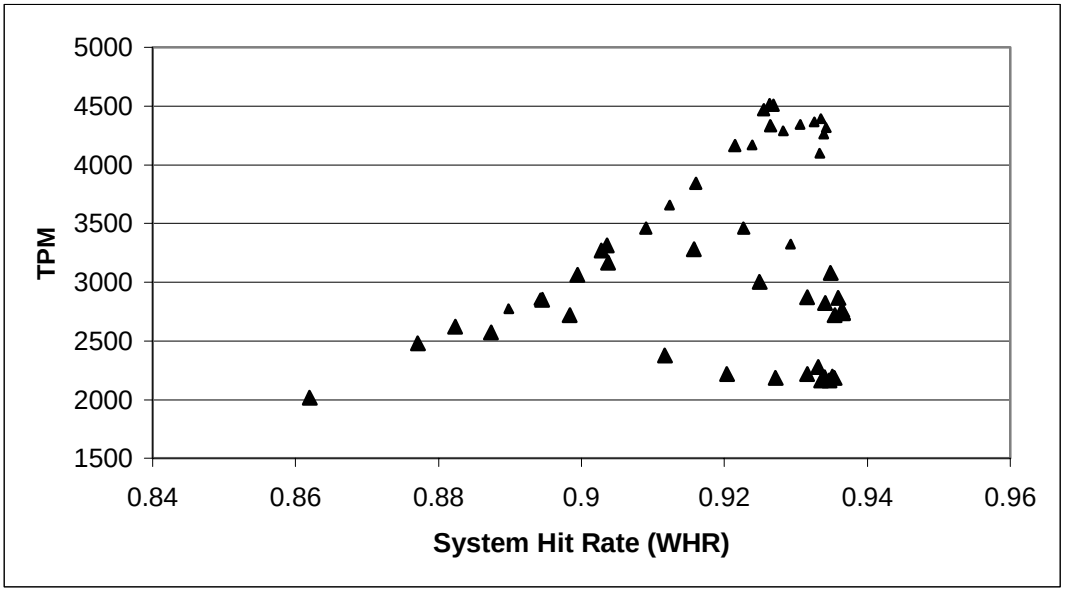

Figure 5- 8 System Hit Rate vs. TPM (3 BPs)

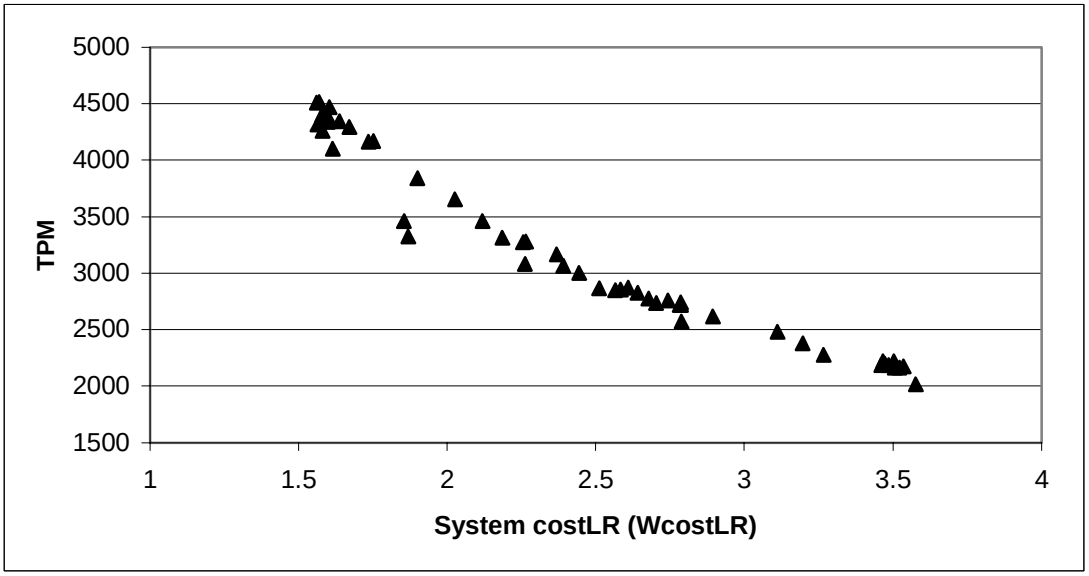

Figure 5- 9 System costLR vs. TPM (3 BPs)

In our experiments to draw the above curves, we obtain the highest TPM, 4667, at buffer pool state <25000, 6000, 19000> and the highest *WHR*, 0.93475, at buffer pool state <17000, 6000, 27000>. Both buffer pool states are very close to "optimal" states in Table 5-4, which are suggested by the sizing algorithms. This therefore qualifies our proposed approach to sizing the buffer pools.

5.3.3.3 Four Buffer Pools

We use a four buffer pool case to further validate our buffer pool sizing algorithms. In this case, the 50,000 4KB page buffer area is divided into 4 buffer pools BP_D1, BP_D2, BP_X1, BP_X2 with the data mapping given in Table 5-1.

We first derive the cost functions by running the system at two randomly chosen buffer pool states: <5000, 10000, 20000, 15000> and <10000, 20000, 10000, 10000>. After the sizing algorithms are applied, we get the suggested "optimal" buffer pool states shown in Table 5-5, where the DAT-Based TPM is 4% higher than Page Fault-Based TPM.

