Unified Fine-Granularity Buffering of Index and Data: Approach and Implementation[1]

Qiang Cao, **Josep Torrellas**, and **H. V. Jagadish**[2]

Department of Computer Science
University of Illinois at Urbana-Champaign, IL 61801
{qiangcao,torrellas}@cs.uiuc.edu jag@eecs.umich.edu
http://iacoma.cs.uiuc.edu

Abstract

Disk I/O is recognized as a major performance bottleneck in many database applications. Consequently, a topic of considerable study in database systems has traditionally been buffer management. Recently, disk pages have been increasing in size, enabling more and more data to fit in a single page. Such a trend suggests that buffering the data at a grain size finer than a page may use memory better. As a result, there has been some interest in fine-granularity buffering.

Past approaches to fine-granularity buffering have proposed buffering either data tuples alone or index entries alone. In this paper, we propose a scheme to support fine-granularity buffering of both index and data entries in a unified manner. The scheme, which we call *Hot-Entry* buffering, can be used in combination with conventional page-level buffering. Through the experimental evaluation of a simple system, we demonstrate the benefits of our scheme over conventional page-level buffering, and over index-only and data-only fine-granularity buffering. In particular, we show that, for a range of parameter values, our buffering scheme speeds-up query execution by 20-60% relative to page-level buffering only, and by 10-20% relative to the best of index-only or data-only fine-granularity buffering.

1 Introduction

Disk I/O is recognized as a major performance bottleneck in many database applications. To reduce the number of I/O accesses, it is necessary to carefully select which disk pages to buffer in memory. Consequently, the topic of buffer management has received considerable attention [6, 7, 9, 13, 14, 15, 17].

Conventional buffer management schemes exploit inter-page access skew to effectively buffer only the most popular pages. However, with the increasing size of disk pages, more and more data today fits on a single disk page. As a result, even within a page, there may be substantial inter-tuple access skew. Consequently, an increasingly attractive approach to obtain higher buffer effectiveness appears to be buffering at a granularity finer than a whole page.

It is useful to draw a parallel with concurrency-control systems. Early database systems implemented page-level locking. A transaction would not only lock the tuple that it was accessing, but all the other tuples in the same page as well. As technology evolved, databases started to implement tuple-level locking. While implementing such fine-granularity locking adds many complications, the performance benefits obtained appear to be worth it in many circumstances. In the same vein, we believe that trends are towards higher benefits for fine-granularity buffering, and expect it to be implemented in mainstream systems in the near future. In fact, Oracle has recently disclosed efforts in this direction [2].

Several fine-granularity buffering schemes have been recently proposed [1, 2, 5, 7, 10, 11, 12, 19]. Each of these systems uses fine-granularity buffering for either index or data, but not for both. In this paper, we present a unified fine-granularity buffering scheme for both index and data. We call our scheme *Hot-Entry* buffering. It is best run in combination with a conventional, page-level buffering scheme, by partitioning the available memory space between the two schemes.

We evaluate our scheme experimentally, using a family of Zipf distributions [8] to generate tuple access patterns. Experimental results using a simple system show that unified fine-granularity buffering of index and data is attractive. For a range of parameter values, our buffering scheme speeds-up query execution by 20-60% relative to page-level buffering only, and by 10-20% relative to the best of index-only or data-only fine-granularity buffering.

The organization of this paper is as follows: Section 2 presents related research; Section 3 introduces our proposed unified fine-granularity buffering scheme; Section 4 discusses related issues in fine-granularity buffering; Section 5 presents our evaluation environment; Section 6 evaluates our scheme; and Section 7 concludes and discusses future work.

2 Background and Related Work

The goal of buffer management is to minimize disk I/O by keeping in a software buffer in memory the pages that are likely to be accessed in the near future. In this buffer, the page management algorithms that work well for the operating system do not necessarily work best for database workloads [17]. In response, there has been much work on buffer management algorithms for databases. Broadly speaking, the algorithms can be classified into workload-aware and statistical.

Workload-aware algorithms use workload information from the query optimizer. For example, Sacco [15] observed that if the buffer allocated to a database system is less than the hot set size of the workload, many misses occur. Based on this observation, he suggested a buffer management algorithm in which each process is allocated a buffer chain equal to its hot set size. Chou and DeWitt [6] studied the page reference patterns of all the common database operations and access methods. Based on the analysis, they introduced the *DBMIN* algorithm, which manages the buffer on a per-table basis. The size and replacement policy for each buffer depends on the table's access method.

Statistical schemes do not require information from the query optimizer. As a result, they are usually simpler. They all modify the basic LRU scheme. For instance, in the 2Q algorithm [9] and the Frequency-Based Replacement algo-

[1]This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP 94-57436, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, and gifts from Intel Corporation.

[2]H. V. Jagadish is now with the University of Michigan, Ann Arbor, Michigan, USA.

rithm [14], there are several queues. Each queue keeps the pages with various "hotness" and has its own replacement policy. By isolating cold from hot pages, the probability of a hot page being replaced by a cold one is greatly reduced. LRU-K [13] is a different generalization of the LRU algorithm: instead of considering the last reference to a page as the only replacement criterion, the history of the last K references is used.

Recently, there has been work on buffering at a finer grain size than a page, which is called fine-granularity buffering. Such work has been mostly done in the context of object-oriented databases. The idea was first proposed in the context of ORION [11, 12] and used in Thor [1] among other systems. Kemper and Kossmann [10] performed a study of dual-buffering strategies, where the data can be cached at the granularity of either page or object. They examined when to copy the objects from the page buffer to the object buffer, and when to relocate the object back to the page buffer. The idea of fine-granularity buffering for objects has taken firm root, at least in the object-oriented, distributed client-server community [5, 19]. One proposal is for an adaptive client-server caching scheme, where the object server decides to send data at the unit of either a page or an object, depending on client and server conditions [19].

In mainstream relational databases, fine-granularity buffering has only been studied recently. We are aware of two pieces of work. The first one is the record caching model of Oracle [2]. In this model, in addition to the regular page cache, there is a separate cache that stores records. Unfortunately, the paper gives very few details on the organization and use of the record cache. One detail given is that the record cache can also be configured to cache B+ tree [16] nodes, but not individual index entries. Goh et al. [7] proposed GHOST, a fine-granularity buffering algorithm for B+ index trees. The work was done in the context of improving B+ tree performance. In their scheme, the buffer space is managed in two parts: a fine-granularity buffer to cache pointers to leaf pages of the index tree, and a conventional page-level LRU buffer to manage data.

In this paper, we present unified fine-granularity buffering. Instead of buffering either index entries as in [7] or tuples (data entries) as in [2], we perform unified fine-granularity buffering for both of them. Consequently, in spirit, but not in implementation or data structure specifics, Oracle's record cache [2] and GHOST [7] can be considered specific instantiations of the general technique introduced here.

3 Hot-Entry Buffering

Data skew exists widely in database applications. We can classify data skew into two categories, namely *data value skew* and *data access skew*. Data value skew refers to the skew in the static data value distribution. Data access skew occurs when certain data records are accessed more frequently than others. Data access skew is what we seek to exploit with the use of fine-granularity buffering.

In our approach, we keep, in a special buffer called *Hot-Entry* buffer, a collection of hot tuples and hot index mappings just as a traditional buffer keeps hot pages (Figure 1). When the processor references a tuple, the corresponding index and data pages are read from disk and copied to the conventional page-level buffer. In addition, however, the index mapping and the tuple referenced are copied to the hot-entry buffer. At any time, when the processor issues a reference, the hot-entry buffer is checked first. In case of a miss, the page-level buffer is checked.

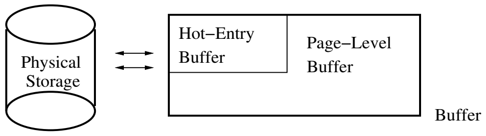


Figure 1: Buffer organization with hot-entry buffering.

In this hot-entry buffer, we keep two types of entries, namely index mappings and actual data tuples. An index mapping is a pair composed of an attribute value and a pointer to the corresponding data tuple, just as it appears at the leaves of a B+ tree [16] index structure. The resulting fine-granularity buffering scheme we call *Unified Index- and Data-Entry* buffering. For comparison purposes, in this paper we also consider fine-granularity buffering of index mappings only (*Index-Entry* buffering) and fine-granularity buffering of data tuples only (*Data-Entry* buffering).

