Towards Automatic Initial Buffer Configuration

by

Fei Yen Ku

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2003

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Buffer pools are blocks of memory used in database systems to retain frequently referenced pages. Configuring the buffer pools is a difficult and manual task that involves determining the amount of memory to devote to the buffer pools, the number of buffer pools to use, their sizes, and the database objects assigned to each buffer pool. A good buffer configuration improves query response times and system throughput by reducing the number of disk accesses. Determining a good buffer configuration requires knowledge of the database workload.

Empirical studies have shown that optimizing the initial buffer configuration (determined at database design time) can improve system throughput. A good initial configuration can also provide a faster convergence towards a favorable dynamic buffer allocation. Previous studies have not considered automating the buffer pool configuration process.

This thesis presents two techniques that facilitate the initial buffer configuration task. First, we develop an analytic model of the GCLOCK buffer replacement policy that can be used to evaluate the effectiveness of a particular buffer configuration for a given workload. Second, to obtain the necessary model parameters, we propose a workload characterization scheme that extracts workload parameters, describing the query reference patterns, from the query access plans. In addition, we extend an existing multifractal model and present a multifractal skew model to represent query access skew.

Our buffer model has been validated against measurements of the buffer manager of a commercial database system. The model has also been compared to an alternative GCLOCK buffer model. Our results show that our proposed model closely predicts the actual physical read rates and recognizes favourable buffer configurations. This work provides a foundation for the development of an automated buffer configuration tool.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Kenneth Salem. His guidance, insight and expressive clarity have served as an inspiration and have helped me tremendously during these past two years. This thesis would not be possible without his patience and advice. I would also like to thank my readers, Professors Tim Brecht and Grant Weddell for taking the time to review my thesis.

I would like to thank all those at the IBM Toronto Lab for answering my endless questions, especially Matt Emmerton, Ian Finlay, Sam Lightstone, Keri Romanufa, and Calisto Zuzarte. Thank you to NSERC and the IBM Centre for Advanced Studies for their financial support.

A special thank you to my family for their endless love, support and encouragement during my studies away from home. Finally, a heartfelt thank you to Gideon for his continuous patience, understanding and love over the years.

Trademarks

- DB2, DB2 Universal Database, AIX and IBM are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.
- Oracle is a registered trademark of Oracle Corporation.
- SQL Server is a registered trademark of Microsoft Corporation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Buffer pools are blocks of memory used in database systems to retain frequently referenced pages. Buffer pools help to reduce query response times and increase system throughput by exploiting temporal and spatial page locality to reduce the number of disk accesses. Configuring the buffer pools is a difficult task that involves determining the amount of memory to devote to the buffer pools, the number of buffer pools to use, their sizes, and the database objects assigned to each buffer pool. Determining an effective buffer configuration requires knowledge of the query reference patterns in the workload. Current buffer configuration methods use database reference traces, obtained after workload execution, as their source of query reference pattern information.

The task of configuring the buffer pools at database design time is called the *initial buffer configuration task*. The default buffer configuration assigns all database objects to one buffer pool. With minimal query reference pattern knowledge at database design time, the initial buffer configuration task typically results in the default configuration.

Configuring and tuning the buffer pools is a manual process that involves gathering query reference pattern information, analyzing this information to determine the workload behaviour and re-configuring the buffers according to the new reference patterns. Users often have neither the time nor the knowledge to search the large space of possible configurations in order to find the best buffer configuration for their workload.

Recent work in self-managing database systems has proposed dynamic buffer allocation algorithms to help simplify the buffer pool tuning process. Given an initial buffer configuration, these algorithms dynamically allocate memory among the buffer pools according to the queries' current buffer needs and pre-specified goals. However, these dynamic algorithms do not consider how the initial buffer configuration is determined, and they do not adjust the number of buffer pools or the objects' buffer pool assignments. Dynamic buffer allocation algorithms and previous buffer configuration methods have not considered how to automate the buffer configuration process.

In this thesis, we introduce a framework to help automate database buffer configuration. We develop an analytic model of the GCLOCK buffer replacement policy that estimates the expected total physical read rate of a workload running with a given buffer configuration. We expect that a decrease in the total physical read rate will result in a system throughput increase. The model can be used to evaluate a candidate buffer configuration for a given workload by predicting the expected total physical read rate. The candidate configuration resulting in the minimal predicted total physical read rate will be the recommended buffer configuration.

Our proposed buffer model requires specific workload input parameters. We have developed a workload characterization scheme that extracts the necessary workload parameters from the query access plans. Additional required parameters describing query access skew and database object statistics are collected from the database administrator and the catalog tables, respectively. In addition, we extend an existing multifractal model and present a multifractal skew model to capture the distribution of query page references. Together, the buffer model and the workload characterization scheme help automate the initial buffer configuration task.

The main contributions of this thesis are as follows:

- We propose a workload characterization scheme that extracts query reference patterns from the access plans. We also extend an existing multifractal model to represent a broad range of access skew using only a few parameters.
- We develop an analytic model of the GCLOCK buffer replacement policy that

considers more generalized weight assignments than an alternative GCLOCK model. Our proposed buffer model is able to closely predict the buffer occupancy and physical read rate of database objects in the workload.

- We present an extensive experimental validation of our buffer model against system measurements. In addition, we compare our buffer model predictions to an alternative GCLOCK Markov model. We also evaluate our model's predictive capability to distinguish between initial buffer configurations that improve system throughput and those that result in throughput degradation.

The remainder of this thesis is organized as follows. In Chapter 2, we present some background information on database buffer management. Specifically, we introduce the database buffer pools and the role of the database buffer manager. In Chapter 3, we review related work and describe previous buffer configuration methods and the only existing analytic GCLOCK buffer model, the approximate GCLOCK Markov model [NDD92]. In Chapter 4, we provide an overview of our initial buffer configuration automation process. In Chapter 5, we present our workload extraction algorithm and the multifractal skew model. We also explain how query page references are modelled. In Chapter 6, we describe our GCLOCK buffer model. In Chapter 7, we present the results from our experimental evaluation of the accuracy and effectiveness of our proposed buffer model. Finally, in Chapter 8, we summarize the main results of this thesis and make some suggestions for future work.

Chapter 2

Database Buffer Management

Modern operating systems provide a storage cache to satisfy read and write requests without having to physically read disk blocks. In this chapter, we present an equivalent concept in database systems known as *buffer management*, where the goal is to minimize the number of disk accesses required to satisfy query requests. We introduce the primary element, the database buffer pools, in Section 2.1. In Section 2.2, we provide an overview of the buffer manager and describe two commonly used buffer replacement algorithms.

2.1 Buffer Pools

In database systems, the cost of retrieving a page from disk is greater than the cost of retrieving it from memory. Satisfaction of query requests by pages in memory saves costly disk accesses. Modern database systems designate a portion of the memory area to buffer copies of database pages. This memory area is often called the *buffer pool* or simply the *buffer*.

Effective use of the buffer pools exploits temporal and spatial locality to retain frequently accessed pages. *Temporal locality* indicates that a currently accessed page will likely be accessed again in the near future. *Spatial locality* indicates that if a page is currently being accessed, then its neighbouring pages will likely be accessed in the near future. By exploiting temporal and spatial locality, the buffer manager is better able to predict and service query page requests. As the buffer

satisfies an increased number of page requests, this leads to fewer disk accesses and decreased query response times.

As incoming query requests occur, the buffer is searched first for the desired page. If the page is found in the buffer, this is known as a *buffer hit*. Otherwise, the page must be retrieved from disk and then placed into the buffer. This is known as a *buffer miss*. A *buffer manager* governs which data reside in the buffer pools. The buffer manager is responsible for (1) searching the buffer pools and determining if a page request is a buffer hit or a miss, and (2) allocating a limited number of buffer pages among a large number of competing queries. The role of the buffer manager is explored further in the next section.

2.2 The Buffer Manager

The buffer manager governs buffer pool usage and is an interface between the buffer pools and other database components. The buffer manager's goal is to minimize the number of physical reads and writes needed to satisfy query requests.

Each query request for a page of data is called a *logical reference*. A request that requires reading a page from disk is also called a *physical reference*. Physical references are a subset of logical references. The buffer manager handles logical references as follows.

The buffer manager searches the buffer for some desired page, p_i, requested by a query Q_i.[1] If p_i is found in the buffer, it is *fixed*, to prevent replacement during its use. The address of the buffer frame containing p_i is passed to the calling component that is evaluating Q_i. When Q_i has completed its work on p_i, it calls the buffer manager to *unfix* p_i, making p_i available for replacement.

If the search for p_i results in a buffer miss, a buffer replacement algorithm (implemented by the buffer manager) selects a candidate buffer page to discard to make room for p_i. If the candidate buffer page has been modified, it is considered *dirty*, and must first be flushed to disk. After writing the dirty page to disk, the buffer manager retrieves p_i from disk and places it in the buffer. The buffer manager

[1] All pages in a buffer pool are of equal size.

fixes and unfixes p_i to prevent replacement, as it did in the case of a buffer hit.

The amount of data stored and accessed in a database system is typically much larger than the buffer size. This causes the buffer pool to be a high contention resource. Since query requests compete for limited buffer space, the buffer manager must [EH84]:

1. Efficiently *search* the buffer for requested pages.
2. Effectively *allocate* buffer pages among its requestors.
3. *Minimize* the number of physical references for a given workload by implementing an effective buffer replacement policy.

2.2.1 Buffer Search

For every logical reference, the buffer manager searches the buffer for the desired page. The search process is a vital task that must be efficiently implemented to avoid excessive query wait times.

A sequential scan of the buffer is the simplest search strategy. With no assumptions of page ordering, the buffer manager checks each buffer page header until the desired page is found. For a buffer of size B, an average of $B/2$ pages are scanned for a buffer hit, and a worst case of B pages are scanned for a buffer miss. To avoid the costly process of scanning the entire buffer for each logical reference, some database systems implement a partial sequential search. A database administrator (DBA) specifies a *search limit* of m pages, indicating the maximum number of buffer pages to examine. A buffer pointer tracks the currently examined buffer page. The buffer pointer starts at the first buffer page, and for each unsuccessful page examined, the pointer is incremented to point to the next buffer page. This process continues until the desired page is found, or m buffer pages have been examined. If after scanning m pages, the desired page is not found, the buffer manager returns a buffer miss. The buffer miss is misleading since the buffer has not been exhaustively searched. A small value for m can lead to unnecessary physical reads, whereas large values of m can cause large query response times. A balance between these two cases must be achieved. At the next logical reference, the buffer manager begins its search

with the buffer pointer pointing to the buffer page immediately following the most recently examined page.

An alternative class of search strategies applies an indirect approach and uses accessory tables to improve the buffer search time. A *sorted table* contains an entry for each buffer page, sorted by page ID. The buffer manager can apply binary search techniques to the sorted entries to locate a desired page. For a buffer of size B, an average of $B/2$ pages must be examined to determine a buffer hit or miss. Creating a binary tree index can help reduce the search space to an average of $log_2 B$ pages. The sorted table approach suffers from the drawback that table updates are costly to maintain. An alternative approach is to implement a hash algorithm. A hash function transforms a page ID into a hash table offset, whose table entry contains the buffer address of the desired page.

2.2.2 Buffer Allocation

A buffer allocation scheme can be classified as *query oriented* or *object oriented*. In a query oriented allocation scheme, the buffer manager decides how many buffer pages to allocate to a particular query. In an object oriented scheme, the buffer manager decides how many buffer pages to allocate to a database object. Our work concentrates on the object oriented allocation scheme. The buffer manager provides different buffer configuration options to the user and/or DBA depending on the allocation scheme. The buffer manager requires that the user/DBA evaluate these options before it allocates any buffer pages.

Buffer Configuration Options

The buffer manager provides various configuration options to a DBA, depending on the implemented buffer allocation scheme (indicated in parentheses):

1. Specification of the total buffer size (query, object oriented)
2. Specification of the number of buffer pools to use (object oriented)
3. Specification of the size of each buffer pool (object oriented)

4. Assignment of each database object to a buffer pool (object oriented)

In an object oriented allocation scheme, the DBA must evaluate all four buffer configuration options to derive a buffer configuration that is used by the buffer manager for page allocation decisions. We first describe a guideline for calculating the total buffer size, followed by a brief description of the query oriented approach. We then discuss the object oriented allocation scheme and its associated configuration requirements.

Calculating the Total Buffer Size

In both allocation schemes, the total buffer size must be given before the buffer manager allocates any buffer pages to queries or database objects.

Intuition indicates that larger buffer pools lead to higher hit rates, which lead to improved system performance. The relationship between the buffer size and the hit rate varies with each workload. However, previous buffer allocation studies have shown the hit rate versus buffer size curve is generally concave [BCL96, TPK97]. This indicates that for small buffer sizes, increasing the buffer size will lead to large hit ratio improvements. As the buffer size increases beyond a *knee point*, the marginal hit rate benefit decreases. The maximum marginal hit rate occurs at the knee point. In limited memory situations, a DBA should select a total buffer size as close as possible to the knee point to capitalize on the large hit ratio gains. Having a total buffer size greater than the knee point will provide minimal hit ratio improvements, and the extra memory may be better utilized elsewhere in the system.

Query Oriented Scheme

Given a total buffer size of B pages, the buffer manager allocates the B pages among the running queries. The success of this buffer allocation scheme relies on the buffer manager exploiting the locality present in the queries' data references. The number of buffer pages to allocate to a query is based on the query's *working set*, which is the set of pages referenced by a query. The size of the working set and its reference types are both considered when deciding how many buffer pages

to allocate. For example, a query performing a sequential reference of n pages will likely receive only a few buffer pages since the reference locality is low. Whereas a looping reference of s pages will benefit greatly from a buffer allocation of s pages, since the looping set of s pages is referenced repeatedly. An evaluation of query oriented buffer allocation strategies is presented in Section 3.1.1.

Object Oriented Scheme

In an object oriented allocation scheme, the buffer manager decides how many buffer pages to allocate to a database object. Before this allocation can be done, a buffer configuration must be provided. This involves evaluating the four configuration options presented earlier. Once the total buffer size has been determined, the DBA decides how many buffer pools to use and the size of each pool. This is not an easy task. Usually an initial number of buffer pools is chosen, and this number can be refined depending if more or less buffer pools are required to help separate conflicting object reference patterns. Most buffer configurations use between two and four buffer pools.

In a multiple buffer pools environment, the database system of an object oriented allocation scheme requires that every database object be assigned to exactly one buffer pool. When the number of database objects is greater than the number of buffer pools, some objects share buffers. The database administrator performs the *assignment task* of mapping database objects to the buffer pools. An example is shown in Figure 2.1. When considering if two or more objects should share a common buffer, the database administrator evaluates each object's access patterns, from all queries, and determines whether commonalities exist among all the observed access patterns. If so, objects exhibiting similar reference behaviour are good candidates to share a buffer. Two common clustering heuristics are used: (1) group database objects that share similar reference patterns, and (2) cluster database objects of the same type, for example, assigning all indices to the same buffer. These heuristics attempt to isolate conflicting reference patterns and page types to avoid thrashing situations. If two conflicting query references share the same buffer, an infrequent query (e.g., a query with a large number of random accesses) may steal all the available buffer pages from a second more active query

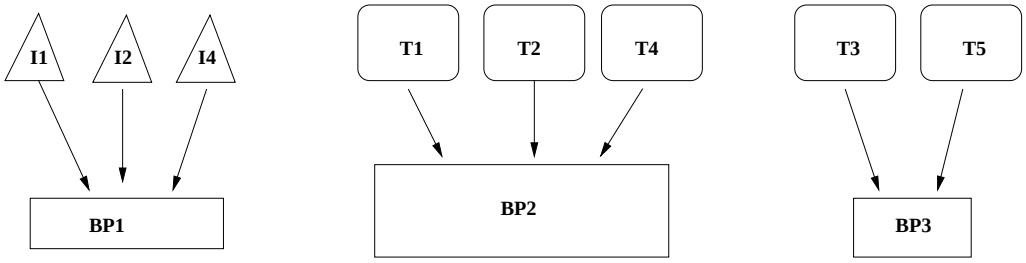

Figure 2.1: A sample buffer assignment. All indices are assigned to buffer pool BP1. Tables T3 and T5 share similar reference behaviour and have their own buffer BP3. The remaining tables are assigned to BP2.

(e.g., a query with looping references), causing the second query to experience frequent buffer misses.

Given a buffer configuration, the buffer manager can then allocate each buffer pool's pages to the objects assigned to that buffer. Consider a buffer assignment, a set of objects $O_i, i = 1..m$, each assigned to the buffer pool B_n of size S_n pages. Depending on the number of pages accessed, and the access patterns of each O_i, the buffer manager will allocate the S_n pages among the m objects to satisfy query requests and to exploit reference locality. Usually, buffer space is allocated dynamically to objects through the actions of the buffer manager's replacement policy. For example, if O_i is a table that is sequentially scanned, it will likely receive only a few buffer pages. The buffer manager does this to avoid flooding B_n with O_i's pages, since they will not be re-referenced in the near future, and to avoid displacing frequently accessed pages from other O_j objects, $j = 1..m, i \neq j$.

Experimental Studies

The buffer manager provides four buffer configuration options, which allow a DBA to customize the buffer pools for a given workload. We conducted a series of experiments to determine whether these configuration options can be used to reduce query response times and increase system throughput. The goal of our experiments was to determine whether initial buffer configuration mattered.

Our system consisted of DB2 v.7, 32-bit on AIX v.5. The machine was equipped

with 4 x 350 MHz processors, 24 x 72 GB external IBM Serial Storage Architecture (SSA) drives and 1 SSA controller. We ran a series of tests using a simulated TPC-C workload on an 11 GB database. Further experimental details such as workload specification and methodology are given in Chapter 7.

We ran several trials of the default, single buffer pool configuration using a total of 1.1 GB of buffer memory. Each trial was executed with an initial 15 minute ramp-up period. System throughput (transactions/min) and average transaction response time (RT) values were collected at the end of a 45 minute test duration for each trial. These values form the benchmark from which we will evaluate the performance of candidate multiple buffer configurations.

Our initial experimental results showed that separating objects that are heavily accessed and re-referenced results in modest performance gains. Results are shown in Table 2.1.

Table 2.1 shows that although the performance gains are small, each of the three non-default configurations help to improve the throughput, response time and the physical read rate. Our results are further supported by other researchers who have conducted similar experiments and have reported larger performance gains: (1) Levy, Messinger and Morris reported a 6% increase in throughput and a 29% decrease in the physical read rate [LMM96], and (2) Xu, Martin and Powley reported a 30% increase in throughput, a 23% reduction in the response time and a 55% reduction in the total physical reads using a TPC-C workload [XMP02]. Further details regarding these results are given in Section 3.3. Xu, Martin and Powley reported significant performance gains. A distinguishing difference between their experimental setup and ours (and Levy et. al) is the total buffer size used. Xu, Martin and Powley used a total buffer size of approximately 1% of the database size whereas we used a total buffer size that was approximately 10% of the database size. These experiments provide an indication that as the total buffer size decreases, buffer configuration leads to larger performance gains, since contention for the limited buffer memory increases. That is, buffer configuration becomes more important as the total buffer size decreases.

Given that the TPC-C workload is not one that we would have expected to benefit from a multiple buffer pool configuration (it has no large scans, no large

Configuration	Throughput	Mean RT (s)	Phys Reads/s
Default: One buffer pool	6124	0.74	2022
BP1: STK Tbl (495 MB) BP2: All remaining objects (605 MB)	6403 (+4.5%)	0.71 (-4%)	1957 (-3.2%)
BP1: STK and OLINE Tbls (710 MB) BP2: STK Idx (150 MB) BP3: All remaining objects (240 MB)	6257 (+2.2%)	0.72 (-2%)	1933 (-4.4%)
BP1: STK Tbl (495 MB) BP2: OLINE Tbl (182 MB) BP3: All remaining objects (423 MB)	6242 (+1.9%)	0.727 (-1.8%)	1969 (-2.6%)

Table 2.1: Results of our TPC-C candidate buffer configurations relative to the default configuration.

sorts, little use of temp space), these results provide an encouraging and positive indication of the performance gains that can be achieved through (multiple) buffer pool configuration.

2.2.3 Buffer Replacement Policies

For each logical reference, if the buffer is full and a buffer miss occurs, the buffer manager calls upon a *buffer replacement policy* to select the best victim for replacement. The replacement policy attempts to select the page with the lowest probability of a re-reference. Some replacement algorithms exploit temporal locality by using a page's reference history to predict its future reference behaviour. Recently and frequently accessed pages are prime candidates to be accessed again in the near future, and thus, will not be selected for replacement. Buffer pages that have been inactive for a long period of time are selected as victims. We describe two commonly used buffer replacement policies: LRU and GCLOCK.

Least Recently Used

The Least Recently Used (LRU) replacement policy exploits temporal locality by victimizing the least recently used buffer page, under the assumption that it will not be accessed in the near future. An LRU buffer can be implemented as a stack.

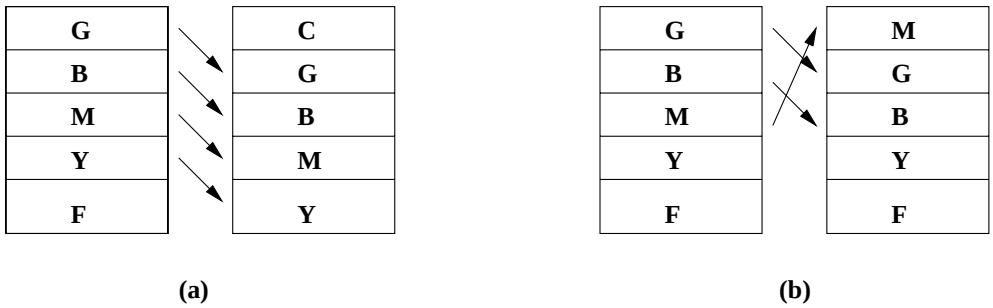

Figure 2.2: An LRU stack buffer. In (a), page C is not found in the buffer, it is fetched from disk and placed at the top of the stack. The least recently used page, F, is removed. In (b), requested page M is a buffer hit. Page M is placed at the top of the stack. Pages G and B each shift down one position.

If a newly requested page is not found in the buffer, it is fetched from disk, and placed at the top of the stack, pushing all other buffer pages down one position. The least recently used page, located at the bottom of the stack, is removed from the buffer. If a requested page is found in the buffer, it is removed from its current stack position j, and placed at the top of the stack. Existing buffer pages in positions 1 to $j-1$ each move down one position, all other buffer pages remain unaffected. An example LRU stack is shown in Figure 2.2.

Generalized CLOCK

The Generalized CLOCK (GCLOCK) replacement algorithm simulates LRU behaviour, but has a simpler implementation. We first describe the CLOCK algorithm, from which the GCLOCK algorithm is based.

