DYNAMIC TUNING ALGORITHMS FOR MULTIPLE BUFFER POOLS IN A DBMS

by

HOI-YING LI

A thesis submitted to the
Department of Computing and Information Science
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
April 1999

Copyright © Hoi-Ying Li, 1999
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-37964-7
Abstract

In order to meet the performance goals of different applications, database administrators (DBAs) must adjust a number of low-level performance "knobs". As the complexity and diversity of data types and database workloads increase, manual tuning by DBAs becomes almost impossible. There is a need for database management systems (DBMSs) to perform automatic tuning, based on high-level performance goals provided by the DBA.

Goal-oriented resource management is the capability of a DBMS to adjust its low-level configuration parameters in order to achieve predefined high-level performance goals. Self-tuning algorithms achieve this. They detect any violations of performance goals and dynamically reallocate database resources to meet those goals.

A self-tuning algorithm for multiple buffer pools is described in this thesis. Buffer pools are a key resource in a DBMS. The algorithm implements the concept of goal-oriented resource management by reallocating the buffer pool resources to meet the performance goals of various on-line transaction classes. All the experimental results
presented in this thesis were based on DB2/UDB and the TPC-C benchmark.
Acknowledgments

I would like to express my gratitude to my supervisor, Dr. Patrick Martin, for his support, advice, and encouragement during my research work. His guidance was critical to this thesis.

Special thanks also go to Wendy Powley and all the other members in the Database Systems Laboratory for their suggestions and feedback on my work. Their friendship has made the course of my research very enjoyable.

I would also like to thank Keri Romanufa at the IBM Toronto Laboratory for her valuable input and comments. Finally, I wish to thank Hugh Stewart for his help and support during my thesis writing.
# Contents

1 Introduction

1.1 Motivation ................................................................. 1
1.2 Goals of Research .......................................................... 3
1.3 Outline of Thesis ............................................................ 4

2 Background and Related Work ........................................... 5

2.1 Goal-Oriented Resource Allocation for DBMSs ......................... 5
2.2 Quartermaster: System Architecture .................................... 8
2.3 Buffer Pool Management .................................................. 11
  2.3.1 Definition and Basics .................................................. 11
  2.3.2 Prefetchers and Page Cleaners ..................................... 14
  2.3.3 Configuration Elements .............................................. 16
2.4 Related Work .............................................................. 19
  2.4.1 Dynamic Tuning Algorithm ........................................ 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.2</td>
<td>Fragment Fencing Algorithm</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Class Fencing Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>Access Cost Estimation</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>Hit Rate Estimation</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Asynchronous Write Rate Estimation</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Access Cost Estimation</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>Buffer Tuning Algorithms</td>
<td>35</td>
</tr>
<tr>
<td>4.1</td>
<td>Global Dynamic Reconfiguration Algorithm</td>
<td>35</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Criterion</td>
<td>36</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Components</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Goal-Oriented Dynamic Reconfiguration Algorithm</td>
<td>40</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Architecture of Tuning</td>
<td>40</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Hit Rate Estimator</td>
<td>41</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Response Time Estimator</td>
<td>42</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Buffer Allocation Mechanism</td>
<td>44</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Summary of the Goal-Oriented Tuning Algorithms</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>Experimental Results</td>
<td>56</td>
</tr>
<tr>
<td>5.1</td>
<td>Experimental Environment</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Response Time Estimator</td>
<td>57</td>
</tr>
</tbody>
</table>
CONTENTS

5.3 Tuning Step Size ........................................... 59
5.4 Memory Allocation Criteria ................................. 61
  5.4.1 Results from the Strict Greedy Scheme .......... 62
  5.4.2 Results from the Global Scheme .................. 65
  5.4.3 Results from the Relaxed Greedy Scheme ........ 67
  5.4.4 Comparison ........................................... 67
5.5 Robustness of the Algorithm ............................. 70

6 Conclusions .................................................. 75
  6.1 Contributions ........................................... 76
  6.2 Future Work ............................................. 78

A TPC-C Benchmark ........................................... 80

Bibliography .................................................... 83
List of Tables

3.1 Experimental Results of Asynchronous Write Rate .......................... 33

5.1 Estimated Response Time and Real Response Time ................. 58

5.2 Tuning Results with Different Tuning Step Sizes ......................... 60

A.1 Relations in TPC-C benchmark .................................................. 81

A.2 Relation Accesses of Transactions in TPC-C benchmark .............. 82
List of Figures

2.1 System Architecture ........................................ 9
2.2 Mapping from Table Spaces to Buffer Pools ................. 13
2.3 Concave Hit Rate Estimator .................................. 24
3.1 Line Approximation for Asynchronous Write Rate ............ 32
4.1 Pseudo-code of Tuning Component ................................. 39
4.2 Architecture of the Algorithms ................................. 41
4.3 Mapping from Transaction Classes onto Buffer Pools .......... 47
5.1 Validation of Response Time Estimator ............................ 59
5.2 Tuning Step Size .................................................. 61
5.3 Strict Greedy Scheme with One Violating Transaction .......... 63
5.4 Strict Greedy Scheme with Two Violating Transactions .......... 64
5.5 Strict Greedy Scheme with Three Violating Transactions .......... 64
5.6 Global Scheme with One Violating Transaction ................... 65
LIST OF FIGURES

5.7 Global Scheme with Two Violating Transactions ............... 66
5.8 Global Scheme with Three Violating Transactions ............... 66
5.9 Relaxed Greedy Scheme with One Violating Transaction ........... 68
5.10 Relaxed Greedy Scheme with Two Violating Transactions ......... 68
5.11 Relaxed Greedy Scheme with Three Violating Transactions ...... 69
5.12 Uniform Initial Allocation with One Violating Transaction ..... 71
5.13 Uniform Initial Allocation with Two Violating Transactions ..... 72
5.14 Uniform Initial Allocation with Three Violating Transactions ... 72
5.15 Skewed Initial Allocation with One Violating Transaction ....... 73
5.16 Skewed Initial Allocation with Two Violating Transactions ....... 74
5.17 Skewed Initial Allocation with Three Violating Transactions ..... 74
Chapter 1

Introduction

1.1 Motivation

As database management systems (DBMSs) become more and more powerful, new application areas are being expanded and explored. Along with traditional short concurrent transactions, we now see workloads containing multimedia data types, such as image, audio and video, and more complicated query processing requirements, such as rules, recursion, and user-defined functions. It is challenging for a DBMS to cope with this diversity of workloads. Studying performance for workloads containing multiple transaction classes is an interesting and important research area.

In a DBMS, a database administrator (DBA) is granted the privileges to adjust some configuration parameters to maintain or improve the performance of the DBMS
1.1 Motivation

[IBM97]. Such manual monitoring and tuning is, however, impractical for workloads with a wide range of resource demands and execution times. The "performance knobs" commonly provided by a DBMS control low-level resource allocation parameters, such as memory management, CPU allocation, load control, disk scheduling, query optimization, and so on [BMC94]. Since each class of transaction in a DBMS has different resource consumption features, a complete solution to the problem of translating performance goals into resource management decisions is difficult to achieve.

The static nature of performance knobs also creates problems for manual performance tuning. In general, a DBMS does not attempt to adjust the performance knobs during execution in response to changes in system load. The workload in an environment with multiple transaction classes (i.e. a multiclass environment) can, however, be unpredictable and dynamic. A DBMS needs a general solution to automate the controls [BCD92].

In a multiclass database workload, each transaction class may have its own performance goal. The high-level performance goals can be defined using different standards: average response time, throughput, and so on. Ideally, a DBMS should be able to accept performance goals for each transaction class as inputs, and then dynamically adjust the relevant low-level performance knobs to achieve those goals. This type of self-tuning capability is called goal-oriented resource management [NFC92]. Finding the mapping from the high-level performance goals for transaction classes to
the appropriate low-level tuning parameters is the critical problem in goal-oriented resource allocation. Our research is targeted at automating this dynamic mapping procedure.

The performance of a DBMS is greatly influenced by the effective use of semiconductor memory as buffer pool space, which is a cache for data pages between database applications and the physical database files [HJ97]. This is true for both on-line transaction processing and decision support query processing. A well-tuned buffer management system can significantly improve performance.

1.2 Goals of Research

The aim of this research is to study the problem of dynamically adjusting the sizes of multiple buffer pools in a DBMS using a goal-oriented approach. The buffer pool sizes are dynamically expanded or contracted by a certain number of memory pages in each tuning interval to achieve a predefined performance goal for each transaction class. All the experiments were run on DB2/UDB [CHA96], and the scenarios for the transaction classes were based on the TPC-C benchmark [TPC93]. The main goals of the research are:

- Explore the research issues of goal-oriented resource allocation in the area of buffer pool tuning. Our research provides solutions for dynamic size adjustments to multiple buffer pools in a DBMS (DB2/UDB in particular).
1.3 Outline of Thesis

- Establish a bridge to map the high-level performance goals to the low-level configuration parameters. We study the relationship between the performance of a transaction class and its resource consumption pattern.

- Create cost models to evaluate resource consumption. The cost models quantify disk access cost for each transaction class.

- Develop a dynamic memory allocation mechanism. In a DBMS with multiple buffer pools, the total number of buffer pages is fixed. We work out strategies for reallocating pages among buffer pools.

1.3 Outline of Thesis

This thesis presents the principle and the components of goal-oriented multiple buffer pool tuning algorithms. Chapter Two presents background knowledge and relevant research, and addresses the originality and challenges in our research. Chapter Three discusses models for access cost estimation. Chapter Four presents a non-goal-oriented multiple buffer pool tuning algorithm with a system-wide approach, and describes our goal-oriented strategies and buffer allocation mechanism. Chapter Five demonstrates and compares the experimental results of our tuning algorithms. Chapter Six draws general conclusions from our research and discusses areas of future research.
Chapter 2

Background and Related Work

2.1 Goal-Oriented Resource Allocation for DBMSs

Goal-oriented resource allocation is the capability of a DBMS to adjust its low-level configuration parameters in order to achieve predefined high-level performance goals [BCL96]. Goal-oriented resource allocation deals with competing transaction classes that require different types of work in a DBMS. In order to allocate resources on a per-class basis, some mechanism must exist to map individual queries and transactions onto a set of transaction classes. The process of defining transaction classes and mapping transactions to classes is critical. In general, classes should consist of transactions that make similar service demands on each system resource. For example, I/O-bound transactions would not ordinarily be in the same class as CPU-bound
transactions. On the other hand, classes must distinguish workload components for which independent performance requirements are to be satisfied. For example, if the response time of a particular transaction type is important, then this type of transaction should not be assigned to the same class as other transaction types.

