future internet

MDPI

Article

Adaptive Multi-Grained Buffer Management for Database Systems

Xiaoliang Wang [1] and Peiquan Jin [1,2,*]

[1] School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China; wxl147@mail.ustc.edu.cn
[2] Key Laboratory of Electromagnetic Space Information, China Academy of Sciences, Hefei 230027, China
* Correspondence: jpq@ustc.edu.cn

Abstract: The traditional page-grained buffer manager in database systems has a low hit ratio when only a few tuples within a page are frequently accessed. To handle this issue, this paper proposes a new buffering scheme called the AMG-Buffer (Adaptive Multi-Grained Buffer). AMG-Buffer proposes to use two page buffers and a tuple buffer to organize the whole buffer. In this way, the AMG-Buffer can hold more hot tuples than a single page-grained buffer. Further, we notice that the tuple buffer may cause additional read I/Os when writing dirty tuples into disks. Thus, we introduce a new metric named *clustering rate* to quantify the hot-tuple rate in a page. The use of the tuple buffer is determined by the clustering rate, allowing the AMG-Buffer to adapt to different workloads. We conduct experiments on various workloads to compare the AMG-Buffer with several existing schemes, including LRU, LIRS, CFLRU, CFDC, and MG-Buffer. The results show that AMG-Buffer can significantly improve the hit ratio and reduce I/Os compared to its competitors. Moreover, the AMG-Buffer achieves the best performance on a dynamic workload as well as on a large data set, suggesting its adaptivity and scalability to changing workloads.

Keywords: buffer management; replacement algorithm; multi-grained buffer; adaptive buffer

Citation: Wang, X.; Jin, P. Adaptive Multi-Grained Buffer Management for Database Systems. *Future Internet* **2021**, *13*, 303. https://doi.org/10.3390/fi13120303

Academic Editor: Devis Bianchini

Received: 1 November 2021
Accepted: 25 November 2021
Published: 26 November 2021

1. Introduction

Buffer manager is a key component in DBMSs (Database Management Systems) [1–3]. A conventional buffer manager organizes the buffer into a collection of fixed-sized pages. Each page in the buffer maintains a flag indicating whether the page is clean or dirty. If the page is updated in the buffer, it becomes a dirty page; otherwise, it is a clean page. When reading a page into the buffer but the buffer is full, the buffer manager has to select a page in the buffer as the victim page and evict it out of the buffer to make free space. In general, the efficiency of a buffer manager is highly impacted by the buffer replacement policy [1].

So far, many buffer replacement policies have been proposed, e.g., LRU (Least Recently Used) [1] and LIRS (Low Inter-reference Recency Set) [2] for hard disks, as well as CFLRU (Clean-First LRU) [4] and CFDC (Clean First Dirty Clustering) [5] for SSDs (Solid State Drives). However, we notice that a user request to data is typically issued with tuple granularity. For example, a SELECT query in relational databases aims to retrieve a set of tuples but not a bunch of pages. A page size is commonly much larger than a tuple size, e.g., the default page size of MySQL is 16 KB but a tuple usually has tens or hundreds of bytes. If a user query requests only a few tuples within a page, buffering the entire page will make memory inefficient [6–8]. As a result, the conventional page-grained buffer retains both hot tuples (frequently requested) and cold tuples in the memory, but the cold tuples do not help improve query performance, which will decrease the hit ratio and worsen the time performance.

To improve the traditional page-grained buffer manager, this paper proposes an adaptive multi-grained buffer management scheme called AMG-Buffer (Adaptive Multi-Grained Buffer). The key idea of AMG-Buffer is to organize the buffer into a page buffer and a tuple buffer and adaptively choose the page buffer or the tuple buffer according to a

new metric called *clustering rate*, which quantifies the hot-tuple rate within a page. Briefly, we make the following contributions in this paper.