In the CLOCK algorithm, buffer pages can be thought of as being arranged in a circular manner, with a pointer advancing among them, as shown in Figure 2.3. The algorithm associates a *reference bit* with each buffer page, indicating whether the page has been referenced during the last revolution of the clock pointer. On a buffer miss, the clock pointer circulates through the buffer pages, examining each page's reference bit. If a page with a reference bit equal to one is encountered, the clock sets it to zero, and advances to the next page. The first buffer page with a

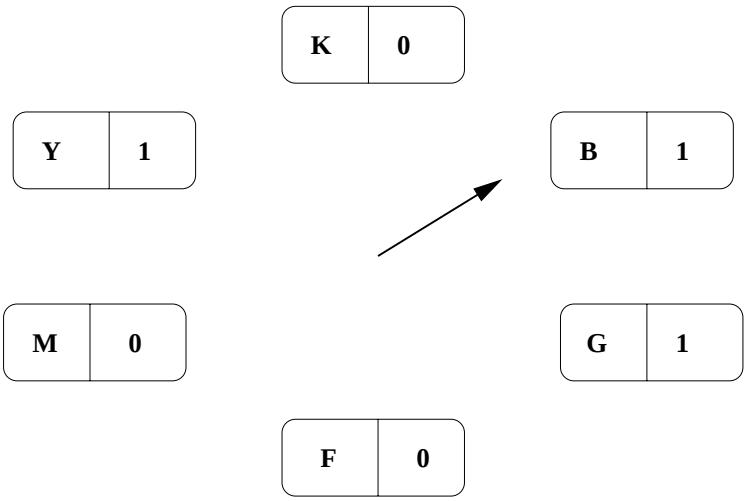

Figure 2.3: The CLOCK replacement policy. The clock pointer will reset page B and page G's weights to zero. Page F will be selected for replacement.

reference bit equal to zero is selected for replacement.

In the GCLOCK algorithm, the reference bit is replaced by a counter known as a *weight*. The initial weight is set when a page is first brought into the buffer. Re-references to buffer pages set the weight to a new value. Different weights can be assigned to different types of pages or to distinguish different reference patterns. For example, index pages may be assigned a higher weight than table pages.

On a buffer miss, the clock pointer cycles among the buffer pages, decrementing the weight of each page it examines, until a zero weight page is found. This zero weight page is selected for replacement. If the selected page is dirty, it is first written to disk. The requested page is brought into the buffer, with its weight set to some initial value. On the next buffer miss, the clock pointer begins its search from the page immediately following the page that caused the previous miss. A GCLOCK buffer page replacement example is shown in Figure 2.4. On a buffer hit, there is no advancement of the clock pointer.

Prefetching

We briefly mention the concept of prefetching due to its widespread implementation in modern database systems and its relationship to buffer replacement policies. A

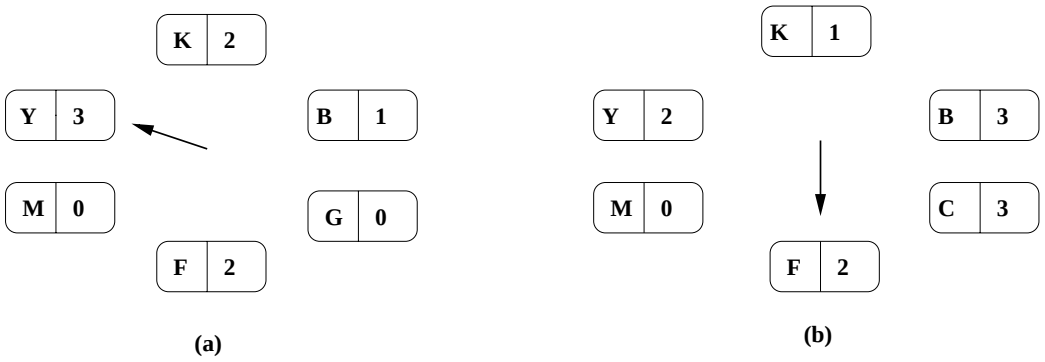

Figure 2.4: The GCLOCK replacement policy. In (a), the clock pointer will examine pages Y, K, and B, and decrement each of their weights by one. Page G will be selected for replacement. Page C is fetched into the buffer replacing page G. If page B is re-referenced before the clock pointer visits it again, the resulting buffer state is shown in (b) (page B is given a new weight equal to 3).

prefetching algorithm exploits spatial locality by not only fetching the requested page into the buffer, but also fetching its neighbouring pages. This is done to minimize data access latency. The number of physical reads is also reduced if a group of pages can be read in one prefetch versus a single page read during demand fetching.

Prefetching is most effective for sequential reads. Many modern database systems can detect that a query is sequentially accessing an object, and will automatically activate prefetching. The prefetch algorithm relies on the buffer manager and the replacement policy to provide the requested number of buffer pages. The number of pages to prefetch depends on the algorithm, but is usually no more than a few page blocks. An aggressive prefetching algorithm, which prefetches far in advance and reads a large number of pages, can increase query response times by replacing buffer pages that might have been re-referenced. Furthermore, a presumptuous algorithm may prefetch pages that are completely unused. Fortunately, most prefetch algorithms provide tunable parameters to limit how far in advance, and how many pages to prefetch. This helps to correct and avoid the undesirable situations described above.

Chapter 3

Related Work

In a database management system, the buffer manager is responsible for searching for available buffers, allocation of buffer pages to requestors, and the implementation of a page replacement policy. Early work in buffer management focused on improving the runtime buffer page allocation strategy by leveraging access pattern and skew information. Analytic models also captured access patterns and access skew to help predict buffer hit rates. More recent work has been directed towards automatic database tuning issues and has explored the multiple buffer pool configuration problem. There has been little work on modelling buffer replacement policies.

In this chapter, we present some of the previous work in these areas. In Section 3.1, we present a survey of buffer allocation strategies and introduce an access skew characterization method. In Section 3.2, we describe an existing analytic GCLOCK buffer model. Proposed solutions to the multiple buffer pool configuration problem are explored in Section 3.3.

3.1 Access Patterns in Buffer Configuration

Different types of workloads access database pages in different ways. For example, online transaction processing (OLTP) workloads consist of many simple transactions which read and update a few pages at a time. Online analytic processing (OLAP) workloads apply more complex queries to voluminous data, typically

performing large scans. The combination of possible query references against a database constitute the database *access patterns*.

The characterization of database access patterns is useful in the following settings:

1. In buffer allocation schemes, to effectively allocate buffer pages to queries in multi-query environments [SS82, CD85, NFS91, CY89, CR93].

2. To help predict buffer hit rates [DYC95, BCL96, XMP01].

3. In multiple buffer pool configuration, to help decide which database objects are best suited to share a common buffer pool [LMM96, XMP02].

Access patterns are usually obtained from reference traces [SS82, CD85, NFS91, CR93, DYC95, LMM96, XMP01, XMP02]. Although access plans also contain access pattern information, they are rarely used to extract this information. Analytic buffer models use access patterns to predict buffer hit ratios [DT90, NDD92]. These models help evaluate *what-if* scenarios by predicting hit ratios of candidate buffer configurations, without having to physically run the configuration itself.

In Section 3.1.1, we present a survey of buffer allocation schemes that exploit access patterns to help allocate buffer pages among competing queries. Section 3.1.2 discusses the importance of access skew in buffer configuration.

3.1.1 Buffer Allocation Schemes

As mentioned in Section 2.2.2, the buffer manager determines the number of buffer pages to allocate to each query or database object. Original buffer allocation strategies applied operating system memory management principles to database systems. They considered only buffer availability at runtime in their allocation decision. Later methods exploited database access patterns to selectively allocate buffer pages to queries and define query admission criteria [SS82, CD85, NFS91, CY89, CR93]. We present a survey of query oriented buffer allocation schemes in this section. We will use the notation adopted in [CR93] and let $[l_{min}, l_{max}]$ represent the minimum and maximum buffer allocations for a given query based on the observed reference pattern.

A primary allocation strategy is to assign enough buffer pages to a query to hold its *hot set* [SS82]. A hot set is a set of pages that have exhibited looping behaviour in the database reference trace. In order for a query to run efficiently, it must be allocated sufficient buffers to hold its hot set. If the number of available buffers is not sufficient to hold a query's hot set, then the query is not admitted for execution. Shortcomings of this approach include infinite waits and long running queries blocking short queries.

Chou and Dewitt improve upon the work of the hot set model by classifying the observed reference patterns into distinct categories [CD85]. Page references are classified as sequential, looping or random. The DBMIN algorithm allocates buffers based on these observed classifications. Each pattern is associated with a fixed number of pages called a *locality set*. The locality set is the estimated number of pages that a query needs to minimize page faults. DBMIN uses the following allocations:

- Sequential references: $[l_{min}, l_{max}] = [1, 1]$.
- Looping references $[l_{min}, l_{max}] = [s, s]$, where $s =$ (number of distinct looping pages). If s is greater than the buffer pool size, a most recently used (MRU) page replacement policy should be used to manage the buffer pool.
- Random references $[l_{min}, l_{max}] = [1, 1]$.

Experimental evaluation showed that DBMIN resulted in 7-13% better throughput than the hot set model. The major shortcomings of DBMIN are a lack of flexibility in its buffer allocation policy, and inefficient use of available buffers. For example, if $(s - 1)$ buffer pages are available, DBMIN will not admit a looping reference query for execution until s buffer pages become available.

The marginal gain algorithm, MG-x-y, was developed to overcome the drawbacks of DBMIN [NFS91]. MG-x-y, where x and y are user parameters, is a flexible buffer allocation strategy based on marginal gains and buffer availability at runtime. MG-x-y is similar to DBMIN except that it allows more flexible allocations, as follows:

- sequential reference: $[l_{min}, l_{max}] = [1, 1]$
- looping reference: $[l_{min}, l_{max}] = [x\% * s, s]$
- random reference: $[l_{min}, l_{max}] = [1, y]$

If the number of available buffer pages is less than a query's locality set size (s), but greater than $x\% * s$, the query is admitted for execution and allocated a set of buffer pages within $[x\% * s, s]$. In the looping and random cases, buffers are allocated up to s and y, respectively, as long as the expected marginal gain (the expected number of page faults reduced per extra buffer allocated) is still positive, and buffer pages are still available. Contention for buffer pool resources between queries is resolved by allocating pages to queries on a first-come, first-served basis.

MG-x-y has been shown to provide greater throughput, higher buffer utilization, and lower query wait times than DBMIN due to its flexible allocation strategy [NFS91]. The success of the algorithm relies on selecting optimal values for x and y. The parameters x, y are static for all queries, which is not a realistic assumption since different queries, although they may have the same reference pattern, can exhibit different fault behaviour. The necessity of selecting suitable parameters x, y is a drawback of this method.

Chen and Roussopoulos proposed a Faulting Characteristic Model (FCM) which collects page access patterns for sequential, looping and random references during query execution [CR93]. The page references are translated to reference strings. The FCM quantifies page fault characteristics for each reference string. Buffer allocation for each reference string is based on its page fault characteristics and current buffer availability. Using query feedback, the model updates the reference strings if changes in the access patterns are detected, and the corresponding buffer allocations are updated. The allocation algorithm (MGR) is based on FCM. Given n concurrent reference strings, MGR allocates buffers in proportion to their average marginal gain ratios.

Simulation results showed MGR provided an average 15-30% throughput improvement over MG-x-y. The advantage of this method is the incorporation of query feedback into the buffer allocation scheme, which provides more accurate

predictions than probabilistic methods and uniformity assumptions.

Chung and Yu proposed a global optimization technique that integrates buffer management and query optimization. The technique exploits access plans to consider the effect of buffers in the query optimization cost function [CY89]. An integer programming approach is used to select the best access plan and its associated buffer allocation. The objective function is determined as follows:

Let
$Q_i, i = 1..N_Q$ be a set of queries
$QS_{ij}, j = 1..n_i$, be the set of possible access plans for Q_i
$X_{ij} = 1$, if plan QS_{ij} is used, and 0 otherwise
D_{ij} be the number of disk pages read, if plan QS_{ij} is used
λ_i be the arrival rate for Q_i

Let the objective function $f = \sum_i \sum_j D_{ij} X_{ij} \lambda_i$.

The following constraints are enforced:

1. $\sum_j X_{ij} = 1, j = 1..n_i$. This states that exactly one plan is used for each query.

2. $\sum_j X_{ij} F_{ij} \leq B$, where F_{ij} is the buffer allocation to Q_i under plan QS_{ij}. This states that the total buffer allocation must be no greater than the total buffer size B.

3. $\sum_i \lambda_i \sum_j X_{ij} F_{ij} RT_{ij} \leq \alpha B$, where RT_{ij} is the average response time of query Q_i under plan QS_{ij}. This states that the average buffer usage for each query must be no greater than some fraction, α, of the buffer size. The parameter α is experimentally determined.

RT_{ij} must initially be estimated for each $Q_i, i = 1..N_Q, j = 1..n_i$. The optimization problem is solved iteratively using the initial RT_{ij} estimates. A queuing

model uses the solution set of query plans and associated buffer allocations to estimate a new set of response times. The integer programming model takes these new response times and uses them to iteratively solve for a new set of query plans and buffer allocations. This process repeats until no changes are observed in either the query plans or in the response times.

3.1.2 Access Skew Characterization

Database access patterns are generally skewed (non-uniform), meaning that some pages are accessed more frequently than others [LD93, DYC95]. Skewed access is an important consideration for buffer resource planning. First, skewed data access increases data contention for frequently accessed pages, also known as hot sets. The buffer manager fixes (i.e., locks) these hot pages during their use, making them unavailable to other requestors. This leads to increased response times due to longer wait periods. Second, skewed access increases the buffer hit ratio, since hot pages reside in the buffer longer, and repeated accesses to hot pages are satisfied by the buffer [DDY94]. Any buffer configuration methodology must carefully consider these two consequences of access skew and ensure that buffer resources are used effectively and that skew is modelled correctly.

It is common practice to model access skew by partitioning the database into hot and cold regions, such that the access probabilities of all pages within a partition are the same [DT90, NDD92, DDY94, DYC95, FMS96, WMCPF02]. The most widely used partition model is the 80-20 model, where 80% of the accesses reference 20% of the data, and the remaining 20% of the accesses reference 80% of the data. Faloutsos, Matias, and Silberschatz proposed a multifractal model to model skewed distributions [FMS96]. The multifractal model recursively divides a unit interval for k levels, where each division bisects the interval, resulting in 2^k total sub-intervals. Hot and cold regions are created by associating an access probability p to the hot region and an access probability $(1-p)$ to the cold region. This process continues recursively for each cold and hot region. Our access skew model, which is described in Section 5.2, is based on this multifractal approach.

Dan, Yu and Chung propose a *binary partitioning algorithm*, which character-

izes the access skew for randomly accessed pages given a reference trace [DYC95]. Assume the database needs to be divided into K partitions to achieve a desired level of buffer hit rate accuracy. The probability of accessing a page within a partition is uniform. A buffer hit rate versus buffer size curve is first derived from a trace driven simulation based on the LRU policy.

The binary partitioning algorithm begins with two partitions, a hot and cold partition, and their corresponding access frequencies and sizes. The hot partition is divided again into a hot and cold partition, and the new sub-partition access frequencies and sizes are re-computed, representing the new access skew. This process continues recursively for each newly created hot partition. The predicted model hit rates [DT90] are successively refined against the simulated hit rates by increasing the number of partitions in the database to K or until a desired level of hit rate accuracy is achieved.

3.2 Analytic Buffer Modelling

Analytic buffer models are used to predict some system performance measurement, e.g., buffer hit rate, system throughput. An accurate model helps to avoid costly and time consuming simulation tests. In this section, we describe the best known, and only, analytic model of the GCLOCK buffer replacement algorithm.

Nicola, Dan, and Dias propose an analytic model of the GCLOCK buffer replacement algorithm [NDD92]. The model assumes the Independent Reference Model (IRM) for page accesses. The objective of the model is to compute the hit probability of each partition, and then the overall buffer hit probability. The model input parameters are: the number of partitions P, the size of each partition s_p, the probability of accessing a partition p page, r_p, the total buffer size B, and the GCLOCK weight L_p for partition p. A brief overview of the model follows.

Consider a database with P partitions, where access to pages within a partition is uniform. When a page miss occurs, a victim page from the buffer is selected (weight=0) and the page causing the miss is fetched into the buffer. An initial weight, L_p, is given to the new page. If a request is fulfilled by an existing buffer page, then the weight for the page is reset to L_p.

Let n_p be the steady-state number of partition p pages in the buffer. The hit rate, h_p, and miss rate, m_p, for each partition can then be computed:

$$h_p = \frac{n_p}{s_p} \tag{3.1}$$

$$m_p = 1 - h_p \tag{3.2}$$

Then $h = \sum_{p=1}^{P} h_p r_p$ is the overall hit rate, and the overall miss rate $m = 1 - h$.

To compute h, n_p must first be determined. A simple approximate Markov model (Approx-GCLK) is developed to estimate n_p, for each p. The Markov chain represents the state of an arbitrary buffer frame at the instant of a random page request. A portion of the Markov chain is shown in Figure 3.1. The model describes the effect of the GCLOCK algorithm on a single buffer frame. State (p, i) indicates that a buffer frame contains a page from partition p with weight i, $1 \leq i \leq L_p$. The Approx-GCLK model has a total of $\sum_{p=1}^{P}(L_p + 1)$ states.

Let $n_{p,i}$ be the steady-state average number of buffer pages from partition p having weight i. We then have,

$$n_p = \sum_{i=0}^{L_p} n_{p,i} \tag{3.3}$$

Since all the buffer pages must sum to the buffer size B, $\sum_{p=1}^{P} n_p = B$. Let $n_0 = \sum_{p=1}^{P} n_{p,0}$ be the number of buffer pages with zero weight.

The Approx-GCLK model assumes the number of misses experienced during a complete cycle of the clock pointer is n_0. Suppose these n_0 pages are randomly distributed in the buffer. On a buffer miss, the clock pointer must decrement the weight of each page located between two zero weight buffer pages (the clock pointer stops when a zero weight buffer page is found for replacement). Thus, the probability of decreasing the weight of a random page is $1/n_0$. In Figure 3.1, the probability of transition from state (p, i) to state $(p, i-1)$ is m/n_0. The probability of replacing a page with weight zero is $(r_p m_p)/n_0$; which is the probability of bringing a new partition p page into the buffer $(r_p m_p)$ multiplied by the probability of selecting a random zero weight buffer page for replacement $(1/n_0)$. A similar explanation

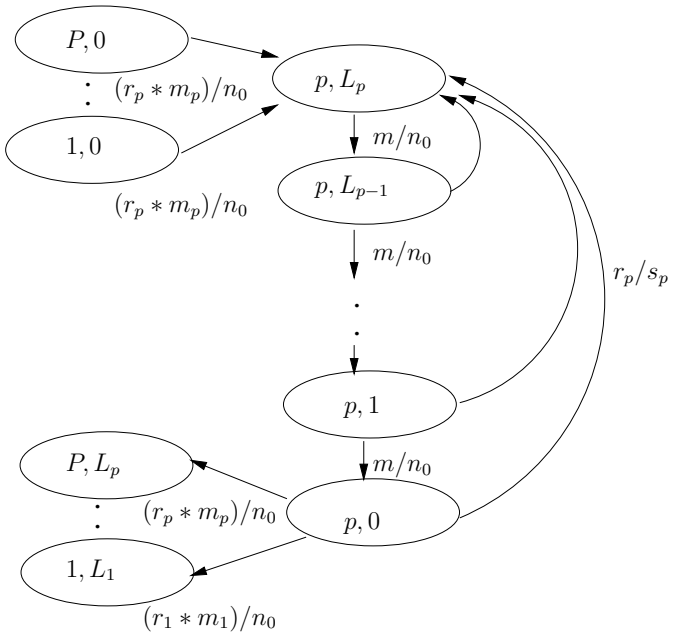

Figure 3.1: The Approximate GCLOCK Markov Model [NDD92].

applies to transitions from some state $(k, 0), 1 \leq k \leq P$, to a state (p, L_p), $k \neq p$, after a request has fetched a new page into the buffer, replacing an old zero weight buffer page. A random partition p page is accessed with probability r_p/s_p. Thus, if a requested page is already in the buffer, its weight is reset to L_p, and the transition probability from state $(p, i), 1 \leq i \leq L_p - 1$ to state (p, L_p) is r_p/s_p.

After deriving and solving the balance equations, a closed form equation for n_p is obtained:

$$n_p = s_p(1 - \frac{1}{(1 + \frac{n_0}{m}\frac{r_p}{s_p})L_p + 1}), 1 \leq p \leq P \tag{3.4}$$

Equation 3.4 can be solved iteratively to compute n_p for each partition, and consequently h_p.

A drawback of the simplified, Approx-GCLK model is it does not consider the distance of a buffer page from the clock pointer. It also assumes the average number of misses in a clock cycle is equal to the number of buffer pages with zero weight,

n_0. This leads to under estimates of buffer hit rate, especially in hot partitions, since pages with a zero weight may have been hit and upgraded to pages with a non-zero weight as the clock pointer traverses through the buffer. This causes over estimates of the number of buffer misses.

Nicola, Dan and Dias refine the Approx-GCLK Markov model by considering the distance between a buffer page and the clock pointer [NDD92]. The refined model corrects the inaccurate buffer miss count assumption by adding a new state variable j, to each state (p, i, j), to represent the distance between the page and the clock pointer. The solution to the simplified model is used as a starting point to iteratively solve the refined model for n_p.

Nicola, Dan and Dias' validation of the refined model showed good results against simulation tests. They further validated the model by comparing its hit rate predictions to hit rates from an optimal static buffer allocation. Initial results showed the model predictions were pessimistic compared to the static allocation's hit rates.

Drawbacks

The Approx-GCLK model considers weights to be assigned on a partition basis, i.e., each page in a partition p is initialized and reset to the same weight L_p. However, in practice, different queries can reference pages in the same partition in very different ways. One query may sequentially scan a set of partition p pages, while another query randomly reads another set of pages. These distinctions in query page reference behaviour dictate different buffer needs. To help exploit temporal locality, repeatedly referenced partition p pages can be tagged by the buffer manager with a high re-reference weight, whereas sequentially read pages can be tagged with a low re-reference weight. The Approx-GCLK model is unable to capture this specific query page reference behaviour and individual page weight assignment.

3.3 Multiple Buffer Pool Configuration

In this section, we discuss two previous solutions to the multiple buffer pool configuration problem; the bursty stream method [LMM96], and the clustering approach

[XMP02]. This problem consists of three sub-problems:

Given O_i database objects, $i = 1 \ldots N_o$ and K total buffer pages.

1. How many separate buffer pools, L, should be configured?
2. How do we allocate the K buffer pages among the L buffer pools?
3. How do we assign each of the O_i objects to the L buffer pools such that system performance is optimized?

In the following subsections, we discuss each of the proposed methodologies.

3.3.1 The Bursty Stream Characterization Method

The bursty stream characterization method models the requests for a given database object as a request stream [LMM96]. Given two objects, the interaction of their request streams is analyzed to determine its burst characteristics - does one stream have more bursts of consecutive page requests that dominate over the other stream at any particular time? Or does mixing two streams result in a fairly interleaved request pattern? Using this stream characterization, the method decides whether it is appropriate for two objects to share a buffer pool. The buffer assignment algorithm assigns objects to buffer pools and allocates buffer pages to the buffer pools. A brief description of the algorithm follows.

The input to the algorithm is the number of buffer pools, L, and a trace of the page requests for each object. The algorithm begins by computing the cumulative depth distribution for each object's request stream. The depth distribution represents the location statistics that describe where each request page is found in the buffer pool. In the second phase of the algorithm, given a page reference stream, P, for all objects, the mean burst lengths for *every* pair of objects, t_1, t_2, must be computed. The next step is to fully split the buffer such that each object is assigned its own buffer pool. The size of each buffer pool is calculated using dynamic programming. The optimal buffer pool sizes are those that maximize overall hit rate.