Some of the major challenges in the design of fine-granularity buffering are how to cache the entries in the buffer, how to access the entries efficiently, how to manage the data, and how to minimize the penalty when an entry is not found in the fine-granularity buffer. Given that millions of entries might be stored in the hot-entry buffer, any search must be efficient. In addition, the data structures must be space efficient.

While our buffering scheme is applicable to most index structures, for ease of explanation, we focus on B+ trees. In the following sections, we present the basic organization of index-entry buffering (Section 3.1) and data-entry buffering (Section 3.2), as we build up to the complete system, unified index- and data-entry buffering (Section 3.3).

3.1 Index-Entry Buffering for B+ Trees

Given an index tree built on an attribute, an index-entry buffer caches the mappings that are found at the leaves of the tree. Consequently, if we query the buffer with one instance value of the attribute, the buffer returns a pointer to the data tuple (or tuples) that have that value.

Based on the issues discussed above, we propose to organize the buffer around a two-level hash. The buffer is organized in a table split into a number of *pages* (Figure 2). These pages are software structures that are not necessarily equal in size to virtual memory pages. Given an attribute value, a first-level hash function determines the page of the index-entry table to visit (Figure 2). A second-level hash function, also based on the attribute value, locates the correct linked list of entries to traverse.

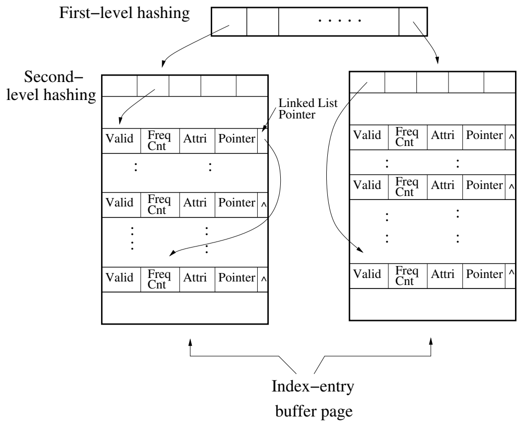


Figure 2: Index-entry buffer.

As shown in Figure 2, each page has a number of hash buckets at the beginning of the page. The rest of the page is devoted to mappings, which we call *Index Entries*, organized in linked lists. Each index entry has five attributes: *Valid*, *Freq Cnt*, *Attri*, *Pointer*, and *Linked List Pointer*. *Freq Cnt* counts the number of references to the entry and its use will be discussed later; *Attri* and *Pointer* are the attribute value and the pointer to the tuple, respectively; *Linked List Pointer* points to the next node in the same linked list. Every valid entry is part of a linked list, whose head is pointed to by a bucket in the second-level hash.

Searches are performed by hashing the attribute value twice and following the linked list. Insertions are handled by hashing the attribute value twice and inserting at the head of the linked list for higher speed. Entry replacements are handled within each page independently. We will consider them in Section 4.1.

For a fixed table size, there is a choice of how many pages to break the table into. If we have few, large pages, the system can utilize them better: there is less chance of having a few unutilized pages and a few overflowing ones. However, large pages require a longer search. Recall that, in the worst case, when all the entries in one page are tied to the same linked list, the whole page may need to be traversed.

3.2 Data-Entry Buffering

A data-entry buffer caches only data tuples. Index pages are buffered as usual in the page-level buffer. When a query is executed, the index tree is traversed as in conventional systems to get pointers to the selected tuples. For each of these pointers, the fine-granularity buffer is accessed and returns the tuple requested. Note that, in data-only buffering, the buffer is accessed using the pointer to a tuple, not the value of an attribute. Only one single tuple can be returned per access.

To facilitate the search of the data-entry buffer, we use a two-level hashing structure (Figure 3) similar to the one for the index-entry buffer. Each of the records in the pages, called *Data Entries*, has five attributes: *Valid*, *Freq Cnt*, *Modified*, *Data Tuple*, and *Linked List Pointer*. The *Valid*, *Freq Cnt*, and *Linked List Pointer* attributes are the same as those in the index-entry table. The *Modified* bit records whether the tuple has been modified since it was cached in the data-entry buffer. Finally, *Data Tuple* is the tuple.

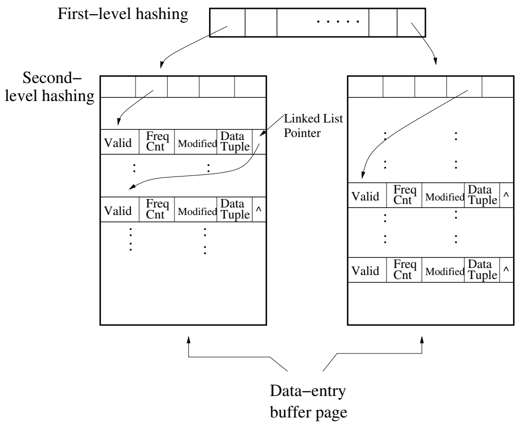


Figure 3: Data-entry buffer.

We address the buffer by hashing the tuple pointer in two steps. All the tuples with pointers that hash to the same location are linked together in a linked list. With these links, we reduce the length on any searches. Search, insertion, and replacement in the data-entry buffer are performed in the same way as in the index-entry buffer.

3.3 Index- and Data-Entry Buffering

Finally, the complete scheme that we propose buffers both index and data entries with fine granularity. The buffer is queried with one attribute value and returns the set of tuples that match such a value.

The buffer proposed contains an index-entry table and a data-entry table, both partitioned into the same number of pages. The organization is a bit different than in the previous two cases. Specifically, the pages of both tables are addressed by hashing the input attribute value with the same function (Figure 4). Index-entry pages are organized with another level of hashing as before, but each individual index entry has an extra field called *Data-Entry Pointer*. This field points to the tuple (or tuples) in the data-entry buffer that have the requested attribute value. These tuples are linked in a list inside a single page of the data-entry table.

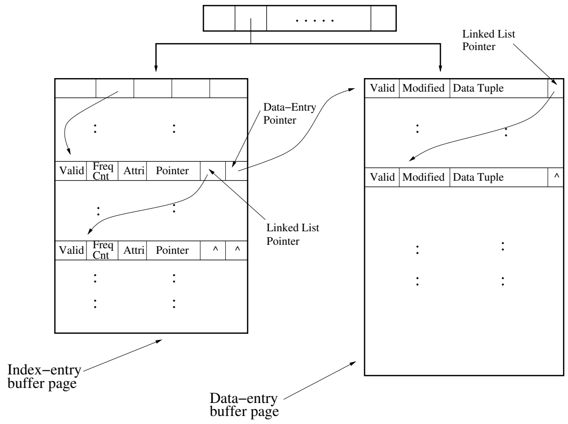


Figure 4: Index- and data-entry buffer.

Data-entry pages do not have any additional level of hashing, since their entries are accessed from the index-entry table. All the tuples that have the same attribute value are linked in a list. For a given attribute value, either all the tuples that have it are in the page or none is. They are brought to the buffer and displaced from it in a group. Each data entry only needs four attributes, namely *Valid*, *Modified*, *Data Tuple*, and *Linked List Pointer*.

When a query for a given attribute value reaches the buffer, the index-entry table is searched, the correct index entry is identified, and the corresponding *Data-Entry Pointer* is checked. If it is not null, it is followed to obtain all the tuples with this attribute value. Otherwise, the conventional page-level buffer or the disk are accessed to get the tuples.

For best performance of the system, the size of a page in the index- and data-entry tables should be related in the following manner. The ratio of index-entry page size to data-entry page size should be equal to the ratio of single index-entry size to the average combined size of the data entries associated with the index entry. If we follow this rule, the average data-entry page will be neither too underused nor too overused. Of course, we may still run out of space and need replacements. In the following section, we address replacements and other issues.

4 Issues in Hot-Entry Buffering

In this section, we briefly examine some issues related to hot-entry buffering: replacements, updates, and multiple indices. We present the high-level ideas only; more detailed algorithms can be found in [3, 4].

4.1 Replacements

Hot-entry replacement, like regular buffer replacement, attempts to keep in the buffer the data that is likely to be used in the near future. However, hot-entry buffering introduces two complications. First, the number of entries that are candidates for buffering is much larger. LRU may not work well when the number of items accessed between two successive accesses to even the hottest items is very large. Secondly, to limit the search necessary to find entries, we keep replacements within each page.

In view of these considerations, we use a frequency-based replacement policy. In index-entry buffering (Section 3.1) and data-entry buffering (Section 3.2), each entry in the buffer has a *Freq Cnt* attribute, which counts the number of accesses to the entry. *Freq Cnt* is initially set to zero and is incremented every time that the entry is accessed. As a way of aging the counters with low overhead, whenever a replacement occurs in a page, we shift right the *Freq Cnt* values of all the entries in the page, therefore dividing them by half. When we need to insert an entry in a page, we select an invalid entry. If there is no invalid entry, we replace the one with the lowest *Freq Cnt*, writing it back if it is modified.