- We propose AMG-Buffer to use two page buffers and a tuple buffer for organizing the buffer pool. AMG-Buffer can adapt to access patterns and choose either the page buffering scheme or the tuple-based scheme according to the user request.
- We present a detailed architecture and detailed algorithms for the AMG-Buffer, including the algorithms of reading a tuple and writing a tuple. In addition, we propose an efficient algorithm to adjust the size of the tuple buffer, which can improve the space efficiency of the buffer and make the AMG-Buffer adapt to dynamic workloads.
- We conduct experiments over various workloads to evaluate the performance of the AMG-Buffer. We compare our proposal with five existing buffer management schemes. The results show that the AMG-Buffer can significantly improve the hit ratio and reduce the I/Os. Moreover, the experiment on a dynamic workload and a large dataset suggests the adaptivity and scalability of AMG-Buffer.

The remainder of the paper is structured as follows. Section 2 briefly reviews the related work. Section 3 presents the details of AMG-Buffer. Section 4 reports the detailed experimental results, and finally, in Section 5 we conclude the entire paper and discuss future work.

2. Related Work

2.1. Traditional Buffer Management

Buffer management is a fundamental component in traditional disk-based database systems [1–3]. The database buffer is organized as a buffer tool, an area of main memory used by DBMSs to maintain recently used data. The buffer pool is divided into many fixed-sized frames. In general, each frame has the same size as a disk page. When a user query requests a page, the DBMS returns the page address in the buffer pool if the requested page is in the buffer pool; otherwise, the buffer manager calls the storage manager to load the requested page from the disk. Due to the limited memory space, when the buffer pool is full, the buffer manager must perform a page replacement policy to select a page, namely, victim page, which will be evicted out of the buffer [9,10]. The design of replacement strategies has a significant impact on the performance of the buffer manager [11,12].

In the past two decades, many well-known buffering policies have been proposed, e.g., LRU and LIRS [2]. Most of these algorithms have been well explained in textbooks; thus, in this paper, we will not give many details about the existing buffering schemes. The literature [1] presents a good survey on traditional buffer management policies. Although in recent years, SSDs have accomplished notable improvements over the years, HDDs are still widely used as enterprise storage [13,14].

Recently, we proposed MG-Buffer [7] that adds a new tuple buffer for the traditional page buffer, which inspires this study. MG-Buffer always moves the hot tuples in the page buffer to the tuple buffer when a page is evicted out of the page buffer. However, MG-Buffer has two limitations. First, MG-Buffer always manages hot data in a tuple granularity, which loses the benefit of spatial locality, because the hot tuples are from different pages. Second, writing dirty pages may incur additional read I/Os. If a dirty page evicted from the page buffer only has few tuples modified, MG-Buffer will always move these tuples to the tuple buffer. Then, when a dirty tuple needs to be written to disks, it has to reload the original page and synchronize the dirty tuple before flushing. Our design improves MG-Buffer by proposing a new adaptive scheme to make MG-Buffer suitable for various access patterns, and our experiments will demonstrate the superiority of our design over MG-Buffer.

2.2. Buffer Management for SSD-Based Database Systems

Nowadays, SSDs have been widely adopted in enterprise computer systems. Recently, due real-time applications' need for in-memory databases, the volatility and low density of DRAM have made it challenging to keep all data in memory [15,16]. SSDs as block devices have the same page-wise access granularity as magnetic hard disks (HDDs). Thus, traditional DBMSs are easy to integrate SSDs into their current storage system. Besides,

SSDs have several advantages over HDDs, such as high read performance and low power consumption. As a result, it has been demonstrated that using SSDs is an efficient way to improve the performance of database systems [17–19].

However, SSDs, due to their unique internal materials, suffer from some limitations [20]. First, they have limited writing endurance, i.e., an SSD can only undergo a limited number of write operations. Second, the read latency of SSDs is far lower than the write latency, which is usually called the read/write asymmetry of SSDs. Third, the read/write performance of SSDs is much worse than that of DRAM, indicating that we still need to pay much attention to the use of DRAM in SSD-based database systems.