From this initial fully split state, the algorithm repeatedly selects two streams to superimpose. If the superposition of the two streams is deemed beneficial, their

associated buffer pools are merged into a single larger buffer pool. The number of streams is decremented, and the number of current buffer pools is decremented until it reaches L, at which point the algorithm terminates. The superposition criteria is based on a greedy approach which merges the pair of buffers that results in the lowest miss rate.

Experimental Evaluation

Levy, Messinger and Morris experimentally evaluated the quality of the algorithm's buffer assignments and allocations. The experimental workload consisted of six update transactions and one complex read-only query accessing a total of 26 database objects on a DB2 database system. The number of non-compulsory misses (avoidable physical reads) and transaction I/Os, and the overall transaction throughput were the performance metrics used to compare the single buffer pool configuration and the configuration recommended by the algorithm. Experimental observations indicated that using greater than three buffer pools provided only marginal gains. Results from running the recommended three buffer configuration over the single default buffer configuration showed:

- 11.4% decrease in non-compulsory misses
- 29% decrease in the average number of physical I/Os per transaction
- 6% increase in overall transaction throughput

3.3.2 The Clustering Approach

Xu, Martin and Powley propose a clustering approach to solve the problem of assigning database objects to the buffer pools [XMP02]. They present a list of buffer partitioning heuristics:

- Separate data and indices into their own buffer pools
- Separate large tables used in sequential scans
- Separate frequently used small tables into their own buffer

- Separate temp tables
- Separate tables that must be accessed quickly and repeatedly

The clustering algorithm defines a feature vector for each database object. An object's feature vector quantifies that object's access pattern. Features include: relative size, relative access rate, read/write mix, sequential, re-reference and random read rates. The idea is to group objects with similar access patterns into one cluster and assign the cluster to a buffer pool. Similarity is based on an object's access pattern features.

The features allow for a similarity measure to be defined by giving different weights to different features and clustering based on these weights. Three clustering algorithms were evaluated: K-Means, Partitioning Around Medoids, and Divisive Hierarchical, with K-Means proving to be superior. The number of buffer pools, L, is pre-specified. Once the objects are clustered into L groups, and each group is mapped into its own buffer pool, the Dynamic Reconfiguration algorithm [MLZRP00] is used to size each buffer pool.

Experimental Evaluation

Xu, Martin and Powley tested the clustering algorithm's proposed configurations against a random configuration (randomly assign objects to buffer pools), an expert configuration, and the default single buffer pool configuration using a TPC-C workload. The experiments used database traces to obtain the feature vectors and a total of three buffer pools were used. The evaluation measurements were the system throughput, the percentage of physical reads, and the overall weighted response time (WRT), which was determined by weighing the transaction response times by the respective transaction frequencies. The K-Means clustering algorithm was used in the experiments. Three candidate buffer configurations were evaluated:

- **Equal Weight Scheme**: Read/write, sequential, re-reference and random features are weighted equally.
- **Heavy Read/Write Scheme**: Read/write patterns are favoured over sequential, re-reference and random access patterns.

Configuration	WRT	Throughput Gain	% Physical Reads
Equal Weight	-20.9%	27.6%	-46.1%
Heavy Read/Write	-20.3%	26.3%	-54.8%
Access Pattern	-22.7%	30.7%	-55.7%

Table 3.1: Results of candidate buffer configurations relative to the default configuration.

- **Access Pattern Scheme**: Sequential, re-reference and random accesses are favoured over read/write accesses.

The experimental results showed that all three configurations out performed the default buffer configuration; details are listed in Table 3.1. The Equal Weight and Access Pattern Schemes resulted in higher throughput and lower overall response times than the random configuration. The Access Pattern Scheme was superior over the other two schemes, and matched the performance of the expert configuration.

3.3.3 Comparison

The bursty stream method and the clustering method apply distinct approaches to solving the multiple buffer pool configuration problem. The bursty stream model applies a bottom-up approach in which buffers are merged pair-wise until the desired number of buffer pools is reached. The clustering approach applies a top-down method where objects are logically associated with a single buffer pool and the algorithm selectively partitions the buffer according to the object clusters. The bursty stream method suffers from the drawback that if there are many database objects, it is a very time consuming process to merge from the initial fully split state to a few buffer pools.

In both algorithms, the target number of buffer pools must be pre-specified. This number is used to provide an algorithm termination condition. The fact that neither algorithm automatically determines the number of buffer pools also hints at the difficulty of selecting this number. Buffer pool sizes are computed using

external methods based on maximization of buffer hit rates.

3.4 Discussion

Multiple buffer pools have the potential to provide system performance improvements when they are tuned correctly. Having an effective initial buffer configuration provides users with other buffer startup options instead of relying on the default single buffer pool. A workload running with an effective initial multiple buffer pool configuration can outperform a single buffer configuration, as we have seen with previous experimental results. In goal-oriented algorithms [BCL93, MLZRP00], a good initial allocation can provide faster convergence towards a favourable dynamic buffer allocation.

Section 3.1 provided a summary of how access patterns and access skew are used in buffer configuration. To achieve effective buffer allocations and accurate buffer hit rate predictions, access patterns and skew must be considered. In most of the methods we have presented, access patterns and access skew are extracted from database reference traces. In addition, the bursty stream and clustering algorithms described in Section 3.3 provided buffer assignments and allocations based on reference traces. At database design time, when trace data is not yet available, an alternative source of access pattern input is required to determine a good initial buffer configuration. Access plans from the query optimizer can be exploited to provide this access pattern information.

We propose a novel workload characterization scheme that extracts access pattern information from the access plans. In addition, we apply and extend the multifractal model to model query access skew. The model allows specification of access skew for particular database objects, if this information is known, and assumes a default skew otherwise.

Previous configuration methods have proposed heuristics and algorithms to address a few of the buffer configuration options. However, tuning and re-configuring the buffers still remains a manual and difficult process. Previous buffer configuration methods have not considered automating the buffer configuration task.

The GCLOCK buffer replacement policy has been known to effectively simulate LRU behaviour. It is simpler to implement than LRU and it exists in many commercial database systems. We propose an analytic model of the GCLOCK buffer replacement policy to help automate the buffer configuration process and to address the drawbacks of the Approx-GCLK Markov model. Our buffer model assumes pages are referenced independently by queries. Our model's objective is to compute the total physical read rate, given a buffer configuration, query arrival rates, and query access patterns.

To address the per partition weight assignment limitation of the Approx-GCLK model, our proposed GCLOCK model adopts a more generalized approach where weights are assigned to individual pages based on the page type and the query type. This approach provides two benefits:

1. It provides greater flexibility to distinguish among individual page references and exploit temporal page locality, while still providing the option of a partition based weight assignment, if required.

2. It is more realistic - not all pages in a partition are referenced in the same manner by different queries. Thus, the pages in a partition should not all be reset to the same weight.

Together, the proposed workload characterization scheme and analytic buffer model help to automate the initial buffer configuration task. Our proposed techniques are introduced next.

Chapter 4

Initial Buffer Configuration Overview

In this chapter, we introduce the main components of our initial buffer configuration methodology. We propose two novel techniques that facilitate initial buffer configuration; a workload characterization scheme and an analytic GCLOCK buffer model. We begin by discussing the various workload inputs needed to generate a suitable workload characterization. In Section 4.2, we present an overview of our workload characterization scheme, which extracts object access patterns from the query access plans. Our proposed buffer model, described in Section 4.3, uses the workload characterization to evaluate the quality of a given buffer configuration.

4.1 Workload Input

Figure 4.1 shows an overview of our framework. The *workload query declaration* input consists of static SQL query statements. These statements may be derived from a DBA, users or application programs. At the time of initial buffer configuration, dynamic query statements are not available, so we rely on static queries to obtain reference pattern information. These static query statements are passed to the query optimizer.

The query optimizer parses the queries for semantic correctness and may re-write

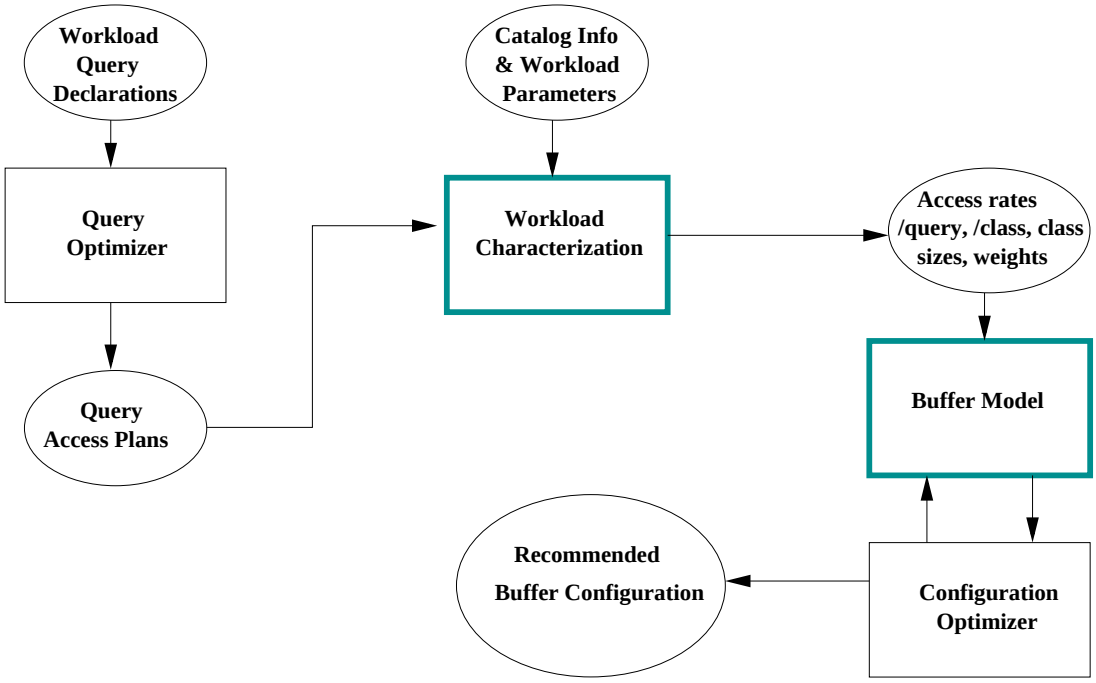

Figure 4.1: Our initial buffer configuration framework. The notation */x, /y*, indicates a quantity is calculated *per x, per y*. For example, access rates are calculated per query, per class.

the query in an equivalent form that facilitates cost based optimization. A query may be executed in many ways and still generate the same result, but the expended resources can vary greatly. A cost-based optimizer generates a collection of possible query execution plans called *access plans*, and associates an estimated cost with each plan. The estimated cost considers factors such as the physical database design, access methods, object access statistics, caching, and CPU resources. The optimizer will select the access plan with the lowest cost.

A workload is represented by a set of queries. The chosen access plans contain useful information indicating how the queries will be executed. Specifically, they indicate which objects will be accessed, the estimated number of pages accessed per object, and the anticipated access methods, e.g., sequential scans or random references. The information in the chosen set of access plans provides an indication of the workload behaviour. Our workload characterization scheme capitalizes on this information and defines it in a manner that is useful for initial buffer configuration.

In addition to the query access plans, *workload parameter* information and the *system catalogs* are required inputs to the workload characterization scheme. The system catalogs provide information on database object sizes and types. Query arrival rates and access skew information are the additional workload parameters (provided by a DBA) that are needed to derive an accurate characterization of the given workload.

4.2 Workload Characterization

Given the query access plans, query arrival rates, object attributes, and access skew inputs, the characterization scheme generates values for the input parameters of the buffer model. Figure 4.2 shows an overview of the workload characterization process. Our characterization scheme consists of four tasks:

1. Given the query reference types and object types, determine a GCLOCK weight assignment scheme for each combination of a workload query and a database object.

2. Given the query access plans, extract the object access patterns from the

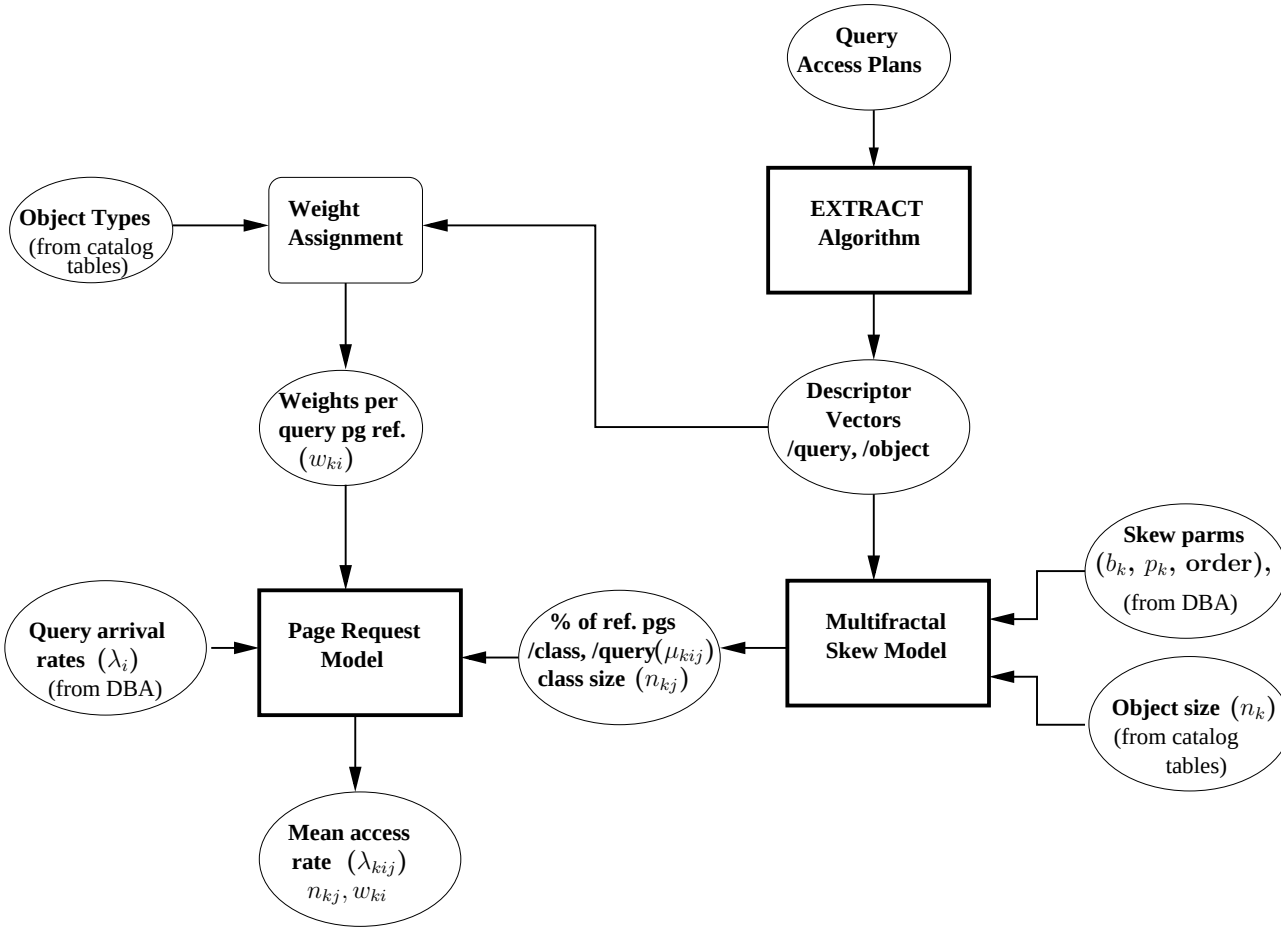

Figure 4.2: Overview of our workload characterization process.

Symbol	Description
$\mathbf{w_{ki}}$	The GCLOCK weight to an object k page from a query i reference.
$\mathbf{p_k}$	Access probability to an obj. k hot class page; range [0.5,1); default = 0.8.
$\mathbf{b_k}$	Fraction of obj. k's pages to allocate to a hot class; range (0,0.5]; default = 0.2.
order	The number of class division iterations; order ≥ 0; default = 2.
$\mathbf{n_k}$	The number of pages in object k.
$\mu_{\mathbf{kij}}$	The utilization factor, the proportion of pages in class kj referenced by query i.
$\mathbf{n_{kj}}$	The number of pages in class kj.
$\lambda_{\mathbf{i}}$	The arrival rate of query i.
$\lambda_{\mathbf{kij}}$	The reference rate to a class kj page by query i.
$\lambda_{\mathbf{kj}}$	The reference rate to a class kj page from all queries.

Table 4.1: Workload characterization and buffer model symbols.

plans to produce a *descriptor vector* per query, per object.

3. Given the skew parameters, object sizes and the per query descriptor vectors for each object, develop a skew model to capture the access skew in query references. The skew model divides each object into *classes*, with uniform access probability within each class. The skew model determines the proportion of pages referenced per class, per query, and the size of each class.

4. Given the proportion of referenced pages per class, per query, and the query arrival rates, model the page request arrival process, and determine the mean access rate per class from each query.

Table 4.1 shows a list of of symbols that we use in our workload characterization scheme and buffer model.

GCLOCK Weight Assignment

Weight assignments are database system specific. The underlying database system implements a weight assignment policy that is used in the GCLOCK replacement algorithm. The weight assignment module shown in Figure 4.2 should assign weights to query/object combinations that correspond to those used by the database system whose buffer manager is being modelled. Since our GCLOCK model is very

general, the weight assignment model has the flexibility to assign weights based on the type of object being referenced or on the type of access (sequential or random) required by the query. This depends on what is actually done by the database system that is being modelled.

Access Pattern Extraction

Given the query plans, the EXTRACT algorithm estimates the number of pages each query operator references per object. For simplicity, we assume each object is referenced by at most one operator in a query plan, although this is not a fundamental limitation of the EXTRACT algorithm. The EXTRACT algorithm performs a depth first traversal of the access plan to extract the number of referenced pages and the reference type of each operator. We define a *descriptor vector* for each operator accessing an object. The descriptor vector consists of two attributes: the number of referenced pages and the reference type. The collection of descriptor vectors, one per query, for each object characterizes the access patterns for that object. Further details of the access pattern extraction method are given in Section 5.1.

Multifractal Skew Model

In general, accesses to database pages are not uniform. Non-uniform page access, where some database pages are accessed more frequently than others, is referred to as *access skew*. The EXTRACT algorithm estimates the number of pages referenced in an object k, and represents this information in a set of descriptor vectors. This estimate, however, does not indicate which pages are actually referenced. We assume that these query page references are skewed. We introduce a multifractal model of this skew. The multifractal skew model divides each object's pages into hot and cold *classes* according to a given set of skew parameters, b_k, p_k and $order$. Within each class, access is assumed to be uniform. The skew model uses the descriptor vectors to model the distribution of the queries' references over the hot and cold class pages of the object k.

The skew parameters may be pre-specified by a database administrator. Otherwise, the default values shown in Table 4.1 are assumed. The skew model deter-

mines μ_{kij}, the proportion of pages referenced in the j^{th} class of database object k by query i, and the size of each class, n_{kj}, $\forall j = 1..2^{order}, i = 1..Q$. We discuss the multifractal skew model in greater detail in Section 5.2.

Page Request Arrival Process

Given the utilization factor, μ_{kij}, and the arrival rate, λ_i, of each query i, we model the arrival of requests to each class kj page as a Poisson process. Each page's requests are modelled independently of the requests to other pages. Assuming uniform access probability within a class, we first calculate λ_{kij}, the expected reference rate to a class kj page, per query. We then aggregate over all queries to determine the mean access rate to a class kj page, λ_{kj}. Section 5.3 gives further details on our model of page requests.

4.3 The Buffer Model

Given the expected page reference rates, class sizes and a weight assignment policy from the workload characterization scheme, and a buffer configuration from the configuration optimizer, our analytic model of the GCLOCK buffer replacement policy estimates the total physical read rate for the given workload executing under the given buffer configuration. The estimated physical read rate provides a suitability measure for the buffer configuration. Buffer configurations with low physical read rates are more appropriate for the given workload than configurations with high read rates, since this indicates that the buffer pools are doing a better job of satisfying page requests. Further details of our GCLOCK buffer model are presented in Chapter 6.

The final component in our framework, the *configuration optimizer*, is an input and output source for our buffer model. The configuration optimizer supplies the buffer model with candidate initial buffer configurations for evaluation. A given buffer configuration includes: the total buffer size, the number of buffer pools, and their respective sizes and object assignments. In turn, our buffer model returns a physical read rate count for each given buffer configuration indicating the quality of

the configuration under the current workload. It is up to the configuration optimizer to decide whether to continue supplying initial configurations to the model, or terminate the process by selecting the configuration with the lowest physical read rate as the recommended buffer configuration.

We are currently manually implementing the configuration optimizer tasks described above. The development of a search algorithm to generate candidate initial buffer configurations is part of our future work.

Chapter 5

Workload Characterization

In this chapter, we describe our workload characterization scheme, which extracts the workload parameters needed by our buffer model to evaluate a given buffer configuration. Specifically, the workload characterization scheme supplies the buffer model with: a query reference rate, one per query, for each object class; all class sizes; and a weight assignment policy.

As discussed in Chapter 4, the underlying database system uses the query reference types and object types to select a weight assignment policy. Since the workload characterization scheme simply mimics this policy, we will focus on how the reference rate and class sizes are calculated. We first describe an access pattern extraction technique that infers the reference type and the number of referenced pages from the query access plans. We then present a multifractal skew model that is capable of generating a broad range of access skew based on a few parameters. Finally, we show how page requests are modelled.

5.1 Access Pattern Extraction

The combination of all query accesses on an object determines that object's *access patterns*. Access patterns can be defined by two attributes: (1) the reference type, e.g., sequential or random access, and (2) the number of referenced pages. Our extraction method infers object access patterns from information in the query access plans.

5.1.1 Query Access Plans

A query access plan describes the execution strategy for a query. Access plans can be represented in a tree-like form in which the leaf level nodes are objects and the internal nodes are *operators*. (An internal node may also be an object such as a temporary (TEMP) table.)

The query plans we describe in this section are derived from the DB2 database system. Specifically, the IO_COST and sequential detection operator attributes exist in DB2 query plans. However, query plans from other commercial database systems, such as Oracle and SQL Server, also contain similar operator attributes. Our extraction method can be applied to any set of database query plans that contain similar IO_COST and sequential detection attributes to those in DB2.[1] A sample access plan for the SQL query *Q1* is shown in Figure 5.1.

Q1:
SELECT C.name, S.ID, S.name
FROM Students S, Courses C
WHERE C.ID = S.CourseID [2]

Connections between the nodes represent data flows. The tuples returned at a lower level node are passed as an input stream to the parent node. The query execution plan shown in Figure 5.1 may be described as follows:

- The query processor sequentially scans the COURSES table and each tuple it reads is passed to the parent operation, the Nested Loop Join (NLJ).
- For each COURSES record received, the NLJ operation passes the C.ID to the index scan operator.
- The index scan reads a secondary STUDENTS index on CourseID, and returns a set of row IDs to the FETCH operation. Each row ID identifies a student record whose CourseID matches the given C.ID.

[1]Minor modifications to the extraction algorithm may be required when calculating the I/O costs of an operator depending on whether the I/O costs in the query plans are cumulative values or not.

[2]For each course, *Q1* returns all the students (their IDs and names) enrolled in that course.