In unified index- and data-entry buffering, we use this same *Freq Cnt*-based replacement scheme for the index-entry table. For the data-entry table, we choose a replacement policy that is closely coupled to the one for the index-entry table. Specifically, when an entry is replaced from the index-entry table, we also displace its corresponding entries from the data-entry table. In addition, when we insert an entry in the index-entry table, we check if there is space in the corresponding page of the data-entry table to buffer the associated data tuples. If there is space, the tuples are brought in and copied to the page; otherwise, the *Data-Entry Pointer* of the newly-inserted index entry is set to *Null*.

When a hit in the index-entry table finds that the *Data-Entry Pointer* is *Null*, we access the requested tuples from outside the hot-entry buffer. Note that, because of the way we dimension the relative sizes of the index-entry and the data-entry tables, this situation should not happen often. However, when it happens, we do not copy the requested tuples into the data-entry table. There are two reasons for it. First, if we did it, we would likely be displacing other useful data from the data-entry table. Secondly, we would need to traverse the page, most likely unsuccessfully, to find free entries.

Overall, coupling the replacements in both tables reduces complexity and avoids thrashing. In addition, we need neither add frequency counters to the data entries nor traverse the data-entry page to find free entries or entries with low *Freq Cnt* values.

4.2 Updates

An important requirement for database structures is that they be able to accommodate incremental changes in response to database updates. Consequently, whenever data is cached in the hot-entry buffer, we must exercise care to ensure that updates do not cause data coherence problems.

There are two write policy schemes, namely write through and write back. While write through does not cause data coherence problems, it is costly because it generates much update traffic. Consequently, we use write back. In this case, tuples in the hot-entry buffer can be updated multiple times and are not written back until they are displaced from the hot-entry buffer.

Note that writing a tuple back from the hot-entry buffer can proceed in two possible ways. If the page where the tuple is allocated is currently resident in the conventional page-level buffer, the tuple can simply be written to it. However, if the page is not in the page-level buffer, the page has to be brought from disk, updated with the new value of the tuple, and then written back to disk.

Fortunately, applications cannot update everything. For example, they cannot directly modify indices. To understand the problem, we classify the updates that applications can perform into the following three types: insertion or deletion of a tuple, update of a non-indexed tuple attribute, and update of an indexed tuple attribute. We consider each type of update in turn.

When a tuple is inserted or deleted, the index tree changes. As a result, the system may reorganize pages of tuples and copy individual tuples to other pages. For these tuples, the index mapping information cached in the index-entry table becomes obsolete.

Consequently, in a tuple insertion or deletion, we first identify the index mappings that are cached in the index-entry table that have become obsolete. For each of these mappings, we invalidate the entry in the index-entry table and follow the entry's *Data-Entry Pointer* link. For each data entry that we find in the linked list, we set it to *Invalid* and, if the *Modified* bit is set, we write the tuple back.

After this, we can proceed with the insertion or deletion. If the operation is a deletion, any information about the tuple is removed from the hot-entry buffer; if the operation is an insertion, the mapping information for the new tuple and the tuple are loaded into the hot-entry buffer. In addition, in either case, the data in the conventional page-level buffer is updated with the change.

The second case is when the application modifies a non-indexed tuple attribute. In this case, we simply set the *Modified* bit of the corresponding data entry. If the data entry is later displaced, we will have to write the tuple back as indicated above.

Finally, the third case is when an indexed tuple attribute is updated. We can consider this case as a delete followed by an insert operation, and follow the procedure outlined for the first case.

4.3 Multiple Indices

A table may be indexed by multiple indices, where each index uses a different attribute. In this case, we reorganize our hot-entry buffer to have, in addition to the data-entry table, as many index-entry tables as there are indices (Figure 5).

To support this organization, we must make two changes to the data-entry table of Figure 4. One change is needed because each tuple in the data-entry table may be linked to as many lists as indices the table has. Consequently, we add more attributes to a data entry: a counter of how many linked lists the data entry is currently linked to (*Link Cnt*), and as many *Linked List Pointer* attributes as indices the table has (Figure 5).

The second change is to eliminate the hashing step for the data entries and, therefore, have only a single data-entry page. The reason is that, if we had several data-entry pages, dif-

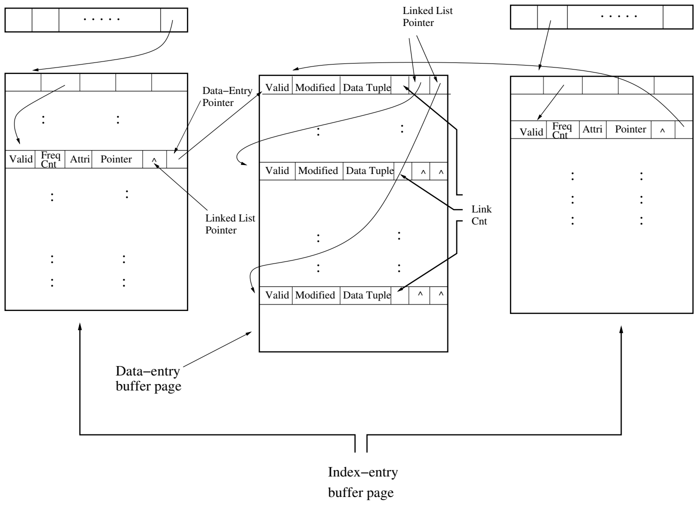


Figure 5: Index- and data-entry buffer with multiple indices.

ferent indices might want to store a given tuple in different pages. Note, however, that having a single, large data-entry page should not slow down our algorithms significantly: our operations typically access the tuples in the data-entry table through the *Data-Entry Pointer* field in the index-entry tables and, therefore, do not need to traverse the data-entry table. The only exception is when we initially look for a free entry to allocate a new data entry. Note, however, that even this operation can be performed without any searching by simply linking all the free data entries in a list.

Having a data entry potentially tied to several linked lists demands extra care when inserting and removing entries. Specifically, suppose that we have inserted a new entry in an index-entry table. Before we can read in and insert its associated data tuples in the data-entry table, we must check whether they are already there. Some or all of them may have already been brought into the data-entry table by other indices.

Consequently, we start by reading the data tuples from disk and identifying what other index entries could have brought them in. Then, the other index-entry tables are checked for these index entries and, if found, their *Data-Entry Pointer* links are followed to see if the desired tuples are already in the data-entry table. If a tuple is already in the table, we link it to the linked list of the new index using the corresponding *Linked List Pointer* and increment *Link Cnt*. Otherwise, the tuple is inserted into the data-entry table and linked to the correct linked list, and *Link Cnt* is set to 1.

Finally, sometimes we replace an entry from an index-entry page and, therefore, must displace the associated tuples from the data-entry table. In this case, we follow the *Data-Entry Pointer* link and, for each entry in the list, decrement *Link Cnt* by one. Entries with *Link Cnt* equal to zero are evicted from the data-entry table. The resulting free entries are linked to the free list.

5 Evaluation Environment

To evaluate our proposed unified index- and data-entry buffering scheme, we compare four different buffer designs. These designs are: page-level buffering only, index-entry plus page-level buffering, data-entry plus page-level buffering, and unified index- and data-entry plus page-level buffering. Each design is evaluated with one of the following page-level buffering algorithms: LRU, LRU-K [13], and 2Q [9]. The LRU-K and 2Q algorithms have been briefly described in Section 2. Since the trends with any of the three algorithms are similar, we only report on experiments using the best-performing algorithm, namely 2Q.

These buffer designs are implemented on a simplified database storage system called *SimpleStorage* that we wrote [3]. *SimpleStorage* includes algorithms to manage a given set of memory pages as a buffer for tables of data and index trees. *SimpleStorage* takes as input a stream of attribute values corresponding to the tuples that we want. Then, it returns a stream of tuples with the attribute values that we requested. Overall, *SimpleStorage* is about 8,000 lines of C++ code.

SimpleStorage exists on top of the Windows NT operating system. The index structures and data tables are stored as regular NT files. While such a storage system is a simplification of what one would find in a real database system, we feel that it is good enough to give us good intuition about the impact of our buffer designs.