	DAT Based	**Page Fault Based**
Suggested Buffer Pool State	<4000,18000,20000,8000>	<4000,13000,22000,11000>
System Hit Rate	0.9380	0.9405
System costLR	1.2724	1.2844
TPM	4913	4722

Table 5- 5 Suggested Optimal BP Sizes by Sizing Algorithms
(4 BPs, Buffer Area = 50,000 4KB Pages, Δ=1000 4KB Pages)

As the previous cases, buffer pool sizes are varied and the related performance curves are drawn. Figure 5-10 shows the *WHR* vs. the TPM. Once again, the highest

system hit rate does not ensure the best system performance. The *WcostLR* vs. the TPM is displayed in Figure 5-11, where we see that a lower system costLR yields a better system performance. This again supports the claim that the *WcostLR* is a more appropriate measurement for optimally sizing multiple buffer pools.

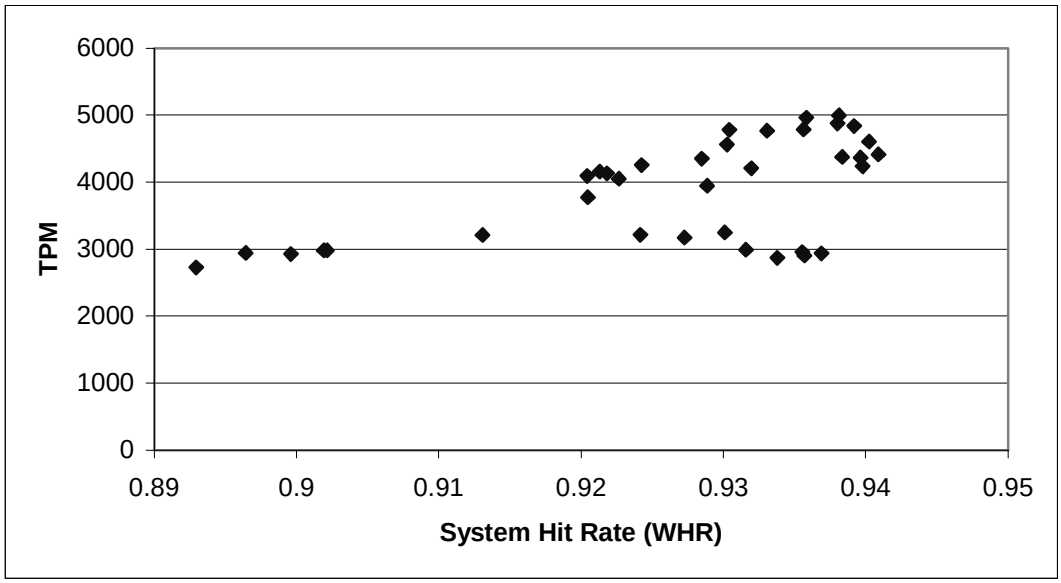

Figure 5- 10 System Hit Rate vs. TPM (4 BPs)

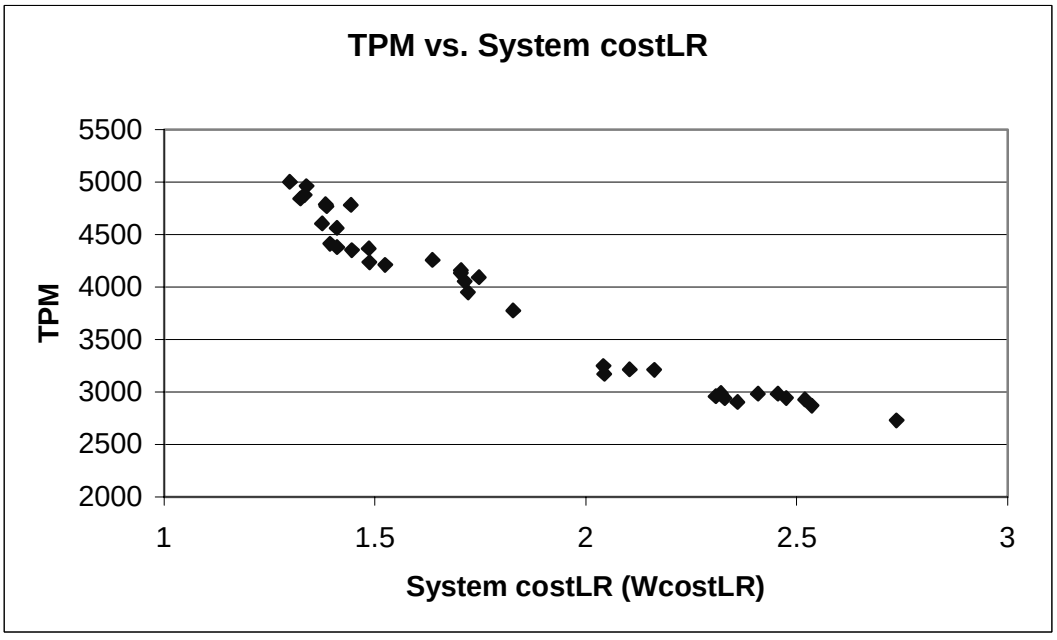

Figure 5- 11 System costLR vs. TPM (4BPs)

In the curves of Figures 5-10 and 5-11, we get the highest TPM, 4999, at buffer pool state <6000, 15000, 20000, 9000> and the highest *WHR*, 0.94089 at buffer pool state <4000, 10000, 25000, 11000>. By comparing them with the results in Table 5-5, we again claim that our sizing algorithms perform pretty well in finding an optimal buffer pool state.

5.3.4 Dynamic Buffer Tuning

A dynamic DBMS resource tuning algorithm is expected to have the capability of dealing with varying workloads. It should be able to sense workload changes and adaptively re-allocate the resources so that optimal system performance is still maintained.

