The VLDB Journal (2006) 15(2): 99–120
DOI 10.1007/s00778-004-0145-1

REGULAR PAPER

Chong Leng Goh · Yanfeng Shu · Zhiyong Huang · Beng Chin Ooi

Dynamic buffer management with extensible replacement policies

Received: 4 December 2002 / Accepted: 10 January 2004 / Published online: 22 July 2005

Abstract The objective of extensible DBMSs is to ease the construction of specialized DBMSs for nontraditional applications. Although much work has been done in providing various levels of extensibility (e.g., extensibility of data types and operators, query language extensibility, and query optimizer extensibility), there has been very limited research in providing extensibility at the buffer management level. Supporting extensibility at the buffer management level is important as it can contribute significantly to overall system performance.

This paper addresses the problem of supporting extensibility of buffer replacement policies. The main contribution is the proposal of a framework for modeling buffer replacement policies. This work is novel in two aspects. First, by providing a uniform and generic specification of buffer replacement policies, the proposed framework unifies existing work in this area. Second, our work introduces a new level of extensibility. None of the existing extensible DBMSs, to our knowledge, provides extensibility at the buffer management level. The proposed framework provides a basis for the construction of an extensible buffer manager as part of a 100% Java-based storage manager. We conducted an extensive performance study to investigate the performance of the proposed framework. The experimental results demonstrate that the proposed framework is indeed feasible for existing DBMSs and improves system performance significantly without costing significant overhead.

Keywords Buffer management · Replacement strategies · Extensible DBMS

Edited by M. Kersten

1 Introduction

Today's new and emerging applications such as scientific and statistical database systems, multimedia database systems, and geographical information systems require support for large complex objects and various types of data: inference rules, procedural data, spatial and temporal data, image, voice, and textual data. Moreover, each of these application areas also requires its own specific set of operations. Lack of support for the requirements of such applications in DBMSs has motivated several research efforts to widen the applicability of database technology to such new kinds of data-intensive applications. This has led to the advent of extensible DBMSs [34, 41, 45].

Extensible DBMSs are primarily designed to meet the following two objectives [6]:

1. They should provide support for the addition of new features or domain-specific extensions so that the system can be customized to support specialized data management and modeling requirements efficiently.
2. They should ease the problem of tracking developments in technology by simplifying the integration of new algorithms and new kinds of storage devices into existing DBMSs.

Since the mid-1980s the database research community has directed considerable effort toward the design and implementation of extensible DBMSs. Examples of such systems include EXODUS at the University of Wisconsin-Madison [5], GENESIS at the University of Austin-Texas [1], POSTGRES at the University of California-Berkeley [44], and Starburst at IBM's Almaden Research Center [21].

DBMS extensibility technology is a confluence of technologies in both databases as well as software engineering. A central issue in extensible DBMS research concerns how DBMSs should be built. Much of the research is focused on producing clean specifications of database components as well as their reusability and extensibility [49]. Existing extensible DBMS prototypes can be broadly classified into three types based on their approaches to providing extensibility.

– The first approach is to build an extensible complete end-user DBMS to handle all new application areas. This approach is based on a fixed architecture that accommodates

C. L. Goh(✉) Y. Shu · Z. Huang · B. C. Ooi
Department of Computer Science, National University of Singapore, Singapore 117543, Singapore

a wide range of features. POSTGRES and Starburst are examples of systems that employ this approach.
– The second approach, known as the toolkit approach, is to construct a DBMS generator that enables rapid customization of application-specific DBMSs. This approach is based on an open architecture that automatically generates a DBMS with the desired features by putting various library modules together. Examples of toolkit-based systems include DBMSs such as GENESIS and EXODUS.
– The third approach, known as component DBMS [13], is to extend DBMSs by adding or replacing components with different functionality. This approach requires a well-defined architecture that supports extension at certain places in the system.

To extend a DBMS component, the first approach entails making modifications to the component. Note that the type of extensions required depends on the interface of the component. For example, a procedural interface would require the addition of new code, while a rule-based interface would involve modification of the rules to incorporate the new capability into the existing system. For the second approach, customizing a component involves modifying and "recompiling" the component's specification to produce a new component. The third approach allows various implementations of a component and allows replacement or inclusion of components. For any approach, a generic and well-designed component interface/specification is critical to its extensibility as well as the ease of carrying out the desired extensions. In this paper, we shall assume the second approach, although the issue we address here can be taken as a single component of a component database.

Apart from the underlying architectural differences, extensible DBMSs also differ in their degree of extensibility. The levels of extensibility supported in various extensible DBMSs include extensions in data types as well as operators [32, 36, 37, 43, 48], query language extensions [30, 38], query optimizer extensions [16, 19, 22, 33], and storage and access method extensions [31].

Extensible DBMSs must be extensible at *all* levels since extensions made at a particular level of a DBMS usually require extension support at other levels [30]. For example, to introduce a new data type into an existing DBMS, the following DBMS components need to be extended: query parser, query optimizer, access method component, and storage manager. A more extensible system is likely to exploit a given application domain better, thereby providing a finer-grained control of the system's performance.

An important challenge in extensible DBMSs is the effective support of extensibility [6] without incurring too high a performance overhead. An important area that has received little attention in extensible DBMS research is the support for extensibility at the buffer management level. The buffer manager component plays a critical role in the performance of DBMSs by caching part of the database in main memory to minimize disk I/O. An important task of a buffer manager is buffer replacement, which determines how buffer pages are replaced when a page fault occurs. The goal is to manage the replacement of buffer pages efficiently to avoid unnecessary page faults. Using an appropriate buffering scheme to exploit a certain access pattern can have significant impact on the overall performance of a DBMS [42].

Conventional buffer managers, however, are built with only a single, fixed buffer replacement policy, which is usually the LRU replacement policy or a variant of it. Since there is no single universal replacement policy that can guarantee good performance for all applications, optimization at the buffer management level *cannot* be fully exploited. In existing extensible DBMSs, the buffer manager is also designed as a fixed component. For example, in EXODUS [5], the storage system (which includes the buffer manager) is a nonextensible component as it is a part of the kernel of the system. The buffer manager only supports two replacement policies, the LRU and MRU policies [15], and it is difficult to extend the buffer manager's repertoire of replacement policies as the code for the supported policies are all hardwired into the system. However, by supporting buffer replacement extensibility, replacement policies can be customized to provide more efficient buffering.

Supporting extensibility at the buffer management level (specifically, replacement policy extensibility) has the following advantages:

– With buffer replacement extensibility, it becomes possible to custom-tailor a "smart" buffer replacement policy that can manage buffer replacements intelligently by exploiting the reference behavior of new access methods. Furthermore, the emergence of more complex applications also provides more opportunities for exploiting domain-specific semantics to improve system performance.
– Providing extensibility at such a low level can contribute significantly to the system performance since disk I/O remains a critical performance bottleneck, in spite of the advances in disk hardware technology.
– Supporting buffer level extensibility facilitates the evaluation and fine-tuning of new buffer replacement policies for domain-specific operators, algorithms, and applications.

In this paper, we address the problem of efficiently supporting buffer replacement extensibility, i.e., enabling customization of buffer replacement policies. A fundamental issue in achieving buffer replacement extensibility is the design of a generic interface to specify buffer replacement policies. The goal of this paper is to provide a framework for specifying buffer replacement policies that will serve as the basis for the construction of an extensible buffer manager. The novel contributions of this work are as follows:

– By providing a uniform and generic specification of buffer replacement policies, the proposed framework unifies work in this area. Using this framework, we can specify existing buffer replacement policies as well as new replacement policies in a standardized way.
– By characterizing a replacement policy into a two-step process (selection of buffer group followed by selection

Table 1 Classification of existing work

Classification	Examples
Traditional buffer management	LRU policy, CLOCK policy.
Query-oriented buffer management	Hotset algorithm [39, 40], DBMIN algorithm [9, 10], LIRS [25], LRU-K [26, 35].
Real-time buffer management	Priority-LRU algorithm [7], Priority-DBMIN algorithm [7], Priority-Hints algorithm [24], Dynamic buffer allocation scheme [28].
Hint-based buffer management	WiSS replacement policy [11], Starburst replacement policy [27].

of buffer page), the framework provides a cleaner separation of concerns, resulting in a clearer conceptualization and modeling of replacement policies.
– This framework is an essential first step toward realizing buffer replacement extensibility. Based on this framework, we have proposed a component-generator-based approach to achieve buffer replacement extensibility. The introduction of extensibility at the buffer management level is novel in that none of the extensible DBMSs, to our knowledge, support extensibility at this level. As explained earlier, supporting extensibility at such a level can have a significant impact on the overall DBMS performance.

This paper is a major extension of [3], where we briefly introduced the framework and demonstrated its feasibility. In this paper, we provide a detailed description of the framework, definition of various popular replacement strategies using the framework, and thorough experimental studies.

The remainder of this paper is organized as follows. Section 2 presents a survey of existing buffer replacement policies. Section 3 introduces our framework for modeling buffer replacement policies. This framework consists of two components: a hierarchical buffer pool model (discussed in Sect. 3) and a priority scheme, which is presented in Sect. 4. Section 5 combines the two models of the previous two sections and presents the overall framework. Section 6 reports our performance study. Finally, we conclude in Sect. 7.

2 Survey of DBMS buffer replacement policies

This section presents a survey of DBMS buffer replacement policies in order to present the motivation behind the proposed framework. Moreover, these existing buffer replacement policies are also used as examples in the subsequent sections of this paper to illustrate the modeling abstractions using the proposed framework.

Early DBMSs either used a buffer management algorithm adapted from virtual memory systems or relied on the buffer management capabilities of the underlying operating system. A discussion of the traditional replacement policies as well as some new variants (e.g., generalized CLOCK and least reference density algorithms) can be found in [14]. However, many DBMSs of today bypass the buffering service of their underlying operating systems by implementing their own buffer managers. This is primarily due to two main reasons [42]:

– Operating systems do *not* support the controlled flushing of buffer pages, which is a requirement of the write-ahead-log protocol used for DBMS recovery.
– The standard global LRU page replacement policy used in operating systems does not offer efficient support for the page reference patterns of DBMSs.

Traditional OS page replacement policies attempt to predict future reference behavior by using past usage statistics. However, in relational DBMSs, the access plan for a query is generated by the query optimizer. Hence, information about the query reference pattern is known a priori and can be exploited. In fact, work on buffer management has progressed from the design of replacement policies that are based on stochastic measures to the design of more sophisticated approaches that integrate additional domain knowledge (e.g., page reference patterns and external domain hints) to obtain more intelligent buffering schemes.

Table 1 shows a classification of existing buffer replacement policies. Note that, except for the first category (traditional approaches), each of the other three categories exploits query semantics in order to improve management of the buffer pages. An overview of these three categories of buffer replacement algorithms will be given in subsequent subsections.

Note that a page in the buffer pool can be in one of two states: fixed or unfixed. When a client (e.g., access method) makes a request for access to a page, the buffer manager returns a handle to the buffer page that is holding the requested page (i.e., that buffer page is said to be fixed). The buffer page is said to be unfixed when the client notifies the buffer manager that it does *not* need that page. We note that only an unfixed page can be selected for replacement. In the case where no unfixed pages are available, some transactions may have to be rolled back in order to make some unfixed pages available for use.

2.1 Query-based buffer management

Unlike conventional stochastic approaches, the query-based approaches exploit knowledge about the reference patterns of queries to further optimize the utilization of buffer pages and minimize buffer faults. The buffer manager in DB2 [46], in a limited way, adopts this approach. Pages in the buffer are classified into two groups: those that are sequentially accessed and those that are randomly accessed; the page replacement policy selects pages from the former group in preference to those in the latter group.

Two such approaches are the hotset algorithm [39, 40] and the DBMIN algorithm [9, 10]. In the hotset model [39, 40], each query is allocated an effective number of buffer pages, known as the query's hotset. The calculation of a query's hotset size is performed during query optimization by analyzing the query's buffer fault behavior as a function of the buffer size. A new query is *not* admitted if the available free buffers cannot satisfy its hotset requirement because if scheduled, it might incur too many unnecessary page faults. Each query's allocation is managed locally using the LRU replacement policy.