Traditional page replacement strategies utilize the temporal locality of page requests to reduce many I/Os. Thus, the first goal of these strategies is to minimize the miss rate. However, minimizing the miss rate does not lead to better I/O performance on SSDs due to its read–write asymmetry [21]. Considering the unique characteristics of SSDs, SSD-aware buffer managers usually prioritize choosing clean pages as victims over dirty pages [22,23].

CFLRU [4] is modified from the LRU algorithm, considering the characteristics of SSD. CFLRU divides the LRU list into two parts: working region and clean-first region. The working region consists of the most recently used pages and is placed at the head of the LRU list. The clean-first region consists of candidates for eviction and is placed at the tail of the LRU list. To reduce write I/Os, CFLRU selects a clean page to evict in the clean-first region. Only when there is no clean page in this region, a dirty page at the end of the LRU list is evicted. Additionally, the size of the working region is determined by a parameter w, called window size. By delaying write dirty pages, CFLRU can reduce write I/Os and improve performance.

CFDC [5] is an enhanced version of CFLRU. CFDC further divides the clean-first region into two parts: an LRU list of clean pages and a priority queue of clusters. Clean pages in the clean-first region are the first choice to be selected for eviction. If there is no clean page available, a cluster having the lowest priority is selected, and the oldest unfixed pages in the cluster are selected as a victim. This design has two benefits. First, it can avoid scanning extra dirty pages in the clean-first region. Meanwhile, CFDC clusters modified buffer pages for the better spatial locality of page flushes.

However, the algorithms mentioned above have an ad-hoc way of determining the victim based on a heuristic. Additionally, they fail to adapt to various degrees of asymmetry and workload characteristics fully. To overcome this problem, FD-buffer was proposed. FD-buffer [24] separates clean and dirty pages into two pools and uses an independent management policy for each pool. FD-buffer can automatically adjust the two pools' size ratios based on the read-write asymmetry and the run-time workload. However, FD-Buffer needs the prior knowledge of the characteristics of storage devices.

In summary, most modern SSD-aware buffer management strategies only focus on the design of different page replacement strategies that are aware of the characteristics of SSDs [16,23,25]. They continue the page management scheme and likewise suffer from low memory utilization, as we discussed before. Instead, our work focuses on the data management strategy of the buffer . Thus, the replacement strategies in our AMG-Buffer actually can be replaced with any other modern ones (e.g., LIRS).

2.3. Key-Value Cache Management

Key-value caches are often used to decrease data latency, increase throughput, and ease the load off back-end systems. For example, Memcached is a general-purpose distributed memory caching system. It is used to speed up dynamic database-driven websites by caching data and objects in DRAM to reduce the number of times an external data source must be read. Compared with Memcached, Redis supports different kinds of abstract data structures, such as strings, lists, maps, and sets. In general, popular key-value caches use LRU or variants of LRU as their eviction policy. By deploying a key-value cache in front of a disk-based database, hot tuples can be extracted from the disk-based database and managed by the separate cache system [12]. Thus, hot tuples rather than hot pages are cached in memory.

However, for two-layer architectures, there are two problems [26–28]. First, hot tuples may reside both in the key-value cache and in the buffer pool, which leads to wasting precious memory resources. Second, databases and key-value caches are two separate independent systems. Thus, a synchronization mechanism is necessary to ensure data consistency. For example, when a tuple is modified, the update is sent to the back-end database. Now, the states of the tuple in the database and key-value cache are different. If the application needs up-to-date tuples, the application must update the object in the cache.

In summary, key-value caches manager data in a fine-grained manner. However, it suffers from data-redundant and complex synchronous mechanisms. Instead of the two-layer architecture, we embed the fine-grained data management into the buffer management to benefit from their respective advantages.