Previous approaches to dealing with DBMS workload changes are to run the resource tuning algorithm once in a pre-defined period, called a tuning interval [17, 19]. During each tuning interval, related system statistics are collected and used by the tuning algorithms to propose new resource allocation schemes. The interval, as a parameter, is predetermined by the DBAs. With this approach, the tuning algorithm is run only when a real workload change occurs, in order to avoid unnecessary overhead. Therefore, a monitoring mechanism is required to detect workload changes. To build such a monitoring mechanism, we need to consider what is going to be monitored and how much the monitoring costs. For example, in goal oriented resource tuning algorithm Dynamic Tuning [17] and DRF [19], goal satisfaction and achievement index are used respectively for monitoring workload changes.

In the case of buffer pool sizing, if the algorithm is aimed at minimizing the system wide page faults, the *WHR* can be monitored. If the sizing algorithm is to minimize the data access time, then we can use the *WcostLR* as the indication of workload changes. This is reasonable because any workload changes (for example, the data access pattern, the data access intensity, as so on) would potentially alter either of them or both. Once a workload change is perceived, the buffer sizing algorithm is rerun to propose the new "optimal" buffer pool sizes.

To acquire buffer pool hit rate and buffer read time, DB2/UDB provides some monitoring APIs for this purpose. Periodically calling these in-built APIs would not cost as much as the tuning algorithms that are based on tracing and analyzing the database activities [22, 44].

5.3.4.1 TPC-C Workload Variant

We also tested our buffer pool sizing approach with a different workload. To achieve a different workload, we changed the TPC-C transaction mixture ratios, which results in different data access frequencies. The changes to the TPC-C transaction ratios are shown in Table 5-6.

Transaction	TPC-C Standard (%)	TPC-C Variant (%)
New Order	43	24
Payment	45	4
Order Status	4	24
Delivery	4	22
Stock Level	4	26

Table 5- 6 A TPC-C Workload Variant

We use the previous three buffer pool case to test the sizing algorithms with the TPC-C variant workload. First of all, we check the relationship between buffer pool

HR and buffer pool *costLR*. Figure 5-12 and Figure 5-13 demonstrate the curves, which again validates the linear assumption in *costLR* estimation.

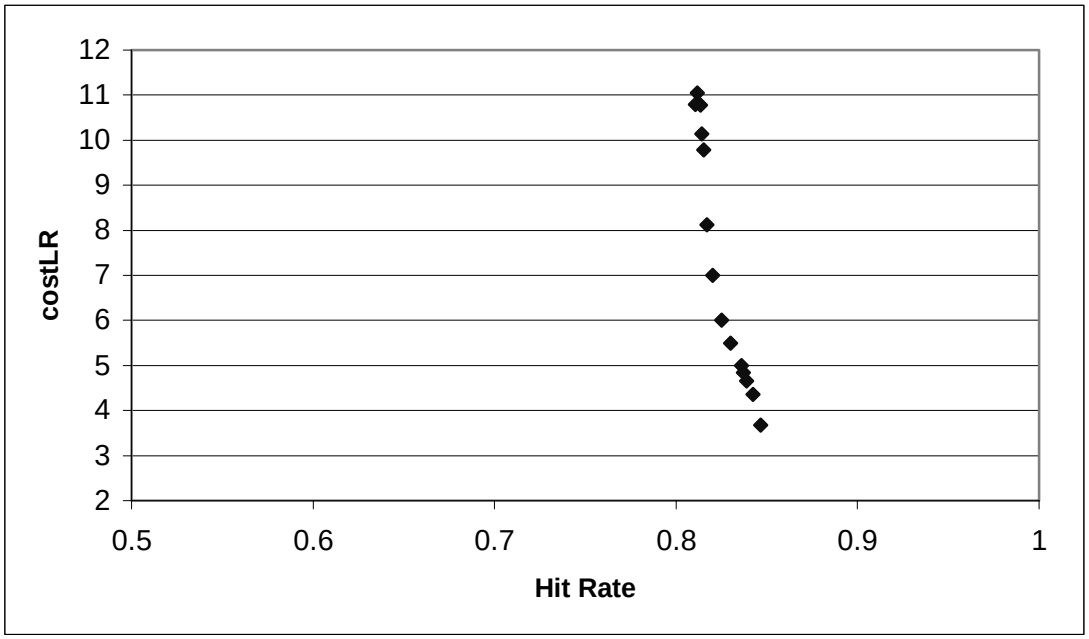

Figure 5- 12 Hit Rate vs. costLR (BP_D)