There are many possible ways to specify database system performance goals. Performance goals for a transaction class with sub-second response times are usually expressed in terms of average throughput, which can be measured in transactions per second. On the other hand, performance goals for long-running transactions with tens of seconds or minutes response times may be more naturally expressed in terms of average response time. Response time metrics can be specified as average, maximum, or percentile values. In addition, combinations of multiple metrics are also common, such as a target response time that is subject to a minimum or an 80th percentile throughput constraint [BCL93b].

Several criteria should be satisfied by any goal-oriented resource allocation algorithm. These criteria are defined using such metrics as average throughput, average response time, etc. The most important criteria are accuracy, responsiveness, stability, overhead, robustness and practicality [BRO95].

- **Accuracy:** The observed performance of classes should be close to their stated goals. An index value can be used to quantify the accuracy. The value we used in this thesis is *achievement index*, which is the ratio of the average response
time goal over the average real response time. An achievement index which is greater than one indicates an achieved goal, while an achievement index which is less than one indicates a violated goal. Accuracy is the most important criterion for the success of a goal-oriented resource allocation algorithm.

- **Responsiveness**: The number of knob adjustments required to bring a class to its goal should be as small as possible, and the adjustment procedure should finish in reasonable time. The responsiveness criterion rules out an exhaustive search strategy. There is a trade-off between the responsiveness and the accuracy of an algorithm. In general a good algorithm should find the best solution within an acceptable period of time.

- **Stability**: For a stable workload, all performance knobs should be left alone once performance goals are achieved. In other words, the low-level knob settings provided by the algorithm should remain stable under the same workload. Also, database performance should not degrade dramatically during the tuning procedure. There can be a trade-off between stability and responsiveness. A goal-oriented resource allocation algorithm should balance these criteria.

- **Overhead**: A goal-oriented resource manager should minimize the extent to which it reduces the overall system efficiency, compared to a non-goal-oriented
system. A goal-oriented resource allocation algorithm should not cause a significant reduction in system capability compared to a non-goal-oriented system.

- **Robustness**: The system should handle as wide a range of workloads as possible, and avoid knob adjustments for a class that cannot be affected by a given knob. Different classes can have different resource access patterns. For example, in general, the more buffer pool space allocated, the more efficiently a transaction class can be serviced. However, if the transaction class is dominated by large file scans and the DBMS has an effective prefetching strategy, then the buffer allocation knob may not affect the performance for that class since the prefetcher already supports a fairly high hit rate with considerably little memory.

- **Practicality**: No unrealistic assumptions about the workload or the DBMS should be made in a goal-oriented resource allocation algorithm. The system should take application requirements into consideration and the design should be based on practical evaluation of workloads and other involved factors.

### 2.2 Quartermaster: System Architecture

*Quartermaster* is the general project whose aim is to study the problem of automating resource management in DBMSs. Our goal-oriented buffer pool allocation algorithm is
part of this project. The project is based on DB2/UDB. Our models and algorithms are designed to support multiple resource management for DB2, and the research will eventually lead to the development of a DBMS which is able to dynamically reconfigure its low-level resource allocations in order to meet the performance goals predefined by its users. This thesis contributes to the overall project by developing algorithms to reallocate the buffer pool resource, which has a major impact on the DBMS performance. The architecture of the project is shown in Figure 2.1.

The Sampler collects performance data of concern from DB2 and stores the information in a repository, which can be accessed at a later date. The sampling
2.2 Quartermaster: System Architecture

mechanism used in the project is snapshot monitoring [IBM98], which takes successive pictures of the state of database activity. The snapshot information is stored in special data structures, which can be accessed through an application programming interface (API) provided by DB2. The snapshot monitor provides cumulative information in the form of counters, and the available statistics are categorized using different monitor switches. For example, the statistics for the number of locks and deadlocks can be collected using the "LOCK" switch, the statistics for the start time, stop time and statement identification of SQL statements can be collected using the "STATEMENT" switch, and the statistics for the number of reads and writes in the buffer pools can be collected using the "BUFFERPOOL" switch. Currently, the sampler collects buffer pool performance data for the multiple buffer pool tuning algorithms, and has a minimal impact on the database server.

The Analyzer checks the performance data in the repository at timed intervals and compares the data with predefined performance goals. Performance goals are defined in terms of average response times for the various transaction classes. The analyzer uses the transaction class's achievement index which is the ratio of the desired response time to the average response time, to determine whether a class is meeting its performance goal or not. An achievement index value which is smaller than one indicates a goal violation. If consecutive violations are detected for a certain transaction class, the analyzer posts a message to the blackboard to report the performance
violation. Ideally, the analyzer would be able to detect unrealistic performance goals and negotiate with the end-user to adjust them.

The Planner checks for violations posted on the bulletin board, and responds with an appropriate action if a violation exists. A group of actions can be included in the planner as candidate solutions, such as buffer pool tuning, CPU allocation, disk scheduling, load control, etc. The planner takes the most suitable actions to help the violating classes to meet their goals. This thesis contributes to the project as a component of the planner. It provides multiple buffer pool tuning strategies as solutions for performance violations.

2.3 Buffer Pool Management

2.3.1 Definition and Basics

Among all the low-level configuration knobs in DB2/UDB, the most important parameter which affects database performance is the size of the buffer pools. A buffer pool is a large data cache between the applications and the physical database files. A buffer pool provides an area of memory storage into which database pages are temporarily read and updated. The purpose of the buffer pool is to improve database system performance by reducing disk accesses. The buffer pool mechanism in DB2/UDB is similar to the virtual memory paging mechanism in operating systems [S92].
Each database in DB2/UDB has at least one buffer pool, which is named "IB-MDEFAULTBP". It is created when the database is created. There can be multiple buffer pools in a database system. All buffer pools reside in global memory, which is available to all applications using the database. If the buffer pools are large enough to hold the required data in memory, fewer disk accesses will occur and the applications will run faster. On the other hand, if the buffer pools are not large enough the overall performance of the database can be significantly reduced due to the high amount of disk activity needed to fetch the data required by applications.

A table space is a storage model that provides a level of indirection between a database and the tables stored in that database [IBM97]. Multiple tables can be aggregated into one table space. The table space mechanism in DB2/UDB provides improved performance, more flexible configuration, and better integrity. The relationship between table spaces and buffer pools is shown in Figure 2.2. Each table space is associated with a specific buffer pool. The mapping from table spaces onto buffer pools is many-to-one (i.e. one table space can only be mapped onto one buffer pool while one buffer pool can hold multiple table spaces).

In a database system with \( m \) table spaces and \( n \) buffer pools, there are \( m^n \) ways to map the table spaces to the buffer pools. There are some rules for reasonable mappings. For example, it is beneficial to assign a large buffer pool to a table space with a non-uniform access pattern, which indicates that some "hot pages" in the
2.3 Buffer Pool Management

Figure 2.2: Mapping from Table Spaces to Buffer Pools

table space are visited much more frequently than the other pages. If “hot pages” are kept in the buffer pool, the response time is reduced. As well, a small table space is more sensitive to the buffer pool size, and in such cases a large buffer pool can help dramatically.

The configuration of one or more buffer pools is the single most important tuning area – for most of the applications connected to the database, most of the data manipulation upon table rows or index entries takes place in the buffer pool. When an application accesses a row of a table for the first time, the database manager places
the page containing that row in the buffer pool. Each time an application requests data, the buffer pool is checked first. If the requested data is found on pages kept in the buffer pool, the database manager does not need to access the disk storage to retrieve the requested data. Avoiding the need to retrieve data from disk storage results in faster performance.

Having multiple buffer pools provides flexibility in setting performance goals for specific applications. For example, for table spaces representing one or more large tables which are accessed randomly, the size of the buffer pool can be reduced, since caching more data pages in the buffer pool might not be beneficial. In a database system with multiple buffer pools, data and indexes can be isolated in separate buffer pools to favour certain applications. The buffer pool which is associated with an important or urgent online-transaction application can be assigned a larger size. In this way the data pages used by the application can be found more frequently in the buffer pool, so that response time is shortened.

2.3.2 Prefetchers and Page Cleaners

Prefetching index and data pages into the buffer pools can help improve performance by reducing the time spent waiting for I/O to complete [CHA96]. A prefetcher is a mechanism that anticipates the use of data or index pages and retrieves them from disk into buffer pools before they are accessed. In most situations, those pages are
read just before they are needed. However, prefetchers can cause unnecessary I/O by reading pages into the buffer pools that will not be used. There are two categories of prefetch:

- **Sequential prefetch.** Sequential prefetch is a mechanism that reads consecutive pages into buffer pools before the pages are required by the application. Reading several consecutive pages into a buffer pool with a single I/O operation can greatly reduce the overhead.

- **List prefetch.** List prefetch is a way to access data pages efficiently, even when the data pages needed are not consecutive. It can be used in conjunction with either single or multiple index access.

Prefetching is started when the database manager determines that prefetching may help to improve performance. Multiple prefetchers can be used. Prefetchers can perform multiple I/O operations in parallel when data pages are scattered over different physical devices. Prefetchers run asynchronously with application agents that perform database applications, so that the applications can retrieve the required pages from the buffer pools immediately. The I/O operations performed by prefetchers are called *asynchronous reads*.

“Dirty” pages are pages in a buffer pool where data has been changed by an application but has not yet been written to disk. Pages are written from the buffer pools to disk when the dirty pages in the buffer pool has exceeded a threshold value, which
is specified as a configuration parameter for the database. *Page cleaners* monitor the buffer pools and asynchronously write dirty pages from the buffer pools to disk. They perform I/O that would otherwise have to be performed by the application agents. The I/O operations performed by page cleaners are called *asynchronous writes*. Page cleaners can run in parallel with the application agents so that transactions are not forced to wait while pages are written to disk. More than one page cleaner can be configured in a database. The goals of the page cleaners are:

- To ensure that application agents will always find free pages in the buffer pools. If an agent does not find free pages in a buffer pool, it has to write them to disk itself, and the associated application will have a poorer response time.

- To speed database recovery if a system crash occurs. The more pages that have been written to disk, the smaller the number of log file records that must be processed to recover the database.