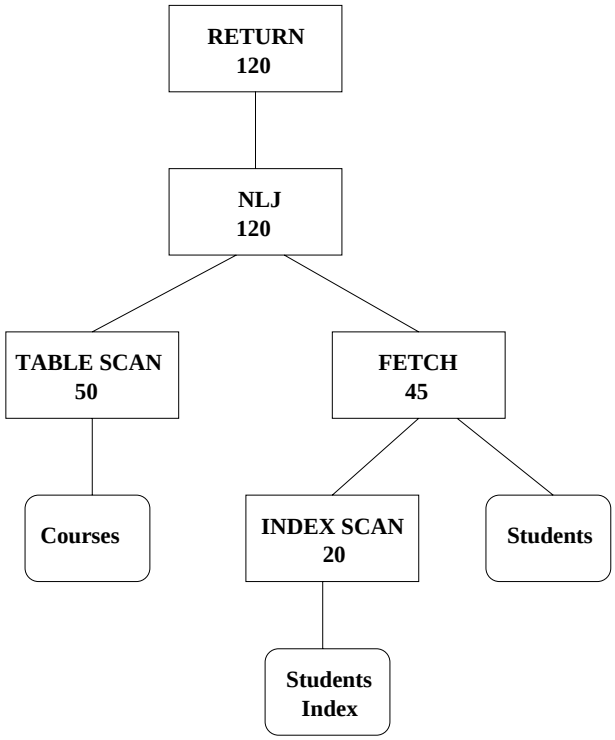

Figure 5.1: A sample query access plan. The value below each operator is the IO_COST for the sub-plan rooted at that operator.

- The FETCH operation uses the row IDs to retrieve the corresponding records from the STUDENTS table, and percolates the qualifying records up to the NLJ. The NLJ combines the matching COURSES and STUDENTS records.
- Finally, the query processor returns the set of qualifying records to the calling program.

IO_COST

Most of today's cost-based query optimizers apply cost models that measure expended CPU, memory and disk I/O resources from proposed query plans. The access plans contain granular breakdowns of exactly how much of each resource is used by each operation in the plan. In particular, the *IO_COST* of an operation, is the estimated number of page I/Os required to execute the plan up to (and including) the current operator. The query optimizer considers object attribute characteristics, object sizes, index clustering, disk overhead and available buffering when computing the IO_COST value.

We use the IO_COST values to estimate the number of pages each operator references per object. For operations that access an object directly (table and index scans) the IO_COST reflects the number of object pages that operator references. Indirect access operations (fetch, insert, update, delete, merge join, hash join), are represented as internal nodes in a query plan. The number of available buffers is used to calculate the IO_COST of an operation. We use the IO_COST values to help define access patterns, which we consequently use to help recommend buffer configurations. This forms a circular dependency. Further details regarding this circular dependency problem are described in Appendix A. Since the IO_COST is a cumulative value, the IO_COST of an indirect operation alone can be estimated as:

$$IO\_COST_{int} = IO\_COST_{cum} - IO\_COST_{child} \tag{5.1}$$

where IO_COST_{cum} is the cumulative IO_COST up to (and including) the current indirect operation, and IO_COST_{child} represents the sum of the cumulative IO_COSTs of its immediate children operations. For the merge join, hash join and

sort operations, which normally receive their inputs via pipelining from lower level operations, their IO_COSTs should be minimal. Large IO_COST_{int} values for these operations occur when object pages spill to disk due to insufficient buffer memory, temp space, or sort memory.

The nested loop join (NLJ) operation normally executes its right input branch repeatedly. This behaviour can typically be inferred from the access plans by observing a larger IO_COST for the NLJ operation. However, we do not have sufficient information to determine how these NLJ page IO_COSTs are distributed among its children operations and database objects and whether repeated references are to the same pages or to distinct pages. Thus, for simplicity, we assume that for each iteration of a looping reference, query references are to the same pages. If the NLJ's IO_COST value is large, our assumption introduces error that can cause under estimates in our model's predictions.

In the sample access plan in Figure 5.1, the table and index scans access an estimated 50 and 20 pages of the COURSES table and STUDENTS index, respectively. The FETCH operation references approximately 25 pages from the STUDENTS table.

Sequential Scan Detection

The access plans contain operator specific attributes indicating changes in access behaviour based on object statistics, available data structures and current system resources. One of these operator attributes is a parameter which we call *seqON*, that indicates whether or not an operator will sequentially read an object's pages. A variant of this parameter can be found in modern query access plans [OR00, DB01], particularly in direct access operator attributes. Sequential access can also be inferred from the access method directly, e.g., a table scan. A seqON = TRUE value indicates the operator will sequentially read the underlying object pages, whereas a FALSE value implies the object pages will be randomly read.

The query optimizer decides whether an operator will sequentially read an object's pages by evaluating factors such as available indices, index selectivity and the page clustering. We are interested in the value of seqON because it may be used by the GCLOCK weight assignment module.

5.1.2 Descriptor Vectors

To characterize the access pattern of each query i's reference to an object $k, i = 1..Q$, $k = 1..C$, or to a temporary (TEMP) table, we define a *descriptor vector*. A descriptor vector consists of two attributes:

1. **pgsRead(k,i)**, gives the number of pages query i reads from object k. By default, pgsRead(k,i) := 0, $\forall k, i$.

2. **seqAccess(k,i)**, a boolean flag indicating if query i sequentially reads object k's pages. The default value, seqAccess(k,i) := FALSE, $\forall k, i$, indicates random page accesses.

Chou and Dewitt categorized query references into three types: sequential, random or looping [CD85]. We adopt a similar categorization to our descriptor vectors to capture sequential and random query references against objects.

We assume a permanent object k is referenced by at most one operator in a query. However, TEMP tables, which are temporary objects that exist only during query execution, may be used by more than one operator in a query plan to handle record spillage. Since we do not know beforehand the number of operators which will use a TEMP table (if any), we simply assign a unique object identifier greater than C (the number of non-TEMP database objects) to each TEMP table that we discover in an access plan.

5.1.3 Methodology

The extraction algorithm determines a value for pgsRead(k,i) from the IO_COST value(s), and determines sequential access either from the seqON parameter or from the access method. We infer the seqON parameter value from the access method in cases where the operation is defined to sequentially read an object's pages (e.g., a table scan) and in cases where the operation reads a TEMP table. In addition, we assume that a sort operation randomly reads a table's pages. Our extraction algorithm performs a depth first traversal of the query plan tree to calculate these values. We assume each node p in the query plan tree contains the following fields:

- **name**, is the operator name, or a unique identifier for the object.
- **type**, indicating the node type: operator or object.
- **IO_COST**, represents the number of page I/Os required to execute the plan up to, and including, the current operator. For object (leaf) nodes, IO_COST = 0.
- **seqON**, is a boolean parameter indicating if an operator will reference the object pages sequentially. For object nodes, seqON = NIL.
- **left**, represents the left child of an operator node. For object nodes, left = NIL. Unary operators have only the left child.
- **right**, represents the right child of an operator node. For object nodes, right = NIL.

We present the access pattern extraction algorithm below. We describe the specific processing details for each access method following the algorithm.

Algorithm

For a given access plan represented as a tree, the plan nodes are visited in a depth first order (for simplicity, we omit the depth first traversal details, and present only the specific processing details). Prior to processing any plan nodes, the algorithm initializes the TEMP tables identifier, $t := C$, which will be incremented each time a descriptor vector is created for a TEMP table. If p is an operator node in query i, EXTRACT(p,i) will determine the number of referenced pages and the reference type for the database object used by operator p.

EXTRACT(p, i)

```
1  if p.type = operator
2    then switch (p.name) // Extract access patterns according to the access method
3           case TBSCAN :
4                 // the child node is the referenced table
5                 pgsRead(left.name,i) ← p.IO_COST
```

```
6           seqAccess(left.name,i) ← TRUE
7
8       case IDXSCAN :
9           // the child node is the referenced index
10          pgsRead(left.name,i) ← p.IO_COST
11          seqAccess(left.name,i) ← p.seqON
12
13      case FETCH :
14          if p is a unary operator
15            then pgsRead(left.name,i) ← p.IO_COST
16                 seqAccess(left.name,i) ← p.seqON
17            else // p is a binary operator and the
18                 right child is the referenced object
19                 pgsRead(right.name,i) ← (p.IO_COST − left.IO_COST)
20                 seqAccess(right.name,i) ← p.seqON
21
22      case INSERT||UPDATE||DELETE :
23          // binary operators, where the right child is the referenced object
24          pgsRead(right.name,i) ← (p.IO_COST − left.IO_COST)
25          seqAccess(right.name,i) ← left.seqON
26
27      case MGJOIN||HSJOIN :
28          // ops don't access objects directly, input is piped from children ops
29          // check if spillage occurs to the TEMP tables
30          if p.IO_COST > (left.IO_COST + right.IO_COST)
31            then t+ = 1
32                 pgsRead(t,i) ← (p.IO_COST−
33                 (left.IO_COST + right.IO_COST))
34                 seqAccess(t,i) ← TRUE
35
36      case TEMP :
37          // a unary operator
```

```
38              t+ = 1
39               pgsRead(t,i) ← (p.IO_COST − left.IO_COST)
40               seqAccess(t,i) ← TRUE
41
42          case SORT :
43               Let n_rows be the number of rows to be sorted
44               Let r_width be the width of a sorted row
45               Let p_size be the sorted table's page size
46               Let ST identify the sorted table
47              nSorted ← (p.n_rows * p.r_width) / p_size
48               pgsRead(ST,i) ← nSorted
49               seqAccess(ST,i) ← FALSE
50              if nSorted > size(sortMem)
51                then t+ = 1
52                        pgsRead(t,i) ← nSorted
53                        seqAccess(t,i) ← FALSE
54
55          case default :
56               break
```

The EXTRACT(p,i) algorithm examines each node p in the query i plan, and if p is an operator node, the algorithm derives the reference type and an estimate for the number of referenced pages to each object k. The derivation steps vary with each access method. We describe each access method's specific processing details.

TBSCAN

A TBSCAN is a sequential scan of a table's pages. Since the TBSCAN is a unary operator, its child node represents the referenced table. An estimated number of referenced table pages is given by the TBSCAN operator's IO_COST value.

IDXSCAN

An IDXSCAN is a unary operator where the child node represents the referenced index. Whether the IDXSCAN references the index pages sequentially or randomly is indicated by its seqON attribute, which updates the seqAccess vector attribute. Since an IDXSCAN reads the index pages directly, an estimated number of referenced index pages is given by its IO_COST value.

FETCH

FETCH can be a unary or binary operator. In either case, the reference type (sequential or random), is indicated by the FETCH operator's seqON attribute. If FETCH is a unary operator, it references object k's pages directly, and an estimate for the number of referenced pages is given by the operator's IO_COST value. If FETCH is a binary operator, the right child is the referenced object and the left child is an operator. The left child passes an identifier to the FETCH to retrieve the matching record from the object. In this case, the FETCH operator's IO_COST is a cumulative value and the estimated number of referenced object pages is calculated by subtracting the cumulative IO_COST of the left child operator from the FETCH operator's IO_COST.

INSERT/UPDATE/DELETE

The INSERT, UPDATE and DELETE operators all have binary inputs, where the left child is an operator and the right child is the referenced object. Similar to the binary input FETCH, a row identifer is passed from the left child to the UPDATE or DELETE operator, which then applies the operation to the matching object record. An INSERT operation inserts a given record into the table or index object. The reference type is indicated by the left child's seqON attribute. The number of pages referenced during the operator's execution is the difference between the current operator's IO_COST value and its left child's IO_COST value.

MGJOIN/HSJOIN

The merge join (MGJOIN) and hash join (HSJOIN) operations are processed in a similar manner. MGJOIN sequentially reads its sorted input records. If there are duplicate records on the join key in the outer table, then each matching record in the inner table will be repeatedly read. The set of matching records will be spilled to a TEMP table to avoid excessive disk reads. HSJOIN sequentially reads a hash table. If the hash table is too large to fit into the given amount of memory, it is spilled to TEMP tables.

Ideally, the IO_COST of the MGJOIN and HSJOIN operations should each be approximately equal to the sum of its children's IO_COSTs, since all inputs are piped from children operations. However, in the cases described above, the MGJOIN and HSJOIN incur extra I/Os (p.IO_COST > (left.IO_COST + right.IO_COST)) to read pages from the TEMP tables. We create a descriptor vector to capture the TEMP table references and to account for the extra buffer pages the TEMP tables will occupy.[3] To accurately identify each referenced TEMP table, the identifier t, is incremented each time a descriptor vector is created for a TEMP table. The estimated number of sequentially referenced TEMP table pages is calculated as p.IO_COST - (left.IO_COST + right.IO_COST).

TEMP

A TEMP operation stores data into, and retrieves data from a TEMP table. TEMP is a unary operator, where its child is normally an operator node. The estimated number of referenced TEMP pages is calculated as (p.IO_COST - left.IO_COST). The pages are assumed to be sequentially read.

[3]When p.IO_COST < (left.IO_COST + right.IO_COST), the MGJOIN, HSJOIN is expected to read only a portion of its inner table or hash table, respectively. This indicates that the optimizer's initial operator IO_COSTs were over estimates. In this case, since the joins receive the necessary pages from lower level operators and no spillage occurs to the TEMP tables, the algorithm does not need to perform any calculations.

SORT

Modern database systems devote a portion of memory specifically for handling SORT operations. Let this sort memory be called *sortMem*, whose size is a database tunable parameter. If the size of sortMem is not large enough to contain the sort (i.e., size(sortMem) $<$ nSorted), the sort spills to TEMP tables. If this occurs, we create a descriptor vector to capture the TEMP table references, and to account for the extra buffer pages that the TEMP tables will use. We calculate the estimated number of pages to be sorted, and use this estimate as the number of pages the SORT operation randomly references in the TEMP table.[4] In addition, we create a descriptor vector to capture the query page references to the table whose pages are to be sorted.

5.1.4 Summary

In this section, we have shown how the EXTRACT algorithm determines the reference type and the number of referenced object pages from information in the query access plans, to produce a descriptor vector per query, per object, for permanent objects. The value of the attribute pgsRead(k,i), indicates the expected number of pages query i references from object k. This attribute does not provide information regarding the distribution of query page references nor which specific pages are referenced. In the next section, we describe how query access skew is modelled and show how the descriptor vector attributes are used to calculate the utilization factor per class, per query.

5.2 Multifractal Skew Model

Database reference patterns are generally skewed, meaning that some pages are accessed more frequently than others [CS89, LD93, DYC95]. The *multifractal model* is closely related to the 80-20 skew law, which says that 80% of the accesses reference 20% of the data. The multifractal model mimics this skew behaviour by dividing the

[4]Modern query optimizers provide the values n_{rows}, r_{width} and p_{size}, in the SORT operator's attributes in the query access plans [OR00, DB01].

data records into two equal halves and setting a bias parameter $p = 0.8$, to indicate that 80% of the accesses occur in one half of the data records and the remaining 20% of accesses occur in the other half. This process then continues recursively [FMS96, WMCPF02].[5] Empirical studies have shown that the multifractal model is quite accurate in modelling skewed distributions [SCH91, FMS96], and network, web and disk I/O traffic [WMCPF02].

We use the multifractal model to represent database access skew. The model divides an object k's pages into *classes*, with uniform access probability within each class. Pages are divided into hot and cold classes, which consequently determines their access frequency, i.e., hot classes contain frequently referenced pages, whereas cold classes contain less frequently referenced pages. We modify the model slightly by introducing a parameter, b_k, to help create a more general access skew representation. The parameter b_k indicates the percentage of object k's pages to allocate to each of the two divided classes, i.e., $b_k = 0.3$ means 30% of object k's pages are distributed to the hot class and the remaining 70% of pages are distributed to the cold class. The original multifractal model divided the object pages into two equal sized classes, this is a special case with $b_k = 0.5$. We present our multifractal *skew* model next.

5.2.1 Details of the Multifractal Skew Model

In our buffer configuration methodology, the multifractal model is used to model skewed query access to object pages. Table 4.1 describes the skew model parameters p_k, b_k, *order* and n_k. Values for these parameters may be specified by the DBA. Otherwise, the default values shown in Table 4.1 are used. In addition to these parameters, the model uses the descriptor vector attribute, pgsRead(k,i). For each object k, the model divides its pages into 2^{order} classes. We will use c_{kj}, $1 \leq j \leq 2^{order}$, to represent the classes of an object k.

Access skew is modelled by associating higher page access probabilities to smaller

[5]The multifractal model extrapolates on the self-similarity property of fractals (patterns seen at a high level are also seen in nested lower levels).

sized classes. The model begins with one class (containing all pages of object k), and at each step n, $0 \leq n < order$, the model recursively divides each of the 2^n classes into a cold class and a hot class. This division process is repeated for $order$ times, resulting in 2^{order} classes. Cold classes are larger than hot classes and typically receive a lower percentage of page references relative to their size.

Suppose we start with a class of N pages of which N_i are referenced by query i. On each iteration, divide the class into two classes:

- A **hot** class has $(N \cdot b_k)$ pages, of which $\min(N \cdot b_k,\ N_i \cdot p_k)$ are referenced by query i.
- A **cold** class has $N(1 - b_k)$ pages, of which (N_i - $\min(N \cdot b_k,\ N_i \cdot p_k)$) are referenced by query i.

Initially, for each object k, we have a single class with $N = n_k$ and $N_i =$ pgsRead(k,i). We perform $order$ iterations for each object k, resulting in 2^{order} classes. Let n_{kj} be the size of class c_{kj} and d_{kij} be the number of pages query i references in class c_{kj}. The output of the model is n_{kj} and the utilization factor $\mu_{kij} = \frac{d_{kij}}{n_{kj}}$, $\forall j$.

Figure 5.2 shows an example of the skew construction process when query referenced pages are distributed among the hot and cold object classes, according to probabilities p_k and $(1 - p_k)$, respectively. Note that at each level, the right sided class is the hotter class.

Figure 5.3 shows the first two steps of how the multifractal skew model determines the class size proportions and class reference probabilities according to b_k and p_k, respectively. In Figure 5.3 (a), a query references all of object k's pages uniformly according to some probability P. In Figure 5.3 (b), at order level = 1, an 80-20 skew is generated; 80% of the total page references go to the hot class (containing 20% of object k's pages). In Figure 5.3 (c), at order level = 2, four classes exist; containing 64%, 16%, 16%, and 4% of object k's pages, and attracting 4%, 16%, 16%, and 64% of object k's total query page references, respectively. The multifractal skew model generates heavier access skew as $b_k \rightarrow 0$ and $p_k \rightarrow 1$.

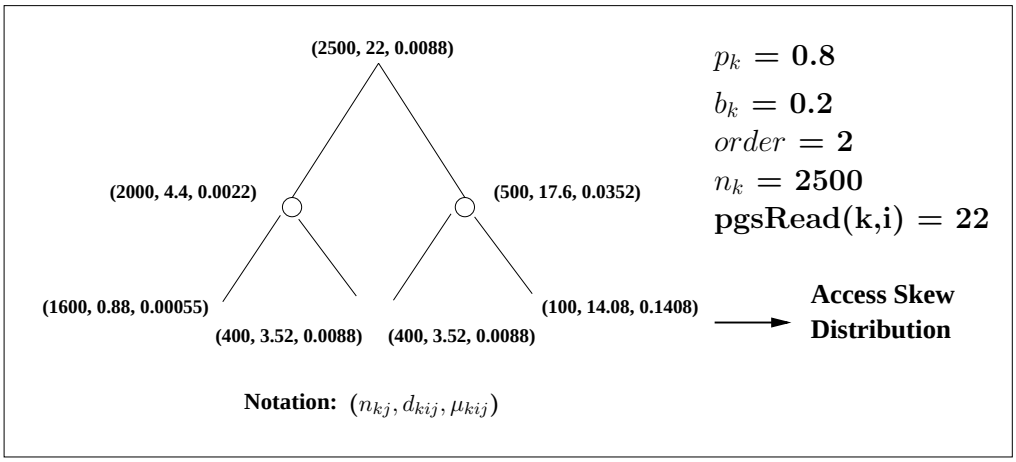

Figure 5.2: The distribution of query page references among the hot and cold classes, according to p_k and $(1 - p_k)$, respectively. Note that $\sum_j n_{kj} = n_k$ and $\sum_j d_{kij} =$ pgsRead(k,i).

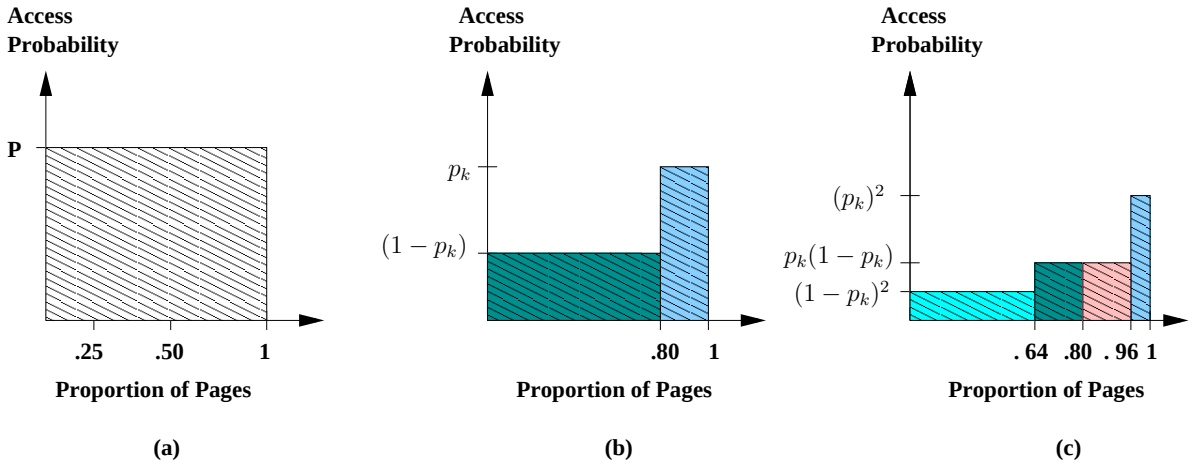

Figure 5.3: The multifractal skew model construction process to determine class size proportions and page access probabilities, for $b_k = 0.2, p_k = 0.8$ and $order = 2$.

The multifractal skew model calculates μ_{kij}, the proportion of class c_{kj} pages referenced by query i, $\forall j = 1..2^{order}$. When a query i reads all the pages in a table, the multifractal skew model returns $\mu_{kij} = 1$, $\forall j$, regardless of the values for p_k and b_k. In the next section, we use μ_{kij} to determine the expected reference rate to a class c_{kj} page.

5.3 Page Request Arrival Process

To model queries' requests for pages, consider a query i, with arrival rate λ_i, which references a fraction μ_{kij} of the pages in class c_{kj} at each arrival. Assuming each class c_{kj} page is equally likely to be referenced, the expected reference rate to an arbitrarily selected class c_{kj} page is:

$$\lambda_{kij} = \lambda_i \cdot \mu_{kij} \tag{5.2}$$

Query requests to a class page are modelled independently of requests to other class pages. Assume references by query i to a class c_{kj} page form a Poisson process with rate λ_{kij}. Assume all queries are independent. Then the aggregate reference process for the page, over all queries, is a Poisson process with rate:[6]

$$\lambda_{kj} = \sum_{i=1}^{Q} \lambda_{kij} \tag{5.3}$$

Consequently, query references to the class c_{kj} page occur with a mean interarrival time of $\frac{1}{\lambda_{kj}}$.

In the next chapter, we show how the output from the workload characterization scheme, namely the parameters: λ_{kij}, n_{kj}, and the weights w_{ki}, are used in our buffer model to determine the total physical read rate, which is the performance metric we use to evaluate a given buffer configuration.

[6]This follows directly from the superimposition property of Poisson processes [JAIN91].