3. Adaptive Multi-Grained Buffer Management

3.1. Architecture of AMG-Buffer

Traditional database buffer manager uses fixed-sized pages to organize the buffer pool, e.g., the buffer pages in MySQL have 16 KB. However, a user query focuses on tuples (records) rather than pages. For example, an SQL query "SELECT * FROM students WHERE age = 20" aims to return the student tuples with an age of 20. Moreover, as a page size is typically larger than a tuple size, the traditional page-grained buffer management will become inefficient when a user query requests only a few tuples within a page. This is because the page-grained buffer manager retains both hot tuples (frequently requested) and cold tuples in a page. However, cold tuples occupy buffer space but do not help improve query performance, which results in the degrading of the hit ratio of buffer management and ultimately worsens the time performance of buffer management.

To overcome the problem of the page-grained buffer manager, we propose AMG-Buffer to use both page buffers and tuple buffers to organize the buffer, i.e., it can intelligently determine whether we should use a page-grained buffering scheme or a tuple-grained scheme. The idea and contributions of AMG-Buffer can be summarized as follows.

1. AMG-Buffer proposes to use both a page buffer and a tuple buffer to organize the whole buffer space. The hot or dirty tuples in the page buffer will be moved to the tuple buffer if the page in the page buffer is evicted out. Thus, the buffered page can be released to offer more buffer space for subsequent requests. With such a mechanism, AMG-Buffer can hold more hot tuples than the conventional page-based buffer, yielding the increasing of hit ratio and overall performance. Note that the requests to hot tuples will still hit in the tuple buffer.
2. We notice that when the tuple buffer manages hot tuples that come from different pages, it loses the benefit of spatial locality, which worses the efficiency of the memory. Furthermore, writing dirty tuples may incur additional read I/Os. In other words, either the page buffer or the tuple buffer is not efficient for all access patterns. Thus, AMG-Buffer introduces *clustering rate* to quantify the hot-tuple rate on a page.

   Definition 1. *Clustering Rate. The clustering rate of a page refers to the ratio of the hot tuples within the page. A higher clustering rate means the page contains more hot tuples, and the page is more suitable to be managed by a page-grained buffer manager. On the other hand, a lower clustering rate means that a tuple buffer should be more efficient.*

3. We develop algorithms to automatically perform tuple migration in AMG-Buffer according to the clustering rate of buffered pages. We experimentally demonstrate that the proposed AMG-Buffer can outperform both conventional buffering schemes, such as like LRU and LIRS, and SSD-aware buffering policies like CFDC and CFLRU. We also show that AMG-Buffer performs better than the non-adaptive multi-grained buffer manager.

Figure 1 shows the general architecture of AMG-Buffer. It is composed of three sub-buffers, in which F-Buffer and P-Buffer are page buffers and S-Buffer is a tuple buffer. The F-Buffer is used to read pages from the disk. It is a page buffer because pages swapping

between memory and disks must be performed with the page granularity. The P-Buffer is used to cache the pages with a high clustering rate because the tuples inside those pages have similar hotness. Thus, a page-grained buffering scheme is more suitable for the pages in the P-Buffer. The S-Buffer is a tuple buffer consisting of the hot or dirty tuples of the pages evicted out of the F-Buffer.

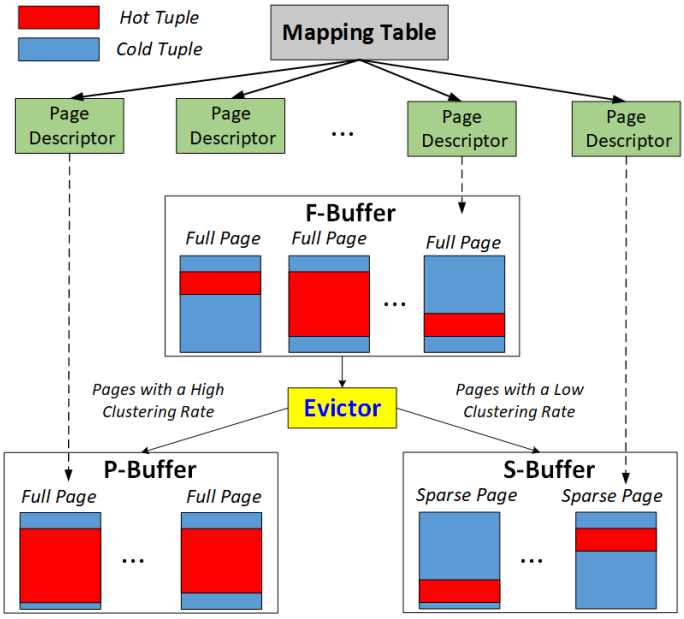


Figure 1. The architecture of AMG-Buffer.