### 2.3.3 Configuration Elements

The database server reads and updates all data from the buffer pools. Data is copied from disk to buffer pools if it is required by applications but is not present in the buffer pools. Pages are loaded in a buffer pool either by the application agent, which is called a *synchronous read*, or by the prefetchers, which is called an *asynchronous read*. Similarly, pages are written to disk from a buffer pool either by the application
agent, which is called a *synchronous write*, or by the page cleaners, which is called an *asynchronous write*. If the server needs to read a page, and that page is already in the buffer pool, then access to that page is much faster than if the page has to be read from disk. It is desirable to hit as many pages as possible in the buffer pool. Avoiding disk I/O is the main issue for performance improvement, and so proper configuration of buffer pools is the most important factor for performance tuning.

The following DB2/UDB-specific metrics provide information that is particularly helpful for buffer pool configuration. Each metric is a statistic for a single buffer pool [IBM98].

- **Data Logical Reads.** Indicates the number of logical read requests for data pages that have gone through the buffer pool, denoted as "pool.data.lreads".

- **Index Logical Reads.** Indicates the number of logical read requests for index pages that have gone through the buffer pool, denoted as "pool.index.lreads".

- **Data Physical Reads.** Indicates the number of logical read requests that required I/O to get data pages from disk into the buffer pool, denoted as "pool.data.p.reads".

- **Index Physical Reads.** Indicates the number of physical read requests to get index pages into the buffer pool, denoted as "pool.index.p.reads".

- **Data Writes.** Indicates the number of times a buffer pool data page was physically written to disk, denoted as "pool.data.writes".
2.3 Buffer Pool Management

- **Index Writes.** Indicates the number of times a buffer pool index page was physically written to disk, denoted as "pool.index.writes".

- **Asynchronous Data Reads.** Indicates the number of read requests for data pages performed by prefetchers, denoted as "pool.async.data.reads".

- **Asynchronous Index Reads.** Indicates the number of read requests for index pages performed by prefetchers, denoted as "pool.async.index.reads".

- **Asynchronous Data Writes.** Indicates the number of times a data page was physically written to disk by page cleaners, denoted as "pool.async.data.writes".

- **Asynchronous Index Writes.** Indicates the number of times an index page was physically written to disk by page cleaners, denoted as "pool.async.index.writes".

The buffer pool hit rate indicates the percentage of time that the database manager retrieved pages from the buffer pool without disk access. The higher the hit rate, the better the performance. The buffer pool hit rate can be calculated as follows:

\[
(1 - \frac{pool.data.p.reads + pool.index.p.reads}{pool.data.l.reads + pool.index.l.reads}) \times 100\%
\]
2.4 Related Work

Goal-oriented buffer pool allocation algorithms can be described in terms of three components: *response time estimator*, *hit rate estimator*, and *buffer allocation mechanism*. A response time estimator evaluates response time as a function of buffer pool hit rate. A hit rate estimator evaluates buffer pool hit rate as a function of memory allocation. A buffer allocation mechanism divides up memory between the competing transaction classes. We study three existing goal-oriented buffer pool tuning algorithms in this section, namely the *Dynamic Tuning* algorithm, the *Fragment Fencing* algorithm, and the *Class Fencing* algorithm.

2.4.1 Dynamic Tuning Algorithm

The *Dynamic Tuning* algorithm [CFW95] adopts observations from Belady’s virtual memory study [BEL66] to formulate the relationship between the buffer pool sizes and the corresponding buffer pool hit rates. For a buffer pool with size $SIZE$, the hit rate estimator is formulated as follows:

$$HIT(SIZE) = 1 - a \times SIZE^b$$

where $a$ and $b$ are constants calculated using the known hit rates for two different buffer pool sizes. The constants are defined in Section 3.1.
The Dynamic Tuning algorithm specifies response time goals with respect to low-level buffer management requests instead of high-level transaction classes' performance metrics. The random access response time $RT$ for a buffer pool with size $SIZE$ and the hit rate function $HIT$ is approximated as follows:

$$RT(SIZE) \approx (1 - HIT(SIZE)) \times DELAY$$

where $DELAY$ is the average time required for moving a page from disk to the buffer pool. The delay time for retrieving a page from the buffer pool is assumed to be negligible.

The concept of performance index is introduced in the Dynamic Tuning algorithm to evaluate the performance of a buffer pool in a multiple buffer pool environment. For each buffer pool and its associated random access response time goal, a performance index $PI(SIZE)$ with the buffer pool size $SIZE$ is defined as follows:

$$PI(SIZE) = \frac{RT(SIZE)}{GOAL}$$

where $RT(SIZE)$ is the random access response time for a buffer pool with size $SIZE$ and $GOAL$ is the corresponding response time goal. If the performance goal is met, the performance index will be less than or equal to 1. The smaller the value the better the performance.

The Dynamic Tuning algorithm attempts to minimize the maximum performance index, and to balance the performance index values among all the buffer pools. Since
memory resources are always limited, the total number of the buffer pages is fixed. By varying the buffer pool sizes, the performance index can be decreased by increasing the buffer pool size, and can be increased by decreasing the buffer pool size. In each tuning interval, the buffer pool with the maximum performance index is made larger with a certain step size. On the other hand, the buffer pool with the minimum performance index is shrunk by the same number of buffer pages. This maintains the total number of buffer pages as a constant.

2.4.2 Fragment Fencing Algorithm

The Fragment Fencing algorithm [BCL93a] is useful in a database with multiple transaction classes sharing one buffer pool resource. A fragment is defined as a set of database pages that share approximately the same access frequency. For example, a single relation or a single level of a tree-structured index could be a fragment.

In the Fragment Fencing algorithm, the response time estimator is based on the assumption that the response time and the buffer miss rate (i.e. 1 - hit rate) are direct proportional. The estimated target hit rate for a fragment that is used by the algorithm in its attempts to achieve the response time goal is computed as:

$$HIT_{\text{target}} = 1 - (1 - HIT^{\text{obs}}) \times \frac{R^{\text{goal}}}{R^{\text{obs}}}$$

where $R^{\text{obs}}$ and $R^{\text{goal}}$ are the observed response time and the corresponding response time goal respectively, and $HIT^{\text{obs}}$ is the observed hit rate that occurs with the
observed response time. Though many other factors, such as CPU allocation and
disk scheduling, could have impacts on the real response time, the assumption of a
linear relationship between response time and miss rate is reasonable for a disk-bound
transaction class.

The Fragment Fencing algorithm estimates a hit rate function for each fragment.
A uniform reference probability is assumed for page accesses, and so the hit rate of a
fragment is estimated to be the percentage of the fragment that is resident in memory.
The goal of the Fragment Fencing algorithm is to determine the minimum number of
pages for each fragment that must be kept in the buffer pool in order to achieve an
overall target hit rate for a transaction class. These minimum amounts are called the
target residencies of the relevant fragments.

When a transaction class' hit rate needs to be increased by some amount to meet
the performance goal, all of the fragments referenced by the transaction class are
sorted in order of decreasing class temperature, which is their size-normalized access
frequency. Based on an assumption of uniform reference probability, the target res-
Residencies for all the involved fragments are increased in turn from hottest to coldest
with respect to their class temperature until the hit rates for all fragments add up to
the overall hit rate required by the transaction class. The hit rate for a higher temper-
atture fragment is increased to 100% before increasing that of any lower-temperature
fragment.
2.4.3 Class Fencing Algorithm

A potential problem with the Fragment Fencing algorithm is the assumption of uniform reference probability. The Class Fencing algorithm [BRO95] is proposed as an improved variation. Instead of building multiple fences to protect the database fragments referenced by a transaction class, the Class Fencing algorithm builds a single fence to protect all of a transaction class' buffer pages.

The Class Fencing algorithm adopts the same response time estimator as the Fragment Fencing algorithm. It assumes that the miss rate and response time are proportional.

In the Class Fencing algorithm, the hit rate estimator is based on the concavity theorem, which declares that regardless of the database reference pattern, the hit rate is a concave function of buffer memory allocation under an optimal replacement policy. In other words, the slope of the hit ratio curve, which represents the marginal increase in hit rate obtained by adding an additional page of memory, never increases as more memory is added to an optimal buffer replacement policy. An optimal buffer replacement policy always chooses pages for memory residency in decreasing order of their values in order to achieve the highest hit rate for a given amount of memory. Since the optimal buffer replacement policy always chooses the most valuable page to be inserted into the buffer pool, the slope of the hit rate curve keeps decreasing. Though perfect optimal replacement policies may be impractical in the real world due
2.4 Related Work

Related Work

to the lack of the knowledge of future reference patterns, according to some empirical studies, the behaviour of industrial-strength DBMS replacement policies are close enough to be optimal so that the hit rate concavity theorem applies. The Class Fencing algorithm assumes the hit rate concavity theorem for the most commonly occurring workloads running on a typical commercial DBMS. The hit rate concavity assumption enables a simple straight line approximation to be used to predict the memory required to achieve a particular hit rate for a transaction class.

The prediction of the required buffer allocation for a transaction class is illustrated in Figure 2.3. The dashed curve represents a hypothetical hit rate function for a
class. The horizontal axis represents memory allocation where $M_{max}$ is the maximum memory allocation for a particular transaction class. The vertical axis represents the hit rate. $H_T$ is the target hit rate given to the hit rate estimator. The basic idea of the straight line approximation is to move the class as quickly as possible to the point "X", which stands for the memory allocation $M_T$ that results in the target hit rate. The point labeled $O_1$ indicates the initial observed hit rate $H_1$ of the transaction class with its naturally occurring memory allocation $M_1$. To estimate the memory required to achieve the target hit rate, a line extending from the origin through $O_1$ is computed. The point at which this line intersects the target hit rate ($E_1$) indicates a lower bound ($M_2$) on the memory allocation that will achieve the target hit rate. Since concavity applies to the hit rate curve, there is no risk that the target hit rate will be exceeded. After increasing the class's memory allocation to $M_2$ and waiting long enough to ensure statistical stability, a second observation $O_2$ occurs and another estimate $E_2$ is computed using points $O_1$ and $O_2$. $E_2$ predicts a required memory allocation of $M_3$. With one more estimate using points $O_2$ and $O_3$, the target hit rate is achieved.

