A Framework for Modeling Buffer Replacement Strategies

Stephane Bressan, Chong Leng Goh, Beng Chin Ooi, Kian-Lee Tan
School Of Computing
National University Of Singapore
Building S-16, Level 5, Room 05/08
3 Science Drive 2, Singapore 117543
steph,gohcl,ooibc,tankl@comp.nus.edu.sg

ABSTRACT

An effective buffer management system is crucial for any database management system. While much work has been expended to provide extensible data types, extensible query languages and even extensible optimizers, there is very limited research in providing extensibility at the buffer management level. Supporting extensibility at the buffer management level is equally, if not more, important as no single strategy can perform well in all applications efficiently. In this paper, we present a uniform framework for modeling buffer replacement policies. The framework allows the buffer manager to be easily extended to provide support for and fine-tuning of different replacement policies. Our work is novel in two aspects. First, the proposed framework unifies existing work in this area. Second, our work introduces a new level of extensibility. To our knowledge, none of the existing extensible DBMSs and storage managers provide extensibility at the buffer management level. We implemented an extensible buffer manager and experimented with different buffer replacement polices. The experimental study illustrates the ease of use and efficiency of the proposed framework.

1. INTRODUCTION

The buffer manager 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, the replacement policy, is to identify a buffer page to be replaced when a page fault occurs. The goal is to manage the replacement of buffer pages efficiently to avoid unnecessary page faults.

Many replacement policies have been proposed in the literature: LRU, LRU-K [14], Hotset [15], DBMIN [5], ranking-aware [11], Priority-LRU [3], Priority-DBMIN [3] and Priority-Hints [10, 4], Hints-based policies [6, 12]. Not surprisingly, none of these techniques has been shown to guarantee good performance for all applications. It is clear that as DBMSs are increasingly being deployed for new and advanced applications, we can expect to see more specialized replacement policies being proposed. One such example is the ranking-aware buffer replacement strategy recently proposed for search engine like retrievals [11].

Unfortunately, most of the existing buffer managers support only one fixed replacement policy, which is typically the LRU scheme. For a system such as Oracle 8i that has different cartridges to support a wide range of applications, such as text, image and spatial applications, it only supports LRU replacement strategy. Even in extensible storage managers such as EXODUS, the buffer manager is also designed as a non-extensible part of the kernel of the system, and supports only two replacement policies [16]. 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.

In this paper, we propose that the buffer manager should be extensible. First, supporting extensibility at the buffer management level is essential since extensions made at one level of a DBMS require extension support at other levels as well [1]. For example, having a new data type extension typically requires making extensions to both the access method and query optimizer. With buffer replacement extensibility, it may be possible to custom-tailor a smart buffer replacement policy to manage buffer replacements intelligently by exploiting the reference behaviour of new access methods or transactions. Second, the emergence of more complex applications also provides more opportunities for exploiting domain specific semantics to improve system performance at the buffer manager level. Finally, supporting buffer level

CIKM 2000, McLean, VA USA

extensibility facilitates the evaluation and fine-tuning of new buffer replacement policies for domain specific operators, algorithms or applications [7, 8, 11, 13].

The remaining of the paper is organized as follow. In Section 2, we introduce our framework for an extensible buffer management component, with its structure, its properties and behavior and the priority scheme. Section 3 shows how the framework can be used to model replacement policies. In particular, we will discuss how three well-known replacement policies, namely LRU, MRU and LRU-2 can be implemented using the proposed priority scheme. Section 4 demonstrates with a complete example how the framework can be used to model more complex buffer replacement policies such as the Priority-Hints, Priority-LRU and Priority-DBMIN Algorithms. In Section 5, we present the performance analysis on our proposed framework. Section 6 concludes with a discussion on the contribution of the work.

2. A FRAMEWORK FOR DESIGNING BUFFER REPLACEMENT POLICIES

Our framework consists of two components: a hierarchical model of buffer pool that supports a two-level model for buffer replacement policies, and a priority scheme that generalizes the representation of replacement criterion.