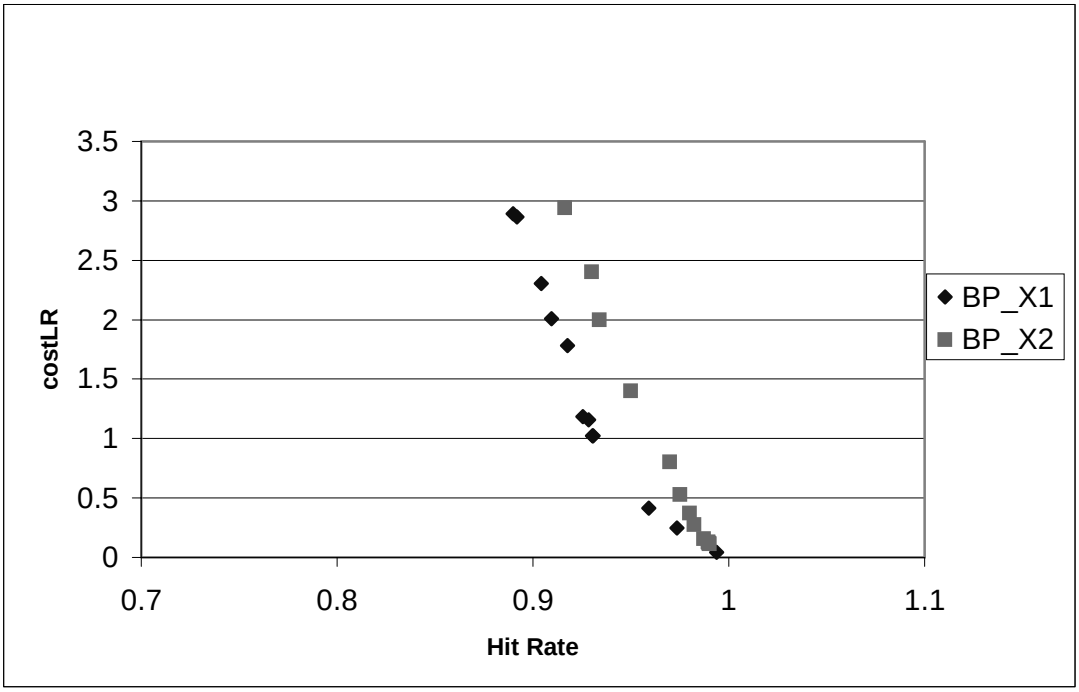

Figure 5- 13 Hit Rate vs. costLR (BP_X1 and BP_X2)

We provide two sampling buffer pool states <10000, 10000, 30000> and <20000, 20000, 10000> to the sizing algorithms, which give the "optimal" results shown in Table 5-7, where the DAT-Based TPM is 30% higher than the Page Fault-Based TPM.

	DAT-Based	**Page Fault-Based**
Suggested Buffer pool State	<20000,4000,26000>	<9000, 7000, 34000>
System Hit Rate	0.9595	0.9682
System costLR	0.9059	1.2780
TPM-C	2714	1896

Table 5- 7 Suggested Buffer Pool Sized by the Sizing Algorithms
(3 BPs, Buffer Area=50,000 4KB Pages, Δ=1000 4KB Pages)

We compare the results of the algorithms with the performance of system as shown in Figure 5-14 and 5-15. We see that the results generated by the greedy sizing algorithms are accurate.

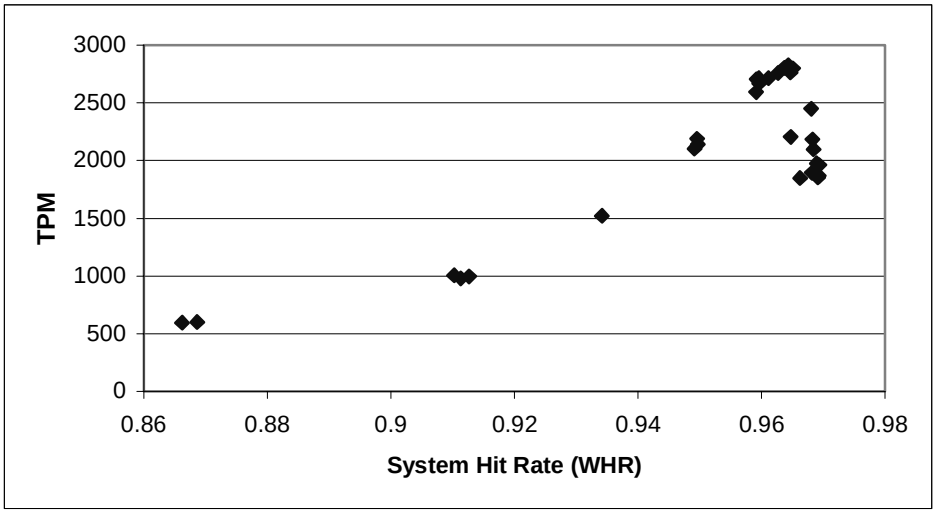

Figure 5- 14 System Hit Rate vs. TPM

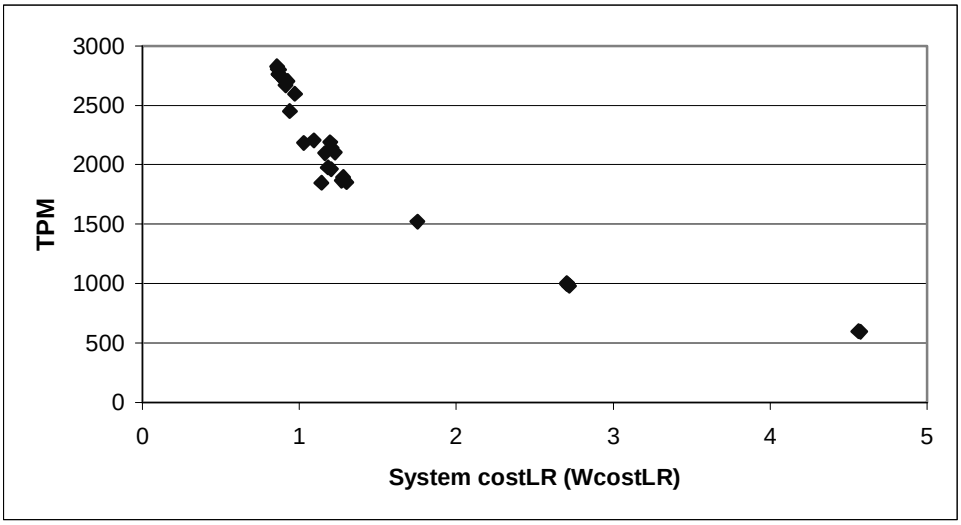

Figure 5- 15 System costLR vs. TPM

5.4 Discussion

We evaluated our proposed buffer pool sizing approach using 2, 3 and 4 buffer pools configurations. Experimental results show that the DAT based cost function is more suitable to perform the task of finding the "optimal" buffer pool state than the page fault based cost function if the system is assessed using the metric of TPM. The results also show that the greedy algorithm is able to efficiently locate an optimal buffer pool state.