The Class Fencing algorithm is extremely responsive because the hit rate estimator allows large increments in memory allocation without overshooting the target. Each goal-oriented transaction class has its own buffer pool manager, while a global buffer pool manager manages pages for no-goal classes and the unfenced pages that belong
to goal classes.
Chapter 3

Access Cost Estimation

This chapter presents the models for access cost. Due to the limitation of the current version of DB2/UDB, a new allocation for multiple buffer pools can not take effect until the running database manager is stopped and restarted. It is therefore unrealistic for an iterative tuning algorithm to reallocate the buffer pool sizes and to collect statistics at each iteration. Instead, we use two approximation functions to estimate the hit rate and the asynchronous write rate of each buffer pool. The statistics for the buffer pool hit rate and asynchronous write rate are used in our access cost estimation function. The cost estimation function is critical to the multiple buffer pool tuning algorithms, which will be described in Chapter 4.
3.1 Hit Rate Estimation

The buffer pool *hit rate* indicates the percentage of time that the database manager retrieves pages from the buffer pool without disk access. Compared with the cost of disk access, the cost of page retrieval from a buffer pool is negligible. With the help of database monitor mechanisms, statistics for the physical reads and logical reads of data pages and index pages can be collected. The hit rate can then be calculated as described in Chapter 2. However, due to the need for a responsive buffer pool tuning algorithm, it is not practical to collect these statistics after each adjustment to the buffer pool sizes. An estimation function is required to formulate the relationship between different buffer pool sizes and the corresponding buffer pool hit rates.

The result from Belady's virtual memory study [BEL66] is adopted by this thesis to quantify the relationship between the size and the hit rate for an individual buffer pool. This formula is also adopted by the Dynamic Tuning algorithm to estimate buffer pool hit rates. With the current buffer pool size $SIZE$ as an input, the hit rate of the buffer pool is estimated as follows:

$$HIT(SIZE) = 1 - a \times SIZE^b$$

where $a$ and $b$ are constants.

To determine the values of $a$ and $b$, the hit rate must be measured experimentally for two different buffer pool sizes. If hit rates $HIT(size_1)$ and $HIT(size_2)$ are collected
for two different buffer pool sizes, $size_1$ and $size_2$, then the values of $a$ and $b$ are:

$$b = \frac{\ln(1 - HIT(size_2)) - \ln(1 - HIT(size_1))}{\ln(size_2) - \ln(size_1)}$$

$$a = \frac{1 - HIT(size_1)}{e^{b \ln(size_1)}}$$

Because of the concavity feature of the hit rate curve in commercial DBMSs, it is acceptable to estimate the hit rates of a buffer pool at different sizes based on two sampling points. This is a trade-off between the accuracy of the hit rate estimation and the responsiveness of the buffer pool tuning algorithm.

### 3.2 Asynchronous Write Rate Estimation

A buffer pool page is written to disk for the following reasons:

- To free a page in the buffer pool so that another page can be read in.
- To flush the buffer pool.

The system does not always have to write a page to make room for a new one. If the page in the buffer pool has not been updated, it can simply be replaced. On the other hand, if a write is required before a new page can be read into the buffer pool, either an asynchronous page cleaner or the application agents can perform that write. Page cleaners can run in parallel with the application agents so that transactions are not forced to wait while the dirty pages are written to disk. As a result, only the
writes that are performed by the application agents need be counted for the access costs.

We study the rate $p$ of the number of asynchronous writes to the total number of writes for both data pages and index pages in a buffer pool, which is:

$$p = \frac{\text{pool\_async\_data\_writes} + \text{pool\_async\_index\_writes}}{\text{pool\_data\_writes} + \text{pool\_index\_writes}} \times 100\%$$

where the numerator is the number of times a data or index page is physically written to disk by page cleaners, and the denominator is the total number of times a data or index page is physically written to disk. These statistics can be collected for each buffer pool by the database monitor mechanisms. However, for the same reason as with the buffer hit rate estimation, it is not realistic to collect the relevant statistics and calculate the asynchronous write rate after each adjustment to a buffer pool's size. Therefore we need to find some appropriate way to estimate the rate for different buffer pool sizes.

We designed experiments to explore the relationship between buffer pool size and asynchronous write rate. Multiple page cleaners were used in a DB2/UDB database with multiple buffer pools. The transaction applications used in the experiments are based on the TPC-C benchmark, which is discussed in the appendix. Two groups of experiments were done with two and three buffer pools in the database respectively. In each group, four, eight, and twenty page cleaners were assigned to the database. (The number of page cleaners is a configuration parameter in a database, and it
3.2 Asynchronous Write Rate Estimation

is specified before the transaction applications are run.) With the number of page
 cleaners as a constant, all the experimental results showed the same tendency: the
asynchronous write rate rises when the buffer pool size increases.

Results from one of the experiments is shown in Figure 3.1. In this experiment, there were four page cleaners and three buffer pools involved (two for data tables and one for index tables). The results for one of the data table buffer pools is shown in Figure 3.1. The solid curve shows how the asynchronous write rate changes while the buffer pool size is enlarged. In this experiment, the buffer pool size increased by 2,500 pages at each step. Each page in the buffer pool is 4KB in size. The dotted line which connects two points on the solid curve (the first sampling point and the last sampling point in this case) can be used as an estimate of the asynchronous write rate under different buffer pool sizes.

Table 3.1 lists the average estimation errors between the real asynchronous write rates and the estimated rates based on a line approximation. BP\_DATA1 and BP\_DATA2 are the two buffer pools for the data tables, and BP\_INDEX is the buffer pool for the index tables.

Experiments were also done to observe how the number of sampling points influences the accuracy of this approximation. In general, with a constant step size (i.e. change to the buffer pool size), more sampling points increase the accuracy.
3.2 Asynchronous Write Rate Estimation

These experiments show that, given the asynchronous write rates at two sampling points with different buffer pool sizes, we can use a line approximation to estimate the asynchronous write rate for other buffer pool sizes. The approximation for the asynchronous write rate \( p(SIZE) \) for a buffer pool of size \( SIZE \) is:

\[
p(SIZE) = k \times SIZE + m
\]

where \( k \) and \( m \) are constants for a given buffer pool. The values of \( k \) and \( m \) can be determined by:

\[
k = \frac{p(S_1) - p(S_2)}{S_1 - S_2}
\]

\[
m = \frac{S_1 \times p(S_2) - S_2 \times p(S_1)}{S_1 - S_2}
\]

where \( p(S_1) \) and \( p(S_2) \) are the asynchronous write rates under the buffer sizes of \( S_1 \) and \( S_2 \) respectively (i.e. at the two sampling points).
3.3 Access Cost Estimation

<table>
<thead>
<tr>
<th>Number of Cleaners</th>
<th>Buffer Pool</th>
<th>Estimation Error (in percentage point units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>BP_DATA1</td>
<td>5.11</td>
</tr>
<tr>
<td>8</td>
<td>BP_DATA1</td>
<td>3.49</td>
</tr>
<tr>
<td>20</td>
<td>BP_DATA1</td>
<td>3.83</td>
</tr>
<tr>
<td>4</td>
<td>BP_DATA2</td>
<td>9.26</td>
</tr>
<tr>
<td>8</td>
<td>BP_DATA2</td>
<td>5.31</td>
</tr>
<tr>
<td>20</td>
<td>BP_DATA2</td>
<td>18.25</td>
</tr>
<tr>
<td>4</td>
<td>BP_INDEX</td>
<td>10.57</td>
</tr>
<tr>
<td>8</td>
<td>BP_INDEX</td>
<td>5.99</td>
</tr>
<tr>
<td>20</td>
<td>BP_INDEX</td>
<td>4.01</td>
</tr>
</tbody>
</table>

Table 3.1: Experimental Results of Asynchronous Write Rate

3.3 Access Cost Estimation

For disk-bound applications, disk access has a major impact on database performance. The fewer disk accesses involved, the better the performance. The hit rate of a buffer pool indicates the percentage of page accesses that do not require disk accesses.

Disk access cost is composed of disk read cost and disk write cost. Since the page cleaners take charge of all the buffer-flushing disk writes, which can be performed in parallel with the applications, the application agents perform disk writes only when a dirty page in the buffer pool has to be written to disk to make room for a new page to be read in. In other words, the disk writes performed by the application agents only occur with a disk read request. Therefore the disk write cost can be regarded as a component of the disk read cost.

The number of physical reads indicates the number of logical reads that required
3.3 Access Cost Estimation

disk access to get the corresponding data and index pages. We estimate the disk access cost $C$ of the applications related to a certain buffer pool as the product of the single access cost for each physical read and the number of physical reads. If we assume that the cost of a read or a write is 1, then the disk access cost $C$ can be approximated as follows:

$$C = (pool\_data\_l\_reads + pool\_index\_l\_reads) \times (1 - H) \times (1 + (1 - p) \times d)$$

where $pool\_data\_l\_reads$ and $pool\_index\_l\_reads$ are the number of logical read requests for data pages and for index pages, $H$ is the hit rate of the buffer pool under current memory allocation, $p$ is the asynchronous write rate of the buffer pool under current memory allocation, and $d$ is the proportion of dirty pages in the buffer pool, which can be estimated as a constant number. $(pool\_data\_l\_reads + pool\_index\_l\_reads) \times (1 - H)$ evaluates the number of physical reads, which are performed by the application agents. $1 + (1 - p) \times d$ evaluates the access cost of each physical read. The cost of one physical read includes an unconditional read request from disk to buffer pool, and a conditional write request to make room for the new pages if all the buffer pages are dirty and the application agents (instead of page cleaners) write dirty pages to disk. The possibility of the conditional write request is $(1 - p) \times d$. 
Chapter 4

Buffer Tuning Algorithms

Based on the disk access cost estimation for each buffer pool, we propose multiple buffer pool tuning algorithms in DB2/UDB. In this chapter we discuss two buffer tuning algorithms: one for *global* dynamic reconfiguration, and the other for *goal-oriented* dynamic reconfiguration, with three variations. The global algorithm described in Section 4.1 was our first attempt at a dynamic reconfiguration algorithm. It laid the foundation for the more sophisticated goal-oriented algorithm described in Section 4.2.

4.1 Global Dynamic Reconfiguration Algorithm

The target of the global dynamic reconfiguration algorithm is to achieve optimal system-wide performance in a DBMS. We present the tuning criterion and the major
components of this algorithm in this section.

4.1.1 Criterion

The evaluation criterion for the multiple buffer pool tuning algorithm with a system-wide approach is to distribute the disk access costs evenly among all the buffer pools. The database system achieves optimal performance when all the buffer pools are achieving their minimum access costs. Since the total number of buffer pool pages is fixed, it is not practical to find a point where all the buffer pools are doing their best. Instead, the access costs are balanced among all the buffer pools.