2.1 Hierarchical Organization of Buffer Pool

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

2.2 Priority-Based Scheduling Scheme

Replacement selection policies are essentially scheduling policies that prioritize buffer pages for replacement based on some selection criteria. For example, the LFU selection policy schedules the less frequently used pages for replacement before the more frequently used ones, while the LRU selection policy schedules the less recently accessed pages for replacement before the more recently accessed buffer pages. We can thus prioritize local and abstract buffer groups, so that local/abstract groups with the lowest priorities are candidates for replacement.

2.3 The Framework for Replacement Policies

Our framework is as follows. Each abstract buffer group is associated with an abstract selection policy, and each local buffer group is associated with a local selection policy. 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. Associated with each local (abstract) selection policy is some information, referred to as local (abstract) 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 abstract selection policy first selects from among the various 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.

3. MODELING BUFFER REPLACEMENT POLICIES

3.1 Modeling Selection Policies

Relating the implementation of this priority scheme to object-oriented concepts, a priority scheme is a class definition where each class instance has an instance variable of type C (control information) and a set of methods for assignment, modification and evaluation of priority values. A new replacement policy class can be defined by inheriting the priority scheme class and overriding the methods defined within the class.

One important note about our priority scheme is the modeling of priority value for different replacement policies. Due to the different criteria of buffer replacement policies, the priority values for buffer page (group) may have to be modeled differently. A higher level of abstraction is needed so that a uniform way of modeling the priority value of buffer page (group) regardless of the buffer replacement policy can

be achieved. This abstraction can be easily realized by *object inheritance*. Using object inheritance, the priority value attached to a buffer page (group) is defined with a generic class. When there is a need for a new form of priority value, the new priority value can be created through object inheritance.

A priority scheme, PS, is modeled as follows.

$PS = (P, C, \alpha, \beta, \gamma, \delta, \epsilon)$

The whole idea of creating the PriorityValue class with a generic Object class, P, is to facilitate the task of modeling the priority value of different buffer replacement policy. Because Object class in Java[1] is automatically inherited by all classes (whether user-defined or its libraries), the attribute class P can be assigned with any user-defined class to model different priority value needed for a new buffer replacement policy. We shall see a few examples of this in the later sections.

Attribute C in class PriorityScheme defines the control information class used in the assignment, update, and evaluation of priority values. For similar reason with attribute P in PriorityValue class, it is defined using a generic Object class in Java.

The five methods are:

1. $\alpha : C_{old} \rightarrow C_{new}$ is an initialization method that initializes the control information associated with a PriorityScheme class.

2. $\beta : S_{old} \rightarrow S_{new}$ is a priority assignment method that assigns an initial priority value to a page (group) associated with a PriorityScheme class. The parameter in β is S_{old} where S_{old} refers to the buffer slot information for the buffer page (group) and the updated slot information S_{new} is returned.

3. $\gamma : S_{old} \rightarrow S_{new}$ is a priority update method that updates the priority value of a re-accessed buffer page (group). The parameter in γ is S_{old} where S_{old} refers to the buffer slot information of a re-accessed buffer page (group) and the updated slot information S_{new} is returned.

4. $\delta : S_i \times S_j \rightarrow S_{lowest}$ is a priority comparison method that compares two buffer pages' (groups'), priority values, S_i and S_j, and returns the priority value of the buffer page (group), S_{lowest}, which is lowest.

5. $\epsilon : S_{ALL} \rightarrow S_i$ is a priority evaluation method that evaluates all the buffer page's (group's) priority values within a buffer group, S_{ALL} and returns the slot location of a buffer page (group) with the lowest priority. This method uses the priority comparison to identify the slot in a buffer page (group) with the lowest priority. A default method based on the above argument is provided. However, in cases where the default evaluation method is not sufficient, the programmer can easily redefine it through method overriding.

[1]We implemented our extensible buffer manager in Java.

3.2 Modeling Local Selection Policy Class

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 it's priority value when the buffer page is re-accessed. 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.