We run our experiments on a 400-MHz Pentium II server running Windows NT. The processor has 16-Kbyte L1 D- and L1 I-caches, and a 512-Kbyte L2 unified cache. All caches are 4-way set-associative and write back. A no-contention round trip from the processor to L2 and to memory takes an average of 4 and 129 cycles, respectively. The machine has 4 Gbytes of main memory and 100 Gbytes of disk, and is completely dedicated to our experiments.

In the experiments, we use the following page sizes. The underlying page size used by *SimpleStorage* and transferred to and from the disk is 8 Kbytes. The size of the pages in the index-entry table is also set to 8 Kbytes for ease of management. Using a multiple of the *SimpleStorage* page for the index-entry pages would result in a page so large that index-entry page traversal overheads would be high. Finally, the size of the pages in the data-entry table is set relative to the index-entry pages as indicated in Section 3.3, and then extended up to the next multiple of the *SimpleStorage* page size. Pages whose size is a multiple of the *SimpleStorage* page size are easier to manage. Overall, for the workload that we evaluate, the ratio of data-entry page size to index-entry page size is 4.

The fraction of buffer space used by the hot-entry buffer and the conventional page-level buffer in our baseline design is as follows: for index-entry buffering, 1% is index-entry table while 99% is conventional buffer; for data-entry buffering, 4% is data-entry table and 96% is conventional buffer; for unified index- and data-entry buffering, 1% is index-entry table, 4% is data-entry table, and 95% is conventional buffer. We also perform experiments varying the fraction of the buffer devoted to the hot-entry buffer. However, the ratio between data- and index-entry table sizes is always 4, which is the one required by the workload that we evaluate.

We perform experiments with different buffer sizes. The size of the complete buffer ranges from 1.0% to 40% of the total database size. The database size includes the size of both index and data.

Workloads

Our experiments involve accessing one data table according to a Zipf distribution [8]. We use a Zipf distribution because it is easy to set its skew. The skew of a Zipf distribution is characterized by the Skew Factor θ [8]. We examine three values of θ: 0.00 (the uniform distribution, which has no skew), 0.50, and 0.85 (which is also called the 80/20 skew because 80% of the accesses go to 20% of the tuples).

The table accessed has 1,280,000 tuples, each of which is 128 bytes. Each tuple has a 4-byte attribute with the unique key. One single B+ index tree is built on the data table. Each index entry is composed of a 4-byte key and an 8-byte tuple pointer. Given the 8-Kbyte page size used and the fill factor of the index page used, it can be shown that the depth of the index tree, from root to leaves inclusive, is 3. Overall, the total size of both index and data tables is about 250 Mbytes.

The workload consists of issuing 10,000 queries in sequence without overlap. Each query accesses a single tuple in the table using the index. The tuple chosen is selected using the Zipf distributions. We perform two experiments: a read-only one and a read-write one.

In the read-only experiment, each query simply takes the selected tuple and reads its 128 bytes with 32 loads. The read-write experiment combines these read-only queries with some read-write queries. A read-write query performs the usual 32 loads followed by a word-store to one of the attributes of the tuple. We use a random number generator to decide which of the two types of queries to perform every time, and set the relative frequency of read-only queries to read-write ones to 3 to 1.

Metrics

In the comparison between the different buffer designs, we use four metrics:

1. Miss rate of the hot-entry table. For a given hot-entry table, this metric is number of misses over the number of accesses to the table. This metric shows how effective the hot-entry table is in caching the hot data.

2. Overall miss rate. For a given buffer design, this metric is the ratio between the number of page-sized I/O reads recorded in our server and the number of pages accessed by our queries in the page-level only buffering algorithm. The latter number, for each individual query, is the height of the index tree (3 in our experiments) plus one, to include the data page. Note that we select as denominator in all cases the pages accessed in the page-level only buffering algorithm. This is done to avoid distortions caused by the fact that the designs with a hot-entry buffer access a different number of pages.

3. Execution time. In our experiments, we report the wall-clock execution time of the average query. To generate this number, we simply measure the time taken to execute the 10,000 queries and divide the resulting time by 10,000.

4. Instruction and Memory Cost. We use the Pentium II built-in hardware event counters to measure, for each query, the number of instructions retired, L1 instruction and data misses, and L2 misses.

6 Experimental Evaluation

In our evaluation, we first examine hot-entry buffering under the read-only workload (Section 6.1), then give some rules of thumb for some good design points (Section 6.2), and finally examine hot-entry buffering under the read-write workload (Section 6.3).

6.1 Evaluation under Read-Only Zipf

In our evaluation, we look at four issues: miss rate of the hot-entry table, overall miss rate, execution time, and CPU and memory activity.

Miss Rate of the Hot-Entry Table

To gain insight into how well hot-entry tables are able to intercept memory accesses, we measure their miss rate for different sizes of the tables. We show the results in Figure 6-(a) and (b). Figures 6-(a) and (b) show the miss rate of the index-entry table and data-entry table respectively, as a function of the ratio of sizes between the hot-entry buffer and the database size. Figure 6-(a) corresponds to a system with index-entry plus page-level buffering, while Figure 6-(b) corresponds to one with data-entry plus page-level buffering. The figures show curves for the three skew factors that we measure.

The figures show that, even with very small hot-entry tables, if there is significant data access skew, fine-granularity buffering is effective. For example, consider the curve with skew factor of 0.85, where 80% of the accesses go to 20% of the tuples. In this case, a hot-entry buffer that is only 0.1% of the database size has a 45% miss rate if it caches index entries only (Figure 6-(a)) or a 63% miss rate if it caches data entries only (Figure 6-(b)). If the size of the buffer increases, the miss rate of the index-entry table quickly decreases to less than 20%, while the data-entry table miss rate remains higher. We observe, therefore, that caching index mappings is more effective.

The figures also show that, if there is no access skew, hot-entry buffering is useless. Specifically, for a skew factor of 0.00, the miss rate stays close to 100% in both tables.

Overall Miss Rate and Execution Time

While it is encouraging that hot-entry tables intercept many requests, they use up space otherwise available to the conventional page-level buffer. Consequently, the important metrics to minimize are the overall miss rate of the combined buffers and, especially, the execution time of the query.

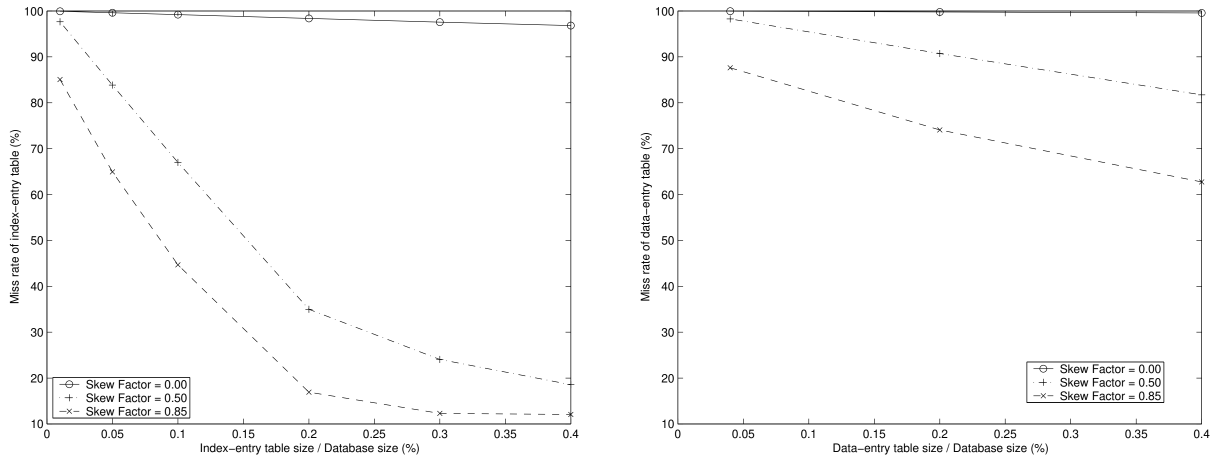
Figures 7-(a), 8-(a), and 9-(a) show the miss rate of the combined buffer as we change its size relative to the database size. The combined buffer includes the hot-entry and the page-level buffers. The figures correspond to skew factors equal to 0.85, 0.50, and 0.00, and show curves for several buffer designs: page-level buffering only (*PAGE-ALG*), index-entry plus page-level buffering (*INDEX-ONLY*), data-entry plus page-level buffering (*DATA-ONLY*), and unified index- and data-entry plus page-level buffering (*INDEX-AND-DATA*).

We see from the figures that, with very small buffer memory, all buffer designs have similar miss rates. In this case, the overall miss rate is around 50%. The reason is because, even with tiny buffers, index pages in the root and first level of the tree are successfully cached and reused.