As shown in Section 3.3, the access cost can be evaluated for each buffer pool under a certain memory allocation. A predefined threshold value is carefully chosen to indicate an acceptable access cost difference between buffer pools. The tuning algorithm is activated when the difference between the maximum and the minimum access cost among all the buffer pools is larger than the predefined threshold value. In each tuning interval, a certain number of buffer pages are transferred from the buffer pool with the minimum access cost to the one with the maximum access cost, and the access costs among all the buffer pools are re-evaluated. The algorithm will not stop until the access costs have been balanced among all the buffer pools (i.e. the difference between the maximum and the minimum access cost is smaller than the threshold value).
4.1 Global Dynamic Reconfiguration Algorithm

4.1.2 Components

Our global reconfiguration strategy has three components: *sampling*, *tuning*, and *resizing*. The *sampling* component collects the statistics for all the buffer pools at the starting point. The *tuning* component repeatedly calculates new buffer allocations during tuning intervals until it finds a solution in which all the buffer pools have equal access costs. The *resizing* component applies this solution by reconfiguring the database.

The sampling component gathers statistics at two points: when the buffer pools are at their initial sizes, and when they are at a different “reference point” size. For each buffer pool, the buffer pool size at the starting point and at the reference point are significantly different. The following statistics are collected for the starting point and the reference point in each buffer pool:

- the current buffer pool size
- the number of physical reads (*p.reads*) for both data and index pages
- the number of logical reads (*L.reads*) for both data and index pages
- the number of asynchronous writes (*async.writes*) for both data and index pages
- the number of writes (*writes*) for both data and index pages

For both the starting point and its reference point, the corresponding hit rates $H_i$
and the asynchronous write rates $p_i$ for each buffer pool $BP_i$ are calculated by:

$$H_i = (1 - \frac{p_{reads}}{l_{reads}}) \times 100\%$$

$$p_i = \frac{async\_writes}{writes} \times 100\%$$

The coefficients in the hit rate estimation function (see Section 3.1) and the asynchronous write rate estimation function (see Section 3.2) are then derived from the hit rates and the asynchronous write rates at the two sampling points.

With the values of the number of logical reads, the hit rate and the asynchronous write rate for each buffer pool captured at the starting point, the initial disk access cost at each buffer pool can be evaluated. If the difference between the maximum and the minimum access cost is larger than the predefined threshold value, the tuning component is activated.

Pseudo-code for the main part of the tuning component is shown in Figure 4.1. The algorithm keeps running until, among all the buffer pools, the difference between the maximum access cost and the minimum access cost is smaller than the predefined threshold value $delta$ (i.e. balanced access costs have been achieved among all the $N$ buffer pools). Over-tuning has occurred ("over-tuned" is true) when, first, the buffer pool with the maximum cost becomes the one with the minimum cost and vice versa, or second, a buffer pool shrinks below a predefined minimum value. Either case indicates that too many pages have been stolen from one buffer pool and given
4.1 Global Dynamic Reconfiguration Algorithm

while ( (max{Ci} - min{Ci}) > δ )
{
    if (over-tuned)
        tuningstep = tuningstep / 2;
    BPmax = BPmax + tuningstep;
    BPmin = BPmin - tuningstep;
    Re-evaluate Cmax;
    Re-evaluate Cmin;
    Sort the buffer pools by Ci = 1..N;
}

Figure 4.1: Pseudo-code of Tuning Component

to the other, and the tuning step size (tuningstep) is then halved to avoid endless loops or unacceptable buffer allocations. In each tuning interval, tuningstep buffer pages are transferred from the buffer pool with the minimum access cost (BPmin) to the one with the maximum access cost (BPmax). This tuning step size is kept fixed for each interval until over-tuning happens.

The resizing component takes the buffer allocation solution from the tuning component and changes the memory allocation for all the buffer pools accordingly. To automate this procedure, JDBC (Java Database Connectivity) [PM97] is used.
4.2 Goal-Oriented Dynamic Reconfiguration Algorithm

In this section we present our buffer management strategies to support goal-oriented database applications. This is the most important research issue explored in this thesis—how to achieve high-level performance goals for complex on-line transactions by tuning multiple buffer pool sizes in a DBMS. Goal-oriented buffer management strategies differ from non-goal-oriented approaches in that, instead of pursuing the best overall system performance, they make more effort to meet user-specified high-level performance goals for all the on-line transaction classes. Goal-oriented strategies compromise overall system performance in order to help a particular transaction class meet its performance goal. Our goal-oriented multiple buffer pool tuning strategy shares the same components as the global dynamic reconfiguration strategy presented in the previous section: sampling, tuning and resizing. The main difference is that the tuning algorithm in our goal-oriented strategy is much more sophisticated.

4.2.1 Architecture of Tuning

Similar to conventional goal-oriented buffer pool tuning strategies, our strategy uses tuning algorithms based on a response time estimator, a hit rate estimator, and a buffer allocation mechanism.
4.2 Goal-Oriented Dynamic Reconfiguration Algorithm

The architecture of the tuning algorithms is captured in Figure 4.2. The *hit rate estimator* estimates the buffer pool hit rate as a function of memory allocation. The *response time estimator* estimates a transaction’s response time as a function of the buffer pool hit rate. The *buffer allocation mechanism* works out appropriate tuning solutions for all the buffer pools based on the transaction classes’ estimated response times.

### 4.2.2 Hit Rate Estimator

The *hit rate estimator* uses Belady’s equation, which is described in Chapter 3, to approximate the hit rate for each buffer pool under a certain memory allocation. The equation is:

\[ HIT(SIZE) = 1 - a \times SIZE^b \]

where \( a \) and \( b \) are constants for each buffer pool.
4.2.3 Response Time Estimator

The response time estimator is based on the assumption that the high-level response time for a certain type of transaction is direct proportional to its corresponding disk access cost in the relevant buffer pools. This assumption is reasonable for data intensive applications.

An On-Line Transaction Processing (OLTP) application in a DBMS can be characterized by a set of transaction classes $T = \{T_1, T_2, ..., T_n\}$. An instance of a particular transaction class $T_i$ ($i = 1 ... n$) is an execution of that type of transaction. All the instances of a particular transaction class access the same set of data objects, which include elements of data tables, index tables, etc., and have the same high-level performance goals. For example, in the TPC-C benchmark, which is described in Appendix A, there are five different types of transactions: New Order transactions, Payment transactions, Order Status transactions, Delivery transactions, and Stock Level transactions. Each of these transaction types is composed of a series of operations upon various data objects in the database system, and each transaction type has a different consumption and access pattern for those data objects.

Database objects that are accessed by the instances of a particular transaction class $t \in T$ can be characterized as $O_t = \{O_1, O_2, ..., O_m\}$. Each instance of class $t$ performs a number of logical reads on each database object $O_i$ ($i = 1 ... m$). Each database object can be shared by multiple transaction classes. The average number of
logical reads of $O_i$ performed by instances of transaction class $t$ is denoted as $L_t(O_i)$. Ideally the value of $L_t(O_i)$ can be estimated by some specific database performance tool, such as the explain facility in DB2/UDB. In this thesis, we use the estimates from the TPC-C specification.

Database objects are buffered in specific buffer pools. We identify the set of buffer pools used by instances of a transaction class $t$ as $BP_t = \{BP_1, BP_2, \ldots, BP_b\}$. We use the notation $O_i \in BP_j$ to indicate that the database object $O_i$ is buffered in buffer pool $BP_j$. One buffer pool holds multiple database objects, while each database object is held in only one buffer pool.

We use the disk access cost model defined in Chapter 3 to estimate the disk access cost $r_i$ for a logical read in a buffer pool $BP_i$, which is formulated as:

$$r_i = (1 - H_i) \times (1 + (1 - p_i) \times d_i)$$

where $H_i$ is the hit rate of the buffer pool under current memory allocation, $p_i$ is the asynchronous write rate of the buffer pool under current memory allocation, and $d_i$ is the proportion of dirty pages in the buffer pool.

For an OLTP application with multiple transaction classes involved, we estimate the average disk access cost for transaction class $t$ as follows:

$$C_t = \sum_{i=1}^{b} \sum_{o \in BP_i} L_t(o) \times r_i$$

For each buffer pool $BP_i$ ($i = 1 \ldots b$) used by instances of class $t$, we sum the cost of the logical reads for each relevant database object $o$ buffered in $BP_i$. We assume the
average response time for a transaction class is direct proportional to its average disk access cost. The response time estimator part of our tuning algorithm can thus use each transaction class’s average disk access cost $C_t$ as its estimate for that transaction class’s average response time.

### 4.2.4 Buffer Allocation Mechanism

For a goal-oriented multiple buffer pool tuning strategy, it is important to determine when to trigger the buffer pool tuning algorithm, and how to tune the multiple buffer pools simultaneously. A buffer allocation mechanism tells us when and how to tune the buffer pools. If a goal violation is detected, the tuning algorithm is triggered. In each iteration of the tuning algorithm, we choose a source buffer pool, which donates a certain number of buffer pages, and choose a target buffer pool, which gains those buffer pages. By transferring buffer pages from the source buffer pool to the target buffer pool, the transaction class that violates its goal will improve its performance.

#### Achievement Index

An achievement index is used to determine whether a transaction class is achieving its predefined response time goal.

Various metrics can be used to address high-level performance goals (see Chapter 2). Average response time is a suitable choice for OLTP applications. Performance
goals can be defined by a database administrator in terms of an average response time for each transaction class. However, internally we use disk access cost to evaluate the performance of each transaction class, which is easier to estimate for a given buffer allocation. We assume a direct proportional relationship between the high-level response time and the low-level disk access cost. The higher the disk access cost for a transaction class, the larger its response time.

For the purposes of our algorithm, we must translate the response time goal into a disk access cost goal. For a certain type of transaction class \( T_i \in T \), the original average response time \( rt_i \) is captured using a database application tool at the database start-up. Its average disk access cost \( c_i \) is also estimated at the database start-up using the access cost estimation function defined in the previous section. Together, these give us the ratio \( q \) of response time to disk access cost for transaction class \( T_i \):

\[
q_i = \frac{rt_i}{c_i}
\]

We use this ratio to convert the performance goal of \( T_i \) from an average response time \( G_i(rt) \) to an average disk access cost \( G_i(c) \). We calculate \( G_i(c) \) as follows:

\[
G_i(c) = \frac{G_i(rt)}{q_i}
\]

We use an achievement index to indicate if a transaction class is meeting its performance goal. The achievement index \( AI_i \) for a certain transaction class \( T_i \) is
defined as:

\[ AI_t = \frac{GoalResponseTime}{ActualResponseTime} = \frac{GoalDiskAccessCost}{ActualDiskAccessCost} \]

If a transaction class is meeting its performance goal, the actual disk access cost is lower than or equal to the goal disk access cost, so the value of its achievement index is larger than or equal to one. On the contrary, if a transaction class is violating its performance goal, the value of its achievement index is less than one. The aim of our buffer allocation mechanism is to help every transaction class whose achievement index value is less than one.