A local selection policy is modeled by first inheriting the PriorityScheme class defined in the above section. New replacement behavior can be created by:

1. overriding the five methods $(\alpha, \beta, \gamma, \delta, \epsilon)$ in the priority scheme, and

2. defining a priority value and control information needed for the priority scheme.

We shall illustrate the modeling of local selection policy class with three sample replacement policies. The first example is the LRU scheme, and demonstrates how easy it is to model the scheme in our framework. The second is the MRU scheme, and it shows even greater ease in modeling when methods from existing replacement policies (LRU in our case) can be reused. Finally, we also show how to model a more complicated LRU-2 scheme. Throughout this paper, we use S.priority to denote the priority value of a buffer page and assume that all these three replacement policies use the default priority evaluation method.

3.2.1 Modeling the Least Recently Used Policy (LRU)

An LRU policy replaces the least recently used page. The priority of a buffer page can therefore be represented by a logical timestamp value such that 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 method 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 re-accessed, 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 method is the same as the priority assignment method. Since a page with a large timestamp value has a higher priority, the priority comparison method simply returns a buffer page's priority value with the smallest value. Using the priority comparison method, the priority evaluation method obtains the buffer page with the smallest priority value and returns it. The algorithm for modeling the LRU local selection policy is outlined in Figure 1.

$PS_{LRU} = \{P_{LRU}, C_{LRU}, \alpha_{LRU}, \beta_{LRU}, \gamma_{LRU}, \delta_{LRU}, \epsilon_{LRU}\ \}$
$C_{LRU} = P_{LRU}$
$P_{LRU} =$ Integer type
$\alpha_{LRU}() \{$
    $c \leftarrow 0$
$\}$
$\beta_{LRU}(S_{old}) \{$
    $c \leftarrow c + 1$
    $S_{old}.priority \leftarrow c$
    return S_{old}
$\}$
$\gamma_{LRU} = \beta_{LRU}$
$\delta_{LRU}(S_i, S_j) \{$
    if $(S_i.priority > S_j.priority)$
        return S_j
    return S_i
$\}$

Figure 1: Modeling of LRU local selection policy.

3.2.2 Modeling the Most Recently Used Policy (MRU)

Unlike the LRU policy, an MRU policy replaces the most recently used page. The modeling of the MRU local selection policy is shown in Figure 2. It is similar to that of the LRU policy, except that for the MRU policy, buffer page with larger timestamp value has lower priority since the MRU policy selects the most recently accessed page (page with largest timestamp value) for replacement. Therefore, the priority evaluation method returns the complement of a buffer page's timestamp value as its priority value.

$PS_{MRU} = \{P_{MRU}, C_{MRU}, \alpha_{MRU}, \beta_{MRU}, \gamma_{MRU}, \delta_{MRU}, \epsilon_{MRU}\ \}$
$P_{MRU} = P_{LRU}$
$C_{MRU} = C_{LRU}$
$\alpha_{MRU} = \alpha_{LRU}$
$\beta_{MRU} = \beta_{LRU}$
$\gamma_{MRU} = \gamma_{LRU}$
$\delta_{MRU} = complement(\delta_{LRU})$

Figure 2: Modeling of MRU local selection policy.

3.2.3 Modeling the Least Recently Used-2 Policy (LRU-2)

The LRU-2 algorithm, an instant of LRU-K [14], uses the Backward 2-distance to decide the replacement page. The Backward 2-distance is defined as follows. Given a reference string known up to the time t, $r_1, r_2, \ldots, r_t$, the backward 2-distance $b_t(p,2)$ is the distance backward to the 2nd most recent reference to the page p:

$$b_t(p,2) = x \quad \text{if } r_{t-x} \text{ has the value of } p \text{ and there has been exactly one } i \text{ with } t-x < i \leq t \text{ where } r_i = p$$
$$= \infty \quad \text{if } p \text{ does not appear at least 2 times in } r_1, r_2, \ldots, r_t$$

The LRU-2 selects the replacement page whose backward 2-distance is the maximum of all pages in the buffer. When there are more than one page with $b_t(p,2) = \infty$, a subsidiary policy is used to select the replacement victim among these pages; for example, classical LRU could be employed as a subsidiary policy. Note that LRU-1 corresponds to the classical LRU algorithm.