However, some problems still exist with the algorithm. The first problem is the efficiency of the sizing algorithm. We observe that a larger Δ shortens the search procedure, so the algorithm converges quickly. In other words, a larger Δ improves the algorithm efficiency. A larger Δ, however, means the two buffer pool state neighbours are far apart, which may result in the real size optimum being missed. To improve the accuracy, we use a smaller Δ, but a smaller Δ causes a longer search procedure, thus lowering the algorithm efficiency. To resolve this conflict, we begin

with a larger Δ, and half its size whenever the algorithm can not proceed further until the Δ is reduced to a reasonably small value, such as 100 or 1000 4KB pages. Our previous experiments were executed using this method, which achieves a compromise between the search time and the accuracy.

The second problem lies in choosing the sampling buffer pool states. We observed that it is very important to start the greedy algorithm with two good sampling buffer pool states if we hope to efficiently reach an optimal set of buffer pool sizes. If an inappropriate sampling state is chosen, the suggested "optimal" buffer pool state will be wrong. If a buffer pool size in a sampling buffer pool state is chosen too large, for example, we cannot establish the correct linear relationship of buffer pool *HR* and buffer pool *costLR*. The reason is as follows.

We saw previouly that the curves of buffer pool *HR* vs. buffer pool *costLR* level off when the hit rates reach some point. An example is shown in Figure 5-16, which is very simlar to Brown's Concavity Theorem [18]. In the example, the linear relationship holds between Point P_1 and Point P_2, but beyond Point P_1 the curve flatens. Samples should be taken before Point P_1. If a size is taken at Point P_1', where the buffer pool is oversized and the linear line drawn between Point P_1' and Point P_2 will not match the actual linear line between Point P_1 and Point P_2. Using this incorrect line to estimate the buffer pool *costLR* for a given *HR* will underestimate the *costLR*s for some hit rates (for example, for hit rate *h3*), or overestimate the *costLR*s for some hit rates (for example, for hit rate *h4*). These incorrect estimations will eventually lead the sizing algorithm to produce an inaccurate "optimal" buffer pool size configuration.

Answers to the questions how we know whether or not a buffer pool is oversized and where the curve starts flattened are beyond the scope of our research. Instead of choosing the good sampling buffer pool states, we adopted the iterative approach to overcome this problem. Once we get a new "optimal" buffer pool state from the sizing algorithm, we run the system at this new buffer pool state and compare the produced results with the results calculated from the two previous sampling states to see if they are close or not. If they are so different, that means one of the sampling states is mistakenly taken. By replacing the wrong sampling state with the new buffer pool state, we can derive the next "optimal" buffer pool state. This approach looks like the method used in [18], where the curve is instead for the buffer hit rate. From the experiments we see that two or three reiterations are enough to approach the real size optimum.

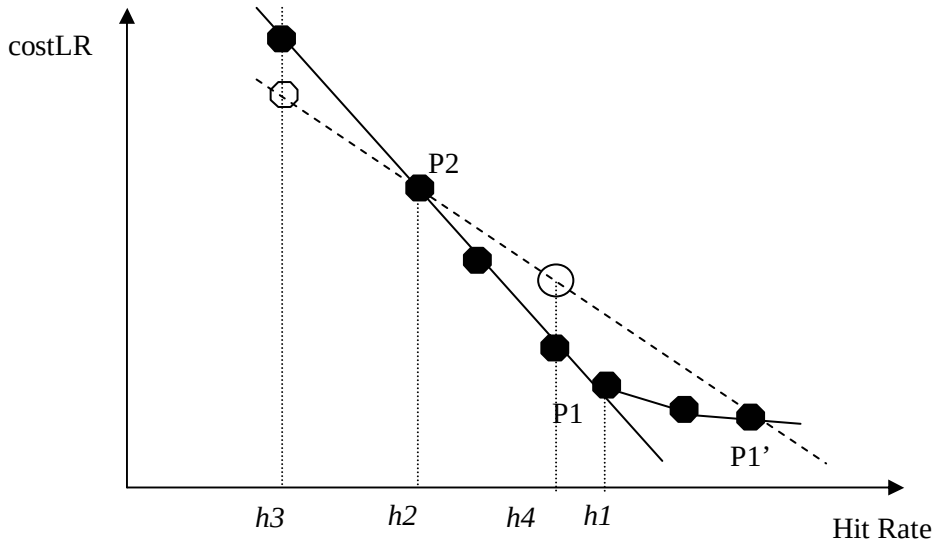

Figure 5- 16 An Example of an Incorrect Sampling BP Size

Chapter 6 Conclusions

"No knobs operation" enables a DBMS to automatically reallocate its resources to maintain acceptable performance in the face of changing conditions. Tuning buffer pool size is crucial to achieving good performance for a DBMS, but this tuning is a complex optimization problem and manual tuning is time consuming. Additionally, manual tuning is not responsive to dynamic workloads. In this thesis we present a method to shift this tuning task to the DBMSs themselves and obviate the need for human intervention. We present contributions of this thesis in Section 6.1 and then conclusions in Section 6.2. Future work is introduced in Section 6.3.

6.1 Thesis contributions

We present the buffer pool sizing problem for DBMSs and outline the requirements for automatic resource management. A heuristic algorithm is proposed for solving the buffer pool sizing problem. Different from previous approaches, a refined, easy-to-use cost model is employed in the algorithm to evaluate a multiple buffer pool size configuration. The subsequent experiments further illustrate that this

is an effective approach. By use of this approach, we can make the DBMS self-tune its buffer pool sizes.