Like Denning's working set model [12] for virtual memory operating systems, the hotset model tries to determine an optimal buffer allocation. The buffer pool is organized into a number of buffer groups, where each query is allocated a buffer group no larger than its hotset size. Each buffer group is managed locally using the LRU replacement policy. When a page request from a query is *not* found in the buffer pool, the least recently referenced unfixed buffer page, which is not shared, is selected from the query's local buffer group for replacement. If no such page is available, the algorithm will either attempt to obtain a page from the free list (which is also managed locally using LRU) or steal an unfixed page from another buffer group. The stealing of buffer pages can cause a buffer group to become *deficient*, i.e., its actual number of buffer pages is less than the hotset size of its associated query. The stealing of buffer pages should preferably be from an already deficient buffer group so as to minimize the number of deficient buffer groups. When a query terminates, its allocated pages are distributed to deficient buffer groups, one at a time; any remaining pages are returned to the free list. To overcome the limitation of LRU, the LRU-K [35] replacement policy has been proposed to collect and use longer historical information. It makes its replacement decision based on the time of the kth-to-last reference to the buffer page. More recently, a buffer replacement strategy called Low Interreference Recency Set (LIRS) [25] was proposed. Based on the assumption that there exists stability on the interreference recency of a buffer page over a certain time period, it uses the LIRS stack, an extension of the LRU stack, to capture the working set, and uses the LIR buffer set to hold its portion most deserved to be in the cache.

Chou [9, 10] proposed the query locality set model (QLSM), which is based on the hotset model, but separates the modeling of the query reference behavior from any particular buffer replacement algorithm. Unlike the hotset model, the QLSM works on a finer grain of decomposition by determining the hotset size on a file-instance basis rather than on a query basis; a separate buffer allocation is determined for each file instance referenced in a query. Moreover, in contrast to the hotset model, which is based on the LRU policy, the QLSM determines an optimal replacement policy for each buffer allocation based on the reference pattern exhibited by the usage of its associated file instance. The QLSM determines the optimal hotset size and replacement policy for each file instance based on a classification of page reference patterns exhibited by common database operations and access methods. The buffer management algorithm based on the QLSM is called the DBMIN algorithm.

2.2 Real-time-based buffer management

Recent interest in real-time DBMSs has motivated the study of special buffer management algorithms for such systems that take into account the priority levels or deadlines of transactions [7, 23, 24]. In [7], two priority-based buffer management algorithms, which allow higher priority transactions to preempt buffer pages from lower priority transactions, were proposed: *Priority-LRU algorithm* and *Priority-DBMIN algorithm*, which are extensions of the LRU and DBMIN algorithms, respectively. The priority-based approach was further developed in [24], where a new algorithm, the *Priority-Hints algorithm*, was proposed.

The Priority-LRU algorithm [7] is a prioritized version of the LRU policy. The buffer pool is organized into n buffer groups (here n is the number of priority levels) and a global free list. Each buffer group is managed locally using the LRU policy and comprises pages owned by transactions with the same priority. The key idea of the Priority-LRU replacement scheme is that the least recently unfixed page of the lowest priority should be chosen as the replacement victim. However, to prevent the most recently accessed pages from being replaced, a threshold parameter (a timestamp value) is used to vary the relative importance between recency and priority in replacement decisions. Therefore, if the free list is empty when a page fault occurs, the search for a replacement starts at the lowest priority group and checks whether the least recently unfixed page in the group falls within the threshold value (i.e., the difference between the global timestamp and candidate's timestamp is less than the threshold value). If it does, the search is repeated at the next higher priority group until either a victim is found or the search is exhausted. If all the replacement candidates fall within the threshold value, the lowest priority candidate is chosen as the default victim.

Like the DBMIN algorithm, the Priority-DBMIN [7] algorithm also allocates buffer pages on a file-instance basis, with each buffer group being managed locally by a suitable replacement policy. If a page is accessed by more than one concurrent transaction, the transaction with the highest priority among the sharing transactions is the owner of the shared page. The selection of a replacement page in the Priority-DBMIN algorithm is determined locally by each group's assigned replacement policy. This is in contrast to the global replacement policy of the Priority-LRU algorithm. The difference between the Priority-DBMIN algorithm and the DBMIN algorithm lies in their transaction admission policies. In the Priority-DBMIN algorithm, the arrival of a higher priority transaction can preempt the unfixed pages owned by transactions of lower priority.

The Priority-Hints algorithm [24] not only incorporates the priority levels of transactions, but also makes use of hints provided by the DBMS access methods to classify the

buffer pages into "favored" pages if they are likely to be rereferenced, and "normal" pages otherwise. The buffer manager receives a page hint with every page request, and "normal" pages are considered for replacement before "favored" pages. The buffer pool is organized into transaction sets, where each set consists of pages owned by a transaction. Pages shared by more than one transaction are owned by the transaction with the highest transaction priority among the sharing transactions. Transaction sets are ordered based on transaction priority and recency of arrival; the latter criterion is used to break ties for transaction sets having the same priority. The buffer pages in each transaction group are partitioned into two groups: a group of fixed pages and a group of unfixed favored pages; unfixed normal pages are returned to a global free list. The free list is managed using the LRU policy, while the unfixed favored pages in each transaction set are managed locally using the MRU policy.

When a buffer fault occurs and the free list is empty, the buffer manager searches the transaction sets in inverse priority order, starting from the lowest priority transaction, to look for an unfixed favored page. If it finds a transaction set of priority lower than that of the requesting transaction with a nonempty set of unfixed favored pages, it will select the most recently unfixed favored page from it as the replacement victim. Otherwise, it will select the most recently unfixed favored page from the requesting transaction itself for replacement.

For video-on-demand applications, a dynamic buffer allocation scheme [28] using the the predict-and-enforce strategy was proposed to address the inherent difficulty of the dynamic buffer allocation: the size of the buffer currently being allocated is dependent on the number and sizes of the buffers to be allocated in the next service period. This is to admit as many concurrent users as possible while maintaining good quality of service.

2.3 Hint-based buffer management

Buffer management algorithms that accept hints from layers above the buffer manager (e.g., access methods and query optimizer) have also been proposed. The Priority-Hints algorithm [24] presented in the previous subsection also falls into this category. The advantage of using hints is that it can provide a more flexible and tighter control over buffer replacement.

The Wisconsin Storage System (WiSS) [11] is a flexible data storage system designed for very high performance. Its buffer manager uses the LRU replacement policy in combination with hints from the system on which pages are important. The hint specified for a page can be *low*, *mid*, or *high*, depending upon its likelihood of being rereferenced. Low hints are assigned to randomly or sequentially accessed pages, mid hints are specified for anchor pages of overflow chains encountered on a sequential scan, and high hints are assigned to B-tree root pages and system directory pages. The page with the lowest hint and the oldest timestamp (i.e., the least recently used page among the least important buffer pages) is selected for replacement. The object manager of O_2 [47] is based on WiSS.

The Starburst DBMS, an extensible relational database system, uses a global allocation and replacement policy [27]. The buffer manager uses a variant of the CLOCK replacement policy and a simple hint mechanism. When a page is unfixed, it receives a hint from the query evaluation system, indicating whether the page is "LOVED" or "HATED." A page is marked as "LOVED" if it is likely to be reused later; otherwise, it is marked as "HATED" and immediately joins the free buffer list if it is not dirtied. The least recently unfixed "LOVED" page is selected for replacement. It is a simpler, as well as a more restrictive, version of the WiSS hint mechanism.

3 A framework for buffer replacement policies

In this section, we propose a uniform framework for modeling buffer replacement policies. Such a framework provides the basis for the construction of an extensible buffer manager that supports extensibility of buffer replacement policies. Buffer replacement policies can be characterized by two main features:

- The structure of the pool of buffer pages, which is organized as groups of pages based on one or more attributes of the pages and/or some external hints (e.g., fixed/unfixed, query owner, transaction priority, and hint level).
- A criterion to select a victim page from among the buffer groups and buffer pages. This selection can be viewed as consisting of two steps: the first step involves selecting a buffer group, while the second step relates to choosing the victim page from the selected buffer group.

Our proposed framework consists of two components corresponding to these two features: a hierarchical model of the buffer pool and a priority scheme that generalizes the representation of replacement criteria. Examples of replacement policies surveyed in Sect. 2 are used to illustrate modeling with the proposed framework.

A summary of the notations used in the rest of this paper is shown in Table 2. In addition, we use the notational convention ti to denote the ith component of an n-component tuple t, where $i = 1, \ldots, n$. We shall simply refer to $t1$ as t. Given an object o, we use $o.a$ to denote an attribute a of the object o.

3.1 Hierarchical buffer pool model

In a multitasking DBMS, applications not only exhibit different resource consumption patterns but also require specific priority and performance objectives [17]. One tuning knob that can achieve performance objectives is memory allocation [4], which partitions buffer space between

Table 2 Notations used in framework

Notation	Meaning
$\mathcal{AP}$	Abstract selection policy type defined as a four-tuple $\mathcal{AP} = (\mathcal{P}, \beta', \gamma', \delta)$.
$\mathcal{B}$	Hierarchical buffer pool model defined as a two-tuple $\mathcal{B} = (\mathcal{N}, \mathcal{E})$.
$\mathbb{B}$	Bit string domain where each b_i $(i = 1, \ldots, k)$ in $b_\mathrm{k}, \ldots, b_1 \in \mathbb{B}$ represents a bit.
$\mathcal{C}$	Local/global control information data type defined as a composition of atomic data types, i.e., $\mathcal{C} = \prod_{\mathrm{i}=1}^{\mathrm{n}} \mathcal{D}_\mathrm{i}$.
c	Local/global control information where $c \in \mathcal{C}$.
comp	Function that computes the 1's complement of a bit string.
$\mathcal{D}_\mathrm{i}$	Atomic data type (e.g., integer, boolean).
$\mathcal{E}$	Set of buffer group type relationships where $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$.
$\mathcal{GP}$	Global selection policy type defined as a five-tuple $\mathcal{GP} = (\mathcal{B}, \mathcal{C}, \mu, \mathcal{S}_\mathrm{A}, \mathcal{M}_\mathrm{A})$.
$\mathcal{LP}$	Local selection policy type defined as a six-tuple $\mathcal{LP} = (\mathcal{P}, \mathcal{C}, \alpha, \beta, \gamma, \delta)$.
$\mathcal{M}_\mathrm{A}$	Function that maps an abstract buffer group type to an abstract selection policy type.
$\mathcal{M}_\mathrm{L}$	Function that maps a local buffer group type to a nonempty subset of local selection policy types.
$\mathcal{N}$	Set of buffer group types where $\mathcal{N} = \mathcal{N}_\mathrm{A} \cup \mathcal{N}_\mathrm{L}$.
$\mathcal{N}_\mathrm{A}$	Set of abstract buffer group types.
$\mathcal{N}_\mathrm{L}$	Set of local buffer group types.
$\mathbb{N}$	Natural number domain.
$\mathcal{O}_\mathrm{g}$	Buffer group type, $\mathcal{O}_\mathrm{g} \in \mathcal{N}$.
$\mathcal{O}_\mathrm{p}$	Buffer page type.
$\mathcal{P}$	Priority type defined as a composition of atomic data types, i.e., $\mathcal{P} = \prod_{\mathrm{i}=1}^{\mathrm{n}} \mathcal{D}_\mathrm{i}$.
$\mathcal{R}$	Buffer replacement policy type defined as a three-tuple $\mathcal{R} = (\mathcal{GP}, \mathcal{S}_\mathrm{L}, \mathcal{M}_\mathrm{L})$.
$\mathcal{S}_\mathrm{A}$	Set of abstract selection policy types.
$\mathcal{S}_\mathrm{L}$	Set of local selection policy types.
α	Initialization function that initializes the local control information.
β	Priority assignment function that initializes the priority value of a buffer page.
β'	Priority assignment function that initializes the priority value of a buffer group.
γ	Priority update function that updates the priority value of a reaccessed buffer page.
γ'	Priority update function that updates the priority value of an accessed buffer group.
δ	Priority evaluation function that interprets the priority value of a buffer page or group.
μ	Global control information update function that updates the global control information when a page fault occurs.

competing applications into buffer pools. Different buffer replacement strategies can then be used for individual applications. However, unnecessary buffer partitioning suffers from the dual disadvantages of reducing the effective available buffer space and also that of incurring the overhead of managing page movement between buffer pools, thereby emphasizing the need for effective tuning strategies.