When a page is evicted out of the F-Buffer, it will be moved to either the P-Buffer or the S-Buffer, according to its clustering rate. To be more specific, if the evicted page has a high clustering rate, it will be directly moved into the P-Buffer. Otherwise, if the page has a low clustering rate, we will move the hot or dirty tuples on the page to the S-Buffer. By keeping the hot tuples of the victim page in the S-Buffer, we can release the page space in the F-Buffer but keep a high hit ratio. The reason for migrating dirty tuples is to reduce the write operations to disks. In such a case, the S-Buffer can be regarded as an update buffer for dirty tuples. The dirty tuples in the S-Buffer will be written back to disks in a batch (for example, when the S-Buffer is full).

The buffer replacement process of AMG-Buffer can be described as follows. When a page in the F-Buffer is replaced, AMG-Buffer checks the clustering rate of the page. If the clustering rate is over a pre-defined threshold, we move the page to the P-Buffer and release the page space in the F-Buffer; otherwise, we move the hot or dirty tuples in the evicted page (in the F-Buffer) to the S-Buffer.

The S-Buffer uses a dynamic memory allocation strategy to allocate memory space to tuples. The tuples extracted from the same page are logically grouped as a virtual page named *Sparse Page*, as shown in Figure 1. Moreover, to access the tuples in the S-Buffer, we maintain a page descriptor for each sparse page to store its memory address. Each page in the F-Buffer and the P-Buffer also has a corresponding page descriptor. As Figure 1 shows, we use a mapping table to index all page descriptors. In addition, we use an LRU list to organize all the pages in each sub-buffer.

Figure 2 shows the data structure of the page descriptor in AMG-Buffer. The meanings of elements in the page descriptor are as follows.

(1) *PageID* is the unique identifier of a page.

(2) *BuffID* indicates the buffer frame identifier in the F-Buffer. *BuffID* is only valid when the page is in the F-Buffer.

(3) *VisitBitmap* marks the tuples that have been accessed since the page was read into the F-Buffer.

(4) *DirtyBitmap* is used to mark dirty tuples.
(5) *DirtyFlag* is used to indicate whether the page contains dirty tuples.
(6) *PageType* indicates whether the page is in the F-Buffer or the S-Buffer.
(7) *SlotArray* is a set of <SlotID, Addr>, which stores the pointers pointing to the tuples in the S-Buffer.

Note that the pages in the P-Buffer can be directly migrated from the F-Buffer by changing the *PageType* (see Figure 2) and moving the descriptor from the LRU list of the F-Buffer to the LRU list of the P-Buffer. Thus, we need not perform tuple migrations for those pages.

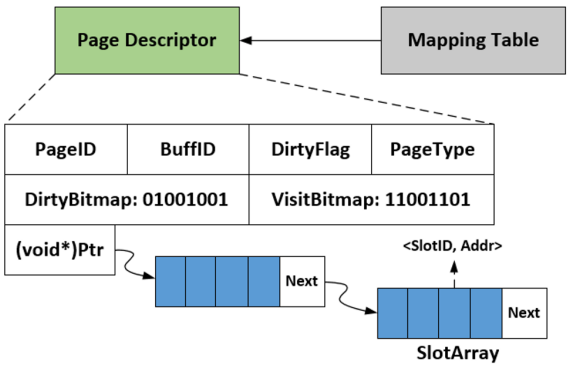


Figure 2. The structure of page descriptor.