As the buffer size increases, however, if the accesses have skew, the more advanced designs have lower miss rates. In all cases, the miss rate decreases as we go from *PAGE-ALG* to *DATA-ONLY*, to *INDEX-ONLY*, and *INDEX-AND-DATA*. The differences are often large. Consider, for example, a buffer equal to 10% of the database size, which means that the index-entry table is 0.1% of the database size. For a θ equal to 0.85, the miss rate of *INDEX-AND-DATA* is 20%, while those of *INDEX-ONLY* and *PAGE-ALG* are 25% and 32%, respectively. As the buffer size increases, the relative differences become higher.

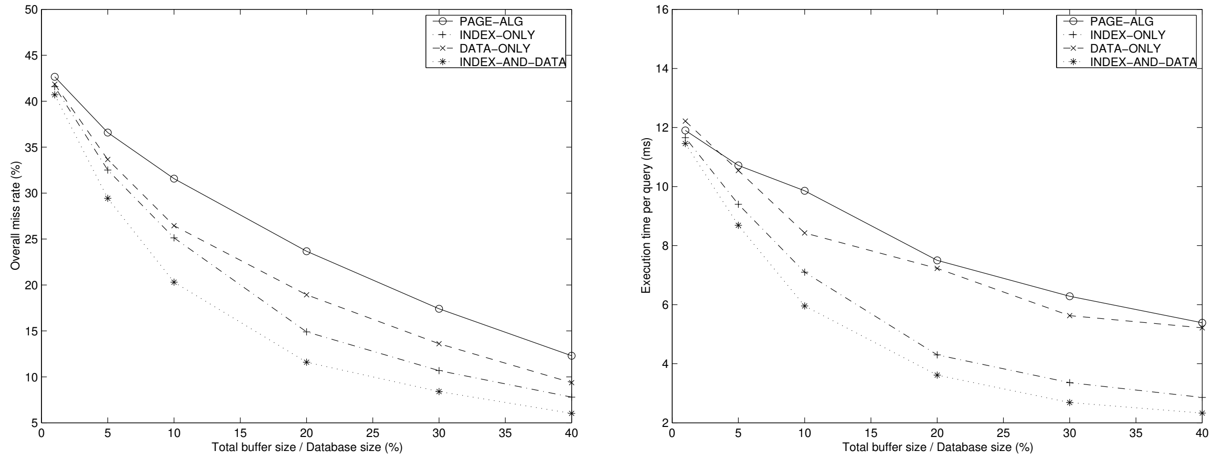
The figures also show that *INDEX-ONLY* has a lower miss rate than *DATA-ONLY*. The reason is that caching index mappings takes less space than caching tuples. Finally, we note that, if there is no access skew as in Figure 9-(a) where θ is 0.00, hot-entry buffering is useless. In fact, it even increases the miss rate because it unnecessarily replicates information in both hot-entry and page-level buffers.

Figures 7-(b), 8-(b), and 9-(b) show how these changes in miss rates translate into changes in the average execution time of a query. These figures correspond to skew factors of 0.85, 0.50, and 0.00, respectively, and again show the effect of changing the ratio between buffer size and database size.



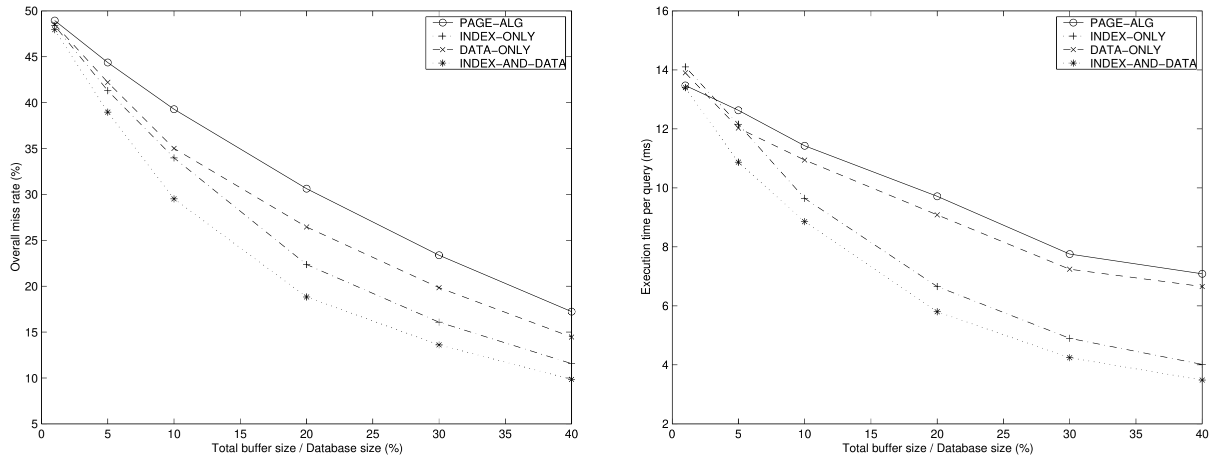
(a) (b)

Figure 6: Miss rate of hot-entry buffering: index-entry table (a) and data-entry table (b).



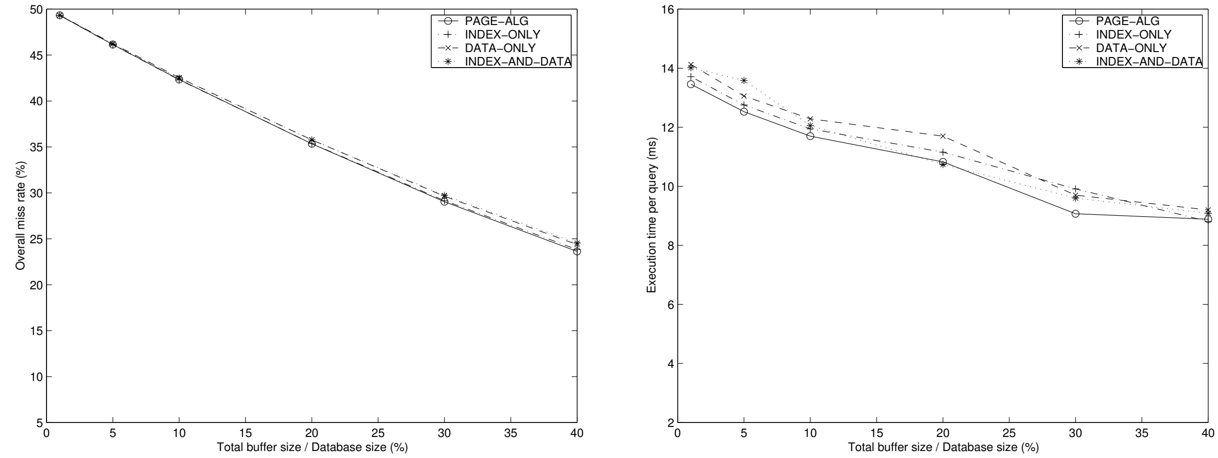
(a) (b)

Figure 7: Overall buffer miss rate (a) and execution time per query (b) for a skew factor equal to 0.85.



(a) (b)

Figure 8: Overall buffer miss rate (a) and execution time per query (b) for a skew factor equal to 0.50.



(a) (b)

Figure 9: Overall buffer miss rate (a) and execution time per query (b) for a skew factor equal to 0.00.

For the skewed distributions, these curves tend to follow the ones for the overall miss rates. This is because, to a first approximation, I/O time constitutes the bulk of the execution time and I/O time is somewhat proportional to the number of I/O reads. As a result, *INDEX-AND-DATA* is the fastest, followed by *INDEX-ONLY*, *DATA-ONLY*, and *PAGE-ALG*. The different buffering schemes often have large speed differences. For instance, in the previous case of a buffer equal to 10% of the database size and a θ equal to 0.85, *INDEX-AND-DATA* is 15% faster than *INDEX-ONLY* and 40% faster than *PAGE-ALG*.

When hot-entry buffering is not useful, namely when there is no skew (Figure 9-(b)) or when the buffer sizes are very small, the extra overheads of hot-entry buffering become noticeable. These overheads are extra instructions to manage the more complicated buffer and possibly extra cache misses. As a result, the conventional *PAGE-ALG* is the fastest (Figure 9-(b)).