Chapter 6

The GCLOCK Query-Weight Buffer Model

The approximate GCLOCK Markov model proposed by Nicola, Dan and Dias [NDD92], assigned weights on a per object basis. Specifically, a set of objects was partitioned, and a weight was selected for each partition. In this chapter, we present an analytic model of the GCLOCK buffer replacement policy that allows weights to be assigned on a per object, per query basis. That is, each type of page may have a distinct weight for each type of query. Given a workload characterization (as described in Chapter 5) and an initial buffer configuration, our buffer model evaluates the initial buffer configuration by estimating the total physical read rate of the workload running with the given buffer configuration. We give an overview of our buffer model, describe its parameters, and then discuss the buffer model in greater detail.

6.1 Buffer Model Overview

To distinguish our GCLOCK model from the approximate GCLOCK Markov model (Approx-GCLK model), we call our model the *GCLOCK Query-Weight* model to highlight our model's ability to accept different weights for different query types. Given a buffer configuration and the workload parameters, λ_{kij}, n_{kj} and w_{ki}, our

Query-Weight buffer model estimates the physical read rate for each class c_{kj}. The buffer model's objective is to calculate the total physical read rate over all classes, which is an indication of the quality of the given buffer configuration for the current workload. That is, a lower physical read rate is preferred over a higher read rate since this indicates that the buffer pools are doing a good job of satisfying a greater number of query page requests.

To determine the total physical read rate, our buffer model first needs to estimate the number of pages from each class that reside in the buffer. This is also known as the *buffer occupancy* for a class. It depends on the length of time a class c_{kj} page spends in the buffer. We define the concept of a page *request lifetime*, as the length of time a class c_{kj} page resides in the buffer after being used by a query.

In the GCLOCK replacement policy, a request lifetime begins when a query i references a page and ends by one of two events: (1) the clock algorithm evicts the page from the buffer when it discovers the page's weight is zero, or (2) another query references the same page, giving the page a new weight and a new request lifetime. Clearly, the page request lifetime depends on the weights and on the request arrival rate. Frequently referenced pages that are given large weights will reside in the buffer longer than infrequently referenced pages with smaller weights.

We assume class pages with a zero weight are evicted from the buffer. As queries reference a buffer page, the page weight either increases or decreases depending on the query reference type. The clock will decrement a page's weight by one on each cycle through the buffer. The clock removes the page from the buffer when the page weight reaches zero. We model the weight transitions of an arbitrary class c_{kj} page using a Markov model. From the Markov model, we determine the probability that the page resides in the buffer (has weight greater than zero). We can then estimate the buffer occupancy and physical read rate values for that class. The sum of the physical read rate estimates, first over all j classes per object, and then over all k objects, determines the model's total physical read rate prediction associated with the given buffer configuration. Further details of our GCLOCK Query-Weight buffer model are given in Section 6.3. We present the model's parameters next.

6.2 Model Parameters

We assume the given buffer configuration provides an assignment of each database object to a buffer pool. Table 4.1 described the workload parameters. Our GCLOCK Query-Weight buffer model uses the following additional parameters and notation:

Parameters

- $\mathbf{B}$, the total number of buffer pools.
- $\mathbf{s_x}$, the size of buffer pool x.

Notation

- $\mathbf{w}$, the maximum weight assigned to a class page, $w = \max_{k,i} w_{ki}$.
- $\frac{\mathbf{1}}{\mathbf{t_x}}$, the clock speed for buffer x, $t_x > 0$.
- $\lambda_{\mathbf{kj}}^{(\mathbf{m})}$, the reference rate to a class c_{kj} page from all queries that set the page weight to m, $m = 0..w$. $\lambda_{kj}^{(m)} = \sum_{i|w_{ki}=m} \lambda_{kij}$.
- $\mu_{\mathbf{m}}$, the clock eviction rate for a buffer page with weight m, $\mu_m = \frac{1}{t_x \cdot m}$.
- $\rho_{\mathbf{kj}}$, the duty cycle of pages in class c_{kj}, i.e., the probability a c_{kj} page is in the buffer.
- $\mathbf{n_{kjx}}$, the expected buffer occupancy of class c_{kj}, i.e., the expected number of c_{kj} pages residing in buffer x. (Given that object k is assigned to buffer x.)
- $\mathbf{n_{kx}}$, the buffer occupancy of object k, $\sum_{j=1}^{2^{order}} n_{kjx} = n_{kx}$.
- $\mathbf{r_{kj}}$, the physical read rate of class c_{kj}.
- $\mathbf{r_k}$, the physical read rate of object k, $\sum_{j=1}^{2^{order}} r_{kj} = r_k$.
- $\mathbf{r_{tot}}$, the total physical read rate from all objects, $\sum_{k=1}^{C} r_k = r_{tot}$.

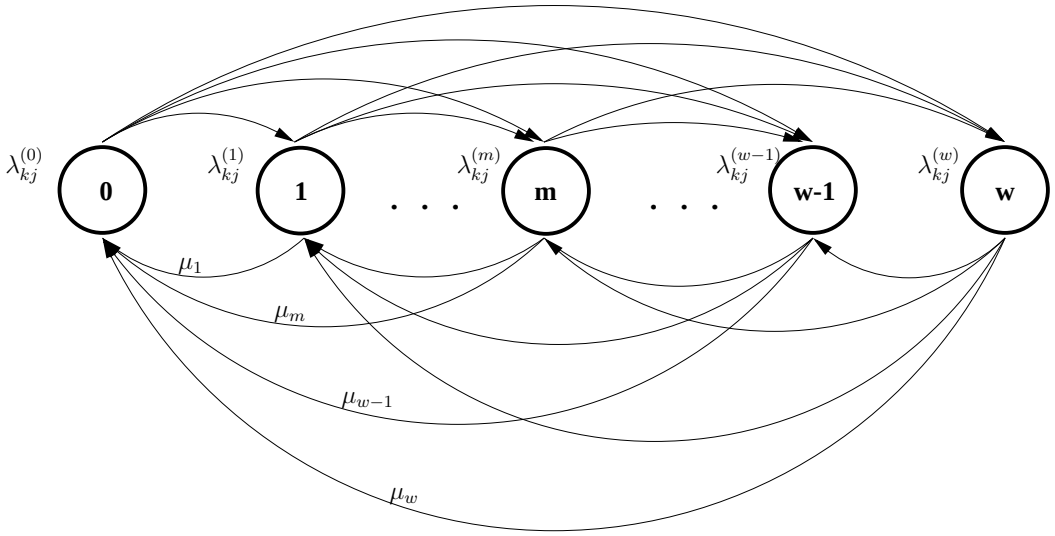

Figure 6.1: Markov model representing the weight of an arbitrary class c_{kj} buffer page.

6.3 Details of the GCLOCK Query-Weight Buffer Model

We model the weight of an arbitrary page from class c_{kj} as a continuous time Markov model, as shown in Figure 6.1.

The Markov model has a total of $(w+1)$ states. State m indicates that the page has weight m. The steady state probability P_m indicates the probability that the page has weight m. If the page is in one of the states $1..w$, then the page resides in the buffer. If it is in state 0, then it is not in the buffer.

When a query of type i references the page, it sets the page weight to w_{ki}. This causes the page to move to state w_{ki}. Transitions to a state m occur with a rate of $\lambda_{kj}^{(m)}$, where $\lambda_{kj}^{(m)}$ is the reference rate of all queries that set the page weight to m. In Section 5.3, we modelled the page request process of a query i to a class c_{kj} page as a Poisson process with rate λ_{kij}. Assuming that the page request processes of the query types are independent, by the superimposition principle of Poisson processes:

$$\lambda_{kj}^{(m)} = \sum_{i|w_{ki}=m} \lambda_{kij} \tag{6.1}$$

The resulting aggregate reference process is also Poisson.

Since the clock decrements the weight of a class buffer page by one on each revolution through the buffer, a page with a weight of m is expected to reside in the buffer for a time of $t_x \cdot m$, if it is not re-referenced, where t_x is the time for the clock to circulate once through the buffer. After this time, the page weight reaches zero and the clock evicts the page from the buffer. To model this, we assume a Poisson eviction process with rate $\mu_m = \frac{1}{t_x \cdot m}$, for a page in state $m > 0$. This is shown by the transitions labelled with μ_i in Figure 6.1. Note that pages with higher weights have a slower eviction process, and will thus remain in the buffer longer before being evicted. Finally, note that the eviction rate is expressed in terms of t_x, called the buffer clock rate, which will be discussed further later in this section.

Having described the state transition rates, we proceed to derive the balance equations for the Markov model in Figure 6.1. Our objective is to derive an expression for P_0, the steady state probability that the page has a zero weight. Equivalently, P_0 indicates the probability that the page does not reside in the buffer.

Let

$$\lambda_{kj} = \sum_{i=0}^{w} \lambda_{kj}^{(i)} \tag{6.2}$$

Generally, for a state m, $0 < m \leq w$, the sum of the flow rates into state m must equal to the sum of the flow rates out of state m, i.e., flow in = flow out.

$$\lambda_{kj}^{(m)} \left[P_0 + P_1 + \ldots + P_{m-1} + P_{m+1} + \ldots + P_w\right] = P_m \left[\mu_m + (\lambda_{kj} - \lambda_{kj}^{(m)})\right]$$

Since we know the normalization condition $\sum_{i=0}^{w} P_i = 1$ must hold, we get:

$$\begin{aligned}
\lambda_{kj}^{(m)}\left[1-P_m\right] &= P_m\left[\mu_m + (\lambda_{kj} - \lambda_{kj}^{(m)})\right] \\
\lambda_{kj}^{(m)} &= P_m\left[\mu_m + \lambda_{kj}\right] \\
P_m &= \frac{\lambda_{kj}^{(m)}}{\mu_m + \lambda_{kj}}, \;\; m \geq 1
\end{aligned} \tag{6.3}$$

To derive P_0, the probability that the page is not in the buffer, we substitute Equation (6.3) into the normalization condition:

$$\begin{aligned}
P_0 &= 1 - [P_1 + P_2 + \ldots + P_w] \\
&= 1 - \sum_{m=1}^{w} P_m \\
P_0 &= 1 - \sum_{m=1}^{w} \frac{\lambda_{kj}^{(m)}}{\mu_m + \lambda_{kj}}
\end{aligned} \tag{6.4}$$

Let P_{0_kj} be P_0 for class c_{kj}. That is, P_{0_kj} is the probability that a c_{kj} page is *not* in the buffer. We define ρ_{kj} as the probability that a class c_{kj} page is in the buffer:

$$\rho_{kj} = 1 - P_{0\_kj} \tag{6.5}$$

We use ρ_{kj} to determine n_{kjx}, the expected number of class c_{kj} pages residing in buffer x, also known as the *buffer occupancy* for class c_{kj}:

$$n_{kjx} = \rho_{kj} \cdot n_{kj} \tag{6.6}$$

where n_{kj} is the size of class c_{kj}, determined from the workload characterization scheme. The buffer occupancy estimate for object k; n_{kx}, is:

$$n_{kx} = \sum_{j=1}^{2^{order}} n_{kjx} \tag{6.7}$$

Up until this point, we have not mentioned how the clock speed, t_x, is determined. In a multiple buffer pool configuration, each buffer $x, x = 1..B$, has an associated t_x, which is determined as follows.

Suppose for a given buffer configuration, u objects are assigned to buffer x, $u \leq C$. From Equations (6.6) and (6.7), we have:

$$\begin{aligned} n_{1x} &= \sum_{j=1}^{2^{order}} \rho_{1j}(t_x) \cdot n_{1j} \\ n_{2x} &= \sum_{j=1}^{2^{order}} \rho_{2j}(t_x) \cdot n_{2j} \\ &\vdots \\ n_{ux} &= \sum_{j=1}^{2^{order}} \rho_{uj}(t_x) \cdot n_{uj} \end{aligned} \tag{6.8}$$

where we have written $\rho_{uj}(t_x)$ to emphasize the fact that ρ_{uj} is a function of t_x (remember that $\mu_m = \frac{1}{t_x \cdot m}$). We must also enforce the constraint:

$$s_x = n_{1x} + n_{2x} + \ldots + n_{ux} \tag{6.9}$$

Equation (6.9) says that the total expected occupancy of all objects assigned to buffer x must be equal to the size of that buffer. Using Equations (6.8) and (6.9), if we start with an initial estimate of t_x and an error tolerance, δ, we can solve for t_x numerically. If $\sum_{v=1}^{u} n_{vx} > (s_x + \delta)$, then t_x should be reduced. Similarly, if $\sum_{v=1}^{u} n_{vx} < (s_x - \delta)$, then t_x should be increased. We are able to solve for t_x using this approach because the function n_{ix}, $i = 1..u$, increases monotonically with t_x.

Finally, we show how to estimate the physical read (miss) rate, r_{tot}, for a workload running under a given buffer configuration. We first estimate r_{kj} and r_k, the physical read rates per class and per object, respectively.

r_{kj} is determined by estimating the number of class c_{kj} pages on disk $(n_{kj} - n_{kjx})$ and summing the query reference rates of these pages:

$$r_{kj} = \lambda_{kj} \left(n_{kj} - n_{kjx} \right) \tag{6.10}$$

It follows that,

$$r_k = \sum_{j=1}^{2^{order}} r_{kj} \tag{6.11}$$

$$r_{tot} = \sum_{k=1}^{C} r_k \tag{6.12}$$

Our buffer model uses r_{tot} as a performance evaluation measurement. A buffer configuration that minimizes the physical read rate will:

- Satisfy a larger number of query page requests from memory than configurations with a higher number of physical reads, and consequently improve buffer hit rates.
- Reduce query response times by reducing the need to wait for disk accesses, and reducing the amount of CPU time and resources required to perform disk accesses.
- Enable the system to generate higher throughput as a result of decreased query response times and increased availability of CPU resources.

Chapter 7

Experimental Evaluation

In this chapter, we present the results of our model validation against a commercial database system. We also present the comparative results between our model predictions and predictions from the approximate GCLOCK Markov model [NDD92]. First, we describe the experimental environment. We then describe the validation methodology. Finally, we present our validation and comparative results and summarize with concluding remarks.

7.1 Experimental Environment

7.1.1 TPC-C Workload Overview

We validated our model using a TPC-C workload. A TPC-C workload is an online transaction processing (OLTP) workload that simulates a generic order entry system. A user may perform one of five transactions to enter and deliver orders, record payments, check the status of orders, or monitor the stock level at the warehouses. The database is scaled according to the number of warehouses. Each WAREHOUSE consists of 10 DISTRICTS, and each district serves 3000 CUSTOMERS. [1] There are 100 000 ITEMS available to be ordered. The inventory levels for each item at each warehouse are maintained in the STOCK relation. When a customer places a

[1] We denote proper relation names by capitalization.

new order, three relations are updated: (1) the ORDER relation maintains a record of each order; (2) the NEW-ORDER relation tracks pending, not-yet delivered orders; and (3) the ORDER_LINE relation maintains a record of every ordered item. A log of payment transactions is kept in the HISTORY relation. Query references to the STOCK, CUSTOMER, and ITEM relations exhibit data access skew [LD93]. The WAREHOUSE, DISTRICT, CUSTOMER, and STOCK relations scale with the number of warehouses. The ORDER, ORDER_LINE, and HISTORY relations grow as orders are processed.

The TPC-C benchmark defines five transaction types. The NEW-ORDER transaction enters an order for 10 items on average, records the order, and updates the inventory level for each item in the STOCK relation. The number of NEW-ORDER transactions completed per minute (TPM) is the TPC-C performance metric. The PAYMENT transaction handles payment from a customer. The ORDER-STATUS transaction returns the status of a customer's last order. The DELIVERY transaction processes 10 pending orders, one for each district, and deletes the pending orders from the NEW-ORDER relation. Finally, the STOCK-LEVEL transaction returns the inventory level of all items that were ordered in the last 20 orders from a district.

The TPC-C benchmark specifies minimum transaction frequency requirements that must be maintained as the database is scaled. For our experiments, we used the following transaction mix, [NEW-ORDER, DELIVERY, PAYMENT, ORDER-STATUS, STOCK-LEVEL] = [0.45, 0.04, 0.43, 0.04, 0.04], which attempts to maximize the NEW-ORDER transaction frequency. Further details regarding the TPC-C benchmark can be found in the TPC-C benchmark specification [TPC02].

7.1.2 System Specification

We conducted our experiments on DB2 v.7 running on AIX v.5. The machine was an IBM pSeries model B80 with 4 x 350 MHz processors, 6 GB total memory, 24 x 72 GB external SSA drives and 1 SSA controller. We used a 100 warehouse scale factor TPC-C database (100 WH SF), with an approximate size of 11 GB.

The database schema consisted of 14 user-defined tablespaces. Each table and index was assigned to its own tablespace with the exception of the WAREHOUSE,

DISTRICT, and ITEM tables and indices, which were assigned to a shared table and index tablespace, respectively. We also created a user-defined TEMP tablespace.

Each tablespace was striped across the 24 disks for increased parallelism. We used 24 prefetchers and page cleaners, corresponding to one prefetcher and one page cleaner per disk. The amount of memory allocated to the buffer pools varied according to each experiment.

7.2 Methodology

Our experimental objective was to validate our model's buffer occupancy and physical read rate predictions against system measured buffer occupancy and miss rate values, under varying buffer configurations and query reference behaviour. We also compared our model's predictions to those from the approximate GCLOCK Markov model. In addition, we examined our model's predictive capability to distinguish buffer configurations that can provide increased system throughput.

The primary performance metric we used is r_{tot}, the total physical read rate across all objects. The buffer occupancy estimates for each object are an intermediate measurement useful for model validation. For our predictive capability tests, we expect our model to predict lower physical read rate totals for buffer configurations that help increase system throughput.

To derive our model estimates, we configured the TPC-C database with a total buffer pool size ranging from 120 MB to 1.2 GB (30 K to 300 K 4 K pages), depending on the particular experiment. In our experimental environment, the optimizer generated 67 query plans for 19 physical objects (9 relational tables, 10 indices). We passed the query plans to the EXTRACT algorithm and the multifractal skew model to derive the required workload model parameters. We assume skewed query references and used two sets of skew parameters: (1) the default skew parameters $b_k = 0.2, p_k = 0.8$ and $order = 2$ for all $k = 19$ objects (uniform skew), and (2) a refined set of skew parameters shown in Table 7.1 (refined skew). Non-default skew parameter values were assigned to objects that were known to exhibit specific access skew [LD93]. The DBMS buffer manager assigned index pages a weight of 2, sequentially referenced pages a weight of 0, and all other page references a

Object	p_k	b_k
STOCK	0.85	0.15
CUSTOMER	0.85	0.15
ORDER_LINE	0.75	0.25
ORDER IDX	0.7	0.3
ITEM	0.75	0.25
Remaining Objects	0.8	0.2

Table 7.1: Refined skew model parameters, skew order level = 2.

weight of 1. We use the same weight assignments in our model. Given the workload model parameters and a buffer configuration specification (i.e., the number of buffer pools, their object assignments and sizes), we define our GCLOCK model parameters according to the given parameters and buffer specification. We use the model to predict the buffer occupancy and physical read rate for each object and r_{tot} for the given buffer configuration.

For the comparative tests, we used the extracted workload parameters as input into the approximate Markov model to determine the access probability to each partition. Further details regarding the comparative testing methodology are given in Section 7.3.2.

To obtain the system measured values associated with the tested buffer configuration, we configure the database with the same buffer configuration. We ran the TPC-C transactions against the configured database to generate workload activity. We executed each test run for an initial 15 minute ramp-up period at 170 clients for a 25 minute duration. We ran the predictive capability tests for a 45 minute duration to verify system stability in our results. The 15 minute ramp-up was the average time needed for the system to achieve a steady state and for the number of physical reads to stabilize. We took a buffer snapshot at the end of each run to determine the actual buffer occupancy value for each database object. We used DB2's tablespace snapshot monitor to measure the physical read rate counts for each tablespace at the end of each run. We then compared the system measured buffer occupancy and physical read rate values to our model's predictions.

A driver application executes the TPC-C transactions and reports the system

throughput as the number of completed NEW-ORDER transactions per minute (TPM). If the driver application reports a higher TPM value with buffer configuration A than with buffer configuration B, we expect our GCLOCK Query-Weight model to predict $r_{totA} < r_{totB}$.

7.3 Experimental Results

7.3.1 Model Validation against System Measurements

We validated our model's predictions against system measurements for a single buffer pool configuration of size 500 MB. Figure 7.1 shows the predicted buffer occupancy of each data object, normalized with respect to its system measured value, ordered by decreasing table size. For clarity, we have omitted the WAREHOUSE and DISTRICT table and index values, whose buffer occupancies were each less than 50. Table 7.2 shows the system measured buffer occupancy value of each data object in Figure 7.1.

For most of the data objects, the uniform skew buffer occupancy predictions show discrepancies from the measured values, i.e., normalized values are greater or less than 1. However, the refined skew model, with corrected individualized skew parameters, (for the STOCK, CUSTOMER, ORDER_LINE, ITEM tables and ORDER index) offers improved buffer occupancy predictions. The existing differences between the refined skew model predictions and the measured values are possibly caused by assuming independent page references. If an object exhibits a large number of correlated page references, the model does not capture this temporal and spatial page locality. Consequently, the model will assume an increased number of independent page references causing an increased buffer occupancy prediction. Further refinement to the access skew parameters may be required for objects whose buffer occupancy predictions are under estimated. Figure 7.1 shows that given correct access skew, the model offers improved accuracy to closely predict the actual object buffer occupancies, particularly for the frequently referenced data objects (i.e., STOCK, CUSTOMER, ORDER_LINE tables and indices).

Figure 7.2 shows the physical read rate prediction of each TPC-C database ob-

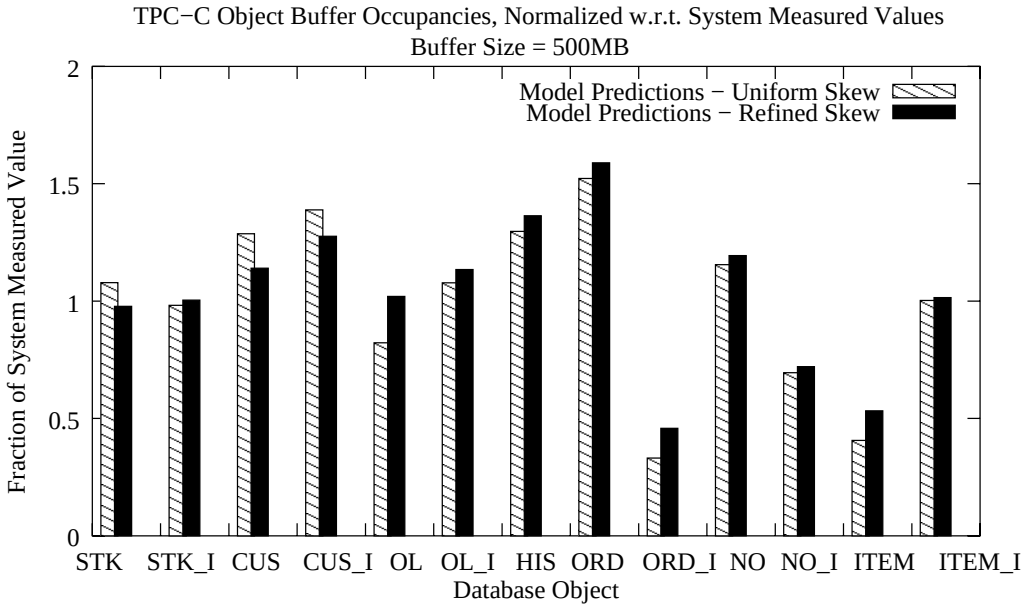

Figure 7.1: Object buffer occupancy predictions.