**Mapping from Transaction Classes onto Buffer Pools**

In an OLTP application, each transaction class is composed of several operations on various database objects, which mainly include the data tables and the corresponding indices. In a database with multiple buffer pools, the data tables and indices are mapped onto the multiple buffer pools as part of the initial configuration of the database system. Each data table or index table can be held in only one buffer pool, while each buffer pool can hold many data tables or index tables.

Mapping from transaction classes onto buffer pools is therefore a many-to-many relationship, as shown in Figure 4.3. One transaction class can use multiple buffer pools, and each buffer pool can hold the pages of the database objects which are required by multiple transaction classes. We use a *bitmap* function to represent the
mapping from transaction classes to buffer pools as follows:

$$\text{bitmap}[T_i][B P_j] = \begin{cases} 
0, & \text{if } T_i \text{ does not use } B P_j, \\
1, & \text{if } T_i \text{ uses } B P_j.
\end{cases}$$

where $T_i$ is a transaction class and $B P_j$ is a buffer pool.

We also study the proportion of a buffer pool's disk access cost that can be attributed to each transaction class. These proportions are known as the transaction classes' \textit{weights} in each buffer pool. The weight that a transaction class carries for the activities in a given buffer pool is determined not only by the number of tuples it accesses, but also by the access pattern it uses. Data tables and indices in a database system are accessed by on-line transactions in different patterns. For example, tuples from some tables are accessed with uniform probability, while tuples from other tables are accessed with different probabilities. When pages that have higher access frequencies are kept in a buffer pool, fewer disk accesses occur. A specific function is defined
in the TPC-C benchmark to simulate such non-uniform access patterns [LD93].

The bitmap function and transaction weights for each buffer pool will help us choose source and target buffer pools for memory reallocation.

Choosing the Source Buffer Pool

The total number of buffer pages in the multiple buffer pools of a database system is fixed. In each iteration of our tuning algorithm, a source buffer pool is picked to donate a certain number of buffer pages. We choose as the source buffer pool the buffer pool that will be least affected by losing pages. We propose three variations of this selection process, namely the strict greedy scheme, the relaxed greedy scheme, and the global scheme [MLR99]. The three schemes differ only in the number of target transaction classes taken into consideration when reallocating buffer pool memory.

In the strict greedy scheme, we choose the buffer pool that has the least impact on the disk access cost of the transaction class whose achievement index is both lower than one and the lowest among all the transaction classes. Based on the achievement index $A_{T_i} (i = 1 \ldots m)$, for each transaction class $T_i \in \mathcal{T}$, we choose $T_t$ as the target transaction class if:

$$A_{T_t} < 1 \text{ and } A_{T_t} < A_{T_i} \text{ where } T_i \in \mathcal{T}, i \neq t.$$  

For each buffer pool $BP_i (i = 1 \ldots n)$, we observe the difference between the disk access cost for the target transaction class $T_t$ before and after the pages are moved.
4.2 Goal-Oriented Dynamic Reconfiguration Algorithm

The difference $\delta_{C_i}$ can be calculated as:

$$\delta_{C_i} = (C_i(size - \delta) - C_i(size)) \times w_t,$$

where $\delta$ is the number of the buffer pages lost, $C_i(size - \delta)$ is the access cost after losing buffer pages at $BP_i$, $C_i(size)$ is the access cost before losing buffer pages at $BP_i$, and $w_t$ is the weight that the target transaction class $T_i$ carries in $BP_i$.

We sort the values of $\delta_{C_i}$ for all the buffer pools in ascending order. The criterion for choosing the source buffer pool is to find the buffer pool that, after losing a certain number of buffer pages, makes the smallest difference in disk access cost with respect to the target transaction class.

In the relaxed greedy scheme, instead of finding a target transaction class with the lowest achievement index, we include all the transaction classes that violate their goals as target transaction classes. For the set of target transaction classes $T'$, $T_i \in T'$ if and only if $AI_i < 1$ ($i = 1 \ldots m$).

For each buffer pool $BP_i$ ($i = 1 \ldots n$), we calculate the difference between the disk access cost for all the target transaction classes before and after the buffer page donation. We sum the weights of all the target transaction classes in $BP_i$, and calculate the difference in disk access cost in $BP_i$ for all the target transaction classes as:

$$\delta_{C_i} = (C_i(size - \delta) - C_i(size)) \times \sum_{t \in T'} w_t.$$
where \( \delta \) is the number of the buffer pages donated, \( C_i(size - \delta) \) is the access cost after losing buffer pages at \( BP_i \), \( C_i(size) \) is the access cost before losing buffer pages at \( BP_i \), and \( w_t \) is the weight that target transaction class \( t \) carries in \( BP_i \).

We sort the values of \( \delta C_i \) in ascending order for all the buffer pools. We choose the buffer pool with the lowest value to be the source buffer pool.

In the global scheme, we choose the source buffer pool as the one which has the least impact on the disk access cost of all the transaction classes. The difference of the disk access cost before and after moving the buffer pages in a buffer pool \( BP_i \) is formulated as:

\[
\delta C_i = (C_i(size - \delta) - C_i(size))
\]

where \( \delta \) is the number of the buffer pages transferred, \( C_i(size - \delta) \) is the access cost after losing buffer pages at \( BP_i \), and \( C_i(size) \) is the access cost before losing buffer pages at \( BP_i \).

Choosing the Target Buffer Pool

The target buffer pool receives the buffer pages from the source buffer pool. We use a greedy approach to choose a target buffer pool that, if enlarged, will most decrease the disk access cost of the target transaction class or classes. The benefit can be estimated as follows: given the new size of the target buffer pool, use the hit rate estimator to get a new hit rate for the target buffer pool, then use the new hit rate
to estimate a new disk access cost and compare it with the old one. The larger the
difference, the better the benefit.

Corresponding to the three schemes for choosing the source buffer pool, we propose	hree variations for choosing the target buffer pool. Each scheme is analogous to the
corresponding scheme for choosing the source buffer pool.

In the strict greedy scheme, we observe the impact on the disk access cost for target
transaction class $T_i$ that has the lowest achievement index value which is smaller than
one. The difference at buffer pool $BP_i (i = 1 \ldots n)$ after gaining buffer pages is:

$$\delta_{C_i} = (C_i(size) - C_i(size + \delta)) \times w_t$$

where $\delta$ is the number of the buffer pages gained, $C_i(size)$ is the access cost before
 gaining buffer pages at $BP_i$, $C_i(size + \delta)$ is the access cost after gaining buffer pages
 at $BP_i$, and $w_t$ is the weight that the target transaction class $T_i$ carries in $BP_i$.

In the relaxed greedy scheme, we count in all the transaction classes $T_i \in T'$ whose
achievement index values are less than one. The disk access cost difference for the
target transaction classes $T_i \in T'$ at $BP_i (i = 1 \ldots n)$ is estimated as:

$$\delta_{C_i} = (C_i(size) - C_i(size + \delta)) \times \sum_{t \in T'} w_t$$

where $\delta$ is the number of the buffer pages gained, $C_i(size)$ is the access cost before
 gaining buffer pages at $BP_i$, $C_i(size + \delta)$ is the access cost after gaining buffer pages
 at $BP_i$, and $w_t$ is the weight that target transaction class $t$ carries in $BP_i$. 
4.2 Goal-Oriented Dynamic Reconfiguration Algorithm

In the *global* scheme, the performance change for all the transaction classes is estimated. The disk access cost difference of all the transaction classes at $BP_i$ ($i = 1 \ldots n$) is calculated as:

$$\delta_{C_i} = (C_i(size) - C_i(size + \delta))$$

where $\delta$ is the number of the buffer pages gained, $C_i(size)$ is the access cost before gaining buffer pages at $BP_i$, and $C_i(size + \delta)$ is the access cost after gaining buffer pages at $BP_i$.

For all the three schemes, we sort $\delta_{C_i}$ for each buffer pool in descending order. The larger the value, the bigger the improvement. We choose the buffer pool with the largest value to be the target buffer pool.

4.2.5 Summary of the Goal-Oriented Tuning Algorithms

The three parts of the tuning algorithm, namely the *hit rate estimator*, the *response time estimator*, and the *buffer allocation mechanism* are used in each algorithm iteration. Before the tuning algorithm is run, statistics are collected about the performance of each buffer pool with the help of the monitoring mechanism in a DBMS. The statistics collected for each buffer pool include the buffer pool size, the number of physical reads, the number of logical reads, the number of asynchronous writes and the number of writes. The statistics are collected at two points. One is the initial buffer pool state for the tuning algorithm and the other is a reference point, at which
all the buffer pools have sizes significantly different from their initial sizes. With
the statistics about the initial and reference states at hand, the hit rate estimation
function, the asynchronous write rate estimation function, and therefore the response
time estimation function for each algorithm iteration are available.

Each transaction class has a performance goal for average response time, which
is given by the database user. Based on the direct proportional relationship between
the response time and the disk access cost for a transaction class, we set up the
performance goal in terms of average disk access cost for each transaction class. With
both the target disk access cost and the current disk access cost for each transaction
class known, the achievement index value is calculated for each transaction class at the
initial buffer pool state. If all the achievement index values are larger than or equal
to one, which indicates that all the transaction classes are meeting their performance
goals at the starting point, no buffer pool tuning will be activated. If one or more
classes have an achievement index value that is lower than one, the tuning algorithm
is triggered.

At each iteration of the tuning algorithm we choose some target transaction classes
first. Once the target transaction classes are chosen, we use a greedy approach to
find the source and target buffer pools.

We choose the source buffer pool based on the disk access cost difference before
and after removing a certain number of pages. We choose the buffer pool in which
removing pages will least hurt the performance of the target transaction classes. We define a threshold value to guarantee that each buffer pool will always keep a certain number of buffer pages. If the size of a buffer pool shrinks to the threshold point after page donation, that buffer pool is not chosen to be the source buffer pool even if it ranks first in the ascending ordering of disk access cost differences. Instead we choose the first buffer pool that will not shrink below the threshold value.