The organization of the buffer pool can be modeled as a hierarchical structure with the buffer pool being successively decomposed into layers of smaller buffer groups such that each group represents a collection of buffer pages that share some common properties (e.g., pages of the same type such as free buffer pages or pages owned by the same transaction). This hierarchical organization can be represented by a rooted tree with the root node representing the entire buffer pool; each leaf node represents a *local buffer group* that consists of a subset of buffer pages and each internal node represents an *abstract buffer group* comprising some other abstract or local buffer groups (represented by its child nodes). Unlike a local buffer group, an abstract buffer group is "abstract" in the sense that it consists of buffer subgroups instead of buffer pages.

Two examples of hierarchical buffer pool organization are shown in Fig. 1, with the circular nodes representing abstract buffer groups and the square nodes representing local buffer groups.

Figure 1a depicts the buffer pool organization for the hot-set buffer management algorithm. The buffer pool is partitioned into a free buffer group that consists of the free buffer pages and a number of query buffer groups. Each query buffer group, which represents buffer pages owned by a query, is further partitioned into a group of unfixed buffer pages and a group of fixed buffer pages. Another example of hierarchical buffer organization is shown in Fig. 1b for the

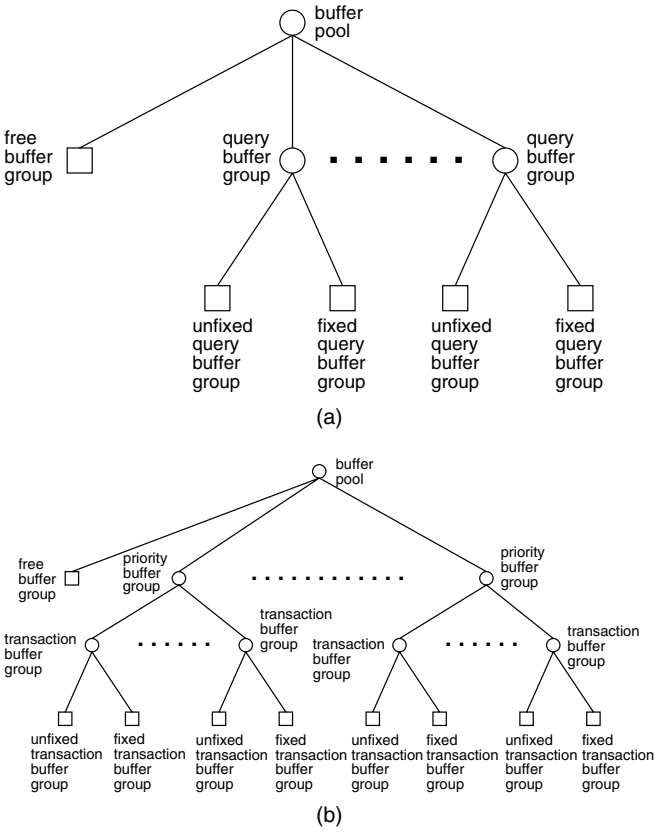

Fig. 1 Examples of buffer pool organization

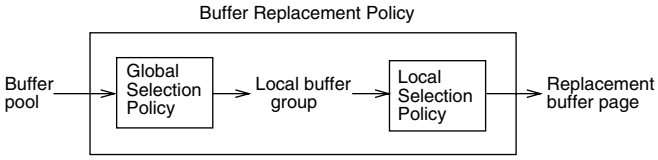

Fig. 2 A two-step model for buffer replacement policies

priority-hints algorithm. The buffer pool is partitioned into a free buffer group that comprises the free buffer pages and a number of priority buffer groups, each of which represents buffer pages owned by transactions with the same transaction priority level. Each priority buffer group is further partitioned into a number of transaction buffer groups, each of which represents buffer pages owned by the same transaction. As in the case of the query buffer groups in the hotset algorithm, each transaction buffer group consists of a buffer group of fixed pages and a buffer group of unfixed pages.

The hierarchical organization of a buffer pool is, therefore, characterized by the types of buffer groups (each buffer group is an instance of some buffer group type) and the containment relationships among these buffer group types. A *hierarchical buffer pool model* for a buffer management policy can be represented by a rooted tree,

$$\mathcal{B} = (\mathcal{N}, \mathcal{E})$$

where $\mathcal{N}$ is a finite set of buffer group types and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is a finite set of buffer group type relationships such that $(N_i, N_j) \in \mathcal{E}$ iff $N_j \in N_i$, that is, each buffer group of type N_j is contained in some buffer group of type N_i. A buffer group type, $N_i \in \mathcal{N}$, is termed as a *local buffer group type* if $\nexists N_j \in \mathcal{N}$ such that $(N_i, N_j) \in \mathcal{E}$ (i.e., N_i is a leaf node of the tree $\mathcal{B}$); otherwise it is termed as an *abstract buffer group type*. Therefore, $\mathcal{N} = \mathcal{N}_\mathrm{A} \cup \mathcal{N}_\mathrm{L}$ and $\mathcal{N}_\mathrm{A} \cap \mathcal{N}_\mathrm{L} = \emptyset$, where $\mathcal{N}_\mathrm{A}$ and $\mathcal{N}_\mathrm{L}$ denote the set of abstract and local buffer group types, respectively. Each instance of a local (abstract) buffer group type is a local (abstract) buffer group. Local buffer groups consist of buffer pages, while abstract buffer groups comprise instances of abstract or local buffer group types.

Since fixed buffer pages are *not* involved in the replacement page selection, we can simplify the modeling of their buffer pool organizations by not modeling the fixed buffer group types. Consider again the examples in Fig. 1. For the hotset buffer management algorithm, $\mathcal{N} = \{N_\text{buffer}, N_\text{free}, N_\text{query-unfixed}\}$ and $\mathcal{E} = \{(N_\text{buffer}, N_\text{free}), (N_\text{buffer}, N_\text{query-unfixed})\}$, where N_buffer, N_free, and $N_\text{query-unfixed}$ denote the buffer pool type, the free buffer group type, and the unfixed buffer group type, respectively. For the priority-hints algorithm, $\mathcal{N} = \{N_\text{buffer}, N_\text{free}, N_\text{priority}, N_\text{trans-unfixed}\}$ and $\mathcal{E} = \{(N_\text{buffer}, N_\text{free}), (N_\text{buffer}, N_\text{priority}), (N_\text{priority}, N_\text{trans-unfixed})\}$, where N_buffer, N_free, N_priority, and $N_\text{trans-unfixed}$ denote the buffer pool type, the free buffer group type, the priority buffer group type, and the unfixed buffer group type, respectively.

3.2 A model for buffer replacement policies

By modeling the buffer pool organization as a hierarchy of local and abstract buffer groups, we can generalize buffer replacement policies into a two-step model. In this model, the buffer pool is associated with a *global selection policy*, and each local buffer group is associated with a *local selection policy*. A global selection policy controls the selection of a local buffer group from the buffer pool, while a local selection policy controls the selection of a buffer page from its associated local buffer group.

Figure 2 shows a schematic diagram of the two-step model for replacement policies. The decomposition of a replacement policy into two levels of selection has several advantages. First, it provides a cleaner separation of concerns and thus enables a clearer and more comprehensive classification of existing replacement policies. Replacement policies, such as LRU and LFU, are more appropriately classified as local selection policies since these policies control the selection of a page from a group of buffer pages. On the other hand, the more sophisticated buffer replacement policies, such as the hotset algorithm and the priority-hints algorithm, involve both global as well as local selection policies. For example, in the hotset algorithm, there are basically three types of buffer pages for selection (unfixed

pages owned by the faulting-query,[1] free buffer pages and unfixed pages owned by other queries), which are partitioned into a number of local buffer groups. Thus, two levels of selection are necessary to select a replacement page: a higher-level selection using the global selection policy first selects a local buffer group; this is followed by a lower-level selection of a buffer page using the local selection policy associated with the selected local buffer group.

Second, the modular separation of a replacement policy into two orthogonal components supports more flexibility, reusability, and extensibility as new replacement policies can be designed by different combinations of existing or new local and global selection policies. In fact, viewed from this framework, both the DBMIN algorithm and the hotset algorithm are similar in that they have the same global selection policy, but they differ in that while the hotset algorithm uses only one fixed local selection policy, the DBMIN algorithm advocates the usage of different local selection policies.

4 Priority scheme

This section presents the second component of our proposed framework. This component is a simple and yet flexible priority scheme for modeling both global as well as local selection policies.

Local selection policies are essentially scheduling policies that prioritize buffer pages for replacement based on certain selection criteria such as LFU and LRU. Similarly, global selection policies are also distinguished by the selection criteria used to select buffer groups for replacement. By treating various selection criteria as different priority schemes, we have a general and flexible abstraction to model selection policies.

The basic idea of our proposed priority scheme is as follows. The selection policy associated with each buffer group assigns priority values to the objects in the buffer group such that whenever the policy makes a selection, it always selects the object with the lowest priority value.[2] Associated with each local (global) selection policy is some information, designated as *local (global) control information*, which the policy uses in the assignment, modification, and evaluation of priority values.

Based on the priority values of buffer groups and buffer pages, a replacement buffer page is selected as follows: the global selection policy first selects from among the various local buffer groups, the one with the lowest priority; the local selection policy associated with the selected local buffer group then selects from it the buffer page with the lowest priority as the replacement page. Thus, using this scheme different selection policies can be modeled by appropriate specifications of priority and control information types in conjunction with functions to initialize, update, and evaluate the priority values and control information. In our priority scheme, priority values are assigned to two types of objects: buffer groups and buffer pages. We shall use $\mathcal{O}_g$ and $\mathcal{O}_p$ to denote a buffer group type and a buffer page type, respectively.

4.1 Modeling of local selection policy types

A local selection policy controls the assignment, modification, and evaluation of priority values for buffer pages contained in a local buffer group. The local selection policy associated with a local buffer group initializes the priority value of a new buffer page when it is fetched into the buffer group and updates its priority value when the buffer page is reaccessed. To select a replacement page from a local buffer group, the associated local selection policy evaluates the priority values of all the pages contained in the group and picks the page with the lowest priority.

We shall use o_p to denote a buffer page and c to denote the local control information associated with a local selection policy. The priority of a buffer page o_p is denoted by o_p.priority.

A *local selection policy type* is modeled by a six-tuple

$$\mathcal{LP} = (\mathcal{P}, \mathcal{C}, \alpha, \beta, \gamma, \delta).$$

$\mathcal{P} = \prod_{i=1}^{n} \mathcal{D}_i, n \geq 1$ defines the *priority type* associated with $\mathcal{LP}$, where each $\mathcal{D}_i$ denotes an atomic data type (e.g., integer type, boolean type). The priority values assigned by local selection policies of type $\mathcal{LP}$ to buffer pages are of type $\mathcal{P}$.

$\mathcal{C} = \prod_{i=1}^{m} \mathcal{D}_i, m \geq 0$ defines the *local control information data type* associated with $\mathcal{LP}$. The local control information c, which is associated with a local selection policy of type $\mathcal{LP}$, is a tuple of type $\mathcal{C}$ and is used in the assignment, update, and evaluation of priority values.

α: $\mathcal{C} \rightarrow \mathcal{C}$ is an *initialization function* that initializes the local control information associated with a local selection policy of type $\mathcal{LP}$.

β: $\mathcal{O}_p \times \mathcal{C} \rightarrow \mathcal{O}_p \times \mathcal{C}$ is a *priority assignment function* that assigns an initial priority value to a page fetched into the local buffer group associated with a local selection policy of type $\mathcal{LP}$. The parameter in β is a tuple (o_p, c) where o_p refers to the new buffer page.

γ: $\mathcal{O}_p \times \mathcal{C} \rightarrow \mathcal{O}_p \times \mathcal{C}$ is a *priority update function* that updates the priority value of a reaccessed buffer page in the local buffer group associated with a local selection policy of type $\mathcal{LP}$. The parameter in γ is a tuple (o_p, c), where o_p refers to the reaccessed buffer page.

δ: $\mathcal{O}_p \times \mathcal{C} \rightarrow \mathbb{B}$ is a *priority evaluation function* that interprets a buffer page's priority value by encoding it into a bit string such that a higher priority value has a larger bit string value. The parameter in δ is a tuple (o_p, c), where o_p refers to the buffer page to be evaluated. The encoding function satisfies the following property:

$$\forall o_p, o'_p \in \mathcal{O}_p, \delta(o_p, c) \geq \delta(o'_p, c)$$

[1] A faulting-query (-transaction) refers to the query (transaction) that causes the page fault.