Object	Abbreviation	Measured Buffer Occupancy (4 K pages)
STOCK Tbl	STK	43505
STOCK Idx	STK_I	21562
CUSTOMER Tbl	CUS	8218
CUSTOMER Idx	CUS_I	7273
ORDER_LINE Tbl	OL	28265
ORDER_LINE Idx	OL_I	3803
HISTORY Tbl	HIS	1138
ORDER Tbl	ORD	1576
ORDER Idx	ORD_I	5033
NEW-ORDER Tbl	NO	896
NEW-ORDER Idx	NO_I	1540
ITEM Tbl	ITEM	1756
ITEM Idx	ITEM_I	336

Table 7.2: Measured object buffer occupancy.

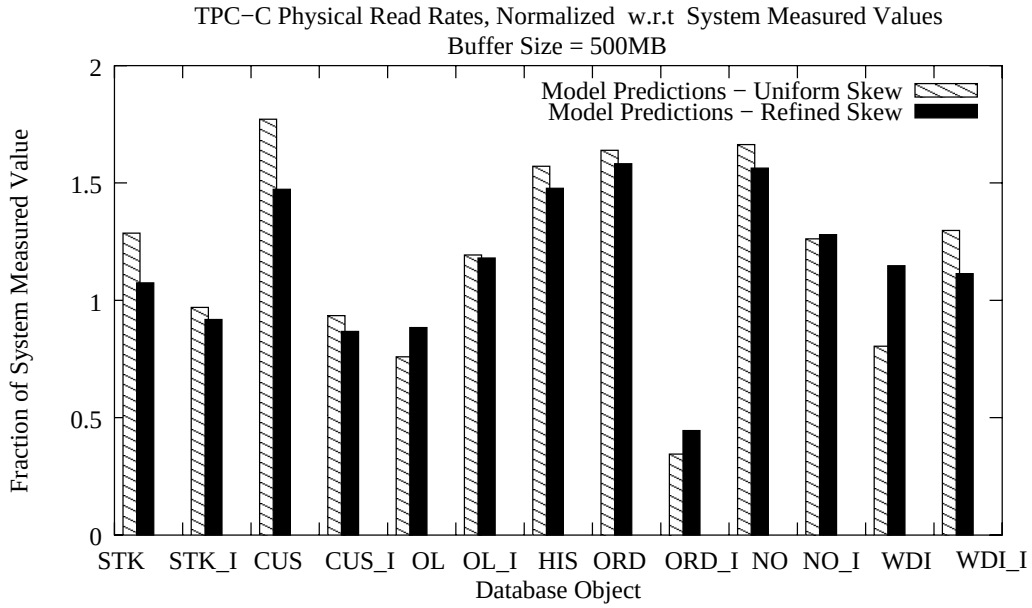

Figure 7.2: Object physical read rate predictions.

ject, normalized with respect to its system measured value. Table 7.3 shows the system measured physical read rates. The physical read rate predictions for the WAREHOUSE, DISTRICT and ITEM tables and indices are reported as WDI and WDI_I, respectively.[2] For objects that were given corrected skew parameter values, the refined predictions offer improved accuracy. However, discrepancies still remain. Over estimates may again be due to the independent page reference assumption, under which an increased number of independent page references (especially for pages not in the buffer) cause an increased number of disk reads. The under estimates for the ORDER_LINE table and the STOCK, CUSTOMER, ORDER indices may be refined by further tuning their individual skew parameter values.

Figure 7.2 shows that with correct skew parameters, our model is able to closely predict the object physical read rate, particularly for frequently referenced data objects. In addition, although not exemplified in Figure 7.2, the model is able to capture the relative physical read rate of each database object. That is, if the

[2]Recall that these tables and indices are assigned to a shared table tablespace and a shared index tablespace. System measured physical read rates are determined on a per tablespace basis.

Object Abbreviation	Measured Physical Reads/Sec
STK	760.35
STK_I	231.77
CUS	125.02
CUS_I	113.23
OL	641.22
OL_I	35.93
HIS	13.63
ORD	17.7
ORD_I	50.87
NO	7.9
NO_I	3.32
WDI	16.02
WDI_I	4.12

Table 7.3: Measured object physical read rates.

measured physical reads for the STOCK table are greater than the physical reads for the ORDER_LINE table, the model predictions follow the same trend.

We are interested in validating the accuracy of the model's total physical read rate prediction r_{tot}. For a buffer size of 500 MB, the measured total physical read rate was 2017.07 reads/sec. Our model predicted $r_{tot} = 2071.76$ reads/sec using the refined skew parameters, giving a relative difference of approximately 3%. Unless otherwise stated, all remaining model predictions will be based on the refined skew parameter values. In addition, figures displaying normalized model predictions are with respect to the system measured values.

Varying Total Buffer Size

We evaluated the Query-Weight model's r_{tot} predictions across varying buffer sizes. We are interested in evaluating the scalability of the model's physical read rate predictions. Figure 7.3 shows the results of our experiments for a total buffer size ranging from 120 MB to 1.2 GB.

As expected, the measured and predicted physical read rates decrease as the total buffer size increases. The model's predictions closely follow the system mea-

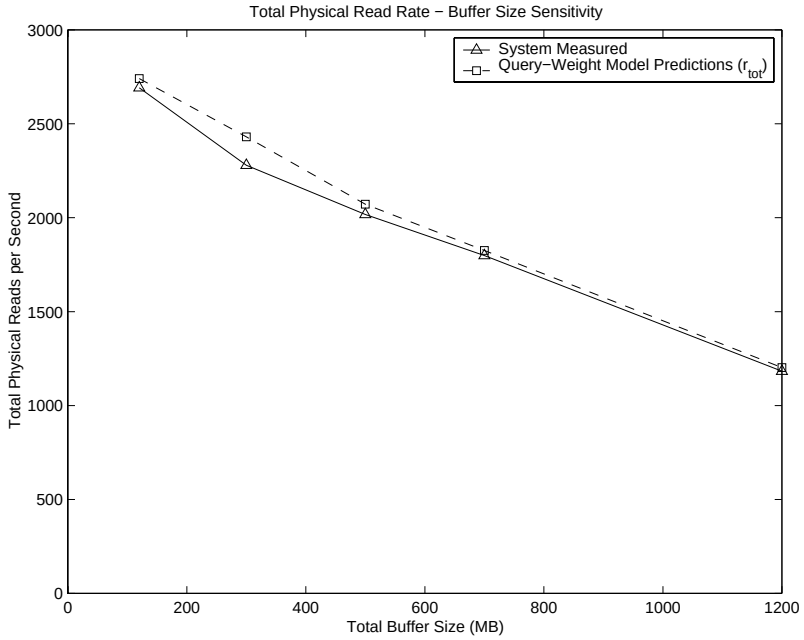

Figure 7.3: Total physical read rates for varying buffer sizes.

sured values throughout the range of buffer sizes. The predictions improve as the buffer size increases, showing a maximum relative error of approximately 7%.

Multiple Buffer Pool Configurations

We evaluated the accuracy of our model predictions under different multiple buffer pool configurations, with a total buffer size of 500 MB. We used the following three buffer configurations:

1. **2 BP:** A two buffer pool configuration where the STOCK table is assigned to its own buffer pool of size 200 MB. All remaining objects are assigned to the default buffer pool.

2. **Table/Index Split:** All tables are assigned to a buffer pool of size 300 MB, and all indices are assigned to another buffer pool of size 200 MB.

3. **3 BP:** The STOCK table is assigned to a buffer pool of size 160 MB, the ORDER_LINE table is assigned to a buffer pool of size 80 MB, and all remaining

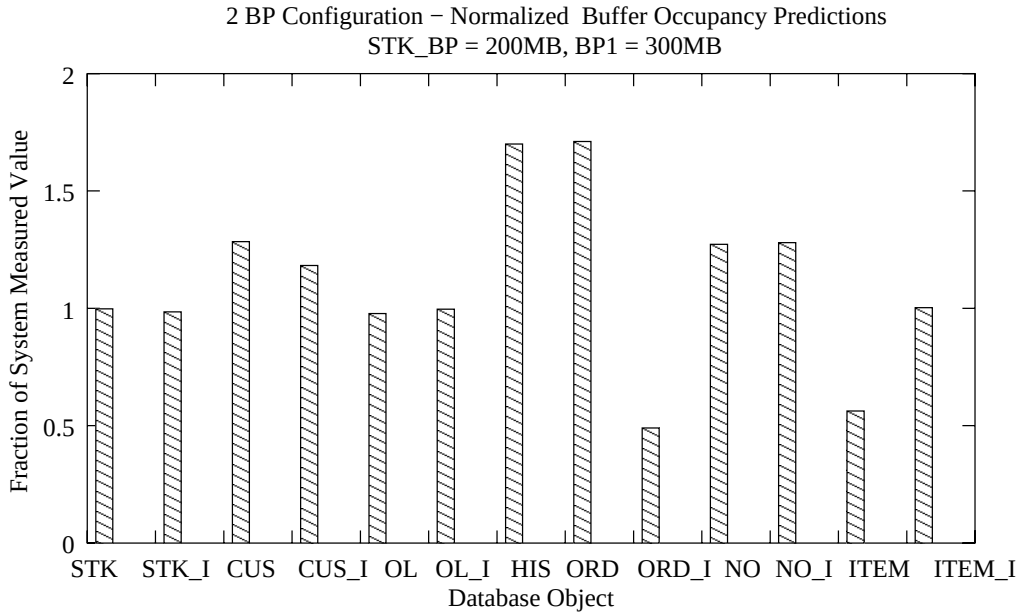

Figure 7.4: Query-Weight object buffer occupancy predictions for the 2 BP configuration.

objects are assigned to the default buffer pool of size 260 MB.

These configurations were derived from preliminary experiments to test the benefits of initial buffer configuration. The focus of these configurations was to devote separate buffer pools to large, frequently referenced tables. By changing the object to buffer pool assignments, the mix of query page references (i.e., query reference types and referenced objects) to a buffer pool changes. We are interested in examining the accuracy of the model's predictions in light of these changes.

Figures 7.4, 7.5 and 7.6 show the buffer occupancy estimate for each object in the 2 BP, split table/index and 3 BP configurations, respectively.

Overall, the figures show that the model is able to quite accurately predict the actual buffer occupancy for most data objects. Differences between the predicted and measured values may be due to required skew parameter tuning for specific objects or an increased number of independent page references. The model's consistent low predictions for the ORDER index may be caused by an under estimated number of query page references for this object. The EXTRACT algorithm deter-

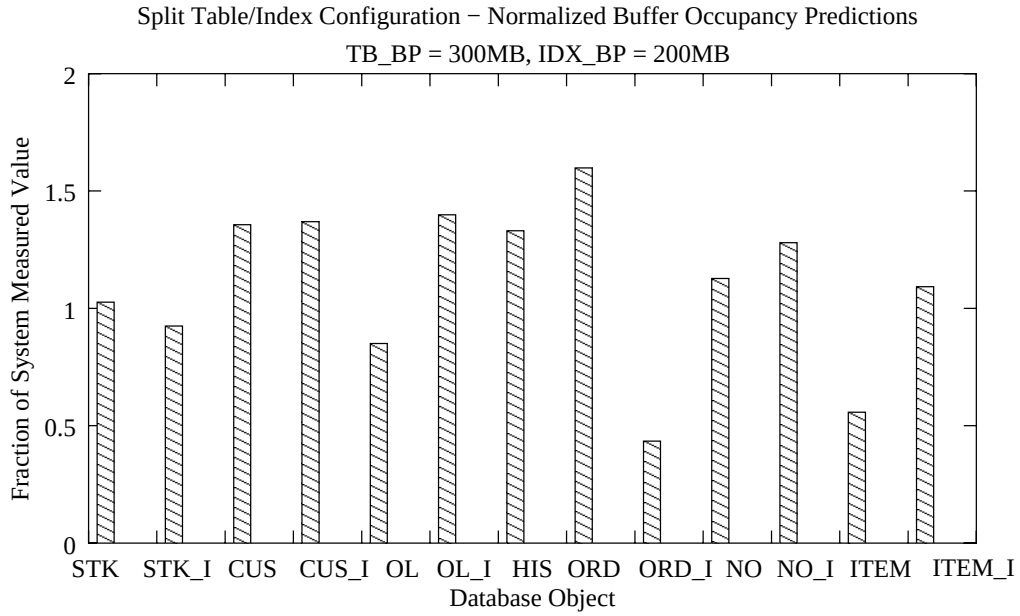

Figure 7.5: Predicted object buffer occupancies for the split table/index configuration.

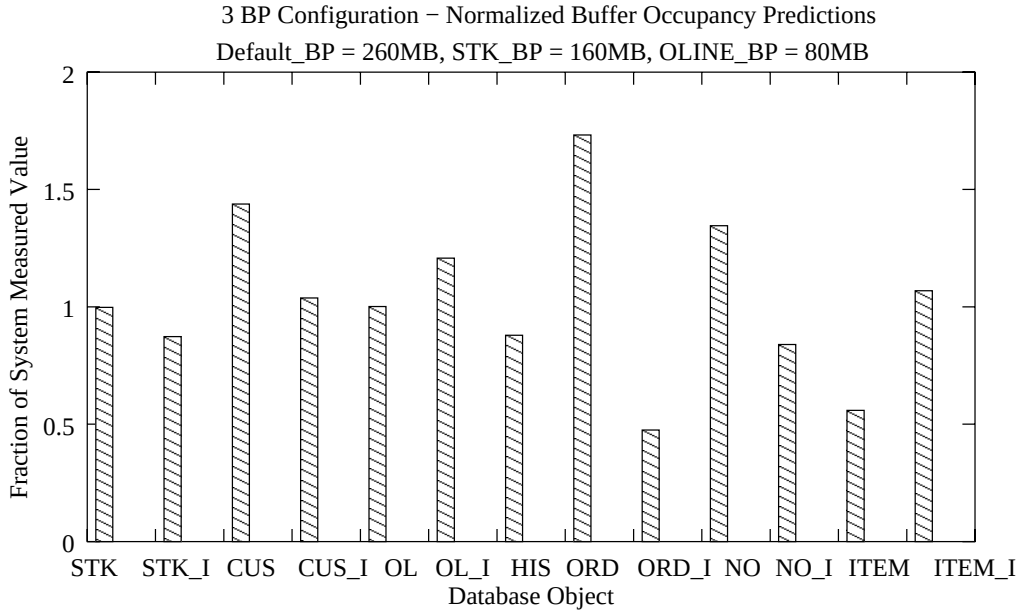

Figure 7.6: Predicted object buffer occupancies for the 3 BP configuration.

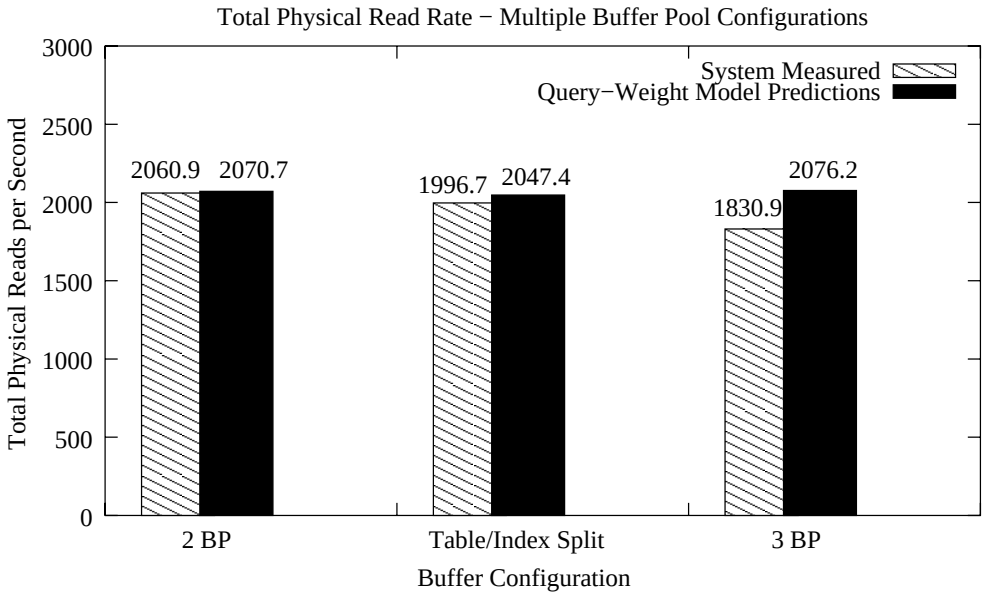

Figure 7.7: Total physical read rates for multiple buffer pool configurations.

mines this estimate from information in the query access plans.

Figure 7.7 shows the total physical read rate for each of the multiple buffer pool configurations. The model is able to accurately predict the actual total physical read rate for the 2 BP and Table/Index split configurations. The prediction for the 3 BP configuration shows a greater discrepancy, about 14%, from the measured total. This is likely due to the increased discrepancies in the 3 BP buffer occupancies, which results in larger deviations between the predicted physical read rate and the system measured physical read rate.

Large Table Scan and Sort Behaviour

The TPC-C workload contains primarily short read and update transactions. We are interested in validating our model on a wider range of workloads, specifically, workloads that contain large table scans and sorts. We modified the TPC-C transactions to include table scans of both large and small tables, and sorts of small and mid-sized tables. Our experiments used a total buffer size of 500 MB. We first discuss the results of our table scan tests followed by the results of our table sort

tests.

Large Table Scans

To test the impact of table scans, we added the following three queries that cause table scans to two of the TPC-C transaction types:

1. **Oline Scan**: in the ORDER-STATUS transaction, we added the query SELECT OL_AMOUNT FROM ORDER_LINE.
2. **History Scan**: in the ORDER-STATUS transaction, we added the query SELECT H_AMOUNT from HISTORY.
3. **Oline_Hist Scan**: in addition to the above two scans, we added the query SELECT OL_DELIVERY_D FROM ORDER_LINE in the PAYMENT transaction.

We expect two results from our table scan tests:

1. The buffer occupancy estimate of each scanned table should be approximately equal to its estimate when there is no table scan. Since the weights of sequentially scanned pages are set to zero, their buffer residency time is minimal.
2. The predicted physical read rate of each sequentially scanned table should increase over its no scan prediction, since an increased number of pages must be fetched from disk to read the table.

Figure 7.8 compares the measured and predicted buffer occupancy values for the ORDER_LINE and HISTORY tables in the Oline Scan and History Scan tests, respectively. As expected, the model prediction for each of the two tables is approximately equal to its original buffer occupancy prediction when there is no table scan. The discrepancies between the model predictions and the system measured buffer occupancies, for the scan cases, may be explained by the following reason. In practice, the buffer manager tags a sequentially scanned table page with a zero weight. The page will reside in the buffer until the clock evicts the page or another query re-references it. If a query re-references the page, this logical reference will

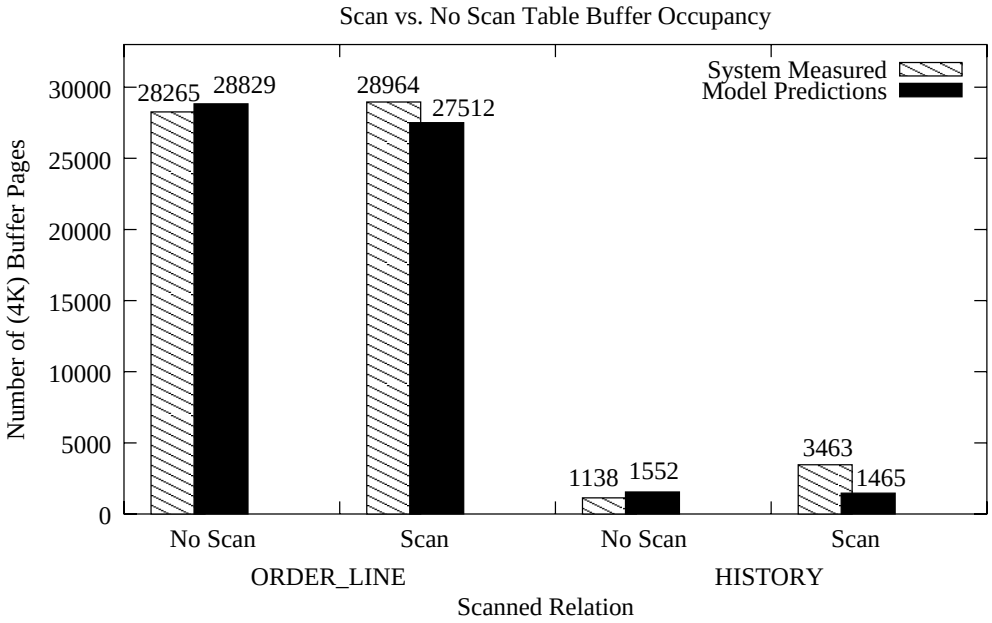

Figure 7.8: Sequentially scanned table buffer occupancy.

correspond to a buffer hit. However, our model assumes zero weight pages are evicted from the buffer and considers this logical reference to be a physical read. This immediate eviction assumption for zero weight pages can cause the model to predict lower buffer occupancy values and higher physical read rates.

Figure 7.9 shows the physical read rate comparison between the measured and the model predictions. As expected, we observe that the physical read rates for both tables have increased from their original estimates when there were no table scans. The model, however, has predicted larger physical read rate increases in both tables than the system measured increases. A possible reason for this increase is the immediate eviction assumption described above.

In the Oline_Hist Scan test, the buffer occupancy measurements and predictions and the physical read rate measurements for the ORDER_LINE and HISTORY tables are similar to the values of the previous two scan tests. However, the model estimates a total of 1248 physical reads/sec for the ORDER_LINE table. This increase is approximately equal to the increase in predicted physical reads of the ORDER_LINE table in the Oline test case over the ORDER_LINE no scan predic-

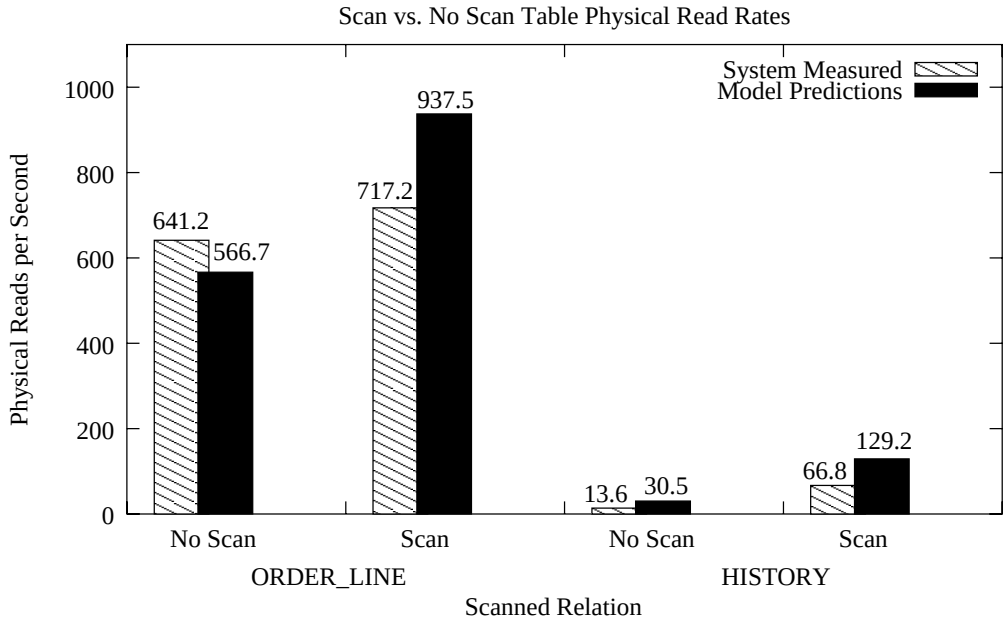

Figure 7.9: Sequentially scanned table physical read rates.

tion. In the Oline_Hist case, the model predicts a larger physical read rate for the ORDER_LINE table because there are two scan queries referencing this table.