In LRU-2, there are two delicate tuning parameters which we must take note of when implementing it. The first is the *correlated reference period*. This is a time period during which a page is retained in the buffer once it has been accessed. The second tuning parameter is the *retained information period* which is the length of time a page's access history is remembered after it is ejected from the buffer.

Due to space constraint, the algorithm for modeling the LRU-2 local selection policy will not be presented here. Instead, it can be found in the original sbumitted paper which can be obtained from this web address
http://www.comp.nus.edu.sg/ gohcl/CIKM/cikm.ps.

3.3 Modeling Abstract Selection Policies

The function of an abstract selection policy is to select a buffer group when a page fault occurs so that the replace-

ment page can subsequently be selected from the appropriate local buffer group. For consistency, the selection of a buffer group from an abstraction buffer group is modeled in the same way as that of the local selection policy, i.e., using the same priority scheme. Each abstract buffer group is associated with a selection policy, referred to as abstract selection policy, which assigns and updates priority values to 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.

An abstract selection policy controls the assignment, modification, and evaluation of priority 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 groups contained in the abstract buffer group and picks the group with the lowest priority. Due to space limitation, and to minimize redundancy, we shall defer the example to Section 4.

3.4 Modeling Buffer Replacement Policy Class

By combining the modeling of local and abstract selection policy classes, a buffer replacement policy class is modeled as follows:

$$R = (S_A,\ M_A,\ S_L,\ M_L)$$

S_A is a finite set of abstract selection policy class. $M_A : N_A \rightarrow S_A$ is a surjective function that maps an abstract buffer group to an abstract selection policy class. Thus, each abstract buffer group of type $N_i \in N_A$ is associated with an abstract selection policy of class $AP_j \in S_A$.

$S_L = \{LP_1,\ \ldots,\ LP_q\}$ is a finite set of local selection policy classes, and $M_L : N_L \rightarrow 2^{S_L}$ is a function that maps a local buffer group class to a non-empty subset of local selection policy class.

Note that while M_A maps an abstract buffer group to an abstract selection policy type, M_L maps a local buffer group to a subset of local selection policy types. This is because the set of abstract selection policy mappings which forms, the first two steps of our model, is fixed and should be determined at compile-time. On the other hand, the mapping of a local selection policy class to a local buffer group can be determined at run-time depending on the page access pattern. For example, in the EXODUS storage manager [16], each local buffer group can be managed by either an LRU or an MRU replacement policy.

4. MODELING THE PRIORITY-HINTS ALGORITHM: A COMPLETE EXAMPLE

So far, we have only illustrated the use of local selection policy. In this section, we will take a closer look on how our proposed abstract and local selection policy can be combined to model a much complex algorithm, the priority-hints algorithm [6, 3, 10].

In the priority-hints algorithm, buffers are organised into "transaction sets" where a transaction set consists of all of the buffers owned by a single transaction. Transaction sets are arranged in priority order, with recency of arrival of the owner transaction being used to break ties if there are multiple transactions of the same priority. Each transaction set is made up of two kinds of buffers: buffers currently fixed by the owner and buffers containing unfixed favored pages of the owner. Besides the transaction sets, the algorithm maintains a free list of pages in LRU order.

Pages referenced by transaction set in priority-hints algorithm, are classified into two groups: pages that are likely to be re-referenced by the same transaction (favored pages) and pages that is likely to be referenced just once (normal pages). Unfixed favored pages are maintained in MRU order. Note that a transaction set never contains an unfixed normal page; whenever a normal page is unfixed, it is freed to the free list.