[2] For a local selection policy, the objects in the associated buffer group refer to buffer pages, while for a global selection policy the objects in the associated buffer group refer to some other buffer groups.

iff o_p has a higher priority than o'_p or the same priority as o'_p. This encoding function is used to select the buffer page with the lowest priority for replacement. Thus, the buffer page o_p is selected for replacement by a local selection policy of type $\mathcal{LP}$ if $\delta(o_p, c) \leq \delta(o'_p, c)$ for each buffer page o'_p in the local buffer group.

Relating to object-oriented concepts, a local selection policy type $\mathcal{LP}$ is like a class definition where each class instance (local selection policy) has an instance variable of type $\mathcal{C}$ (local control information). All instances of the same class share a set of four methods — α, β, γ, and δ. For example, α is an object initialization method that initializes the instance variable for each new instance of the class.

4.1.1 Examples of local selection policy type modeling

In this section, we illustrate the modeling of local selection policy types with the following examples: LRU, LRD, and WiSS local selection policy types. For each example, we first describe how it is modeled before presenting its specification.

Least recently used policy type (LRU). Figure 3 depicts the modeling of the LRU local selection policy type. An LRU policy replaces the least recently used page. Hence, the priority of a buffer page can be represented by a logical timestamp value such that a more recently used page has a larger timestamp value that represents a higher priority. Hence, the replacement page is the page with the smallest timestamp value (lowest priority value). The local control information, c, is also a logical timestamp count that stores the timestamp value of the most recently accessed page. This value is initialized by the initialization function, α_{LRU}, to zero before it is used for updating priority values. When a new page is fetched into the buffer group or when a page in the buffer group is reaccessed, the local control timestamp count is first incremented by one and then assigned as the new priority value of the buffer page. Thus, the priority update function, γ_{LRU}, is the same as the priority assignment function, β_{LRU}. Since a page with a larger timestamp value has a higher priority, the priority evaluation function, δ_{LRU}, simply returns a buffer page's timestamp value as its priority value.

Least referenced density policy type (LRD). In [14], Effelsberg and Haerder proposed the least reference density replacement (LRD) policy, which is an improvement over the least frequently used (LFU) policy. In the LFU policy, each buffer page P_i is associated with a reference counter $\mathrm{RC}(i)$, which is initialized to zero and incremented by one after each reference. The least frequently used page (i.e., the page with the lowest reference count) is selected for replacement. The drawback of this approach is that a page that is heavily referenced for a short period will not be replaced even if it is never referenced again. Instead of using just the reference frequency (i.e., the absolute number of references), the LRD policy improves on the LFU policy by considering the relative frequency over a reference interval.

The LRD policy incorporates the age of a page in its replacement criteria. The age of a page P_i is defined as the number of elapsed references (to all buffer pages) since the first reference to P_i. The age of a page is measured in units of logical references and is determined as follows. Let GRC (global reference count) be the total number of logical references. For each buffer page P_i, the time of its first reference is $\mathrm{FC}(i)$. Thus the reference interval of the age of a page P_i is $\mathrm{GRC\text{-}FC}(i)$. The reference density of a page P_i is denoted by $\mathrm{RD}(i)$. The formula for $\mathrm{RD}(i)$ is as follows:

$$\mathrm{RD}(i) = \frac{\mathrm{RC}(i)}{(\mathrm{GRC} - \mathrm{FC}(i))} \quad \text{where} \quad \mathrm{GRC\text{-}FC}(i) \geq 1.$$

The LRD policy replaces the page with the least reference density.

Thus the priority of a buffer page P_i comprises two components – $\mathrm{FC}(i)$ and $\mathrm{RC}(i)$. The local control information consists of the GRC. Pages with a lower reference density ($\mathrm{RD}(i)$) have lower priority. The modeling of the LRD local selection policy type is depicted in Fig. 4.

WiSS replacement policy type (WRP). The WiSS replacement policy selects the least recently used page among pages with the lowest hint for replacement. The priority of a buffer page can be represented by a composite data value consisting of a hint value (which can be *low*, *mid*, or *high*) and a timestamp value. The local control information is of the same type as the page priority with its timestamp component being initialized and updated similarly as in an LRU policy. Note that as the hint value of a buffer page is determined by the system, it is assigned via the local control information updated by the system. The hint value of a buffer page does not change when the page is reaccessed, only its timestamp value is updated as in the LRU policy. Figure 5 shows the modeling of the WiSS local selection policy type. The priority evaluation function, δ_{WRP}, encodes the respective priority values of buffer pages such that a page with a lower hint value has a lower priority (encoded by $b_{k+2}b_{k+1}$), and among buffer pages with the same hint value, those with smaller timestamp values have lower priorities (encoded by $b_k, \ldots, b_1$).

4.2 Modeling of global selection policy types

The function of a global selection policy is to select a local buffer group when a page fault occurs so that the replacement page can be selected from it. Here we propose a simple and consistent way of modeling global selection policies.

By modeling the buffer pool as a hierarchical graph, the selection of a local buffer group can be viewed as finding a path from the root node (buffer pool group) to a leaf node (local buffer group). This path is found incrementally by the following sequence of selections: Starting at the root node, one of the child nodes of the root node is selected; if the selected node is a leaf node, then a local buffer group is found and the selection process terminates; otherwise, the selection

$\mathcal{LP}_{LRU} = (\mathcal{P}_{LRU}, \mathcal{C}_{LRU}, \alpha_{LRU}, \beta_{LRU}, \gamma_{LRU}, \delta_{LRU})$ where

$\mathcal{P}_{LRU} = \mathbb{N}$ represents the timestamp priority type.

$\mathcal{C}_{LRU} = \mathcal{P}_{LRU}$

$\alpha_{LRU}(c)$

{

    $c := 0$; /* initialize local control timestamp count */

    return c;

}

$\beta_{LRU}(o_p, c)$

{

    $c := c + 1$; /* update local control timestamp count */

    $o_p.priority := c$; /* assign local control timestamp count as page priority */

    return (o_p, c);

}

$\gamma_{LRU} = \beta_{LRU}$

$\delta_{LRU}(o_p, c) = o_p.priority$

Fig. 3 Modeling of LRU local selection policy type

process is repeated by selecting one of the child nodes of the selected internal node (abstract buffer group), and so on until a leaf node is selected. The local buffer group selection process is illustrated in Fig. 6.

Like the selection of a buffer page from a local buffer group, the selection of a buffer group from an abstract buffer group can be modeled using the proposed priority scheme. Each abstract buffer group is associated with a selection policy, referred to as *abstract selection policy*, which assigns and updates priority values for buffer groups contained in the abstract buffer group such that a selection from the abstract buffer group always picks the buffer group with the lowest priority value.

A global selection policy is thus characterized by a set of abstract selection policies, each of which is associated with an abstract buffer group. The selection of a local buffer group by the global selection policy can be expressed as a composition of a sequence of abstract buffer group selections. A global selection policy is also associated with some control information, designated as *global control information*, which is used in the evaluation of priority values of buffer groups. The global control information consists of information related to the faulting-transaction/query and is updated when a page fault occurs prior to the local buffer group selection.

A *global selection policy type* is modeled by a five-tuple $\mathcal{GP} = (\mathcal{B}, \mathcal{C}, \mu, \mathcal{S}_\mathrm{A}, \mathcal{M}_\mathrm{A})$.

$\mathcal{B} = (\mathcal{N}, \mathcal{E})$ is the buffer pool model (as explained in Sect. 3.1).

$\mathcal{C} = \prod_{\mathrm{i}=1}^{\mathrm{m}} \mathcal{D}_\mathrm{i}, m \geq 1$ defines the global control information data type. The global control information c, which is associated with a policy of type $\mathcal{GP}$, is a tuple of type $\mathcal{C}$.

μ: $\mathcal{C} \rightarrow \mathcal{C}$ is an update function that updates the global control information with details of the faulting-transaction/query when a page fault occurs.

$\mathcal{S}_\mathrm{A} = \{\mathcal{AP}_1, \ldots, \mathcal{AP}_\mathrm{k}\}$ is a finite set of abstract selection policy types.

$\mathcal{M}_\mathrm{A}$: $\mathcal{N}_\mathrm{A} \rightarrow \mathcal{S}_\mathrm{A}$ is a surjective function that maps an abstract buffer group type to an abstract selection policy type. Thus, each abstract buffer group of type $N_i \in \mathcal{N}_\mathrm{A}$ is associated with an abstract selection policy of type $\mathcal{AP}_\mathrm{j} \in \mathcal{S}_\mathrm{A}$.

An abstract selection policy controls the assignment, modification, and evaluation of priority values for buffer groups contained in an abstract buffer group. The abstract selection policy associated with an abstract buffer group initializes the priority value of a new buffer group created in the abstract buffer group and updates its priority value when the buffer group is accessed. A local buffer group is said to be *accessed* when a buffer page is added or removed from it; an abstract buffer group is said to be accessed when any of the buffer groups contained in it is accessed. To select a buffer group from an abstract buffer group, the associated abstract selection policy evaluates the priority values of all the buffer

$\mathcal{LP}_{LRD}(\mathcal{P}_{LRD}, \mathcal{C}_{LRD}, \alpha_{LRD}, \beta_{LRD}, \gamma_{LRD}, \delta_{LRD})$ where

$\mathcal{P}_{LRD} = \mathcal{D}_{LRD,1} \times \mathcal{D}_{LRD,2}$ where

$\mathcal{D}_{LRD,1} = \mathbb{N}$ represents the first reference count priority type, and

$\mathcal{D}_{LRD,2} = \mathbb{N}$ represents the reference frequency count priority type.

$\mathcal{C}_{LRD} = \mathbb{N}$ represents the global reference count data type.

$\alpha_{LRD}(c)$

{

$c := 0;$ /* initialize global reference count to zero */

return c;

}

$\beta_{LRD}(o_p, c)$

{

$c := c + 1;$ /* increment global reference count */

o_p.priority[1] := c; /* initialize first reference count */

o_p.priority[2] := 1; /* initialize reference frequency count to one */

return (o_p, c);

}

$\gamma_{LRD}(o_p, c)$

{

$c := c + 1;$ /* increment global reference count */

o_p.priority[2] := o_p.priority[2] + 1; /* increment reference frequency count */

return (o_p, c);

}

$$\delta_{LRD}(o_p, c) = \begin{cases} \frac{o_p.\text{priority}[2]}{(c - o_p.\text{priority}[1])} & \text{if } c - o_p.\text{priority}[1] \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Fig. 4 Modeling of LRD local selection policy type

groups contained in the abstract buffer group and picks the group with the lowest priority.

An *abstract selection policy type* is modeled by a four-tuple

$$\mathcal{AP} = (\mathcal{P}, \beta', \gamma', \delta).$$

$\mathcal{P} = \prod_{i=1}^{q} \mathcal{D}_i$, $q \geq 1$ defines the *priority type* associated with $\mathcal{AP}$, where each $\mathcal{D}_i$ is an atomic data type. An abstract selection policy of type $\mathcal{AP}$ assigns priority values of type $\mathcal{P}$ to buffer groups.

β': $\mathcal{O}_g \rightarrow \mathcal{O}_g$ is a *priority assignment function* that assigns an initial priority value to a new buffer group that is added to an abstract buffer group associated with an abstract selection policy of type $\mathcal{AP}$. The parameter in β' refers to the new buffer group.

γ': $\mathcal{O}_g \rightarrow \mathcal{O}_g$ is a *priority update function* that updates the priority value of an accessed buffer group contained in an abstract buffer group associated with an abstract selection policy of type $\mathcal{AP}$. The parameter in γ' refers to an accessed buffer group

δ: $\mathcal{O}_g \times \mathcal{C} \rightarrow \mathbb{B}$ is a *priority evaluation function* similar to that in a local selection policy type, except that it is used in the evaluation of priority values of buffer groups, and the parameter in δ is a tuple (o_g, c), where o_g and c refer to a buffer group and the global control information, respectively.

5 Modeling of buffer replacement policy types

By combining the modeling of the buffer pool (Sect. 3.1), the local selection policy types (Sect. 4.1), and the global

$\mathcal{LP}_{WRP} = (\mathcal{P}_{WRP}, \mathcal{C}_{WRP}, \alpha_{WRP}, \beta_{WRP}, \gamma_{WRP}, \delta_{WRP})$ where

$\mathcal{P}_{WRP} = \mathcal{D}_{WRP,1} \times \mathcal{D}_{WRP,2}$ where

$\mathcal{D}_{WRP,1} = \{0, 1, 2\}$ represents the hint priority type,

with 0, 1 and 2 representing low, mid and high hint respectively, and

$\mathcal{D}_{WRP,2} = \mathbb{N}$ represents the timestamp priority type.