Figure 7.10 shows the total physical read rate, across all objects, in each of the three table scan tests. The larger discrepancy in the Oline_Hist test can be explained by the increased ORDER_LINE physical read rate estimate mentioned previously. Our model is sensitive to changes to the object buffer occupancy and physical read rate due to sequential scan behaviour. However, our results have shown that the model over estimates the physical read rate of sequentially scanned tables. Query re-references to zero weight buffer pages can cause the model to predict increased object and overall total physical read rates.

Table Sorts

To validate our model on a workload that contains table sorts, we added sort activity to the standard TPC-C workload by using one client to sequentially generate sort queries with no think time. The client was capable of generating the following queries. One query is used per experiment:

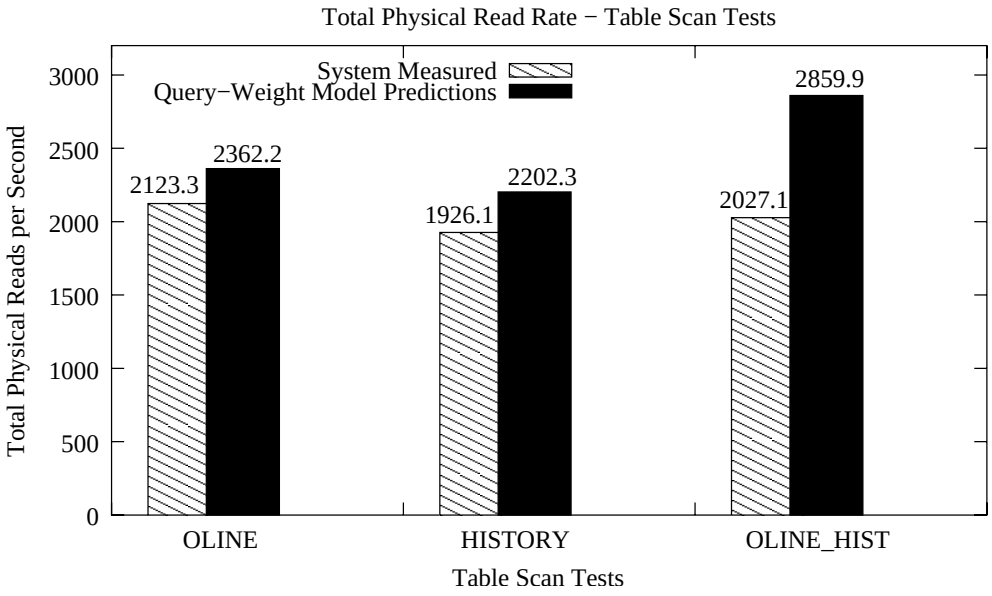

Figure 7.10: Total physical read rates for table scan tests.

1. **History Sort:** A dynamic sort query on the HISTORY table: SELECT H_DATE FROM HISTORY ORDER BY H_DATE.

2. **Item Sort:** A dynamic sort query on the ITEM table: SELECT I_PRICE FROM ITEM ORDER BY I_PRICE.

Figures 7.11 and 7.12 show the model's physical read rate predictions for the HISTORY and ITEM tables, respectively. In Figure 7.11, the model under estimates the HISTORY table's physical read rate and accurately predicts the physical read rate of the TEMP table. The model's under estimation for the HISTORY table may be caused by a few factors. One possibility is that the query optimizer under estimated the number of table pages that were to be sorted, leading the model to predict fewer physical reads. Another possible cause is that the EXTRACT algorithm assumes the sort routine will read the table pages once from disk. The algorithm does not consider repeated physical reads to the table pages, which can occur during the sort routine if the sort spills to the disk, causing the model to predict fewer physical reads.

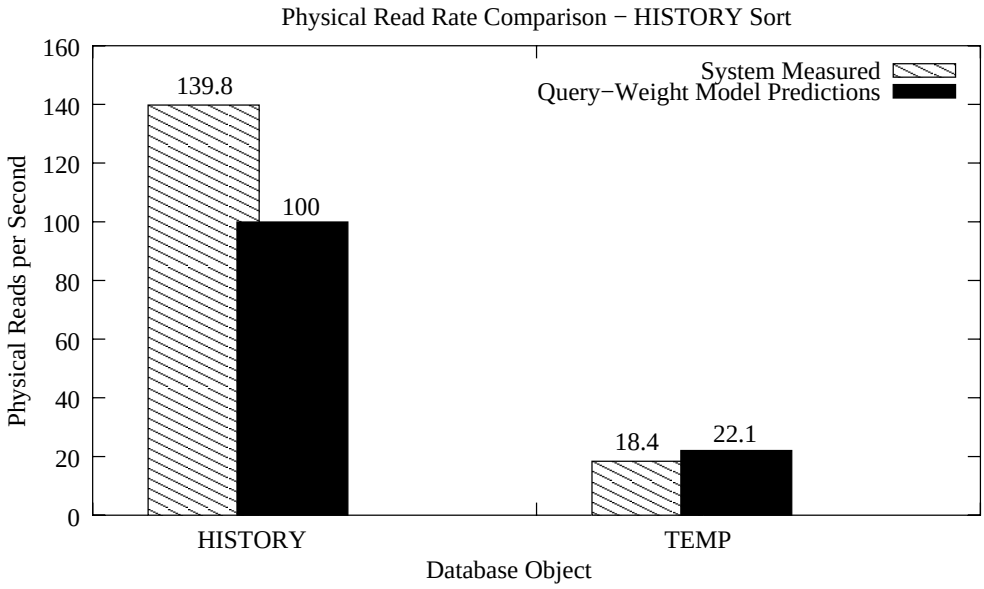

Figure 7.11: Object physical read rate comparison for History sort.

Figure 7.12 shows that the model over estimates the physical read rate of both the ITEM and TEMP tables. This may be a result of fewer disk accesses needed to perform the sort, since most of the sort can be performed within the sort memory.

The total physical read rate of each sort workload is shown in Figure 7.13. The measured and model predicted values are almost indistinguishable. The sort workloads generate a minimal increase in the total physical read rate over the standard TPC-C workload. Sorts on larger tables will likely generate a noticeably larger number of total physical reads.

The results from Figure 7.13 show that our model is able to accurately predict the total physical read rate of workloads with minimal sort activity. However, the model's physical read rate predictions for individual sorted tables requires further improvement.

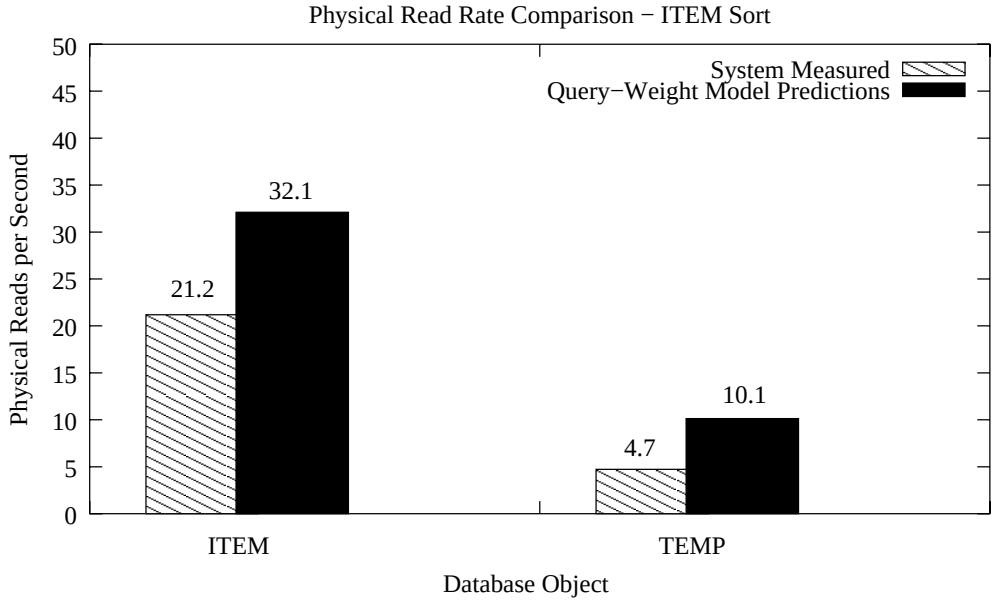

Figure 7.12: Object physical read rate comparison for Item sort.

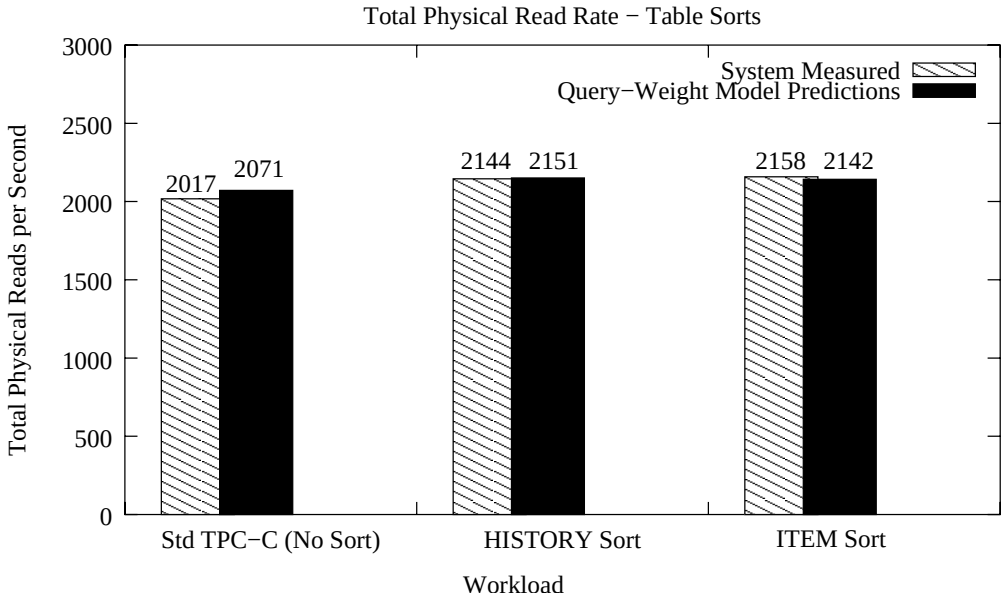

Figure 7.13: A comparison of our varied TPC-C sort workloads against the standard TPC-C workload.

7.3.2 Comparison between our Query-Weight Model and the Approximate Markov Model

We are interested in studying how our Query-Weight model performs relative to the approximate GCLOCK Markov model (Approx-GCLK Model) [NDD92], by comparing the predictions from both models against system measured values. First, we evaluated both models using the standard TPC-C workload under a set of buffer configurations with varying total buffer sizes. Second, we evaluated both models under a modified TPC-C workload that included large table scans. For both experiments, we used a 100 WH SF database. For the second experiment, we used a 500 MB total buffer size.

To calculate the Approx-GCLK model predictions, we used a total of 76 partitions (19 objects times 4 classes/object), corresponding to the number of classes used by our Query-Weight model. The size of each partition was equal to the size of our corresponding class. We used the extracted workload parameters to derive the parameter r_p, the access probability to each partition p, which is required by the Approx-GCLK model. We assigned a weight of 1 to all table pages and a weight of 2 to all index pages in the Approx-GCLK model. We first describe our experimental results with the varying buffer size configurations, followed by our results from the table scan tests.

Model Comparison - Varying Buffer Sizes

Figure 7.14 shows the total physical read rate prediction from each model across buffer sizes ranging from 120 MB to 1.2 GB. Our model tends to slightly over estimate the actual total physical read rate, while the Approx-GCLK model significantly under estimates. Our model offers improved accuracy across the entire range of evaluated buffer sizes.

Model Comparison - Sequential Table Scans

Our model considers query based weight assignments, i.e., weights are based on the query reference type. For workloads involving large table scans, we expect our

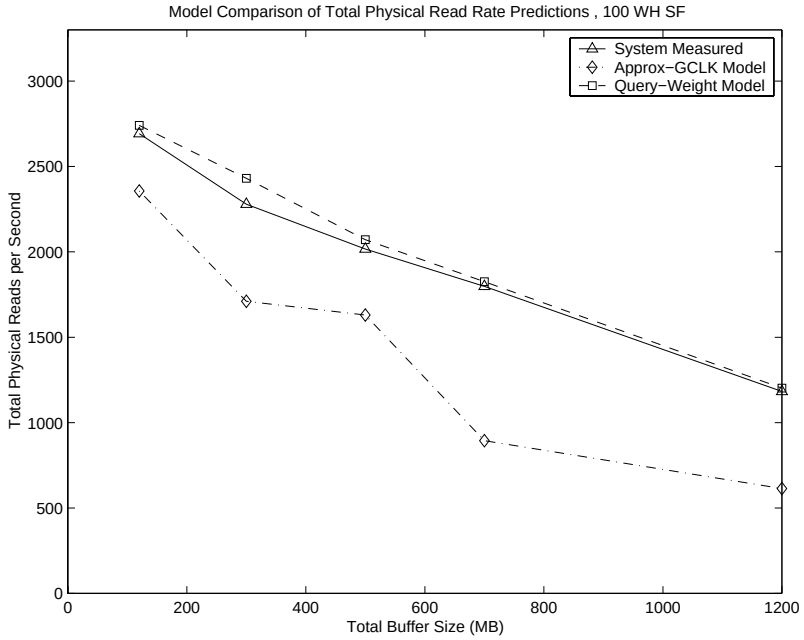

Figure 7.14: A comparison of the Query-Weight versus Approx-GCLK total physical read rate predictions.

model to predict more accurate buffer occupancy and physical read estimates than the Approx-GCLK model.

We evaluated both models on a modified TPC-C workload that included large table scans. Specifically, we modified the TPC-C PAYMENT transaction to include the following queries:

- SELECT s_ytd FROM stock
- SELECT ol_delivery_ d FROM order_line
- SELECT c_ytd_payment FROM customer

Figure 7.15 shows the normalized buffer occupancy estimates for each database object. We observe that with the exception of the CUSTOMER and ITEM tables, our model estimates are more accurate than the Approx-GCLK estimates.

Figure 7.16 shows the normalized object physical read rate predictions. For most of the data objects in the workload, our model predictions offer greater ac-

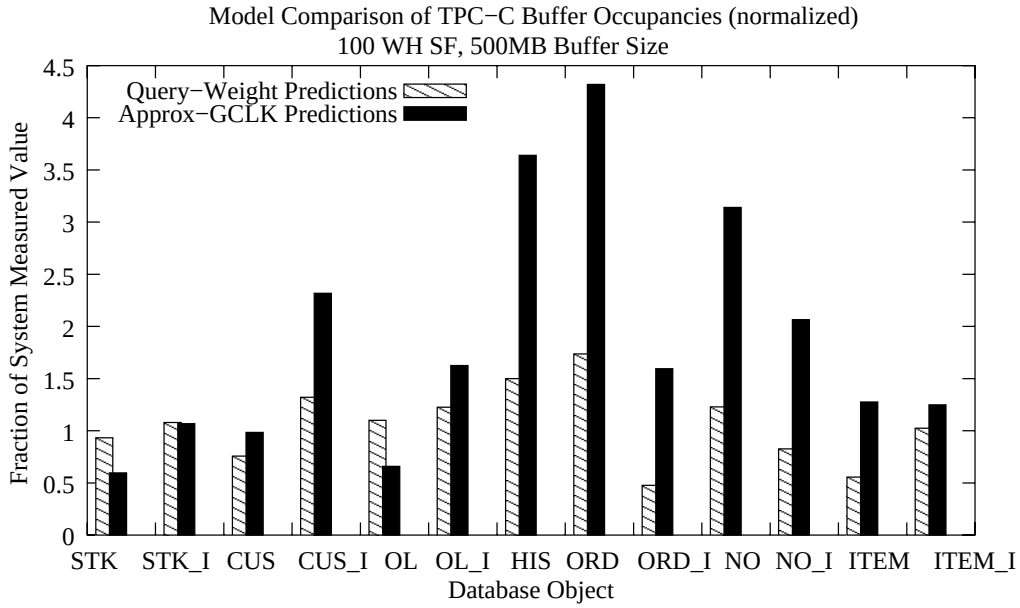

Figure 7.15: A comparison of the Query-Weight versus Approx-GCLK object buffer occupancy predictions using a modified TPC-C workload with sequential table scans.

curacy than the Approx-GCLK predictions. The Approx-GCLK model provides better predictions for the ORDER_LINE, HISTORY and ORDER tables. For each sequentially read table, both models over estimate the actual number of physical reads. This is a likely consequence of both models assuming independent page references and an increased number of disk accesses required to sequentially read the table pages.

The system measured a total physical read rate of 2394 physical reads/sec. Our Query-Weight and the Approx-GCLK model predicted 3212 and 2805 physical reads/sec, respectively. Our initial results indicate that for most of the database objects in the TPC-C workload, our model offers improved accuracy when predicting the individual object buffer occupancy and physical read rate. However, the Approx-GCLK model has shown better predictions for a small set of tables and for the total physical read rate. Further experiments using scan intensive workloads are needed to investigate these discrepancies.

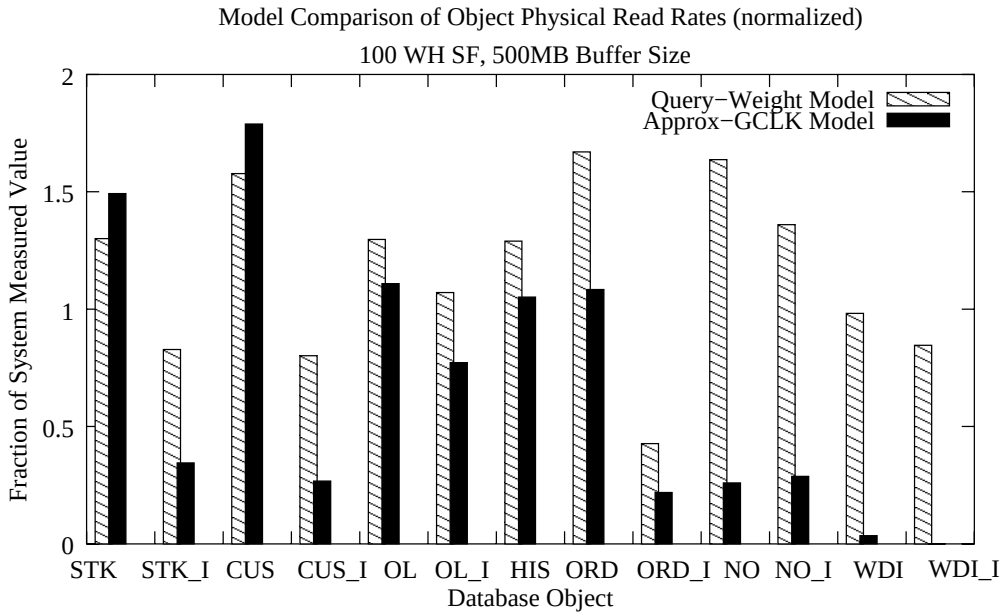

Figure 7.16: A comparison of the Query-Weight versus Approx-GCLK object physical read rate predictions using a modified TPC-C workload with sequential table scans.

7.3.3 Validating Predictive Capability

We conducted experiments to study our model's ability to distinguish given buffer configurations that provide increased system throughput over those that will result in throughput degradation. As this section shows, reducing the number of total physical reads increases the system throughput. We therefore use the total number of physical reads as the performance metric to evaluate our model's system throughput predictive capability.

We evaluated our model's predictive capability on a set of buffer configurations, shown in Table 7.4. For these experiments, we used a total buffer size of 1.1 GB. The experiments were executed for 45 minutes with a 15 minute ramp-up period.

These buffer configurations were derived with the objective of assigning separate buffer pools to large, frequently referenced tables, specifically, the STOCK, CUSTOMER and ORDER_LINE tables, in order to minimize the total number of physical reads. We also evaluated a case where the STOCK index was assigned a separate buffer pool to determine if that would help improve system throughput.

Figure 7.17 shows the mean system measured TPC-C throughput for each configuration. Buffer configurations C1, C2 and C3 each show a throughput gain. The remaining configurations C4-C10 each show a decline in the throughput relative to the default buffer configuration, with C10 performing the worst. For buffer configurations C1, C2 and C3, we expect that the model will predict a total physical read rate close to or less than the physical read rate of the default configuration. For each buffer configuration in the unsatisfying set C4-C10, we expect the model to predict a larger total physical read rate than the default configuration. Furthermore, we expect the model to predict higher read rates for buffer configurations that perform more poorly than others.

Figures 7.18 and 7.19 show the system measured and model predicted total physical read rates, respectively, for each buffer configuration. The results from Figure 7.18 show that each of the C1-C3 buffer configurations had fewer total physical reads than the default configuration. In contrast, all buffer configurations that showed a decline in throughput had a greater number of total physical reads. As expected, configurations C9 and C10, which showed the worst throughput performance, had the greatest physical read rate increase.

Configuration	**Buffer Pool Assignments**
Default	BP1 (1.1 GB): All database objects.
C1	BP1 (495 MB): STK BP2 (605 MB): all remaining objects.
C2	BP1 (495 MB): STK BP2 (182 MB): OL BP3 (423 MB): all remaining objects.
C3	BP1 (177 MB): STK, OL BP2 (150 MB): STK_I BP3 (240 MB): all remaining objects.
C4	BP1 (670 MB): STK, CUS BP2 (430 MB): all remaining objects.
C5	BP1 (495 MB): STK BP2 (175 MB): CUS BP3 (430 MB): all remaining objects.
C6	BP1 (100 MB): STK_I BP2 (1 GB): all remaining objects.
C7	BP1 (250 MB): CUS BP2 (850 MB): all remaining objects.
C8	BP1 (390 MB): OL BP2 (710 MB): all remaining objects.
C9	BP1 (580 MB): OL BP2 (520 MB): all remaining objects.
C10	BP1 (500 MB): CUS BP2 (600 MB): all remaining objects.

Table 7.4: Buffer configurations used to test the model's predictive capability.