CPU and Memory Activity

To better understand these non-I/O overheads, we program the hardware event counters in the processor to measure the instructions executed and the cache misses recorded in the execution of the queries. Figures 10-(a) and (b) show the average number of instructions retired and L1 data misses, respectively, per query execution. Instructions include both user and kernel, and misses include both reads and writes. The data corresponds to a θ equal to 0.85 and, as usual, is shown as a function of the ratio between buffer size and database size.

Figure 10-(a) shows that, in all the buffering schemes, the number of instructions decreases with larger buffer sizes. The reason is that I/O accesses have a significant instruction cost, since much kernel code needs to be executed. As a result, for larger buffers, since there are fewer I/O accesses, fewer instructions execute. The figure also shows that the more advanced buffering schemes execute fewer instructions. Again, this is due to having fewer I/O accesses. In particular, *INDEX-AND-DATA* has the lowest instruction count. The only exception to this rule occurs when the buffer is very small. In this case, the extra management required in the more advanced buffering schemes tends to increase the number of instructions executed.

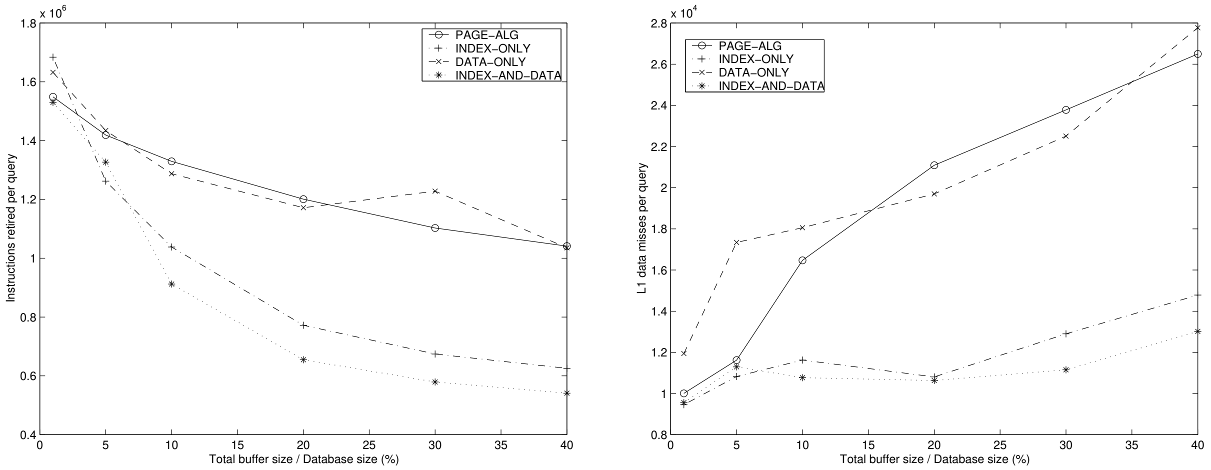
Figure 10-(b) shows that, in general, the number of misses increases as we increase the buffer size. We attribute this effect to the data being now scattered over larger memory sizes and, therefore, causing more cache conflicts. The figure also shows that *INDEX-AND-DATA* and *INDEX-ONLY* have the lowest number of misses. The reason is that, by concentrating the hot entries in a small memory area, we increase the spatial locality of the references, thereby reducing the number of misses. We note, however, that *DATA-ONLY* does not do as well. Its higher number of misses suggests that the data-entry table does not deliver good locality or that there are significant cache conflicts.

We also measured the number of L1 instruction misses and overall L2 misses. The absolute number of misses is very small. For this reason, we do not show them.

With all this information, we can finally estimate the breakdown of the execution time of a query into its components. One component is the CPU stall due to I/O activity (*I/O*). We estimate it with the value of the idle time as measured by Windows NT. A second component of the time is the CPU stall due to cache misses. We estimate this time by multiplying the number of misses recorded by the event counters, by the no-contention latency of a cache miss as given in Section 5. We do this for each type of miss, and get the *L1-I*, *L1-D*, and *L2* times. In reality, we are probably over-estimating these stall times because the latency that we should use is the portion of the miss latency that is not overlapped with other times. Finally, the remaining time we call *Busy* time, and includes instruction execution plus all pipeline hazards not related to the memory system.

Figure 11 shows the contribution of these components to the execution time of the average query for the four buffering schemes that we consider. The figure corresponds to a ratio of total buffer size to database size of 10% and a skew factor of 0.85. The figure shows that *I/O* dominates the execution time by far. Furthermore, the absolute value of *I/O* decreases with the most advanced buffering schemes, which was our goal. It reaches the smallest value for *INDEX-AND-DATA*. Specifically, the *I/O* time in *INDEX-AND-DATA* is only about 55% of that in *PAGE-ALG*. Consequently, our proposed scheme reduces the stall due to I/O significantly.

The cache miss times are very small. The *Busy* time, however, is sizable and varies across schemes. Fortunately, the data shows that fine-granularity buffering often decreases the *Busy* time. This is consistent with the previous observation that advanced buffering reduces instruction overhead.



(a) (b)

Figure 10: Instructions retired (a) and L1 data misses (b) per query for a skew factor equal to 0.85.

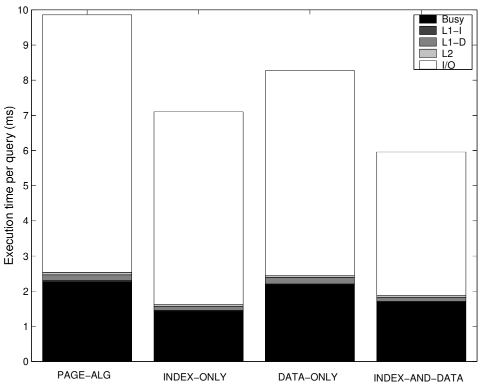


Figure 11: Estimated breakdown of the execution time. The figure corresponds to a ratio of total buffer size to database size of 10% and a skew factor of 0.85.

Total Buffer Size / Database Size (%)	Index-Entry Table Size / Total Buffer Size (%)	Overall Miss Rate (%)			
		PAGE ALG	INDEX ONLY	DATA ONLY	INDEX AND DATA
5	0.5	36.5	34.5	35.0	32.8
	1.0		32.5	33.6	29.4
	2.0		28.5	31.1	23.0
	4.0		22.0	31.0	16.5
10	0.5	31.6	28.4	28.9	25.7
	1.0		25.1	26.4	20.3
	2.0		19.3	26.2	14.8
	4.0		18.3	26.9	14.0
20	0.5	23.7	19.4	19.2	15.4
	1.0		14.9	18.9	11.5
	2.0		14.1	19.4	11.1
	4.0		14.2	20.5	11.5
30	0.5	17.4	12.6	13.3	9.6
	1.0		10.6	13.6	8.3
	2.0		10.7	14.2	8.6
	4.0		10.8	15.4	9.3
40	0.5	12.3	8.1	9.0	6.0
	1.0		7.8	9.3	6.0
	2.0		7.8	10.0	6.4
	4.0		8.0	11.4	7.3

Table 1: Overall buffer miss rates for different buffer parameters. The data corresponds to a θ equal to 0.85. The best designs are marked in bold face.

6.2 Evaluating Various Design Points

The previous section has evaluated the baseline buffer design, where the index-entry table always takes 1% of the whole buffer space. We now evaluate different buffer designs. Table 1 lists the overall buffer miss rate for different designs under a skew factor equal to 0.85. In the table, the ratio between total buffer and database size changes from 5 to 40%, while the fraction of buffer space taken by the index-entry table changes from 0.5 to 4%. Recall from Section 5 that, in designs with data-entry table, the latter is 4 times as large as the index-entry table is (or would be).

The table shows that *INDEX-AND-DATA* has the lowest miss rate in all cases, followed in order by *INDEX-ONLY*, *DATA-ONLY*, and *PAGE-ALG*. Overall, *INDEX-AND-DATA* has a miss rate that is on average 20% and 45% lower than *INDEX-ONLY* and *PAGE-ALG*, respectively.

The table shows that, for small buffer sizes (for example, 5% of the database), hot-entry buffers tend to perform relatively better when they use a relatively larger fraction of the buffer (for example, 4% of the buffer). For large buffer sizes like 40%, the opposite is often true. Although not shown in the table, these same trends occur for access patterns with less skew, although they are weaker.

Based on these and other observations, we offer the following broad recommendations to size hot-entry buffers. In the following, buffer sizes are always relative to the database size, while hot-entry buffer sizes are always relative to the buffer size.