Explaining priority-hints in our terms would mean that the free list is modeled as a local buffer group with a LRU local selection policy and the list of transaction sets is modeled as a set of abstract buffer groups, each consists of two local buffer groups ($N_{trans-unfixed}$ and $N_{trans-fixed}$). A root abstract buffer group (N_{buffer}) is made up of a local buffer group (N_{free}) and a set of abstract buffer groups ($N_{priority}$).

The replacement criteria in the priority-hints algorithm can be explained using this hierarchical buffer organization as follows. The free list local buffer group (instance of N_{free}) will be selected as the replacement local buffer group (if it is

non-empty); otherwise, the selection of the local buffer group involves two levels of selection. The abstract selection policy attached to the N_{buffer} will first select a priority abstract buffer group (instance of $N_{priority}$) from the buffer pool and then the abstract selection policy attached to the selected priority abstract buffer group will select the transaction unfixed local buffer group (instance of $N_{trans-unfixed}$).

In the first selection, the policy will select the non-empty priority abstract buffer group with the lowest transaction priority level not equal to or greater than the priority level of the faulting-transaction. If no such group exists, the priority abstract buffer group with transaction priority level equal to that of the faulting transaction is selected. In the second selection, if the selected priority abstract buffer group has the same priority level as the faulting-transaction, the transaction local buffer group owned by the faulting-transaction will be selected; otherwise, the non-empty transaction local buffer group owned by the 'oldest' transaction[2] will be selected. Figure 3 shows the overall modeling of the priority-hints algorithm.

$R = (S_A,\ M_A,\ S_L,\ M_L)$ where
$S_A = \{AP_1,\ AP_2\}$
$M_A = \{(N_{buffer},\ AP_1),\ (N_{priority},\ AP_2)\}$
$S_L = \{PS_{LRU},\ PS_{MRU}\}$
$M_L = \{(N_{free},\ \{PS_{LRU}\}),\ (N_{trans-unfixed},\ \{PS_{MRU}\})\}$
where the definition for PS_{LRU} and PS_{MRU} are shown in Figure 1 and Figure 2 respectively

Figure 3: Modeling of priority-hints algorithm.

The eventual outcome of AP_1 is either the free list local buffer group or an abstract buffer group within the list of $N_{priority}$. If the free list local buffer group is selected, the replacement page is obtained through the LRU local selection policy. If one of the abstract buffer group from the list of $N_{priority}$ is selected, then the job of finding the replacement page is passed down to the abstract selection policy, AP_2 which is attached to the selected priority abstract buffer group.

Again, due to space constraint, the modeling of abstract selection policy AP_1 and AP_2 using our framework are not presented. They can be found in the original submitted paper which can be obtained at this web address http://www.comp.nus.edu.sg/ gohcl/CIKM/cikm.ps.

When there is a buffer miss, the control information c_2 in AP_1 will be set to the faulting transaction. This control information will then be used by the buffer manager to filter out all the priority abstract buffer groups whose priority values are greater than the faulting transaction, before the eligible set of buffer groups is being passed into the ϵ method in AP_1.

AP_2 is similar to AP_1. In the case of AP_1, the selection is done on a mixture of local buffer group (N_{free}) and a set of abstract buffer group ($N_{priority}$). For AP_2, we only need to make selection from two local buffer groups. Furthermore, one of the local buffer groups will never get selected because it represents the fixed pages hold by the current transaction.

[2]The age of a transaction is measured by its admission time; the oldest transaction therefore refers to the transaction that was admitted the earliest among the existing transactions.

5. IMPLEMENTATION AND PERFORMANCE ANALYSIS

We have implemented the extensible buffer manager as a component in StorM [9]. StorM is a 100% Java-based (JDK 1.2) storage manager, and consists of a set of Java classes and objects for the development and tuning of non-standard data intensive applications. It does not require a special compiler, a special abstract machine, or a set of (possibly OS specific) native methods. Exploiting the serialization properties of Java objects, StorM offers support for persistency for almost any Java object. StorM also supports an independent notion of persistent heterogeneous and homogeneous collection (files) of Java objects. The latter feature can be used for instance for the efficient implementation of relations.