3.2. Operations of AMG-Buffer

First, we access the mapping table to determine whether the page descriptor has been loaded in the memory. Next, according to *PageType* in the page descriptor, we can know whether the page is in the F-Buffer, the P-Buffer, or the S-Buffer. If the page is in the F-Buffer or the P-Buffer, we can retrieve the page address based on *BuffID* of the corresponding page descriptor. If the page is in the S-Buffer, we need to check the tuple's existence using *VisitBitmap*.

3.2.1. Read-Tuple Operation

Algorithm 1 shows the read-tuple operation of AMG-Buffer. We first get the page descriptor via the mapping table. If not found, we load the original page from SSDs and place it into the F-Buffer. Otherwise, we determine which sub-buffer the tuple is placed in by checking *PageType*. If the page is in the F-Buffer or the P-Buffer, the *PageType* is *Full_Page* and we read the tuple from the intact page and update *VisitBitmap*. If not, the *PageType* is *Sparse_Page* and we need to check the existence of the tuple through *VisitBitmap*, i.e., if the bit corresponding to the tuple is 1, we can get the memory address of the tuple by looking up in *SlotArray*; if not, we reload the original page into the F-Buffer and read the target tuple.

3.2.2. Tuple Migration

When loading a target page into the F-Buffer, we need to perform a migrating operation to move a victim page into the P-Buffer or the S-Buffer for releasing the space of the F-Buffer. When the clustering rate of the victim page is below the threshold, i.e., the page has a low clustering rate, we perform a tuple migration operation to move the hot or dirty tuples in the victim page to the S-Buffer. In our implementation, we use the widely-used LRU policy to select a victim page in the F-Buffer. LRU is a popular and efficient buffer replacement algorithm that has been adopted by Oracle, MySQL, PostgreSQL, etc. We identify the hot or dirty tuples by checking the *VisitBitmap*. If a tuple is hot (frequently accessed) or dirty, the bit in the *VisitBitmap* that belongs to the page descriptor of the page containing the tuple is set to 1. Thus, by checking the *VisitBitmap*, we can quickly know whether the tuple needs to be migrated or not. If the bit is 1, we copy the tuple to the S-Buffer and add a <*SlotID*, *Addr*> pair into the *SlotArray* in the page descriptor.

Algorithm 1: Read Tuple

```
Input  : PageID page_id, SlodID slot_id
Output : Tuple
desc = FindDescriptor(page_id) ;
if desc!=NULL then
    if desc → page_type is Full_Page then
        page = PagePool.getPage(desc → buff_id);
        t = page → getTuple(slot_id);
        desc → visit_bitmap.set(slot_id);
        return t;
    else
        if desc → vist_bitmap.get(slot_id) is True then
            t = desc → slot_array.getTuple(slot_id);
            return t;
        end
    end
end
/* Load the page into memory                                          */
desc = PagePool.loadPage(page_id);
page = PagePool.getPage(desc → buff_id);
t = page → getTuple(slot_id);
desc → visit_bitmap.set(slot_id);
return t;
```

3.2.3. Merging

The merging operation aims to reduce redundant data in the buffer. This operation is triggered when a page is loaded into the F-Buffer, and there is a sparse page with the same *PageID* in the S-Buffer. During a merging operation, we visit the *DirtyBitmap* in the page descriptor to determine whether a tuple on the page is dirty. If a tuple is dirty, we read the dirty tuple via the *SlotArray* and copy it to the page in the F-Buffer. After that, we delete all tuples pointed by *SlotArray* to release the memory space of the S-Buffer.