- For large buffers, it is usually best to allocate small hot-entry buffers, while for small buffers, large hot-entry buffers are usually better. A good design for 40% buffers is to use 0.5% hot-entry buffers, while a good one for 5% buffers is to use 4% hot-entry buffers.
- For low-skew access patterns, the hot-entry buffer should be redesigned as follows. It should be smaller when the buffer is small, and it should decrease in size only slowly as the buffer size increases.
- When the skew is unknown, it is best to use small hot-entry buffers to limit the overhead.
- When the buffer is quite small ($\leq$ 0.5%) or quite large ($\geq$ 30%), it is best to use small hot-entry buffers, no larger than 5%.
- For large hot-entry buffers ($\geq$ 5%), especially in large buffers ($>$ 20%), *INDEX-AND-DATA* may have a higher overall buffer miss rate than *INDEX-ONLY* due to data replication.

6.3 Evaluation under Read-Write Zipf

In the presence of read-write queries, the I/O traffic increases. To see how, Figure 12 shows the types of traffic present. When a page is read or written and the buffer misses, the page is read from the disk into the page-level buffer. This is shown as *Read* traffic in the figure. If the access was a write, the update occurs on the page-level buffer. After this, the requested index mapping or data tuple is loaded into the hot-entry buffer. From now on, accesses to the same index mapping or tuple are intercepted by the hot-entry buffer.

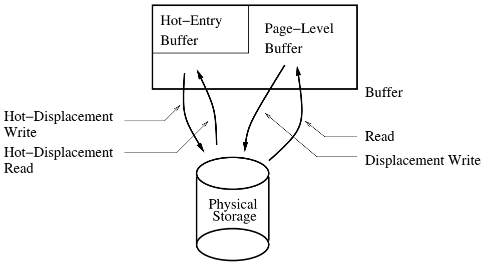


Figure 12: Types of I/O traffic. All I/O operations are page sized.

During execution, a dirty page may be replaced from the page-level buffer, creating what we label *Displacement Write* traffic. A page displaced from the page-level buffer may leave dirty bits of it in the hot-entry buffer. These entries may be eventually displaced from the hot-entry buffer. When they are, if their page is not currently in the page-level buffer, the page is read from disk (*Hot-Displacement Read* traffic), updated in place, and immediately written back to disk (*Hot-Displacement Write* traffic).

It is interesting to compare the resulting I/O traffic to that in the read-only experiments of Section 6.1. To this end, we count the total number of pages of I/O measured in the server and divide the count by the number of pages accessed by the queries in the page-level only buffering algorithm. Recall from Section 5 that, for each individual query, the latter number is the height of the index tree plus 1, which comes up to 4. The resulting ratio, for different relative buffer sizes and skew factors, is shown in Figures 13-(a), 14-(a), and 15-(a). These figures show bars for all the buffer designs, broken down into the different I/O components. These figures are directly comparable to Figures 7-(a), 8-(a), and 9-(a) except that, to show the contribution of each I/O component, we have to reformat them a bit.

The figures show that most of the I/O traffic comes from reading-in pages into the buffer as per processor requests (*Read*). A small fraction of the traffic comes from displacing dirty pages from the page-level buffer (*DispWrite*). The traffic resulting from the displacement of hot entries that are not backed-up in the page-level buffer is negligible (*HotDispRead* and *HotDispWrite*).

We can compare Figures 7-(a), 8-(a), and 9-(a) to Figures 13-(a), 14-(a), and 15-(a). We see that the miss rate in the read-only experiments is quite similar to the rate of *Read* I/O in the read-write experiments. Furthermore, the trends for the different buffering schemes in the two sets of experiments are very similar. Consequently, adding read-write queries has simply added the *Displacement Write* traffic and not changed the relative advantages of *INDEX-AND-DATA*.

We reach a similar conclusion if we examine the average execution time of the queries. Indeed, if we compare the execution time of the read-write queries (Figures 13-(b), 14-(b), and 15-(b)) and the execution time of the read-only ones (Figures 7-(b), 8-(b), and 9-(b)), we see that our conclusions on the relative performance advantages of the advanced buffering schemes do not change. The trends observed as the buffer size increases are the same. In all cases, if there is skew, *INDEX-AND-DATA* is significantly faster than the rest of the schemes.

We have also repeated the experiments of Figure 11 for the read-write environment. Although not shown in the paper, the results are similar. The details are shown in [3].

7 Conclusions and Future Work

Typical database accesses tend to have both inter- and intra-page skew. However, conventional page-level buffering can only exploit inter-page skew. To exploit intra-page skew, we use fine-granularity buffering.

Previous research studied fine-granularity buffering for either indices alone or data alone. In this paper, we introduced unified fine-granularity buffering of both indices and data. We introduced the data structures and, using a simple system, experimentally evaluated the performance of such a technique. Our experimental results showed that, for a range of parameter values, our buffering scheme sped-up query execution by 20-60% relative to page-level buffering only, and by 10-20% relative to the best of index-only or data-only fine-granularity buffering.

Our future work is to evaluate our buffering scheme under realistic data access patterns, like those in popular benchmarks. In addition, we hope to incorporate our algorithms into a complete database system, to demonstrate that it can be done with acceptable impact on code complexity and system performance.

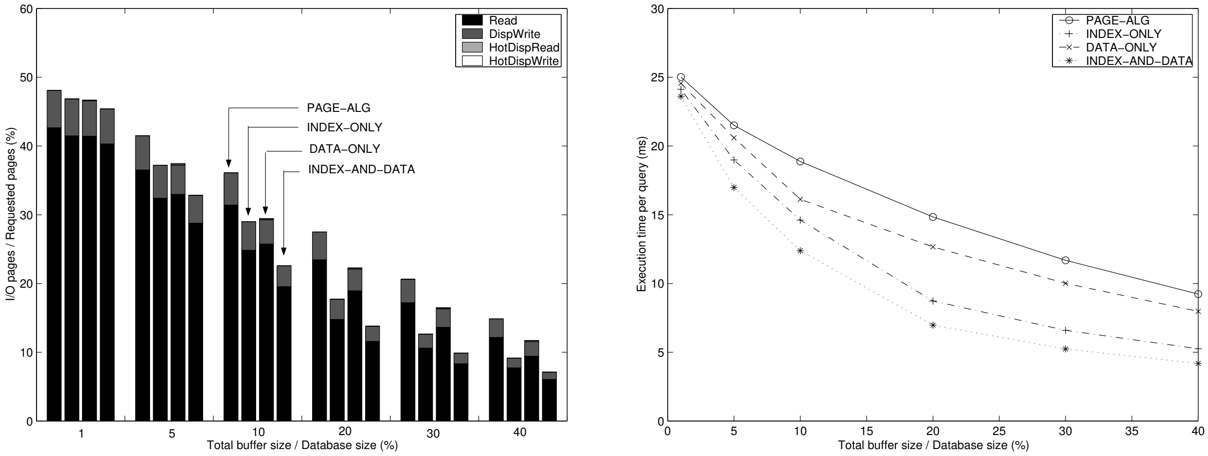
To this end, we have recently started evaluating our buffering scheme under the access patterns of TPC-C [18], an on-line transaction processing benchmark from the Transaction Processing Performance Council. TPC-C models a wholesale supplier that accepts orders from customers, distributes the products, and maintains the stock. TPC-C, like many other workloads, exhibits data access skew. Therefore, it should work well with hot-entry buffering.

We have measured the skew of the accesses to one of the data tables in TPC-C. This table, called *Item*, has 100K 82-byte read-only tuples, and is accessed through one index structure. The access patterns are generated by a non-uniform random number generation function [18].

To see the skew of the accesses to the table, we issue 1 billion accesses to the *Item* table using the non-uniform random number generation function. Figure 16 shows a count of the number of accesses that reach each individual tuple, for the first 10,240 tuples of the table. The pattern shown in the figure repeats every 10,000 tuples. From the figure, we can see that there is dramatic skew in the tuple accesses. Furthermore, since there is a one-to-one mapping between a tuple and the corresponding tuple pointer in a leaf page of the index tree, there is also significant skew in the accesses to the tuple pointers. Consequently, both index- and data-entry buffering should work well.