The contributions of the thesis are as follows:

- A formal description of buffer pool sizing problem.

  We described the importance of sizing multiple buffer pools to a DBMS's performance, built a formal optimization model for describing this sizing problem, and analyzed the complexity of finding an optimal size configuration.

- Cost functions to evaluate a buffer pool state.

  In solving the sizing optimization problem, two cost functions to evaluate buffer pool states are proposed. One is to evaluate the system page faults, and the other one is to evaluate the cost of a logical read.

- An approach to estimating the cost of performing a logical read for a buffer pool.

  The approach assumes the relationship between the cost of performing a logical read and the buffer pool hit rate is linear and our experiments support this assumption. We get the linear function with the help of two sampling data. This approach is simpler than that used in DRF [19].

- A greedy algorithm for solving the problem of buffer pool sizing.

  We introduce a greedy algorithm to find an optimal set of buffer pool sizes. We prove that this greedy algorithm is sufficient to find an optimal set of buffer pool sizes. The subsequent experiments verify this result.

6.2 Conclusions

Based on our experiments and investigations, we can draw the following conclusions:

- The cost function based on the cost of performing a logical read is more accurate than the page fault based cost model for sizing multiple buffer pools.

  Optimizing global (system) hit rate is therefore not the best goal for a sizing algorithm. The experimental results showed that a higher global hit rate does not yield a better system performance, but a lower system *costLR* does.

- The proposed approach to estimating the cost of a logical read is much simpler than the goal-oriented approach used by DRF [19].

  The proposed approach in this thesis only requires buffer pool hit rate to estimate *costLR* while DRF has to consider more components.

6.3 Future work

The work presented in this thesis leaves some interesting topics for future exploration:

- A better buffer pool hit rate estimation method is required.

  Belady's equation is not designed for buffer pool hit rate estimation and it is not always effective for some kinds of workload [18].

- We would like to integrate our approach into the DBMS to simplify the process of buffer pool size tuning.

  The current version of DB2/UDB [46] supports dynamically changing the pool sizes, but it does not support automatic size tuning as the workload varies. We are interested in integrating the proposed sizing algorithm into the DBMS to make the DBMS self-tuning.

- DAT-Based cost function does not consider the impact of data writes.

From the experimental results, we find that some buffer pools are used for frequent updates by the transactions. Ignoring the data writes renders the cost model inaccurate. The linear relationship between buffer pool hit rate and buffer pool *costLR* does not hold when the buffer pool experiences heavy write activities because data access time now is caused mainly by data writes, not data reads. Therefore, incorporating the effect of data writes into the cost function for some buffer pools is required.

References

1. G. Weikum, C. Hasse, A. Moenkeberg and P. Zabback, The COMFORT Automatic Tuning Project, *Information System Vol. 19 No. 5*, 1994.
2. S. Hayes, The IDUG Solutions Journal, Spring 2000, Volume 7, Number 1, http://www.idug.org/member/journal/mar00/optimize.cfm.
3. C. Mullins, A DB2 for z/OS Performance Roadmap, eServer Magazine, Mainframe Edition, http://www.dbazine.com/mullins_db2-perf.html.
4. Y. An and P. Shum, DB2® Tuning Tips for OLTP Applications, *DB2 Universal Database Performance & Advanced Technology, IBM Toronto Lab, IBM Canada,* Jul. 2001.
5. Transaction Processing Performance Council. Benchmark descriptions, http://www.tpc.org/tpcc/.
6. Transaction Processing Performance Council. Benchmark descriptions, http://www.tpc.org/tpch/.
7. S. Chaudhuri and U. Dayal, An Overview of Data Warehousing and OLAP Technology, *ACM SIGMOD Record 26(1)*, March 1997.
8. M. Chen, J. Han and P. Yu, Data mining: An Overview from Database Perspective, *IEEE Transaction On Knowledge and Data Engineering, 8(6),* Dec. 1996.

9. T. Rakow, E. J. Neuhold and M. Lohr, Multimedia Database Systems – The Notions and Issues, in G. Lausen, editor, Tagungsband GI-Fachtagung Datebanksysteme in Buro, Technik und Wissenschaft (BTW), Dresden Marz 1995, pp. 1~29, Springer Verlag, Informatik Aktuell.
10. P. Bernstein, M. Brodie and S.Ceri, et al., The Asilomar Report on Database Research, *ACM SIGMOD Record, 27(4),* Dec. 1998.
11. S. Chaudhuri, G. Weikum: Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. *Proc. of 26th Int. Conf. on Very Large Databases*, Cairo, Egypt, Sept. 2000.
12. P. Scheuermann, G. Weikum and P. Zabback. Automatic Tuning of Data Placement and Load Balancing in Disk Arrays. Database *Systems for Next Generation Applications: Principles and Practice*, 1991.
13. K. Brown, M. Carey, and D. DeWitt et al., Resource Allocation and Scheduling for Mixed Database Workloads, Computer Sciences Technical Report #1095, Department of Computer Sciences, University of Wisconsin, Madison, July 1992.
14. K. Brown, M. Carey and M. Linvy, Towards an Autopilot in the DBMS Performance Cockpit, *High Performance Transaction Systems Workshop*, 1993.
15. K. Brown, M. Mehta, M. Carey and M. Livny, "Towards Automated Performance Tuning for Complex Workloads, *Proc. of 20th Int. Conf. on Very Large Databases*, Santiago, Chile, Sept. 1994.