$\mathcal{C}_{WRP} = \mathcal{P}_{WRP}$

$\alpha_{WRP}(c)$

{

$c[1] := 0;$ /* initialize local control hint value to low */

$c[2] := 0;$ /* initialize local control timestamp count */

return (c);

}

$\beta_{WRP}(o_p, c)$

{

o_p.priority[1] := $c[1]$; /* initialize hint value of page priority */

$c[2] := c[2] + 1;$ /* update local control timestamp count */

o_p.priority[2] := $c[2]$; /* initialize timestamp count of page priority */

return (o_p, c);

}

$\gamma_{WRP}(o_p, c)$

{

$c[2] := c[2] + 1;$ /* update local control timestamp count */

o_p.priority[2] := $c[2]$; /* update timestamp count of page priority */

return (o_p, c);

}

$\delta_{WRP}(o_p, c) = b_{k+2} \ldots b_1$ where

k is the number of bits required to encode the timestamp value and

$$b_{k+2} b_{k+1} = o_p.\text{priority}[1]$$

$$b_k \ldots b_1 = o_p.\text{priority}[2]$$

Fig. 5 Modeling of WiSS local selection policy type

selection policy types (Sect. 4.2), a *buffer replacement policy type* is modeled as follows:

$$\mathcal{R} = (\mathcal{GP}, \mathcal{S}_{\text{L}}, \mathcal{M}_{\text{L}}),$$

where $\mathcal{GP} = (\mathcal{B}, \mathcal{C}, \mu, \mathcal{S}_{\text{A}}, \mathcal{M}_{\text{A}})$ is a global selection policy type, $\mathcal{S}_{\text{L}} = \{\mathcal{LP}_1, \ldots, \mathcal{LP}_{\text{q}}\}$ is a finite set of local selection policy types, and $\mathcal{M}_{\text{L}}$: $\mathcal{N}_{\text{L}} \rightarrow 2^{\mathcal{S}_{\text{L}}}$ is a function that maps a local buffer group type to a nonempty subset of local selection policy types. Note that while $\mathcal{M}_{\text{A}}$ maps an abstract buffer group type to an abstract selection policy type, $\mathcal{M}_{\text{L}}$ maps a local buffer group type to a subset of local selection policy types. Moreover, the set of abstract selection policy mappings that forms the global selection policy is determined at compile time, while the mapping of a local selection policy to a local buffer group can be determined at run time depending on the page access pattern. For example, in the EXODUS storage manager [15], each local buffer group can be managed by either an LRU or an MRU replacement policy.

Hence, using the proposed framework, the specification for a buffer replacement policy type essentially consists of a collection of types (buffer group types, local selection policy types, and abstract selection policy types) and mappings among the types (among buffer group types and between buffer group types and selection policy types).

5.1 Examples of buffer replacement policy type modeling

In this section, we illustrate the modeling of two buffer replacement policy types, namely, the hotset buffer management algorithm and the priority-hints algorithm. For each buffer management algorithm, we first explain its buffer page replacement criteria in terms of its hierarchical buffer pool organization, and then we present the details of the modeling using the proposed framework.

5.1.1 Hotset algorithm

The buffer pool organization for the hotset algorithm, as explained in Sect. 3.1, is depicted in Fig. 7.

Based on the hierarchical organization shown in Fig. 7, we can explain the replacement criteria in the hotset algorithm as follows. The global selection policy selects the query buffer group owned by the faulting query (an instance of $N_{\text{query-unfixed}}$) if it is nonempty; otherwise, it selects the free buffer group (the only instance of N_{free}) if it is nonempty. However, if the free buffer group is also empty, the global policy selects the query buffer group owned by the most *deficient* query. Since the hotset algorithm is based on the LRU policy, the replacement buffer page is the least recently accessed page in the selected local buffer group. The overall modeling of the hotset algorithm is presented in Fig. 8.

For the global selection policy to identify the query buffer group owned by the faulting transaction, we define the global control information type, $\mathcal{C}$, and the global control information update function, μ, such that the global control information, c, is updated with the query identifier of the faulting query when a page fault occurs. This information is used by the abstract selection policy associated with the buffer group (instance of $\mathcal{AP}_1$) in its selection of a local buffer group.

Figure 9 shows the modeling of the abstract selection policy type $\mathcal{AP}_1$. The abstract selection policy (instance of $\mathcal{AP}_1$) associated with the buffer pool group (instance of N_{buffer}) selects either a free buffer group (instance of N_{free}) or a query buffer group (instance of $N_{\text{query-unfixed}}$) based on the following information (definition of $\mathcal{P}_1$):

1. The type of the buffer group, whether it is a free or a query buffer group.

```
/* Return a local buffer group */
GLOBAL_SELECT () : buffer_group_type
{
        return SELECT (root buffer group);
}

SELECT (g : buffer_group_type) : buffer_group_type
{
        if (g is a local buffer group) then
                return g
        else
                return select (g') where
                        g' is the buffer group contained in g selected by the abstract selection policy
                        associated with g;
}
```

Fig. 6 Selection of local buffer group

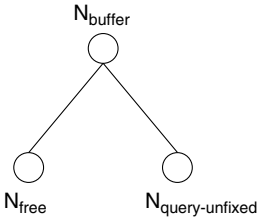

Fig. 7 Hierarchical buffer pool model for hotset algorithm

2. The query owner, which is represented by a query identifier; the purpose is to identify the faulting query buffer group by comparing with the global control information.
3. The locality set size of the buffer group.
4. The current allocated size of the buffer group; both the locality and the allocated size are used to determine the query's deficiency.

Thus, the priority values assigned to the free buffer group and the query buffer groups contained in the buffer pool group are composite values consisting of these four components.

To encode the selection criteria for the local buffer group, we define the priority evaluation function, δ_1, such that

1. Nonempty buffer groups have lower priority than empty buffer groups (encoded by b_{k+3}).
2. Among the nonempty buffer groups, the buffer group owned by the faulting query has lower priority than the other buffer groups (encoded by b_{k+2}), and the free buffer group has lower priority than other query buffer groups owned by the nonfaulting queries (encoded by b_{k+1}).

$\mathcal{R} = (\mathcal{GP}, \mathcal{S}_L, \mathcal{M}_L)$ where

$\mathcal{GP} = (\mathcal{B}, \mathcal{C}, \mu, \mathcal{S}_A, \mathcal{M}_A)$ where

$\mathcal{B} = (\mathcal{N}, \mathcal{E})$ where

$\mathcal{N} = \{N_{buffer}, N_{free}, N_{query-unfixed}\}$

$\mathcal{E} = \{(N_{buffer}, N_{free}), (N_{buffer}, N_{query-unfixed})\}$

$\mathcal{C} = \mathbb{N}$ represents the query identifier type.

$\mu(c)$

{

$c :=$ query identifier of faulting query;

return c;

}

$\mathcal{S}_A = \{\mathcal{AP}_1\}$

$\mathcal{M}_A = \{(N_{buffer}, \mathcal{AP}_1)\}$ where $\mathcal{AP}_1$ is defined in Figure 9.

$\mathcal{S}_L = \{\mathcal{LP}_{LRU}\}$

$\mathcal{M}_L = \{ (N_{free}, \{\mathcal{LP}_{LRU}\}), (N_{query-unfixed}, \{\mathcal{LP}_{LRU}\}) \}$ where

the definition of $\mathcal{LP}_{LRU}$ is shown in Figure 3.

Fig. 8 Overall modeling of hotset algorithm

3. Among the query buffer groups owned by the nonfaulting queries, those owned by the more deficient queries have lower priority (encoded by $b_k, \ldots, b_1$).

$\mathcal{AP}_1 = (\mathcal{P}_1, \beta'_1, \gamma'_1, \delta_1)$ where

$\mathcal{P}_1 = \prod_{i=1}^{4} \mathcal{D}_{\mathcal{P}_1,i}$ where

$\mathcal{D}_{\mathcal{P}_1,1}$ = { FREE-TYPE, QUERY-TYPE } represents the buffer group type,

$\mathcal{D}_{\mathcal{P}_1,2} = \mathbb{N}$ represents the query identifier type,

$\mathcal{D}_{\mathcal{P}_1,3} = \mathbb{N}$ represents the locality set size type, and

$\mathcal{D}_{\mathcal{P}_1,4} = \mathbb{N}$ represents the actual allocated buffer size type.

$\beta'_1(o_g)$

{

o_g.priority[1] = buffer group type of o_g;

if (o_g.priority[1] = QUERY-TYPE) then

{

o_g.priority[2] = query identifier of owner of o_g;

o_g.priority[3] = locality set size of o_g;

} else {

o_g.priority[2] = 0;

o_g.priority[3] = 0;

}

o_g.priority[4] = size of o_g;

return o_g;

}

$\gamma'_1(o_g)$

{

o_g.priority[4] = size of o_g;

return o_g;

}

$\delta_1(o_g, c) = b_{k+3} \ldots b_1$ where k is the number of bits needed to encode the query deficiency, and

$$b_{k+3} = \begin{cases} 0 \text{ if } o_g.\text{priority}[4] > 0 \\ 1 \text{ otherwise} \end{cases}$$

$$b_{k+2} = \begin{cases} 0 \text{ if } o_g.\text{priority}[1] = \text{QUERY-TYPE and } o_g.\text{priority}[2] = c \\ 1 \text{ otherwise} \end{cases}$$

$$b_{k+1} = \begin{cases} 0 \text{ if } o_g.\text{priority}[1] = \text{FREE-TYPE} \\ 1 \text{ otherwise} \end{cases}$$

$b_k \ldots b_1 = comp(o_g.\text{priority}[3] - o_g.\text{priority}[4])$

Fig. 9 Abstract selection policy type of hotset algorithm

5.1.2 Priority-hints algorithm

The buffer pool organization for the priority-hints algorithm, as explained in Sect. 3.1, is depicted in Fig. 10.

The replacement criteria in the priority-hints algorithm can be explained using the hierarchical buffer organization in Fig. 10 as follows. The free buffer group (instance of N_{free}) is selected as the replacement local buffer group if it is nonempty; otherwise, the selection of the local buffer group involves two levels of selection: the global selection policy first selects a priority buffer group (instance of N_{priority}) from

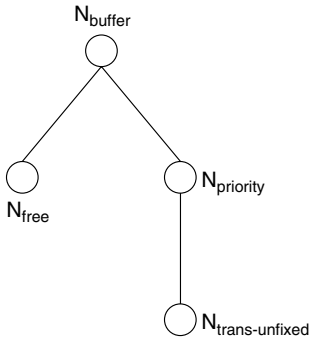

Fig. 10 Hierarchical buffer pool model for priority-hints algorithm

the buffer pool and then selects a transaction buffer group (instance of $N_{\text{trans-unfixed}}$) from among the transaction buffer groups contained in the selected priority buffer group. In the first selection, the policy selects the nonempty priority buffer group with the lowest transaction priority level that is not equal to or greater than the priority level of the faulting transaction. If no such group exists, the priority buffer group with a transaction priority level equal to that of the faulting transaction is selected. In the second selection, if the selected priority buffer group has the same priority level as the faulting transaction, the transaction buffer group owned by the faulting transaction is selected; otherwise, the nonempty transaction buffer group owned by the "oldest" transaction is selected.[3]

The selection criteria for the replacement buffer page depends upon the local buffer group from which it is selected. If the selected local buffer group is the free buffer group, the least recently accessed page is selected for replacement; otherwise, the most recently accessed page is selected from the selected transaction buffer group. Figure 11 depicts the overall modeling of the priority-hints algorithm.

For the global selection policy to select a local buffer group, the following information concerning the faulting transaction must be included in the global control information:

1. The transaction identifier of the faulting transaction and
2. The transaction priority level of the faulting transaction.

$\mathcal{C}$ and μ are defined such that the global control information, c, is updated with this information when a page fault occurs.