We coded a number of buffer replacement strategies using the interface provided by StorM. In this paper, we use LRU-K [14], a very efficient replacement strategy, to illustrate the programming of our extensible buffer manager. We also coded LRU-K in Java based on the C codes provided by [14] to provide us a reference point for comparisons and cross checking.

5.1 Experiments Using LRU-K

We conducted three sets of experiments. The objective is two fold. We want to show that none of the techniques are superior in all cases. This calls for the need to easily extend existing buffer replacement policies. Second, being able to compare several different replacement policies demonstrate the ease in implementing them in our extensible buffer manager.

5.1.1 Two-Pool Experiment

In this set of 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 [14].

Basically, we considered two pools of disk pages, Pool 1 with N_1 pages and Pool 2 with N_2 pages, where $N_1 < N_2$. In this two-pool experiment, alternating references are made to Pool 1 and Pool 2; then a page from that pool is randomly chosen to be 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, and each page of Pool 2 has probability $b_2 = \frac{1}{2N_2}$. This experiment is meant to model the alternating references to index and record pages. Simulation results of the two-pool experiment, with disk page pools of $N_1 = 100$ pages and $N_2 = 10,000$ pages were conducted. The buffer hit ratio (number of hits divided by the number of logical references) of the LRU-2 implemented on StorM is measured and compared to those given in [14]. The results are shown in Table 1. The difference between the results obtained using StorM implementation and those of [14] is due to random nature of the data set. We, however, obtained the same results when we ran both implementations on the same data set.

Buffer Size	*LRU-1*		*LRU-2*		AO
	StorM	[14]	StorM	[14]	
60	0.14	0.14	0.274	0.291	0.300
80	0.18	0.18	0.368	0.382	0.400
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
Summary: Max Averaging Diff. in LRU-1 Hit Ratio = 0.0015 Max Averaging Diff. in LRU-2 Hit Ratio = 0.0033					

Table 1: Performance results for the two-pool experiment.

5.1.2 *Zipfian Random Access Experiment*

In this set of experiment, we use a synthetic workload with random references to a set of pages with a Zipfian distribution of reference frequencies as in [14].

Again the buffer hit ratio of LRU-2 modeled using our framework is obtained and compared to the results in [14].

Buffer Size	*LRU-1*		*LRU-2*		AO
	StorM	[14]	StorM	[14]	
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
Summary: Max Averaging Diff. in LRU-1 Hit Ratio = 0.0073 Max Averaging Diff. in LRU-2 Hit Ratio = 0.0090					

Table 2: Performance results for Zipfian distribution of references.

As can be seen from Table 2, we obtain similar results.

5.1.3 *Range Queries on B^+-Trees*

Finally, this last set of experiments study how knowledge about data structures can call for novel schemes to be designed. Instead of using OLTP traces as in [14], we used B+-tree range query traces to conduct this experiment.

When managing index data structures, the additional knowledge about the data structure itself (a tree structure) and its access patterns (depth first, backtrack free for B^+-tree), can help devise tailored replacement policies for the buffer pool for the index pages [4, 13, 14]. An interesting but very simple 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 as they are more likely to be visited by other queries. In buffer replacement, a random choice is made between buffer pages with the same lowest priority.

We generated 500,000 random numbers from 0 to the maximum integer value, and inserted them into a B^+-tree. We keep the node size small, with a fan-out of 20, to yield a taller tree. 500 queries were randomly generated, with each requiring the scanning of about 10% of leaf nodes, and their traces of traversal are dumped into a file. To simulate the situation of 10 concurrent transactions, we picked 10 queries, and among them, we randomly picked the next node to visit. When a range query was completed, we picked the next query in the list. For the performance evaluation, we dropped the first 1000 references to allow the algorithms to reach a quasi-stable state. Table 3 summarises the buffer

hit rate ratio for each algorithm, priority-based replacement and LRU-2. Due to space constraints, we only present the average hit ratio respectively of 5000 references and 40000 references. 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 (5000 vs. 40000) represent contrasting situations: (1) when traversal of a particular index is infrequent and interleaved with other index traversal; (2) when traversal of an index is intensive and continuous.