16. K. Brown, M. Carey and M. Livny, Managing Memory to Meet Multiclass Workload Response Time Goals, *Proc. of 19th Int. Conf. on Very Large Databases*, Dublin, Ireland, Aug. 1993.
17. J. Y. Chung, D. Ferguson and G. Wang, Goal Oriented Dynamic Buffer Pool Management for Database Systems, *Proc. of Int. Conf. on Engineering of Complex Systems (ICECCS'95)*, Nov. 1995.
18. K. Brown, M. Carey and M. Livny, Goal-oriented buffer management revisited, *ACM SIGMOD Record, 25 (2)*, June 1996.
19. P. Martin, H. Li, M. Zheng, K. Romanufa and W. Powley, Dynamic Reconfiguration Algorithm: Dynamically Tuning Buffer Pools, *Proc. of 11th Int. Conf. on Database and Expert Systems Applications*, London, UK, Sept. 2000.
20. IBM, DB2 Universal Database, http://www.software.ibm.com/ data/db2/udb.
21. Oracle, Oracle8i, http://www.oracle.com/ip/deploy/database/8i/index.html.
22. X. Xu, A Clustering Approach to Configuring Buffer Pools in a Database Management System, *Master thesis, Queen's University*, 2001.
23. Transaction Processing Performance Council. Benchmark descriptions, http://www.tpc.org/tpcw.
24. D. Cornell and P. Yu, Integration of Buffer Management and Query Optimization in Relational Database Environment, *Proc. of 15th Int. Conf. on Very Large Databases*, Amsterdam, The Netherlands, Aug. 1989.

25. S. Chaudhuri and V. Narasayya, AutoAdmin 'What-if' Index Analysis Utility, *Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data,* Seattle, US, Jun. 1998.
26. S.Chaudhuri and V. Narasayya, Microsoft Index Tuning Wizard for SQL Server 7.0, *Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of Data,* Seattle, US, Jun. 1998.
27. P. Yu, A. Leff and Y. Lee, On Robust Transaction Routing and Load Sharing, *ACM Transactions on Database Systems 16(3)*, 1991.
28. AutoAdmin: Self-Tuning and Self-Administering Databases**,** http://research.microsoft.com/dmx/autoadmin/.
29. G. Lohman and S. Lighstone, SMART: Making DB2 (More) Autonomic, *Proc. of 28th International Conference on Very Large Databases*, Hongkong, China, Aug. 2002.
30. AutoDBA, www.cas.ibm.com/toronto/projects/public/viewReport?REPORT=71.
31. J. Teng and R. Gumaer, Managing IBM Database 2 Buffers to Maximize Performance, *IBM Systems Journal, 23(2)*, 1984.
32. W. Effelsberg and T. Haerder, Principle of Database Buffer Management, *ACM Transaction on Database Systems, 9 (4)*, Dec. 1984.
33. E. O'Neil, P. O'Neil and G. Weikum, The LRU-K Page Replacement Algorithm for Database Disk Buffering, *Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data,* Washington, US, Jun. 1993.

34. G. Sacco and M. Sckolnick, A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hot Set Model, *Proc. of 8th International Conference on Very Large Databases*, Mexico City, Mexico, Sept. 1982.
35. H. Chou and D. DeWitt, An Evaluation of Buffer Management Strategies for Relational Database Systems, *Proc. of 11th International Conference on Very Large Databases*, Stockholm, Sweden, Aug. 1985.
36. R. Ng, C. Faloutsos and T. Sellis, Flexible Buffer Allocation based on Marginal Gains, *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, Denver, US, May 1991.
37. C. Faloutsos, R. Ng and T. Sellis, Predictive Load Control for Flexible Buffer Allocation, *Proc. of 17th International Conference on Very Large Databases*, Barcelona, Catalonia, Spain, Sept. 1991.
38. C. Chen and N. Roussopoulos, Adaptive Database Buffer Allocation Using Query Feedback, *Proc. of 19th International Conference on Very Large Databases*, Dublin, Ireland, Aug. 1993.
39. S. Kirkpatrick, C. D. Gelart Jr. and M.P. Vecchi. Op- timization by simulated annealing, *Science*, 1983.
40. E. Lawler, J. Lenstra, A. Rinnooy Kan and D. Shmoys, The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, *Wiley-Interscience Series in Discrete Mathematics*.
41. J. Kruskal. On the shortest spanning tree of a graph and the traveling salesman problem. *Proc. of the American Mathematical Society*, 7:48-50, 1956.

42. R. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1), 1985.

43. D. Yu and J. Chung, Database Access Characterization for Buffer Hit Prediction, *Proc. 9th Int. Conf. on Data Engineering*, Vienna, Austria, Apr. 1993.

44. Y. Xi, Analytical Modeling for Buffer Hit Rate Prediction, *Master Thesis, Queen's University*, 2001.

45. Belady, A Study of Replacement Algorithms for Virtual Storage Computer, *IBM System Journal, 5(2)*, Jul. 1996.

46. What's new in DB2/UDB Version 8.1, http://www-3.ibm.com/software/data/db2/udb/pdfs/db2q0.pdf

Appendix A TPC-C benchmark

The Transaction Performance Council (TPC) approved a series of benchmarks to measure the performance and price/performance of transaction processing systems. The TPC-C benchmark is an online transaction processing (OLTP) benchmark. It models a medium complexity OLTP workload with multiple transaction types. The benchmark is centered around the principle activity of an order-entry environment, which includes entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of the stock at the warehouse.

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

Table A-1 Relations in TPC-C Benchmark

The logical database design is composed of nine relations as listed in Table A-1 and shown in Figure A-1. The overall database consists of a number of *warehouses*.