We then choose the target buffer pool in an analogous way. Among all the buffer pools, the target buffer pool provides the biggest improvement in the disk access cost for the target transaction classes.

After both the target buffer pool and the source buffer pool are determined, a certain number (step size) of buffer pages are transferred from the source buffer pool to the target buffer pool. The disk access costs for all the transaction classes are re-evaluated under the new allocation. If any transaction class is not meeting its goal, a new tuning iteration begins. The step size in each iteration keeps fixed unless the performance of the target transaction classes starts to decline (i.e. until the disk access cost of the target transaction classes increases after the tuning). A performance decrease indicates the tuning was over-done. In this case we halve the step size and re-tune the buffer pools. The relationship between the step size and the speed of convergence of the algorithm will be shown in Chapter 5. The tuning algorithm stops when all the transaction classes are meeting their goals or, if some performance goal
is not achievable, when the step size for a tuning iteration approaches zero.

We propose three variations for tuning multiple buffer pools to meet the performance goals of all transaction classes. The main difference among the three variations lies in the criterion for selecting target transaction classes, which is the target of the performance improvement. In the strict greedy scheme, we choose the transaction class whose achievement index is the lowest among all the transaction classes that are failing to achieve their performance goals. In the relaxed greedy scheme, we choose all the transaction classes that are failing to achieve their goals. In the global scheme, we choose every transaction class. The three schemes are compared experimentally in Chapter 5.
Chapter 5

Experimental Results

A series of experiments were performed to evaluate the validity and performance of our goal-oriented multiple buffer pool tuning algorithms. In this chapter, we start by demonstrating the direct proportional relationship between the high-level response time and the low-level disk access costs of a transaction class. We then explore the relationship between tuning step size and the speed of convergence. Finally, experimental results of the three algorithm variations discussed in Chapter 4 are presented and compared.

5.1 Experimental Environment

All the experiments presented in this chapter were run using DB2/UDB Version 5 under Windows NT on an IBM PC Server 704 (PentiumPro 200). The machine was
configured with three 200 MHz processors, 256 MB of RAM and ten disks managed with RAID (level 0). Disk page size was 4K bytes, and 144 MB of memory (36,000 pages) was allocated to buffer pools. Four page cleaners were configured in the DBMS.

We used the TPC-C benchmark as the transaction workload simulator. The database schema and the transaction workload in the TPC-C benchmark are presented in the appendix.

5.2 Response Time Estimator

Our goal-oriented multiple buffer pool tuning algorithm is based on the assumption that, for each transaction class, the high-level response time is direct proportional to the low-level disk access cost. We first carried out an experiment to verify this relationship. We had three buffer pools for the database: \texttt{BP\_DATA1} for the data tables of the \textit{Warehouse, District, Stock, Order, New Order, Order-Line, and History} relations (the initial buffer allocation was 100 pages), \texttt{BP\_DATA2} for the data tables of the \textit{Customer} and \textit{Item} relations (the initial buffer allocation was 100 pages), and \texttt{BP\_INDEX} for all the index tables of all the relations (the initial buffer allocation was 35,800 pages). The initial estimated response time for the \textit{New Order} Transaction was 21.903 K disk access requests, and its performance goal was 16 K disk access requests. We found that, after the first 6 minutes, the TPC-C driver stabilizes (i.e. the average response time of each transaction class in each predefined interval stops
changing dramatically). For each transient buffer allocation given by our algorithm, we changed the size of the buffer pools accordingly and ran the TPC-C driver for 10 minutes. We used the average of the response time collected in the last 4 of those 10 minutes as the real response time of the transaction class under the specific buffer allocation. In Table 5.1 both the estimated response time (in disk accesses) and the real response time (in seconds) are listed. The average standard deviation of the real response time collected by the TPC-C driver after the first 6 minutes in each 10-minute run is 0.8292, which shows that the TPC-C driver runs stably. The buffer allocation in each tuning iteration is shown in the form "BP\_DATA1/BP\_DATA2/BP\_INDEX". The tuning step size was 800 pages per iteration.

The comparison of the estimated response time and the real response time for the New Order Transaction is shown in Figure 5.1. The experiment demonstrates that the assumption of a direct proportional relationship between the real response time and the disk access cost for a transaction class is reasonable.
5.3 Tuning Step Size

The tuning step size is the number of buffer pages that are transferred from the source buffer pool to the target buffer pool in each tuning iteration. In a reasonable range, the bigger the tuning step size, the faster the tuning algorithm converges.

We did an experiment to compare the convergence speed of our algorithm under different tuning step sizes. The relaxed greedy scheme was used. We had three buffer pools for the database: *BP_DATA1* for the data tables of the Warehouse, District, Stock, Order, New Order, Order-Line, and History relations (the initial buffer allocation was 12,000 pages), *BP_DATA2* for the data tables of the Customer and Item relations.
relations (the initial buffer allocation was 12,000 pages), and BP_INDEX for all the index tables of all the relations (the initial buffer allocation was 12,000 pages). The initial estimated response time for the New Order Transaction was 33.056 K disk access requests, and its performance goal was 26.5 K disk access requests.

We use an achievement index to evaluate how well each transaction class performs. The achievement index of a transaction class is the ratio of its response time goal to its real response time. An achievement index lower than one indicates a performance violation of the corresponding transaction class, which will trigger the tuning algorithm. Figure 5.2 shows the speed of convergence compared under different tuning step sizes.

The final buffer allocations in the form “BP_DATA1/BP_DATA2/BP_INDEX” and the corresponding disk access costs (for New Order Transaction) are shown in Table 5.2. The experiment demonstrates that the tuning procedure converges faster when the tuning step size is larger. The tuning step size has a significant impact on the speed of convergence of the algorithm, but not on the tuning results.

Table 5.2: Tuning Results with Different Tuning Step Sizes

<table>
<thead>
<tr>
<th>Tuning Step Size (pages)</th>
<th>Final Buffer Allocation (pages)</th>
<th>Final Estimated Response Time (K access requests)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>8600/4800/22600</td>
<td>26.498</td>
</tr>
<tr>
<td>400</td>
<td>8400/4400/23200</td>
<td>26.363</td>
</tr>
<tr>
<td>800</td>
<td>8000/4800/23200</td>
<td>26.475</td>
</tr>
</tbody>
</table>
5.4 Memory Allocation Criteria

With reasonable performance goals predefined for all the transaction classes, the tuning algorithm will not stop until the achievement index values of all the transaction classes are larger than one, in other words, all the transaction classes are meeting their performance goals.

We did a group of experiments to compare the tuning results from the three variations of our algorithm (the strict greedy scheme, the relaxed greedy scheme, and the global scheme), which are based on different memory allocation criteria. For this
5.4 Memory Allocation Criteria

group of experiments, we had four buffer pools in the database: BP_DATA1 for the data tables of Warehouse, District, Stock relations, BP_DATA2 for the data tables of Customer and Item relations, BP_DATA3 for the data tables of Order, New Order, Order-Line, and History relations, and BP_INDEX for all the index tables of all the relations. The initial buffer allocation was 6,000 pages each for BP_DATA1, BP_DATA2 and BP_DATA3, and 18,000 pages for BP_INDEX. We did three experiments using each tuning scheme, initially with one (the New Order Transaction), two (the New Order and Payment Transactions), and three (the New Order, Payment and Delivery Transactions) transaction classes violating their performance goals. The tuning step size was 400 pages.

5.4.1 Results from the Strict Greedy Scheme

The strict greedy scheme chooses the transaction class with the lowest achievement index value which is less than one as the target transaction class. In each tuning iteration, a certain number of buffer pages are transferred from the source buffer pool to the target buffer pool. The buffer reallocation attempts to give the best benefit to the target transaction class.

Results from the three experiments using the strict greedy scheme are shown in Figures 5.3, 5.4, and 5.5. The algorithm converges to a point where the achievement index of all the transaction classes are larger than one (i.e. all the transaction classes
are meeting their predefined goals). We found that the performance of all the transaction classes was influenced during the tuning procedure. In attempts to improve the New Order Transaction's performance, the strict greedy scheme hurts the performance of the Order Status Transaction and the Delivery Transaction. The BP_DATA1 buffer pool, which was heavily used by the New Order transaction class, gained buffer pages at the expense of the buffer pools used by the Order Status and Delivery transaction classes to improve the performance of the New Order Transaction. On the other hand, the Payment and Stock Level transaction classes benefited from the tuning because they shared buffer pools with the New Order target transaction class.

![Image](image.png)

Figure 5.3: Strict Greedy Scheme with One Violating Transaction
5.4 Memory Allocation Criteria

Figure 5.4: Strict Greedy Scheme with Two Violating Transactions

Figure 5.5: Strict Greedy Scheme with Three Violating Transactions
5.4 Memory Allocation Criteria

5.4.2 Results from the Global Scheme

In the global scheme, we choose all the transaction classes as target transaction classes. In each tuning iteration, the source and target buffer pools are selected to help all the transactions.

We did three experiments using the global scheme. The results are shown in Figures 5.6, 5.7, and 5.8. The global scheme successfully takes care of all the transaction classes; none suffers significantly in order to improve the performance of others. However, compared with the strict greedy scheme, it takes more tuning iterations for the transaction class that suffers the most to reach its performance goal.

Figure 5.6: Global Scheme with One Violating Transaction
5.4 Memory Allocation Criteria

Figure 5.7: Global Scheme with Two Violating Transactions

Figure 5.8: Global Scheme with Three Violating Transactions
5.4.3 Results from the Relaxed Greedy Scheme

In the relaxed greedy scheme, we choose every transaction class whose achievement index is less than one as a target transaction class. In each tuning iteration, the source and target buffer pools are selected to help all (and only) the transactions that are violating their performance goals.

We did three experiments using the relaxed greedy scheme. The results are shown in Figures 5.9, 5.10, and 5.11. Like the strict greedy scheme, the performance of some transaction classes decreased when the algorithm improved the performance of the transactions violating their goals. When the performance of these "victim" classes violates their goals, however, the relaxed greedy scheme stops harming them and starts helping them. This makes the algorithm converge faster.