Buffer Size	*5000 ref.*		*40000 ref.*	
	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 3: Performance results for range queries on B^+-trees.

The results show that when the access patterns are short and highly unstable, the priority-based replacement strategy is more superior. As the references become stable after 40000 page references, the LRU-2 becomes more effective when it detects locality of leaf page references, while the priority-based replacement strategy fills the buffer with index pages closer to the root.

6. CONCLUSION

In this paper, we have proposed a framework for modeling buffer replacement policies. This framework is based on a hierarchical buffer pool model and a priority scheme. Using this framework, a buffer replacement policy is modeled by a collection of types (buffer group types, local selection policy types, and abstract selection policy types) and mapping among the types (among buffer group types, and between buffer group types and selection policy types). This abstraction provides a uniform and generic interface that can serve as a basis for achieving buffer replacement extensibility.

We have illustrated the expressive power and flexibility of this framework by using it to model some popular replacement policies. We still need to establish formally the expressive power of our framework. In particular we are currently trying to establish the correctness of our encoding of the various policies discussed in this paper. Finally, we have implemented an extensible buffer manager that incorporates the framework to demonstrate how quickly and easily a replacement policy can be defined and coded, and how effective and efficient the result can be [2].

Acknowledgement

We would like to thank Chee Yong Chan and Hongjun Lu for their contribution in conceptualizing the concept of extensible buffer management. This work is partially supported by the NUS ARC Research Grant 960694.

7. REFERENCES

[1] B.Lindsay and L.Haas. Extensibility in the starburst experimental database system. In *Proceedings of International Symposium on Database Systems of the 90s*, pages 217–248, 1990.

[2] S. Bressan, C. L. Goh, B. C. Ooi, and K. L. Tan. Implementing the buffer replacement policies with the storm java persistent storage manager (demo paper). In *ACM-SIGMOD*, May 2000.

[3] M.J. Carey, R Jauhari, , and M. Livny. Priority in dbms resource scheduling. In *VLDB*, pages 397–410, 1989.

[4] C.Y. Chan, B.C. Ooi, and H.J. Lu. Extensible buffer management of indexes. In *VLDB*, 1992.

[5] H. Chou and D.J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *VLDB*, pages 127–141, 1985.

[6] H. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug. Design and implementation of the wisconsin storage system. *Software Practice and Experience*, 15(10):943–962, 1985.

[7] B. George and J. R. Haritsa. Secure buffering in firm real-time database systems. In *VLDB*, pages 364–475, 1998.

[8] C. H. Goh, B. C. Ooi, D. Sim, and K.L. Tan. Ghost: Fine granularity buffering of indexes. In *VLDB*, pages 339–350, 1999.

[9] C. L. Goh, M. Anirban, S. Bressan, and B. C. Ooi. Storm: a 100In *OOPSLA workshop on Java and Databases: Persistence Options*, 1999.

[10] R. Jauhari, M.J. Carey, and M.Livny. Priority-hints: An algorithm for priority based buffer management. In *VLDB*, pages 708–721, 1990.

[11] B. T. Jonsson, M. J. Franklin, and D. Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *ACM-SIGMOD*, pages 118–129, June 1998.

[12] M. Lee. Interaction between the query processor and buffer manager of a relational database system. Technical Report RJ6884 (65710), IBM Almaden Research Center, 1989.

[13] S.T. Leutenegger and M. A. Lopez. The effect of buffering on the performance of r-trees. In *ICDE*, pages 164–171, 1998.

[14] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. In *ACM-SIGMOD*, pages 297–306, 1993.

[15] G.M. Sacco and M.Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *VLDB*, pages 257–262, 1982.

[16] The EXODUS Project. Using the exodus storage manager v1.3. Technical report, University of Wisconsin-Madison, 1991.