The global selection policy type is defined in terms of two abstract selection policy types, $\mathcal{AP}_1$ and $\mathcal{AP}_2$, which are associated with the buffer group types N_{buffer} and N_{priority}, respectively. Figure 12 shows the modeling of the abstract selection policy type $\mathcal{AP}_1$. The abstract selection policy (instance of $\mathcal{AP}_1$) associated with the buffer pool group (instance of N_{buffer}) selects either the free buffer group (instance of N_{free}) or a priority buffer group (instance

[3] The age of a transaction is measured by its admission time. Hence, the oldest transaction refers to the transaction that had been admitted the earliest among the set of existing transactions.

$\mathcal{R} = (\mathcal{GP}, \mathcal{S}_L, \mathcal{M}_L)$ where

$\mathcal{GP} = (\mathcal{B}, \mathcal{C}, \mu, \mathcal{S}_A, \mathcal{M}_A)$ where

$\mathcal{B} = (\mathcal{N}, \mathcal{E})$ where

$\mathcal{N} = \{N_{buffer}, N_{free}, N_{priority}, N_{trans-unfixed}\}$

$\mathcal{E} = \{(N_{buffer}, N_{free}), (N_{buffer}, N_{priority}), (N_{priority}, N_{trans-unfixed})\}$

$\mathcal{C} = \prod_{i=1}^{2} \mathcal{D}_{\mathcal{P}_1,i}$ where

$\mathcal{D}_{\mathcal{P}_1,1} = \mathbb{N}$ represents the transaction identifier type, and

$\mathcal{D}_{\mathcal{P}_1,2} = \mathbb{N}$ represents the transaction priority level type.

$\mu(c)$

{

$c[1] :=$ transaction identifier of faulting transaction;

$c[2] :=$ transaction priority level of faulting transaction;

return c;

}

$\mathcal{S}_A = \{\mathcal{AP}_1, \mathcal{AP}_2\}$

$\mathcal{M}_A = \{(N_{buffer}, \mathcal{AP}_1), (N_{priority}, \mathcal{AP}_2)\}$ where

$\mathcal{AP}_1$ and $\mathcal{AP}_2$ are defined in Figure 12 and Figure 13 respectively.

$\mathcal{S}_L = \{\mathcal{LP}_{LRU}, \mathcal{LP}_{MRU}\}$

$\mathcal{M}_L = \{(N_{free}, \{\mathcal{LP}_{LRU}\}), (N_{trans-unfixed}, \{\mathcal{LP}_{MRU}\})\}$

Fig. 11 Overall modeling of priority-hints algorithm

of N_{priority}) based on the following information (definition of $\mathcal{P}_1$):

1. The type of the buffer group, whether it is a free or priority buffer group;
2. The transaction priority level of a priority buffer group; and
3. The state of the buffer group, i.e., whether the buffer group is empty.[4]

Thus, the priority values of the free buffer group and the priority buffer groups are of type $\mathcal{P}_1$.

To encode the selection criteria of the abstract selection policy type $\mathcal{AP}_1$, the priority evaluation function, δ_1, is defined such that

1. Nonempty buffer groups have lower priority (encoded by $b_{\text{m+3}}$).
2. Among the nonempty buffer groups, the free buffer groups have lower priority than the priority buffer groups (encoded by $b_{\text{m+2}}$).
3. Among the priority buffer groups, those that have transaction priority level lower than or equal to the faulting transaction have lower priority (encoded by $b_{\text{m+1}}$).
4. Among priority buffer groups with transaction priority level lower than or equal to the faulting transaction, those with a lower transaction priority level have lower priority (encoded by $b_{\text{m}}, \ldots, b_1$).

The modeling of the abstract selection policy type $\mathcal{AP}_2$ is depicted in Fig. 13. The abstract selection policy (instance of $\mathcal{AP}_2$) associated with each priority buffer group (instance of N_{priority}) selects a transaction buffer group (instance of $N_{\text{trans-unfixed}}$) based on the following information (definition of $\mathcal{P}_2$):

1. The transaction owner, which is represented by a transaction identifier; this is to enable identification of the faulting transaction buffer group by comparing with the global control information;
2. The admission time of the owner transaction; and
3. The state of the buffer group, i.e., whether the buffer group is empty.

Transaction buffer groups are assigned priority values of type $\mathcal{P}_2$.

[4] A buffer group is empty if the size of the buffer group is zero. The size of a local buffer group is equal to the number of buffer pages contained in it. The size of an abstract buffer group is equal to the sum of the respective sizes of all the buffer groups contained in it.

$\mathcal{P}_1 = \prod_{i=1}^{3} \mathcal{D}_{\mathcal{P}_1,i}$ where

$\mathcal{D}_{\mathcal{P}_1,1}$ = { PRIORITY-TYPE, FREE-TYPE } represents the buffer group type,

$\mathcal{D}_{\mathcal{P}_1,2} = \mathbb{N}$ represents the transaction priority level type, and

$\mathcal{D}_{\mathcal{P}_1,3} = \{false, true\}$ represents the buffer group state type.

$\beta'_1(o_g)$

{
  o_g.priority[1] = buffer group type of o_g;
  if (o_g.priority[1] = PRIORITY-TYPE) then
    o_g.priority[2] = priority level of owner transaction of o_g
  else
    o_g.priority[2] = 0;
  if (size of o_g = 0) then
    o_g.priority[3] = $true$
  else
    o_g.priority[3] = $false$;
  return o_g;
}

$\gamma'_1(o_g)$

{
  if (size of o_g = 0) then
    o_g.priority[3] = $true$
  else
    o_g.priority[3] = $false$;
  return o_g;
}

$\delta_1(o_g, c) = b_{m+3} \ldots b_1$ where

$m = \lceil \log_2(\text{highest transaction priority level}) \rceil$ is the number of bits required to encode the transaction priority level, and

$$b_{m+3} = \begin{cases} 0 \text{ if } (o_g.\text{priority}[3] = false) \\ 1 \text{ otherwise} \end{cases}$$

$$b_{m+2} = \begin{cases} 0 \text{ if } (o_g.\text{priority}[1] = \text{FREE-TYPE}) \\ 1 \text{ otherwise} \end{cases}$$

$$b_{m+1} = \begin{cases} 0 \text{ if } (o_g.\text{priority}[2] \leq c[2]) \\ 1 \text{ otherwise} \end{cases}$$

$$b_m \ldots b_1 = o_g.\text{priority}[2]$$

Fig. 12 Abstract selection policy type $\mathcal{AP}_1$ of priority-hints algorithm

To encode the selection criteria of the abstract selection policy type $\mathcal{AP}_2$, the priority evaluation function, δ_2, is defined such that

1. The transaction buffer group owned by the faulting transaction has lower priority than other transaction buffer groups (encoded by b_{n+2}). Note that this only applies when the selected priority buffer group has a transaction priority level equal to that of the faulting transaction.
2. Nonempty transaction buffer groups have lower priority (encoded by b_{n+1}).
3. Transaction buffer groups that are owned by older transactions have lower priority (encoded by $b_n, \ldots, b_1$).

6 Performance study

This section reports the performance study associated with the proposed extensible buffer management framework. First, the issues concerning the implementation of the proposed framework are discussed. Then, the experimental results associated with the LRU-K replacement policy are presented. Finally, the role of the proposed scheme in improving the performance of indexes is investigated.

6.1 Implementation issues

The extensible buffer manager has been implemented as a component in StorM [18]. Moreover, we have coded a number of buffer replacement strategies using our proposed framework. In order to verify the effectiveness of the implemented buffer manager, we have implemented the LRU-K [35] strategy using the proposed framework and run the experiments that have been described in [35]. Furthermore, we have conducted an experiment to identify the amount of overhead incurred when using the framework. This has been done by implementing one copy of the LRU-2 replacement policy using the framework and another copy of the same policy without using the framework. The two copies are given the same set of buffer page requests as input, and the CPU time incurred by each copy is recorded for comparison purposes.

A buffer pool comprises two components: an array of buffer slots (or frames) and a replacement policy. A buffer slot, in turn, consists of a priority value, a dirty bit, a page identifier (identifier of the page on the disk), and a main memory copy of the page. A buffer pool, its frames, and the pages are all instances of predefined StorM Java classes. The priority associated with a slot reflects the priority of the page that it currently contains and is an object instance of a subclass of a generic abstract class for slot priorities. Although in theory all priorities can be implemented (encoded) as a single number, we found it more convenient to use a more adequate data structure by refining and implementing a subclass of the generic abstract class. Similarly, the replacement policy is an abstract class that needs to be refined and implemented by inheritance to arrive at a tailored replacement

$$\mathcal{AP}_2 = (\mathcal{P}_2, \beta'_2, \gamma'_2, \delta_2) \text{ where}$$

$$\mathcal{P}_2 = \prod_{i=1}^{3} \mathcal{D}_{\mathcal{P}_2,i} \text{ where}$$

$$\mathcal{D}_{\mathcal{P}_2,1} = \mathbb{N} \text{ represents the transaction identifier type,}$$

$$\mathcal{D}_{\mathcal{P}_2,2} = \mathbb{N} \text{ represents the timestamp data type, and}$$

$$\mathcal{D}_{\mathcal{P}_2,3} = \{false, true\} \text{ represents the buffer group state type.}$$

$$\beta'_2(o_g)$$

$$\{$$

$$o_g.\text{priority}[1] = \text{transaction identifier of owner transaction;}$$

$$o_g.\text{priority}[2] = \text{transaction admitted timestamp;}$$

$$\text{if (size of } o_g = 0) \text{ then}$$

$$o_g.\text{priority}[3] = true$$

$$\text{else}$$

$$o_g.\text{priority}[3] = false;$$

$$\}$$

$$\gamma'_2(o_g) = \gamma'_1(o_g);$$

$$\delta_2(o_g, c) = b_{n+2} \ldots b_1 \text{ where}$$

$$n \text{ is the number of bits required to encode the transaction}$$

$$\text{admission timestamp value and}$$

$$b_{n+2} = \begin{cases} 0 \text{ if } o_g.\text{priority}[1] = c[1] \\ 1 \text{ otherwise} \end{cases}$$

$$b_{n+1} = \begin{cases} 0 \text{ if } o_g.\text{priority}[3] = false \\ 1 \text{ otherwise} \end{cases}$$

$$b_n \ldots b_1 = o_g.\text{priority}[2]$$

Fig. 13 Abstract selection policy type $\mathcal{AP}_2$ of priority-hints algorithm

policy. Once the new replacement policy is modeled as a subclass of the priority scheme class, a buffer pool object is associated with one instance of the new replacement policy.

A replacement policy implementation defines the data structure holding the set of global priorities to be taken into account (as in LRU-K or LFU) as well as the data structure maintaining a global counter providing the logical timestamps. These two data structures are persistent since they may have to be maintained across transactions and restored in the case of system failures. There is only a negligible overhead if they are not used for a given replacement policy that uses local priorities only (e.g., LRU).

In addition to these two data structures, the refinement of a concrete replacement policy class comprises the implementation of five methods:

- The *initialization method*, which initializes the global data structure when the buffer pool is created or reset.
- The *priority-assignment method*, which assigns the local priority to a slot and updates the global priority of the page when the page is fetched from the disk into the slot.
- The *priority-update method*, which updates the local and the global priorities when a logical reference is made to the page present in the buffer.
- The *priority-comparison method*, which defines the priority order between two pages in the buffer by comparing their global and local priorities. This method is private to the replacement policy.
- The *priority-evaluation method*, which selects the slot containing the pages with the smallest priority for replacement. This method uses the *priority-comparison method* to determine the page with the minimum priority. By default, it breaks ties by a random selection in the set of candidates. If this is undesirable, ties must be resolved in the *priority-comparison method* itself. Since most policies' evaluation methods are expected to be the same, this method can be directly inherited from the default method without any additional programming. In certain rare situations, when the default approach may *not* be satisfactory or efficient, the method can be redefined to override the default method.

Since most of the generic data structures and methods are either inherited or reused, the proposed framework requires much less implementation effort than the traditional way of implementing buffer replacement policies. This allows the programmer to focus on the definition and implementation of the specific aspects of the replacement policy to be realized. Imposing a standard framework based on priorities for the definition of a buffer replacement policy may be restrictive to the designer as well as the programmer. However, a unified framework provides the flexibility to implement several buffer pools associated with different replacement policies for the purpose of satisfying the needs of different sets of objects in diverse applications such as indices, collections of data, and collections of metadata. In order to realize this feature, files (the unit for object collections in StorM) are associated with a particular buffer pool, and hence a particular replacement policy. Several files may be associated with the same buffer pool, and a file may be opened in several buffer pools (with possibly different policies). According to the hierarchical model, local buffer pools, when created, can be associated with abstract buffer pools. The default available replacement policy provided for programmers who wish to ignore such issues is the standard LRU scheme associated with a single local buffer pool.