3.2.4. Write-Tuple Operation

Algorithm 2 shows the write-tuple operation. This operation is similar to the read-tuple operation. First, we find the page descriptor via the mapping table. If the descriptor is in the F-buffer, we update the page containing the tuple in the F-Buffer directly. We also update both *VisitBitmap* and *DirtyBitmap*. On the other hand, if the descriptor belongs to the S-Buffer but the tuple is not in the S-Buffer (the bit of the tuple in the *VisitBitmap* is 0), we allocate space and write the tuple in the S-Buffer and update the page descriptor, including *VisitBitmap* and *DirtyBitmap*. Note that when the S-Buffer is full, we will evict a sparse page from the S-Buffer via the LRU algorithm. Because a sparse page in the S-Buffer only contains some tuples of a normal page, we need to reload the original page and synchronous the dirty tuples to keep consistency. After that, we write the modified page into SSDs.

3.3. Adjustment of the P-Buffer

One key issue in AMG-Buffer is to determine the appropriate size of the P-Buffer. Allocating a large P-Buffer is not effective if most pages in the F-Buffer have a low clustering rate. On the contrary, if most pages have a high clustering rate, we need to allocate a large P-Buffer.

Algorithm 2: Write Tuple

```
Input :PageID page_id, SlotID slot_id, Tuple t
desc = FindDescriptor(page_id) ;
if desc!=NULL then
    if desc → page_type is Full_Page then
        page = PagePool.getPage(desc → buff_id);
        page → putTuple(slot_id, t);
        desc → visit_bitmap.set(slot_id);
        desc → dirty_bitmap.set(slot_id);
        return;
    else
        if desc → vist_bitmap.get(slot_id) is True then
            desc → slot_array.Update(slot_id, t);
            return;
        else
            addr = TuplePool.alloc(t.size());
            memcpy(addr,t.data(), t.size());
            desc → slot_array.putTuple(slot_id, t);
            desc → visit_bitmap.set(slot_id);
            desc → dirty_bitmap.set(slot_id);
            return;
        end
    end
end
/* Load the page into memory                                          */
desc = PagePool.loadPage(page_id);
page = PagePool.getPage(desc → buff_id);
page → putTuple(slot_id, t);
desc → visit_bitmap.set(slot_id);
desc → dirty_bitmap.set(slot_id);
```

The F-Buffer is used to collect history access information and determine where the page belongs. Thus, the size of the F-Buffer is static in AMG-Buffer. We denote the memory space of both the P-Buffer and the S-Buffer as M. Next, we let W be the time window for adjustment and S be the migration unit for tuning the size ratio of P-Buffer and S-Buffer. M_p denotes the size of the P-Buffer. Then, the proportion of the P-Buffer is M_p/M. The basic idea of the tuning algorithm is that the buffer absorbing new tuples frequently requires more memory space to the cache. Thus, we count the number of pages that enter the P-Buffer, which is denoted as Ep. Similarly, the number of pages that enter the S-Buffer in the time window is Es.

Here, we set two thresholds to trigger the adjustment, namely, T_{min} and $Tmax$. If $|M_p/M - Ep/E|$ is below T_{min}, we do not take any actions to resize the P-Buffer. To avoid frequent adjustments, we extend W to $2W$ (but not exceed W_{max}) and shorten S to $S/2$ (but not less than S_{min}). If $|M_p/M - E_p/E|$ is more than T_{max}, we first resize the P-Buffer, and then we shorten W to $W/2$ (but not less than W_{min}) and extend S to $2S$ (but not more than S_{max}). Then, we resize the P-Buffer as well as the S-Buffer by adding or removing S memory space. Algorithm 3 summarizes the process of adjusting the size ratio of the P-Buffer and the S-Buffer in AMG-Buffer.

Our experimental results show that the proposed adjusting algorithm can make AMG-Buffer adaptive to the change of access patterns. For example, when the workload changes from requesting the pages with a high clustering rate to requesting the pages with a low clustering rate, our algorithm can dynamically adjust the size of the P-Buffer and the S-Buffer to make the whole buffer work efficiently for the current workload. In particular, when requests focus on the pages with a high clustering rate, we will increase the size of the P-Buffer; otherwise, we will increase the size of the S-Buffer.