Each *warehouse* is composed of ten *districts*, and each *district* has 3,000 (4 K) *customers*. There are about 100 K *items* stocked by each *warehouse*. The stock level for each *item* at a *warehouse* is maintained in the *Stock* relation. *Orders* are maintained in three relations: the *Order* relation maintains a permanent record of each order, the *New-Order* relation maintains pending orders, and the *Order-Line* relation maintains an entry for each item in an order. A history of payment transactions is maintained in the *History* relation.

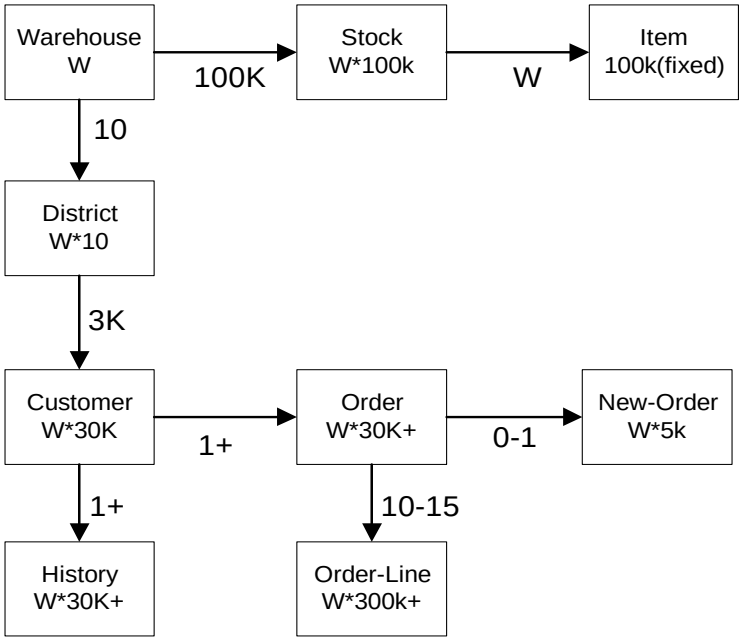

Figure A-1 Schema of TPC-C

There are five transactions in TPC-C and each corresponds to a different transaction class. The TPC-C transactions are as follows:

- *New Order* places an order for, on average, 10 items from a warehouse, inserts the order, and for each item updates the corresponding stock level.

- *Delivery* processes orders corresponding to 10 pending orders, one for each district, with 10 items per order. The corresponding entry in the New-order relation is deleted.
- Payment processes a payment for a customer and updates balances and other data in the Warehouse, District and customer relations.
- Order Status returns the status of a customer's last order.
- Stock Level examines the quantity of stock for the items ordered by each of the last 20 orders in a district.

Table A-2 summarizes the transactions based on the percentage of the workload each transaction comprises, and the number of selects, updates, inserts, deletes, non-unique selects, and join for a relational model.

Transaction	Percent	Selects	Updates	Inserts	Deletes	Non-Unique select	Join
New Order	43	23	11	12	0	0	0
Payment	45	4.2	3	1	0	0.6	0
Order Status	4	11.4	0	0	0	0.6	0
Delivery	4	130	120	0	10	0	0
Stock Level	4	0	0	0	0	0	1

Table A-2 Summary of Transactions in TPC-C Benchmark

Appendix B Confidence Intervals

Real system performance is often affected by various factors. DB2 TPC-benchmark transactions perform in a consistent manner, however, to be more scientifically accurate, we still need to specify boundaries by which we can express our confidence in the data we collect. Normally, confidence intervals are placed on the mean values of results to express the precision of these results.

Consider the results of N system runs for the same experiment: X_1, X_2, …, X_N. We assume that these results are statistically independent. The sample mean $\overline{X}_N$, of these results is given by

$$\overline{X}_N = \frac{1}{N}\sum_{i=1}^{N} X_i$$

and the variance of the distribution of the sample values, S_X^2, is denoted by

$$S_X^2 = \frac{1}{N-1}\sum_{i=1}^{N}(Xi - \overline{X}_N)^2$$

The standard deviation of the sample mean is given by $\frac{S_X}{\sqrt{N}}$.

The mean of system runs may fall in the interval $\pm\varepsilon$ within the actual mean with a certain probability drawn from the t-distribution,

$$\varepsilon = \frac{S_X t_{\alpha/2;N-1}}{\sqrt{N}}$$

where $t_{\alpha/2;N-1}$ is the $(1-\frac{\alpha}{2})$ - quantile of the t-distribution with *N-1*degrees of freedom.

The confidence intervals with respect to the system results have upper and lower limits, which are defined as follows:

$$\text{Lower Limit} = \overline{X}_N - \varepsilon$$

$$\text{Upper Limit} = \overline{X}_N + \varepsilon$$

In this thesis, a 95% confidence level was used. 5% confidence intervals for each data point were obtained.

Glossary of Acronyms

BPSS	Buffer Pool State Space
DAT	Data Access Time
DBA	Database Administrator
DBMS	Database Management System
DRF	Dynamic Reconfiguration Algorithm
DSS	Decision Support System
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
TPC	Transaction Processing Performance Council
TPC-C	Transaction Processing Performance Council Benchmark C
TPC-D	Transaction Processing Performance Council Benchmark D

Vita

Name: Wenhu Tian

Education: M.Sc. in School of Computing

Queen's University, Kingston, Ontario, Canada, 2000-2002

M.Sc. in Computer Science

Northeastern University, Shenyang, Liaoning, China, 1995-1998

B.Sc. in Computer Science

Northeastern University, Shenyang, Liaoning, China, 1991-1995

Experience: Research Assistant, School of Computing

Queen's University, 2000-2002

Teaching Assistant, School of Computing

Queen's University, 2000-2002

Awards: Graduate Award, Queen's University, 2000-2001