6.2 Verification of correctness

In this section, we wish to verify the accuracy of our implementation of LRU-K against the experimental results obtained from [35].

Table 3 Experimental results for the two-pool experiment

	LRU-1		LRU-2		
Buffer size	*StorM*	[35]	StorM	[35]	AO
100	0.22	0.22	0.449	0.459	0.500
120	0.26	0.26	0.493	0.496	0.501
140	0.29	0.29	0.500	0.502	0.502
160	0.32	0.32	0.502	0.503	0.503
180	0.35	0.34	0.503	0.504	0.504
200	0.37	0.37	0.503	0.505	0.505
250	0.42	0.42	0.505	0.508	0.508
300	0.45	0.45	0.510	0.510	0.510
350	0.48	0.48	0.512	0.513	0.513
400	0.50	0.49	0.513	0.515	0.515
450	0.50	0.50	0.518	0.517	0.518

6.2.1 Two-pool experiment

In this experiment, we studied the codes available from ftp://ftp.cs.umb.edu/pub/lru-k/lru-k.tar.Z and ran the experiments as described in [35].

We considered two pools of disk pages, Pool 1 with N_1 pages and Pool 2 with N_2 pages, where $N_1 < N_2$. Alternating references are made to Pool 1 and Pool 2 and then a page from that pool is randomly selected as the sequence element. Thus, each page of Pool 1 has a probability of reference, $b_1 = \frac{1}{2N_1}$, of occurring as any element of reference string w, while the corresponding probability for each page of Pool 2 is $b_2 = \frac{1}{2N_2}$. The objective of this experiment is to model the alternating references to index and record pages. We conducted simulations with disk page pools of $N_1 = 100$ pages and $N_2 = 10,000$ pages. The buffer hit ratio (ratio between the number of hits and the number of logical references) of the LRU-2 scheme implemented on StorM was measured and compared with those in [35]. The results are shown in Table 3. The difference between the results obtained using our StorM implementation and those of [35] is due to the random nature of the dataset. When the experiments are run on the same dataset, the results obtained are identical.

We also ran the same experiment for LRU-3, LRU-4, LRU-6, and LRU-8. The results generally indicate slight improvement over LRU-2, with some converging toward AO (the optimum policy's hit ratio). However, the improvement is *not* drastic. Incidentally, this confirms the superiority of LRU-2 when the history of access patterns is short and unstable.

6.2.2 Random access experiment

In this experiment, we used a synthetic workload with random references to a set of pages with a Zipfian distribution of reference frequencies. As in [35], we generated references to $N = 1000$ pages, numbered from 1 to N, with a Zipfian distribution of reference frequencies. In this experiment, the probability of referencing a page with page number less than

Table 4 Experimental results for Zipfian distribution of references

	LRU-1		LRU-2		
Buffer size	StorM	[35]	StorM	[35]	AO
40	0.53	0.53	0.59	0.61	0.640
60	0.58	0.57	0.63	0.65	0.677
80	0.62	0.61	0.66	0.67	0.705
100	0.64	0.63	0.68	0.68	0.727
120	0.66	0.64	0.70	0.71	0.745
140	0.67	0.67	0.71	0.72	0.761
160	0.70	0.70	0.74	0.74	0.776
180	0.71	0.71	0.75	0.73	0.788
200	0.73	0.72	0.75	0.76	0.825
300	0.79	0.78	0.81	0.80	0.846
500	0.86	0.87	0.87	0.87	0.908

or equal to i was $(i/N)^{\log a/\log b}$, where the constants a and b are between 0 and 1. The relationship between a and b is that a fraction a of the references accesses a fraction b of the N pages, and the same relationship holds recursively within the fraction b of hotter pages and the fraction $(1-b)$ of colder pages [35].

Again the buffer hit ratio of LRU-2 modeled using our framework was obtained and compared with the results in [35]. Table 4 indicates that the results are similar. We also ran the same experiment for LRU-3, LRU-4, LRU-6, and LRU-8. The improvement shown by LRU-K ($K > 2$) was found to be more significant as compared to the case of the two-pool experiment. Moreover, the results depict the converging trend toward the optimum policy's hit ratio.

6.3 Extensibility vs. overhead

Many commercial systems employ the approach of providing "configuration" knobs to meet different organizational needs. This is usually achieved with a well-designed backend and a slight overhead as compared to a monolithic specialized system. The overhead must be small in comparison with the overall performance and efficiency gain. Likewise for our case, the extensibility should not cause prohibitive overhead.

In our implementation of the proposed extensible buffer manager, we made heavy use of the concepts of object inheritance and polymorphism supported by Java. Different replacement policies (e.g., the LRU Scheme Class) can be created by inheriting the parent PriorityScheme class and overriding the parent's methods. The buffer manager uses a polymorphism to invoke the correct child methods during the assignment, update, and evaluation of the priority value of a buffer slot. While these facilitate ease of programming and extensibility of the system, the use of a polymorphism and the need to resolve the methods from their respective parent classes to the corresponding correct child classes do incur additional overhead.

To study the effect of inheritance and polymorphism on the performance of the buffer manager, we also implemented the monolithic LRU-2 scheme (also in Java) and timed the execution of both with different buffer pool sizes. The study was conducted on a SUN Solaris machine with eight CPUs. The buffer pool size was incremented gradually until no further significant improvement on the hit ratio could be observed. Our findings reveal that the extensible buffer manager incurs almost the same amount of overhead as compared to the monolithic LRU-2 scheme. In fact, experimental results demonstrate that running the buffer manager with the framework is actually faster than running it without the framework as the buffer pool size increases. Table 5 illustrates the CPU time required to run the LRU-2 replacement strategy.

Table 5 CPU comparison

Buffer size	Hit ratio	Without framework (in sec)	With framework (in sec)
100	0.4527	1.713	2.074
120	0.4893	1.924	2.164
140	0.498	2.121	2.665
160	0.5003	2.404	2.628
180	0.502	2.711	2.793
200	0.5037	2.888	2.829
250	0.505	3.204	3.268
300	0.508	3.586	3.709
350	0.5103	3.923	4.304
400	0.516	4.310	4.447
500	0.5203	5.002	5.282
600	0.5263	5.883	5.897
700	0.5317	8.438	7.028
800	0.538	18.044	8.673
900	0.5433	40.981	11.887
1,000	0.5473	63.344	16.425
1,200	0.5537	109.859	25.383
1,400	0.5597	207.730	41.776
1,600	0.5613	357.085	80.911
1,800	0.5633	569.117	139.192
2,000	0.5633	844.770	201.619

In the monolithic LRU-2 implementation, a history reference hash table (each element in this hash table is a linked list) has to be kept globally to keep track of the reference history of each buffer slot. Whenever a buffer slot is accessed/replaced, this hash table has to be scanned to obtain/update the priority value of a buffer slot. This scanning process requires CPU time. In contrast, our framework stores the priority values as an array within the slot itself. Therefore, the priority value of a slot can be obtained/updated directly.

When the buffer size is small, the number of elements in the hash table of the monolithic LRU-2 implementation is small. Consequently, the time required for obtaining the priority value of a given buffer slot from this hash table is small. As the buffer size increases, the number of elements in the hash table also increases. In this situation, the monolithic LRU-2 implementation requires more time than our extensible framework since accessing an array is faster than accessing a hash table. Hence, our framework performs more efficiently as the buffer pool size increases.

Table 6 Hit ratio of B^+-tree queries under different replacement policies

Buffer size	LRU-1	MRU	LRD	LRU-2	LUU
100	0.266	0.030	0.337	0.450	0.164
200	0.476	0.070	0.529	0.537	0.359
300	0.594	0.109	0.604	0.582	0.433
400	0.659	0.146	0.653	0.614	0.472
500	0.692	0.184	0.687	0.643	0.512
600	0.707	0.222	0.718	0.670	0.555
700	0.719	0.259	0.748	0.695	0.585
800	0.731	0.297	0.779	0.719	0.616
900	0.752	0.334	0.810	0.743	0.645
1000	0.785	0.372	0.834	0.763	0.669

6.4 Improving index performance

Since every DBMS employs indexes in order to facilitate speedy retrieval of data, issues concerning performance of the indexes are critical to the overall system performance. The most widely used indexes for one-dimensional data are the B-tree [2] and its variants (e.g., the B^+-tree), while the R-tree [20] and its variants are the most popular index structures for indexing multidimensional data. For databases of huge size, such indexes are usually too large to reside in the main memory, and hence a good buffer replacement policy, which can maximize the hit ratio of index pages, is of paramount importance. In this section, we demonstrate the the ease in implementing several different replacement policies in our extensible buffer manager and show how the flexibility of our proposed framework can be exploited in case of index structures.

6.4.1 Point queries on a B^+-tree

For this experiment, we created a B^+-tree using StorM with a fanout of 52. We randomly generated one million numbers in the range of 1–10,000,000 and inserted them into the B^+-tree. The fanout of 52 was derived from the maximum number of entries that can fit into a physical data page of StorM. Note that a data page of StorM is set at 4 kbytes. We randomly generated 100,000 B^+-tree point queries based on the 10–90% rule where 10% of the one million records are accessed 90% of the time while the remaining records are accessed 10% of the time. The same set of queries was run on different replacement policies available in StorM, namely, LRU, MRU, LRU-2, LRD, and LUU with varying buffer sizes ranging from 100–1,000. To simulate a "hot" buffer,[5] the first 1,000 queries were ignored in the computation. We obtained the hit ratio needed for each buffer replacement policy to process all the queries. Table 6 summarizes the results of this experiment. The replacement policy, which is best suited to a given query processing pattern, can be deduced on the basis of the results obtained.

[5] "Hot" buffer refers to a buffer in which all the slots are filled.

The results of this experiment indicate that when the buffer size is small (less than or equal to 200), LRU-2 is the clear choice as the replacement policy. However, as the buffer size increases (above 300), both the LRD and LRU are better candidates as compared to LRU-2. Hence it is clear that LRU-2 may *not* always be the winner in all cases. This underlines the importance of a framework that is flexible enough to model different replacement policies to cater to varying applications' needs.

6.4.2 Range queries on B^+-trees

This experiment examines how knowledge about data structures can be employed to design novel schemes. We have used B^+-tree range query traces to conduct this experiment. This is in contrast to the corresponding experiment in [35], where OLTP traces were used.

When managing index data structures, additional knowledge about the data structure itself and its access patterns can facilitate in devising replacement policies that are tailored for the buffer pool of the index pages [8, 28, 29, 35]. An interesting yet very simple as well as effective strategy for B^+-tree buffering is to use the level of the page in the tree to determine the page priority. Pages at higher levels in the tree have higher priority since they are more likely to be visited by other queries. For purposes of buffer replacement, a random choice is made between buffer pages with the same priority.

For this experiment, we generated 500,000 random numbers from 0 to the maximum integer value and inserted them into a B^+-tree. The node size was kept small (with a fanout of 20) in order to yield a taller tree. Five hundred queries were randomly generated, each requiring the scanning of about 10% of the leaf nodes, and their traces of traversal were put into a file. To simulate the situation of 10 concurrent transactions, we picked 10 queries, and from those we randomly picked the next node to visit. When a range query had been executed to completion, we picked the next query in the list. For the performance evaluation, we dropped the first 1,000 references in order to allow the algorithms to reach a quasistable state. Table 7 summarizes the buffer hit ratio for the respective algorithms, namely, the priority-based replacement algorithm and the LRU-2 algorithm. We present the average hit ratio of 5,000 references and 40,000 references, respectively. In real-world applications, multiple indexes are maintained for each table for different purposes such as for answering popular queries and generating infrequent but expensive reports. These two references (5,000 vs. 40,000) represent contrasting situations: (1) when traversal of a particular index is infrequent and interleaved with other index traversals and (2) when traversal of an index is intensive and continuous.