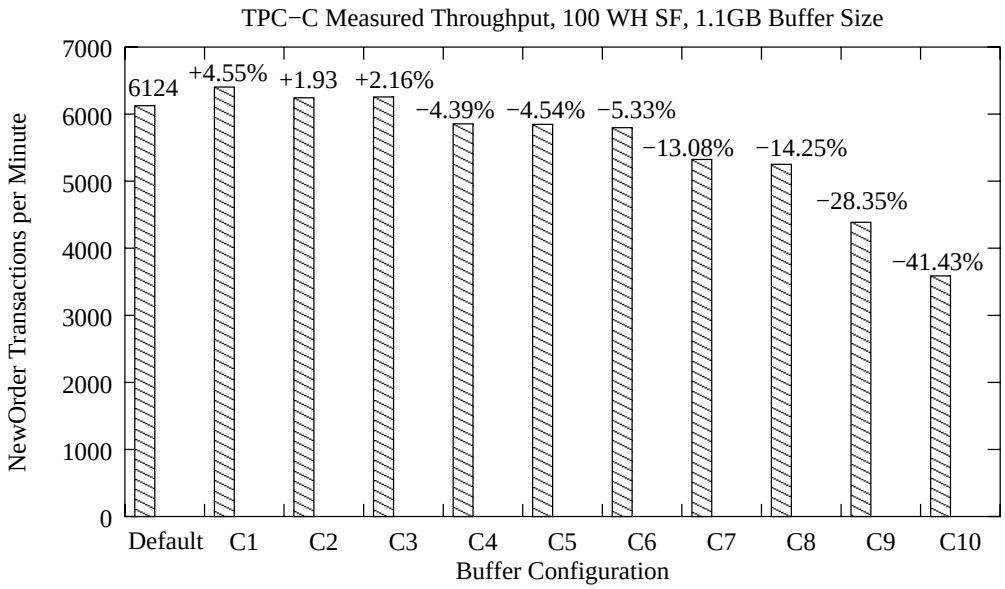

Figure 7.17: Mean TPC-C throughput.

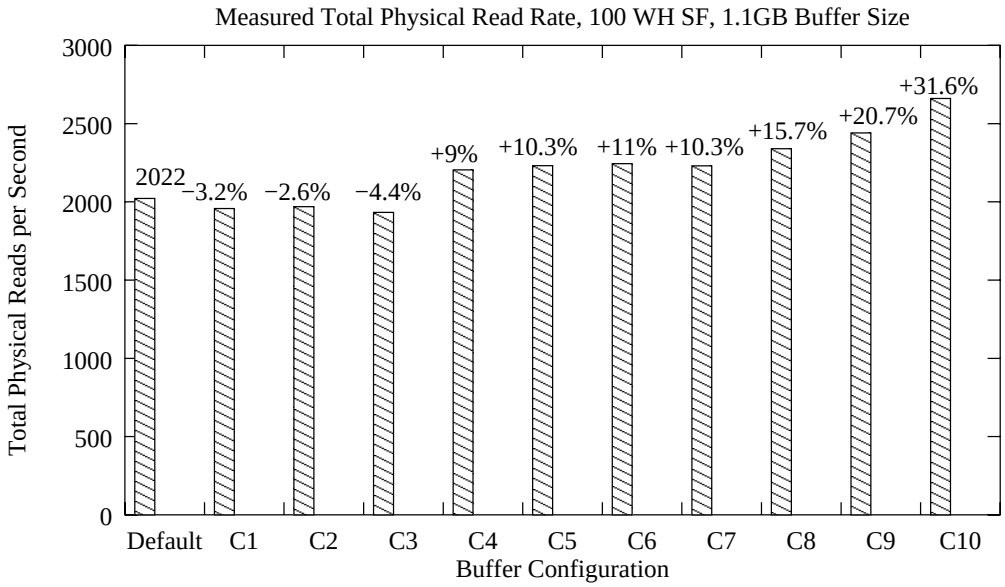

Figure 7.18: Measured mean total physical read rates.

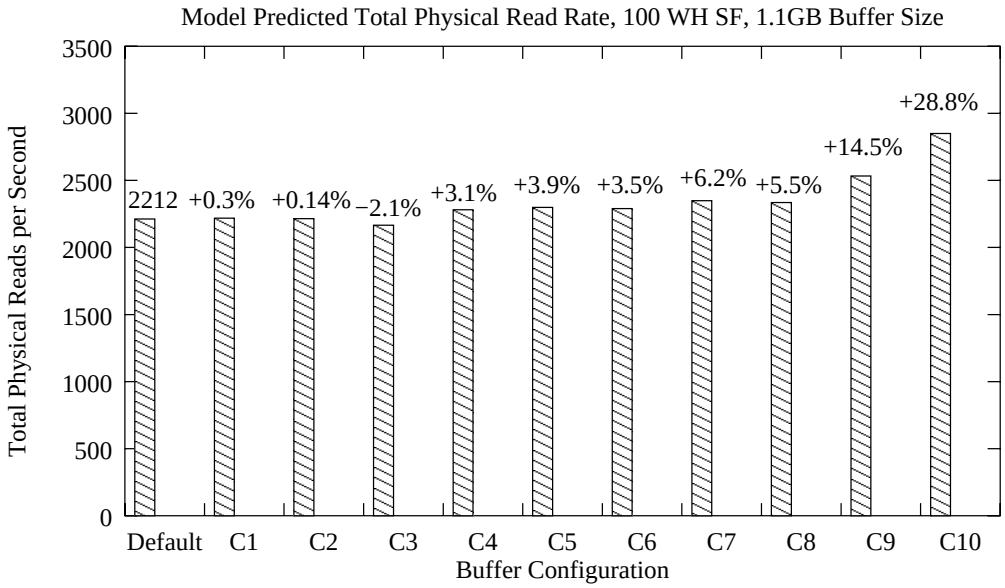

Figure 7.19: Model predicted total physical read rates.

When comparing Figure 7.19 with Figure 7.18, one can see that the model's total physical read rate prediction of 2212 physical reads/sec is about 9.5% greater than the measured total of 2022. The predictions in Figure 7.19 show that our model is able to distinguish the preferable configurations, C1, C2 and C3. The predictions for C1 and C2 are approximately equal to the physical read rate of the default configuration. For configuration C3, which showed the greatest physical read rate decrease over all configurations at -4.4%, the model also predicted the greatest decrease over all buffer configurations at -2.1%. As expected, for each buffer configuration that demonstrated a throughput gain, the model predicted values close to or less than the default configuration's total physical read rate.

For each buffer configuration in the unfavourable set C4-C10, the model's estimates were between 3%-30% larger than the default configuration's physical read rate. Furthermore, the results show that our model is able to distinguish those buffer configurations that perform extremely poorly relative to other configurations. The model estimates the greatest physical read rate increases for configurations C9 and C10, at 14.5% and 28.8%, respectively.

From our experimental results, we observe that our model is able to distinguish

favourable versus unfavourable buffer configurations. Favourable buffer configurations, which show an improvement in system throughput, are a consequence of a decrease in the number of total physical reads. In contrast, unfavourable buffer configurations show a reduction in throughput; a consequence of an increased number of physical reads. In addition, our model is able to detect those configurations that perform extremely poorly (large throughput reductions), by predicting a larger total physical read rate.

Our model's ability to distinguish favourable versus unfavourable buffer configurations is useful for automatic database buffer configuration. Assuming a mechanism that generates candidate buffer configurations, our model is able to evaluate a given configuration and predict whether applying the buffer configuration will improve or degrade system throughput.

7.4 Summary

We draw the following conclusions from our experiments:

- The model quite accurately predicts the buffer occupancy and physical read rate for frequently referenced objects in the TPC-C workload assuming a uniform access skew. The model provides more accurate predictions when given improved access skew values for an object. Discrepancies between the measured values and the model predictions are possibly caused by the independent page reference assumption.

- The model's total physical read rate predictions scale well with increasing buffer size.

- For multiple buffer pool configurations, our model closely predicts the buffer occupancy for the majority of the data objects in each buffer pool, and the total physical read rate.

- Our model is sensitive to changes to the object buffer occupancy and physical read rate due to sequential scan behaviour. Query re-references to zero weight

buffer pages can cause the model to predict less accurate (increased) object and total physical read rates.

- Our initial results showed that our model is able to accurately predict the total physical read rate of workloads with minimal sort activity. However, further experiments are needed to validate the model's ability to handle sort intensive workloads. In addition, the model's physical read rate predictions for individual sorted tables needs to be improved.

- For a wide range of increasing buffer sizes, our model offers improved accuracy over the Approx-GCLK model for predicting total physical read rates.

- For the modified TPC-C table scan workload, our results indicate that for the majority of the database objects in the workload, our model offers improved accuracy over the Approx-GCLK model when predicting the object buffer occupancy and physical read rate. However, the Approx-GCLK model provides better predictions for a small set of tables and for the total physical read rate. Further experiments using scan intensive workloads are needed to further investigate these model distinctions.

- Our model is able to distinguish between favourable and unfavourable buffer configurations. For all favourable configurations, our model's total physical read rate predictions were either less than or within 1% of the default configuration's physical read rate. The model's ability to detect favourable buffer configurations helps an automatic buffer configuration tool recommend non-default buffer configurations that are expected to improve system throughput. Furthermore, our model is able to detect those buffer configurations that perform extremely poorly relative to other configurations, by predicting larger physical read rate totals.

Chapter 8

Conclusions

In this thesis, we evaluated the benefits of buffer configuration at database design time. Our experimental results showed that having a good initial buffer configuration can increase system throughput and reduce both the mean transaction response time and the total physical read rate. These results are further supported by previous empirical studies [LMM96, XMP02]. Determining an appropriate buffer configuration at database design time requires knowledge of the workload. Previous methods have used reference traces as their source of query access pattern information. Since reference traces are not available at database design time, the initial buffer configuration task is challenging without sufficient query access pattern information.

When the buffer pools are effectively configured and tuned, they help to reduce query response times and increase system throughput and buffer hit rates by minimizing the number of disk accesses. Configuring and tuning the buffer pools is a manual and difficult process. Users and database administrators are often hindered by limited knowledge and time to determine the best buffer configuration for their workload. Having a DBA manually evaluate all possible buffer configurations is clearly impractical. Previous methods have not considered automating the buffer configuration process.

We have developed an analytic model of the GCLOCK buffer replacement policy that can be used to evaluate the effectiveness of a particular buffer configuration for a given workload. To derive the workload parameters required by our model,

we proposed a workload extraction algorithm that extracts query reference patterns from the access plans. The workload extraction algorithm can be applied at database design time to obtain the necessary query access pattern information. In addition, we extended an existing multifractal model and introduced a multifractal skew model that models the distribution of query page references over object classes. The skew model is able to generate a broad range of access skew based on a few parameters.

We have given an overview of how our proposed buffer model can be used in conjunction with our workload characterization scheme and a configuration optimizer to automate the initial buffer configuration task. The buffer model can also be used after database design time to help automate the buffer configuration process by evaluating candidate buffer configurations.

Our buffer model improves upon the only alternative GCLOCK model (Approx-GCLK) [NDD92] by considering a more generalized weight assignment policy, which distinguishes query page reference behaviour. Our model improves accuracy by considering weights to be assigned based on the page type and the query reference type. Comparative tests between our buffer model and the Approx-GCLK model showed that for increasing buffer sizes, our model offers improved accuracy for predicting the total physical read rate. For a modified TPC-C workload involving large table scans, the comparative model results were mixed. Our model showed greater accuracy in predicting the buffer occupancy and physical read rate for most of the database objects in the workload. However, the Approx-GCLK model provided better predictions for a small set of tables and the total physical read rate. Further experiments using scan intensive workloads are required to investigate these model differences.

We conducted an extensive set of experiments to validate our model predictions against system measurements using a wide range of buffer configurations. Our results showed that given correct access skew parameters, our model is able to quite accurately predict the buffer occupancies and physical read rates of frequently referenced objects in the TPC-C workload. The model's total physical read rate predictions scale well with increasing buffer size. In addition, our model is sensitive to multiple buffer pool configurations. It closely predicts the buffer occupancy for

the majority of the data objects in each buffer pool, and the total physical read rate. We also validated our model using a modified TPC-C workload containing table scans and minimal sort activity. Our initial results indicate that our model is sensitive to changes in the object buffer occupancy and physical read rate due to the scan and sort activity. However, discrepancies in the physical read rate predictions indicate that further investigation is needed to refine the model to better handle scan and sort intensive workloads. Finally, our predictive capability experiments showed that our model is able to accurately distinguish and predict those buffer configurations that improve system throughput from those that degrade throughput.

The above results are an encouraging indication that our model provides accurate predictions for OLTP type workloads. Our proposed buffer model provides the foundation for an effective mechanism that evaluates candidate buffer configurations in an automated buffer configuration tool.

The following are suggestions for future work:

- Further model validation using a wider range of workloads. In particular, database workloads that involve large sequential scans, sorts and longer running queries.
- Development of the configuration optimizer, which generates candidate buffer configurations for evaluation by the buffer model.
- Consideration of temporal and spatial page locality by modelling the query page reference process using heavy tailed distributions, which consider correlated page reference characteristics.
- Extension of the EXTRACT algorithm to consider a larger number of operators.

Appendix A

Circular Dependency Problem

In Chapter 5, we showed how the EXTRACT algorithm exploited the access plans to derive a characterization for a given workload. Our buffer model used this workload characterization to evaluate and recommend a candidate buffer configuration. However, the access plans are originally derived from a buffer configuration - resulting in a cyclic dependence. We describe the circular dependency problem and explain how it affects our buffer configuration methodology.

Figure A.1 shows an overview of the dependency cycle. A DBA passes a buffer configuration (consisting of the total buffer size, the number of buffer pools and their respective sizes and object assignments) and a workload declaration to the query optimizer. The optimizer generates a set of access plans based on its inputs. Our workload characterization (i.e., EXTRACT algorithm) and buffer model use these access plans to evaluate the quality of a given candidate buffer configuration. The problem is, how can our proposed techniques evaluate and recommend a buffer configuration when one of its inputs is also based on a buffer configuration?

The break in the cycle comes from the fact that query optimizers do not necessarily consider all four buffer configuration options when evaluating their access plan decisions. For some database systems, the optimizer considers only the total buffer size, i.e., if the total buffer size changes, new access plans must be generated. If the number of buffer pools, or their respective sizes or object assignments change, the access plans are unaffected. Thus, the dependency cycle is partially broken between steps 1 and 2, where the access plans are insensitive to all buffer

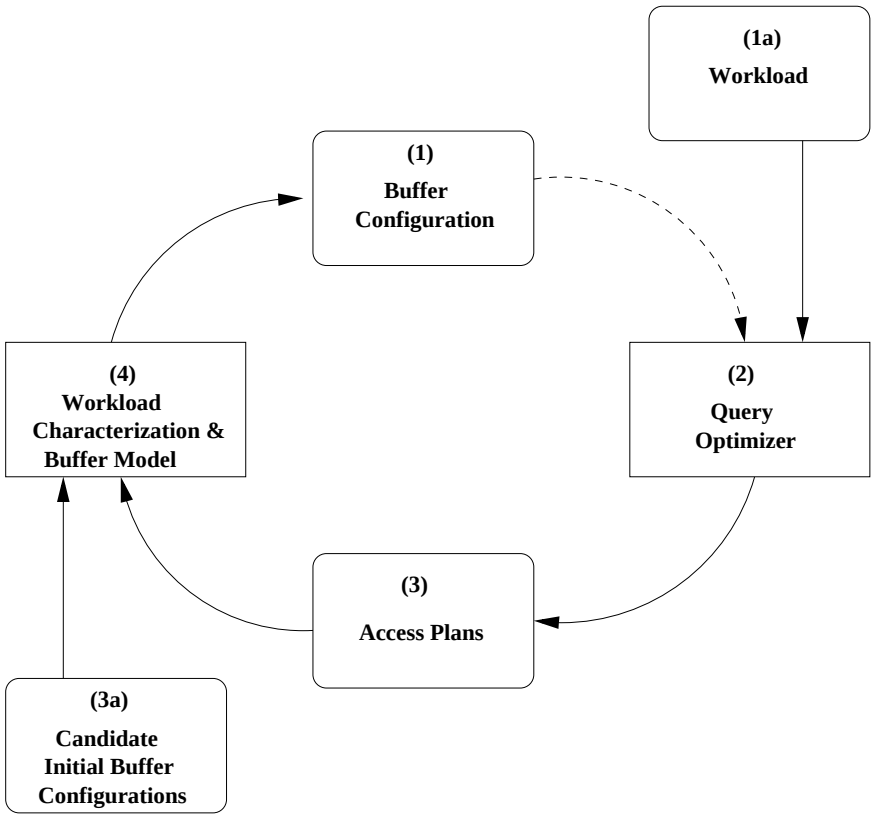

Figure A.1: The circular dependency problem.

configuration changes, except the total buffer size.

Therefore, for a given fixed total buffer size, our buffer configuration methodology remains valid to evaluate a set of candidate initial buffer configurations, possibly with changing object assignments, buffer sizes or the number of buffer pools. Any change to the total buffer size requires that the user re-generate the access plans and re-execute the EXTRACT algorithm.

If future query optimizers are enhanced to consider the remaining buffer configuration options, then any buffer configuration change will require that new access plans be generated. The effect on our buffer configuration methodology is that the buffer model will evaluate only the buffer configuration on which the access plans are based. The evaluation of each new buffer configuration will involve generating a new access plan. However, our workload characterization scheme and buffer model are still valid to effectively evaluate the quality of a given buffer configuration for a workload without having to actually run the workload.

Bibliography

[AS01] Yongli An and Peter Shum. *DB2 Tuning Tips for OLTP Applications*, DB2 Universal Database Performance and Advanced Technology, IBM Toronto Lab, IBM Canada, July 2001.

[BCL93] Kurt P. Brown, Michael J. Carey, Miron Livny. *Managing Memory to Meet Multiclass Workload Response Time Goals*, VLDB 1993: 328-341.

[BCL96] Kurt P. Brown, Michael J. Carey, Miron Livny. *Goal-Oriented Buffer Management Revisited*, SIGMOD Conference 1996: 353-364.

[BEL66] L. Belady. *A Study of Replacement Algorithms for a Virtual-Storage Computer*, IBM Systems Journal, 5(2), July 1966.

[CD85] Hong-Tai Chou, David J. DeWitt. *An Evaluation of Buffer Management Strategies for Relational Database Systems*, VLDB 1985: 127-141.

[CFWNT95] J.Y. Chung, D. Ferguson, G. Wang, C.Nikolaou, J.Teng. *Goal Oriented Dynamic Buffer Pool Management for Database Systems*, Workshop on Quality of Service in Open Distributed Processing. Brisbane, Australia, 1995.

[CLR92] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*, MIT Press, 1992.

[CR93] Chung-Min Chen, Nick Roussopoulos. *Adaptive Database Buffer Allocation Using Query Feedback*, VLDB 1993: 342-353.

[CS89] Ignacio R. Casas, Kenneth C. Sevcik. *A Buffer Management Model For Use In Predicting Overall Database System Performance*, ICDE 1989: 463-469.

[CY89] Douglas W. Cornell, Philip S. Yu. *Integration of Buffer Management and Query Optimization in Relational Database Environment*, VLDB 1989: 247-255.

[CY93] Douglas W. Cornell, Philip S. Yu. *Buffer Management Based on Return on Consumption in a Multi-Query Environment*, VLDB Journal 2(1): 1-31, 1993.

[DB01] *DB2 SQL Reference v.7.2*, IBM Corporation, 2001.

[DDY94] Asit Dan, Daniel M. Dias, Philip S. Yu. *Buffer Analysis for a Data Sharing Environment with Skewed Data Access*, TKDE 6(2): 331-337, 1994.

[DT90] Asit Dan, Don Towsley. *An approximate analysis of the LRU and FIFO buffer replacement schemes*, ACM SIGMETRICS Performance Evaluation Review, Proceedings of the 1990 ACM SIGMETRICS, April 1990.

[DYC95] Asit Dan, Philip S. Yu, Jen-Yao Chung. *Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability*, VLDB Journal 4, 127-154, 1995.

[EH84] W. Effelsberg, T. Haerder. *Principles of Database Buffer Management*, ACM TODS 9(4), pages 560-595, Dec. 1984.

[FMS96] C. Faloutsos, Y. Matias, A. Silberschatz. *Modeling skewed distributions using multifractal and the 80-20 law*, VLDB 1996: 307-317.

[HAYES00] Scott Hayes. *Bufferpool Tuning*, IDUG Solutions Journal, Spring 2000, vol. 7, no.1.

[HSY01] W. Hsu, A.J. Smith, H. Young. *Analysis of the Characteristics of Production Database Workloads and Comparison with the TPC Benchmarks*, IBM Systems Journal, Vol. 40(3), 2001.

[JAIN91] R. Jain. *The Art of Computer Systems Performance Analysis*, Wiley-Interscience, April 1991.

[JCL90] R. Jauhari, M. Carey, M. Livny. *Priority-Hints: An algorithm for priority based buffer management*, VLDB 1990: 708-721.

[KD89] J. P. Kearns and S. Defazio. *Diversity in database reference behaviour*, Performance Evaluation Review, 17(1):11-19, May 1989.

[LD93] Scott T. Leutenegger, Daniel M. Dias. *A Modeling Study of the TPC-C Benchmark*, SIGMOD Conference 1993: 22-31.

[LM96] Hanoch Levy, Robert Morris. *Should Caches be Split or Shared? Analysis using the Superposition of Bursty Stack Depth Processes*, Performance Evaluation 27 & 28, 1996, pgs.175-188.

[LMM96] Hanoch Levy, Ted Messinger, Robert Morris. *The Cache Assignment Problem and its Application to Database Buffer Management*, Transactions on Software Engineering, November 1996 (Vol. 22, No. 11).

[MLZRP00] Patrick Martin, Hoi-Ying Li, Min Zheng, Keri Romanufa, Wendy Powley. *Dynamic Reconfiguration Algorithm: Dynamically Tuning Multiple Buffer Pools*, DEXA 2000: 92-101.

[NDD92] V. F. Nicola, A. Dan, and D. M. Dias. *Analysis of the generalized clock buffer replacement scheme for database transaction processing*, ACM SIGMETRICS and Performance, 1992.

[NFS91] Raymond T. Ng, Christos Faloutsos, Timos K. Sellis. *Flexible Buffer Allocation Based on Marginal Gains*, SIGMOD Conference 1991: 387-396.

[OR00] *Oracle8i Designing and Tuning for Performance*, Release 2 (8.1.6). Oracle Corporation, 2000.

[SCH91] Manfred Schroeder. *Fractals, Chaos, Power Laws, Minutes from an Infinite Paradise*, pages 187-189, 1991.

[SS82] Giovanni Maria Sacco, Mario Schkolnick. *A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model*, VLDB 1982: 257-262.

[STO96] A. Storkey. *The Fractal and Multifractal Nature of Traffic*, UTSG Conference, January 1996.

[TPC02] Transaction Processing Council. *TPC Benchmark C*, www.tpc.org/tpcc/, December 2002.

[TPK97] T. Tsuei, A. Packer, and K. Ko. *Database buffer size investigation for OLTP workloads*, SIGMOD Conference 1997: 112-122.

[TSW92] D. Thiebaut, H. Stone, J. Wolf. *Improving Disk Cache Hit-Ratios Through Cache Partitioning*, IEEE Transactions on Computers, Vol. 41, No. 6, June 1992.

[WMCPF02] M. Wang, T. Madhyastha, N.H. Chan, S. Papadimitriou, C. Faloutsos. *Data Mining Meets Performance Evaluation: Fast Algorithms for Modeling Bursty Traffic*, 18th International Conference on Data Engineering, 2002.

[XMP01] Y. Xi, Patrick Martin, Wendy Powley. *An Analytical Model for Buffer Hit Rate Prediction*, Proceedings of CASCON 2001: November 2001.

[XMP02] Xiaoyi Xu, Patrick Martin, Wendy Powley. *Configuring Buffer Pools in DB2 UDB*, Proceedings of CASCON 2002: 171-182.

[XU01] Xiaoyi Xu. *A Clustering Approach to Configurting Buffer Pools in a Database Management System*, M.Sc. Thesis, Dept. of Computing and Information Science, Queen's University, 2001.