5.4.4 Comparison

In general, the relaxed greedy scheme is the most robust and efficient version of our algorithm. The strict greedy scheme and the global scheme can be regarded as special cases of the relaxed greedy scheme. In a certain tuning iteration, if only one transaction class is failing to meet its performance goal the strict greedy scheme works in the same way as the relaxed greedy scheme. On the other hand, if all the transaction classes are violating their performance goals, the global scheme works in the same way as the relaxed greedy scheme.
5.4 Memory Allocation Criteria

Figure 5.9: Relaxed Greedy Scheme with One Violating Transaction

Figure 5.10: Relaxed Greedy Scheme with Two Violating Transactions
The experiments show that, in most cases, the relaxed greedy scheme is the fastest in converging to the point where all the transaction classes are meeting their pre-defined goals. Furthermore, the relaxed greedy scheme can meet more challenging performance goals than the other two versions. For each iteration in the strict greedy scheme, only one transaction class is taken care of; although the convergence speed for that violating transaction class is fast, the performance of other transaction classes is harmed. On the other hand, the global scheme tries to help all the transaction classes simultaneously without paying special attention to those that are failing to meet their performance goals, and therefore suffers from a slow convergence speed. The relaxed greedy scheme is a compromise between the strict greedy scheme and the
global scheme. It improves the performance of the violating transactions relatively quickly without seriously hurting other transactions.

5.5 Robustness of the Algorithm

To test the robustness of our multiple buffer pool tuning algorithm, we ran two groups of experiments with different initial buffer allocations, using the relaxed greedy scheme. The relationship between the buffer pools and the data and index tables is the same as that described in the previous section. For each group, we did three experiments with one (the New Order Transaction), two (the New Order and Payment Transactions), and three (the New Order, Payment and Order Status Transactions) transaction classes initially violating their performance goals.

In the first group of robustness experiments, we assigned 9,000 pages initially to each of $BP\_DATA1$, $BP\_DATA2$, $BP\_DATA3$, and $BP\_INDEX$. The results are shown in Figures 5.12, 5.13, and 5.14. The experiments show that the size of $BP\_INDEX$ is the most important for the violating transaction classes and it is the biggest winner among all the buffer pools. Since all the transaction classes share $BP\_INDEX$ for the indexes, all the transaction classes benefited when $BP\_INDEX$ gained buffer pages. With these experiments' initial buffer allocations it took more iterations for the algorithm to converge to the point where all the transaction classes met their
goals. Finally, although there were large differences between these initial buffer allocations and those of the experiments presented in the previous section, there was no significant difference in the convergence speed.

Figure 5.12: Uniform Initial Allocation with One Violating Transaction

In the second group of robustness experiments, the buffer pools were given a skewed initial allocation: 35,400 pages for \textit{BP\_DATA1}, and 200 pages each for \textit{BP\_DATA2}, \textit{BP\_DATA3} and \textit{BP\_INDEX}. The results are shown in Figures 5.15, 5.16, and 5.17. In terms of the relationship between the buffer pools and database tables, the initial buffer allocations in this group of experiments were the worst of all the experiments presented in this chapter. It left the largest room for performance improvement. Even under these rather extreme initial conditions, the algorithm was able to make all the
5.5 Robustness of the Algorithm

Figure 5.13: Uniform Initial Allocation with Two Violating Transactions

Figure 5.14: Uniform Initial Allocation with Three Violating Transactions
transaction classes achieve their performance goals.

Figure 5.15: Skewed Initial Allocation with One Violating Transaction

The experiments described in this chapter show that our algorithm's underlying assumption is reasonable, that our algorithm is robust, and that the "relaxed greedy" version of our algorithm is the most efficient, although all versions are successful.
5.5 Robustness of the Algorithm

Figure 5.16: Skewed Initial Allocation with Two Violating Transactions

Figure 5.17: Skewed Initial Allocation with Three Violating Transactions
Chapter 6

Conclusions

"Automatic resource management" is an attempt to move the burden of tuning low-level configuration parameters in a database management system (DBMS) from the database administrator (DBA) to the database system itself. With an embedded intelligent dynamic reconfiguration mechanism, a DBMS is able to automatically reallocate its resources to maintain acceptable performance or to achieve performance goals predefined by the DBA. In this thesis, we presented a goal-oriented multiple buffer pool tuning algorithm. This algorithm reallocates the size of the buffer pools in a DBMS to meet the high-level performance goals of on-line transactions.
6.1 Contributions

As with earlier self-tuning algorithms, our multiple buffer pool tuning algorithm is comprised of a response time estimator, a hit rate estimator, and a memory allocation mechanism. Our response time estimator assumes a direct proportional relationship between high-level response times and low-level disk access costs. Our hit rate estimator uses Belady’s equation to quantify the buffer pool hit rate under a certain buffer allocation. Our memory allocation mechanism chooses a source buffer pool and a target buffer pool and transfers a certain number of buffer pages between them.

The three versions of our algorithm (namely the strict-greedy scheme, the relaxed-greedy scheme, and the global scheme) differ from each other in how they select the target transaction class(es) whose performance is to be tuned. We have shown that the relaxed-greedy scheme is a good compromise between rapid convergence and the overall performance of all the transaction classes.

We have implemented our multiple buffer pool tuning algorithm and conducted a series of experiments to evaluate the performance of the algorithm with the TPC-C benchmark under DB2/UDB. Unlike all the previous algorithms, our algorithm relates the performance goals of transaction classes to the access costs for database objects (tables, indexes, temporary tables). As well, our algorithm breaks new ground in the following areas:

- Response time estimate. Our algorithm looks into the impact of dirty pages
and page cleaners, and proposes a more accurate estimate for the response time of a transaction class. The only factor that the other algorithms take into account when estimating response time is buffer pool hit rates.

- **Data sharing.** Our algorithm considers the relationship between the on-line transaction classes and the database they access. The different transaction classes weight the access costs for a given database object differently if that object is shared by multiple transaction classes. Data sharing, which is inevitable in a practical database system, is not studied in the dynamic tuning and fragment fencing algorithms.

- **Data-oriented model.** Our algorithm adopts a data-oriented model. It takes high-level performance goals for all the transaction classes from the DBA, and maps those goals to access costs for the relevant database objects in multiple buffer pools. This is superior to the dynamic tuning algorithm, which uses a buffer pool-oriented model that tunes the performance of buffer pools without considering how the performance of the buffer pools relates to the performance of real-world database applications. Our algorithm is also superior to the fragment and class fencing algorithms, which use a transaction class-oriented model. Those algorithms use only one buffer pool, and so less flexibility in balancing the performance among all the transaction classes is possible.
6.2 Future Work

The work presented in this thesis opens up several interesting areas for future exploration:

- **Types of performance goals.** In this thesis, high-level performance goals for the on-line transaction classes in the TPC-C benchmark are defined in terms of average response times. Later, we will investigate how to efficiently support a variety of types of performance goals, such as average, minimum, or percentile throughput.

- **Relationships among buffer pools.** In this thesis, we studied the relationship between transaction classes and the relevant buffer pools, and the tuning strategies are based on the knowledge of these relationships. During our work, we discovered that, in certain situations, there are relationships among buffer pools. For example, if a transaction class contains a query that requires a join of two tables that are held in different buffer pools, the size of both pools will need to be increased to improve the performance of the transaction class. In the future, it will be beneficial to explore and take advantage of such relationships in a tuning strategy.
6.2 Future Work

- **TPC-D Workload.** All the experiments presented in this thesis are based on the TPC-C benchmark, which models a medium complexity online transaction processing workload with multiple transaction types. TPC-D is another database benchmark, which is designed to model a decision support environment in which complex ad hoc business-oriented queries are submitted against a large database [TPC94]. Dynamic SQL queries are supported in TPC-D, and the queries access larger portions of the database than TPC-C. Queries in TPC-D have some characteristics that those in TPC-C do not have, such as multi-table joins, extensive sorting, grouping and aggregation, etc. It will be interesting to investigate buffer tuning strategies for the TPC-D workload.

- **Explain Facility.** In this thesis, the number of logical reads performed by a certain type of transaction class in each buffer pool is estimated from the TPC-C specification. We would like to explore a more general approach for the estimation with the help of the explain facility of DB2/UDB [CHA96].

- **Prefetch Mechanism.** In this thesis, we studied how the page cleaner mechanism of DB2/UDB affects performance. However, we ignored the role that the prefetch mechanism could play for improving the database performance. A prefetcher anticipates the use of data and index pages and retrieves them from disk into buffer pools before they are accessed. The impact of asynchronous reads which are performed by prefetchers will be studied in the future.
Appendix A

TPC-C Benchmark

The TPC-C benchmark is a database benchmark approved by the Transaction Processing Performance Council. TPC-C benchmark is designed to model a medium-complexity online transaction processing workload with multiple transaction types. It is patterned after an order-entry workload.

The logical database design of the TPC-C benchmark is composed of nine relations. Non-uniform access is specified within individual relations. The TPC-C logical database is summarized in Table A.1. In TPC-C, the overall database consists of a number of warehouses. In our experiments, we have 40 warehouses in the database (i.e. $W = 40$). Each warehouse is composed of ten districts, and each district has 3,000 customers. There are 100,000 items that are stocked by each warehouse. The Stock relation maintains the stock level of each item at each warehouse. Orders are
maintained in three relations: a permanent record of each order is maintained in
the Order relation, pending orders are maintained in the New-Order relation, and
an entry is made for each item ordered in the Order-Line relation. A history of the
payment transaction is appended to the History relation.

There are five transaction types in TPC-C: the New Order transaction, the Pay-
ment transaction, the Order Status transaction, the Delivery transaction, and the
Stock Level transaction. The five transactions access the relations with different ac-
cess patterns, which are summarized in Table A.2. In the table, U(x) indicates that
x tuples are chosen uniformly from the relation, NU(x) indicates that x tuples are
selected non-uniformly using a certain function defined in the TPC-C benchmark,
A(x) indicates that x tuples are appended to the relation, and P(x) indicates that x
tuples which were recently accessed are picked. On average, the New Order transac-
tion places an order for ten items from a warehouse, inserts the order, and updates
the corresponding stock level. The Payment transaction processes a payment from a customer and updates related data, such as the balances in the Warehouse, District and Customer relations. The Order Status transaction returns the status of a customer’s last order. The Delivery transaction processes ten pending orders, one for each district, with ten items per order, and deletes corresponding entry in the New-Order relation. The Stock Level transaction examines the quantity of stock for the items ordered by each of the last twenty orders in a district.

In our experiments we categorized each type of transaction as a transaction class (i.e. there are five transaction classes in our experiments). The external workload source for the database is modeled by a fixed set of simulated clients, each of which submits a stream of transactions. Forty clients were scheduled in our experiments.
Bibliography