In situation (1), most of the 5,000 references access mostly the internal nodes of the B^+-tree rather than the leaf nodes. Since the priority-based replacement strategy gives higher priority to the internal nodes of the B^+-tree, it is likely to achieve a good hit ratio. In contrast, the nature

Table 7 Performance results for range queries on B^+-trees

	5,000 ref.		40,000 ref.	
Buffer size	Priority	LRU-2	Priority	LRU-2
100	0.0390	0.0255	0.0295	0.0301
120	0.0450	0.0305	0.0350	0.0347
140	0.0495	0.0322	0.0384	0.0392
160	0.0555	0.0347	0.0427	0.0433
180	0.0612	0.0355	0.0461	0.0486
200	0.0677	0.0385	0.0522	0.0541
220	0.0732	0.0410	0.0566	0.0590
240	0.0782	0.0417	0.0594	0.0623
260	0.0845	0.0425	0.0628	0.0658
280	0.0908	0.0430	0.0668	0.0695
300	0.0972	0.0437	0.0726	0.0734

Table 8 Hit ratio of R-tree queries under different replacement policies with buffer size 100

Replacement policies	Hit ratio
LRU	0.603
MRU	0.595
LRD	0.595
LRU-2	0.643
LUU	0.625

of such access patterns makes it difficult for the LRU-2 algorithm to detect locality of page references. As observed in the result of the experiment with 5,000 references, the priority-based replacement strategy is superior.

In situation (2), most of the references access the leaf nodes instead of the internal nodes. Since all the leaf nodes in the B^+-tree are chained at the leaf level, the locality of page references in such access patterns is easily detected by the LRU-2 algorithm. On the other hand, the priority-based replacement strategy is likely to incur more "buffer misses" because the buffer pool is filled with internal nodes. As observed in the result of the experiment with 40,000 references, the performance of the two buffer strategies are comparable.

6.4.3 Experiment on the R-tree

A similar experiment was also conducted for the R-tree. In the case of the R-tree, the data stored is a set of one million rectangles obtained from a real dataset known as *Greece Roads* (`http://dias.cti.gr/~ytheod/research/datasets/spatial.html`). We randomly generated 100,000 *containment* queries from the dataset. To simulate "hot" buffer, the first 1,000 queries were ignored. The buffer size for the R-tree experiment was set at 100. Table 8 summarizes the results for this experiment. The experimental results reveal that the LRU-2 replacement policy clearly outperforms the rest.

7 Conclusion

In this paper, we have proposed a framework for modeling buffer replacement policies. We have illustrated the expressibility as well as the flexibility of our proposed framework by using it to model some of the existing buffer replacement policies. This framework can provide the basis for realizing buffer replacement extensibility, which is lacking in existing extensible DBMSs. Supporting extensibility in this layer of the system can have a great impact on system performance since disk I/O still remains a critical performance bottleneck. The proposed framework was implemented and a performance study conducted. The results of our performance study clearly demonstrate that an extensible buffer replacement policy component based on the proposed framework can be constructed efficiently.

Acknowledgements We would like to thank Anirban Mondal, Chee Yong Chan, Stephen Bressan, and Kian-Lee Tan for their contributions and the referees for their suggestions in improving the paper.

References

1. Batory, D.S., Barnett, J.R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.C., Wise, T.E.: GENESIS: an extensible database management system. IEEE Trans. Softw. Eng. **14**(11), 1711–1729 (1988)
2. Bayer, R., McCreight, E.: Organization and maintenance of large ordered indices. Acta Informatica **1**(3), 173–189 (1972)
3. Bressan, S., Goh, C.L., Ooi, B.C., Tan, K.L.: A framework for modeling buffer replacement strategies. In: Proceedings of ACM CIKM international conference on information and knowledge management, pp. 62–69 (2000)
4. Brown, K.P., Carey, M.J., Livny, M.: Goal-oriented buffer management revisited. In: Proceedings of the 1996 ACM-SIGMOD international conference, pp. 353–364 (1996)
5. Carey, M.J., DeWitt, D.J., Graefe, G., Haight, D.M., Richardson, J.E., Schuh, D.T., Shekita, E.J., Vandenberg, S.L.: The EXODUS extensible DBMS project: an overview. In: Zdonik, S., Maier, D. (eds.) Readings in object-oriented database systems, pp. 474–499. Morgan Kaufmann, San Francisco (1990)
6. Carey, M.J., Haas, L.M.: Extensible database management systems. ACM SIGMOD Rec. **19**(4), 54–60 (1990)
7. Carey, M.J., Jauhari, R., Livny, M.: Priority in DBMS resource scheduling. In: Proceedings of 15th international conference on very large data bases, pp. 397–410. Amsterdam (1989)
8. Chan, C.Y., Ooi, B.C., Lu, H.: Extensible buffer management of indexes. In: Proceedings of 18th international conference on very large data bases, pp. 444–454. Vancouver, Canada (1992)
9. Chou, H.-T.: Buffer management of database systems. PhD thesis, University of Wisconsin-Madison. In: Computer Sciences Technical Report 597, University of Wisconsin-Madison (1985)
10. Chou, H.-T., DeWitt, D.J.: An evaluation of buffer management strategies for relational database systems. In: Proceedings of 11th international conference on very large data bases, pp. 127–141. Stockholm (1985)
11. Chou, H.-T., DeWitt, D.J., Katz, R.H., Klug, Design, A.C.: Implementation of the Wisconsin storage system. Soft. Pract. Exp. **15**(10), 943–962 (1985)
12. Denning, P.J.: The working-set model for program behaviour. Commun. ACM **11**(5), 323–333 (1968)
13. Dittrich, K., Geppert, A.: (eds.) Component Databases. Morgan Kaufmann, San Francisco (2001)

14. Effelsberg, W., Haerder, T.: Principles of database buffer management. ACM Trans. Database Syst. **9**(4), 560–595 (1984)
15. University of Wisconsin-Madison. Using the EXODUS storage manager, vo1. 3. (1991)
16. Freytag, J.C.: A rule-based view of query optimization. In: Proceedings of ACM SIGMOD conference, pp. 173–180. San Francisco (1987)
17. George, B., Haritsa, J.R.: Secure buffering in firm real-time database systems. In: VLDB Conference, pp. 364–475 (1998)
18. Goh, C.L., Bressan, S.: Storm: a 100% Java persistent storage manager. In: Java and Databases L'objet, vol. 6, Hermes Science Publications, Paris, pp. 305–316 (2000)
19. Graefe, G., DeWitt, D.J.: The EXODUS optimizer generator. In: Proceedings of ACM SIGMOD conference, pp. 160–172. San Francisco (1987)
20. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of ACM SIGMOD conference, pp. 47–57. Boston (1984)
21. Haas, L.M., Chang, W., Lohman, G.M., McPherson, J., Wilms, P.F., Lapis, G., Lindsay, B., Pirahesh, H., Carey, M.J., Shekita, E.: Starburst mid-flight: as the dust clears. IEEE Trans. Knowl. Data Eng. **2**(1), 143–160 (1990)
22. Haas, L.M., Freytag, J.C., Lohman, G.M., Pirahesh, H.: Extensible query processing in Starburst. In: Proceedings of ACM SIGMOD conference, pp. 377–388. Portland, OR (1989)
23. Huang, J., Stankovic, J.A.: Buffer management in real-time databases. COINS Technical Report 90-65, University of Massachusetts-Amherst (1990)
24. Jauhari, R., Carey, M.J., Livny, M.: Priority-hints: an algorithm for priority-based buffer management. In: Proceedings of 16th international conference on very large data bases, pp. 708–721. Brisbane, Australia (1990)
25. Jiang, S., Zhang, X.: LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In: Proceedings of SIGMETRICS, pp. 31–42 (2002)
26. Kim, I., Yeom, H.Y., Lee, J.: Analysis of buffer replacement policies for www proxy. In: Proceedings of ACM symposium on applied computing table of contents, pp. 98–103. Atlanta, GA (1998)
27. Lee, M.K.: Interaction between the query processor and buffer manager of a relational database system. Master's thesis, Massachusetts Institute of Technology, 1989. In: Research Report RJ6884 (65710), IBM Research Division, Almaden Research Center, San Jose, CA (1989)
28. Lee, S.H., Whang, K.Y., Moon, Y.S., Song, I.Y.: Dynamic buffer allocation in video-on-demand systems. In: Proceedings of ACM SIGMOD conference, pp. 343–354 (2001)
29. Leutenegger, S.T., Lopez, M.A.: The effect of buffering on the performance of r-trees. In: ICDE, pp. 164–171 (1998)
30. Lindsay, B., Haas, L.: Extensibility in the Starburst experimental database system. In: Blaser, A. (eds.), Lecture Notes in Computer Science 466, pp. 217–248. Berlin Heidelberg New York: Springer (1990) In: Proceedings of international symposium on database systems of the 90s, Muggelsee, Berlin (1990)
31. Lindsay, B., McPherson, J., Pirahesh, H.: A data management extension architecture. In: Proceedings of ACM SIGMOD conference, pp. 220–226. San Francisco (1987)
32. Linnemann, V., Kuspert, K., Dadam, P., Pistor, P., Erbe, R., Kemper, A., Sudkamp, N., Walch, G., Wallrath, M.: Design and implementation of an extensible database management system supporting user defined data types and functions. In: Proceedings of 14th international conference on very large data bases, pp. 294–305. Los Angeles (1988)
33. Lohman, G.M.: Grammar-like functional rules for representing query optimization alternatives. In: Proceedings of ACM SIGMOD conference, pp. 18–27. Chicago (1988)
34. McLeod, D.: 1988 VLDB panel on future directions in DBMS research: a brief, informal summary. ACM SIGMOD Rec. **18**(1), 27–30 (1989)
35. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The lru-k page replacement algorithm for database disk buffering. In: Proceedings of ACM SIGMOD conference, pp. 297–306 (1993)
36. Ong, J., Fogg, D., Stonebraker, M.: Implementation of data abstraction in the relational database system ingrres. ACM SIGMOD Rec. **14**(1), 1–14 (1984)
37. Osborn, S.L., Heaven, T.E.: The design of a relational database system with abstract data types for domains. ACM Trans. Database Syst. **11**(3), 357–373 (1986)
38. L. Rowe and Stonebraker, M.: The POSTGRES data model. In: Proceedings of 13th international conference on very large data bases, pp. 83–96. Brighton, UK (1987)
39. Sacco, G.M., Schkolnick, M.: A mechanism for managing the buffer pool in a relational database system using the hot set model. In: Proceedings of 8th international conference on very large data bases, pp. 257–262. Mexico City (1982)
40. Sacco, G.M., Schkolnick, M.: Buffer management in relational database systems. ACM Trans. Database Syst. **11**(4), 473–498 (1986)
41. Silberschatz, A., Stonebraker, M., Ullman, J.: Database systems: achievements and opportunities. ACM SIGMOD Record **19**(4), 6–22 (1990) The "Lagunita" report of the NSF invitational workshop on the future of database systems research. Palo Alto, CA (Feb. 22–23, 1990)
42. Stonebraker, M.: Operating system support for database management. Commun. ACM **24**(7), 412–418 (1981)
43. Stonebraker, M.: Inclusion of new types in relational database systems. In: Proceedings of 2nd international conference on data engineering, pp. 262–269. Los Angeles (1986)
44. Stonebraker, M., Rowe, L.A., Hirohama, M.: The implementation of POSTGRES. IEEE Trans. Knowl. Data Eng., pp. 125–142 (1990)
45. Stonebraker, M., Rowe, L.A., Lindsay, B., Gray, J., Carey, M.J., Brodie, M., Berstein, P., Beech, D.: Third-generation database system manifesto. ACM SIGMOD Rec. **19**(3), 31–44 (1990). Committee for Advanced DBMS Function
46. Teng, J.Z., Gumaer, R.A.: Managing IBM database 2 buffers to maximize performance. IBM Syst. J. **23**(2), 211–218 (1984)
47. Velez, F., Bernard, G., Darnis, V.: The O_2 object manager: an overview. In: Proceedings of 15th international conference on very large data bases, pp. 357–366. Amsterdam (1989)
48. Wilms, P.F., Schwarz, P.M., Schek, H.-J., Haas, L.M.: Incoporating data types in an extensible database architecture. In: Beeri, C., Schmidt, J.W., Dayal, U. (eds.) Proceedings of 3rd international conference on data and knowledge bases: improving usability and responsiveness, pp. 180–192. Jerusalem (1988)
49. Zdonik, S.B., Maier, D.: Relational extensions and extensible database systems. In: Zdonik, S.B., Maier, D. (eds.) Readings in object-oriented database systems. Data Management Systems, pp. 445–449. Morgan Kaufmann, San Francisco (1990)