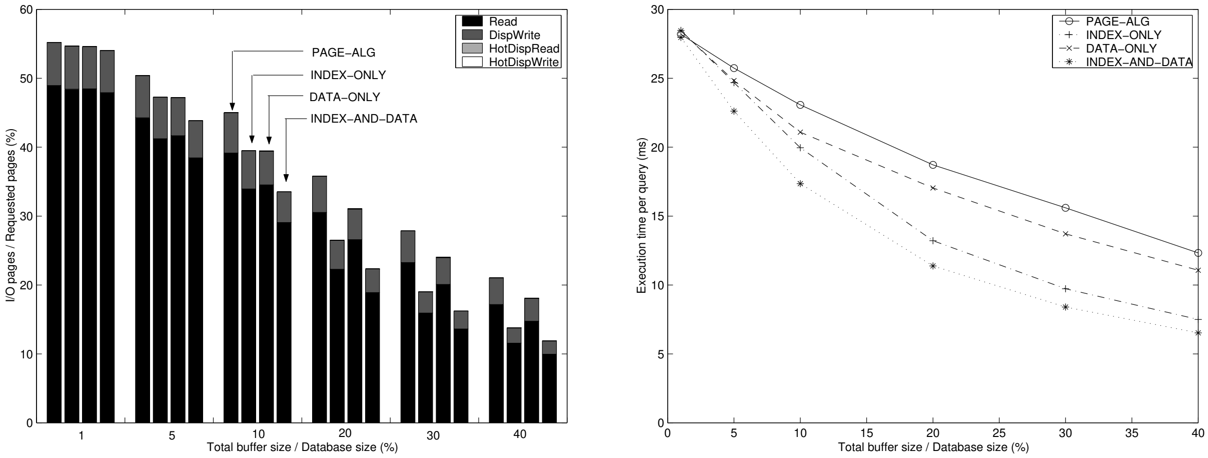
We are in the process of evaluating our algorithms under TPC-C. Initial results are encouraging. For example, Table 2 shows overall buffer miss rates, assuming that we only buffer the *Item* table. We compare all the buffering schemes discussed in this paper. We use different buffer sizes, shown as a fraction of the *Item* table size. For this experiment, the index-entry table is set to 2% of the total buffer size and the data-entry table to twice that.

We can see from the table that the more advanced buffering schemes have a lower miss rate than the page-level algorithm. We are in the process of performing a detailed evaluation of the whole system.



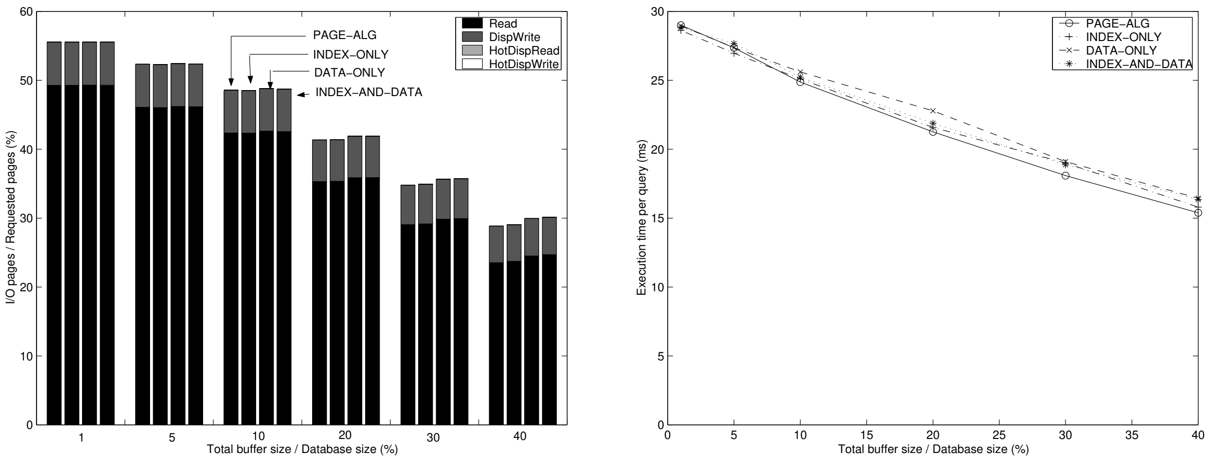
(a) (b)

Figure 13: Number of I/O pages per requested page (a) and execution time per query (b) for a skew factor equal to 0.85.



(a) (b)

Figure 14: Number of I/O pages per requested page (a) and execution time per query (b) for a skew factor equal to 0.50.



(a) (b)

Figure 15: Number of I/O pages per requested page (a) and execution time per query (b) for a skew factor equal to 0.00.

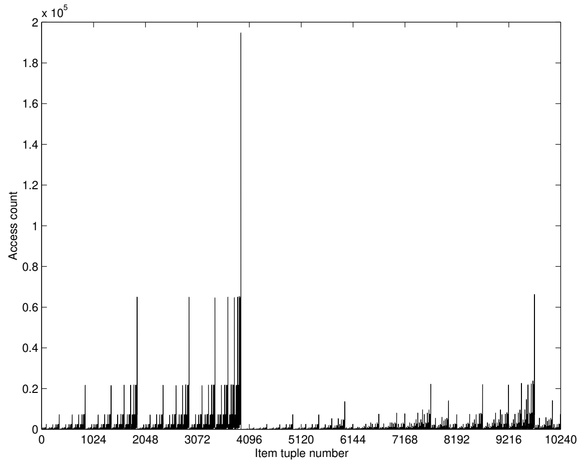


Figure 16: Counting the accesses to each of the first 10,240 tuples in the *Item* table, for a 1-billion access run.

Buffer Size / *Item* Size (%)	Overall Miss Rate (%)			
	PAGE ALG	INDEX ONLY	DATA ONLY	INDEX AND DATA
1	46.8	44.7	46.7	44.4
5	30.8	24.0	29.9	23.7
10	22.1	14.3	21.3	14.2
20	12.9	6.9	12.4	6.6
30	8.0	4.9	7.5	4.9
40	5.0	3.5	4.6	3.7

Table 2: Overall buffer miss rates for the *Item* table in TPC-C.

Acknowledgments

We would like to thank Pedro Trancoso, Josep-Lluis Larriba-Pey, the referees, and the graduate students in the I-ACOMA group for their comments on this work.

References

[1] A. Adya, M. Castro, B. Liskov, U. Maheshwari, and L. Shrira. Fragment Reconstruction: Providing Global Cache Coherence in a Transactional Storage System. In *Proc. of the 17th Int. Conference on Distributed Computing Systems (ICDCS)*, 1997.

[2] R. Anderson, G. Arun, and R. Frank. Oracle Rdb‘s Record Caching Model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, pages 526–527, June 1998.

[3] Q. Cao. Performance Characterization and Buffer Memory Optimization of Databases. Ph.D. Thesis UIUCDCS-R-2000-2164, Department of Computer Science, University of Illinois at Urbana-Champaign, August 2000.

[4] Q. Cao, J. Torrellas, and H. V. Jagadish. Hot-Entry: a Fine-Granularity Disk Buffer Management Scheme. Technical Report UIUCDCS-R-2000-1569, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2000.

[5] J. R. Cheng and A. R. Hurson. On the Performance of Object-Based Buffering. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 30–37, December 1991.

[6] H. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica*, 1(3):311–336, 1986.

[7] C. Goh, B. Ooi, D. Sim, and K. Tan. GHOST: Fine Granularity Buffering of Index. In *Proceedings of the 25th International Conference on Very Large Databases*, September 1999.

[8] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly Generating Billion-Record Synthetic Databases. *SIGMOD Record*, 23(2):243–252, June 1994.

[9] T. Johnson and D. Shasha. 2Q: a Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Databases*, pages 439–450, 1994.

[10] A. Kemper and D. Kossmann. Dual-Buffering Strategies in Object Bases. In *20th International Conference on Very Large Databases*, pages 427–438, September 1994.

[11] W. Kim, N. Ballou, H. Chou, J. F. Garza, D. Woelk, and J. Banerjee. Integrating an Object-Oriented Programming System with a Database System. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, pages 142–152. 1988.

[12] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. In *IEEE Transactions on Knowledge and Data Engineering*, pages 109–124. 1990.

[13] E. O'Neil, P. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *SIGMOD Record*, 22(2):297–306, June 1993.

[14] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. Technical Report, 981, IBM Research Division. T. J. Watson Research Center, 1989.

[15] G. M. Sacco. A Mechanism for Managing the Buffer Pool in a Relational Database System Using the Hot Set Model. Technical report, IBM San Jose Research Laboratory, December 1981. Also published in: Proceedings of the 8th Conference on Very Large Databases, September 1982.

[16] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Publishing Company, Inc., third edition, 1997.

[17] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[18] *TPC Benchmark C Standard Specification, Revision 3.4*, 1998.

[19] K. Voruganti, M. T. Ozsu, and R. C. Unrau. An Adaptive Hybrid Server Architecture for Client Caching Object DBMSs. In *25th International Conference on Very Large Databases*, September 1999.