Algorithm 3: Adjust the P-Buffer

Input : E_p, E_s

$E = E_s + E_s$;
if $|M_p/M - E_p/E| < T_{min}$ **then**
  $S = (S/2 > S_{min})\ ?\ S/2 : S_{min}$;
  $W = (W*2 < W_{max})\ ?\ W*2 : W_{max}$;
else
  if $|M_p/M - E_p/R| > T_{max}$ **then**
    $S = (S*2 < S_{max})\ ?\ S*2 : S_{max}$;
    $W = (W/2 > W_{max})\ ?\ W/2 : W_{min}$;
  end
  if $M_p/M < E_p/E$ **then**
    Migrate S Memory from S-Buffer into P-Buffer;
  else
    Migrate S Memory from P-Buffer into S-Buffer;
  end
end

The current adjusting scheme is actually an empirical algorithm. Although our experiments suggest the effectiveness of the algorithm, it is more attractive to employ a machine-learning algorithm [29,30] to make AMG-Buffer learn the access pattern, which can be further used to adjust the size of the P-Buffer.

3.4. Theoretical Analysis

In this section, we theoretically analyze the efficiency of AMG-Buffer. The parameters of the buffer manager are shown in Table 1, which will be used in the following analysis. First, we assume that the buffer size is 20% of the database size. The access pattern has a locality of 80–20, which is set according to the Pareto Principle, meaning that 80% of the requests access 20% of the tuples.

Since the hot tuples are randomly stored in pages, we can infer that the rate of the hot tuples in one page is about 20%. For the traditional buffer manager, the number of the hot tuples that the buffer can hold is estimated by Equation (1).

$$\frac{M}{P} \cdot \frac{0.2 \cdot P}{T} = \frac{0.2 \cdot M}{T} \tag{1}$$

For AMG-Buffer, we assume that the sizes of the F-Buffer, the P-Buffer, and the S-Buffer are 20%, 40%, and 40% of the memory size, respectively. Additionally, the number of the hot tuples that the F-Buffer can hold is estimated by Equation (2).

$$\frac{M}{P} \cdot \frac{(0.2 + 0.4) \cdot 0.2 \cdot P}{T} = \frac{0.12 \cdot M}{T} \tag{2}$$

When a page is evicted out of the F-Buffer, the probability of the page's hot tuples having been accessed is about 80%. Thus, the number of the hot tuples that the S-Buffer can hold is estimated by Equation (3).

$$\frac{M}{P} \cdot \frac{0.4 \cdot 0.8 \cdot P}{T} = \frac{0.32 \cdot M}{T} \tag{3}$$

As we assume that the access locality is 80–20, we can infer the hot tuples in the database is $0.2 \cdot D$, and 80% of the requests can be answered by these hot tuples. Thus, we can estimate the hit ratio of the traditional buffer manager and AMG-Buffer by Equations (4) and (5), respectively.

$$\frac{\frac{0.2 \cdot M}{T}}{\frac{0.2 \cdot D}{T}} \cdot 0.8 + \frac{\frac{(1-0.2) \cdot M}{T}}{\frac{0.8 \cdot D}{T}} \cdot 0.2 = \frac{M}{D} (M < D) \tag{4}$$

$$\frac{\frac{0.44 \cdot M}{T}}{\frac{0.2 \cdot D}{T}} \cdot 0.8 + \frac{\frac{(1-0.44) \cdot M}{T}}{\frac{0.8 \cdot D}{T}} \cdot 0.2 = \frac{1.9 \cdot M}{D} (M < \frac{5 \cdot D}{11}) \quad (5)$$

We summarize the analysis result in Table 2, which indicates that AMG-Buffer can maintain more hot tuples than the traditional buffer manager. To this end, when processing the same requests, AMG-Buffer can maintain a higher hit ratio than the traditional buffer manager.

Table 1. Parameters of the buffer manager.

Symbol	Description
D	The database size (e.g., 10 GB)
P	The page size (e.g, 4 KB)
T	The tuple size (e.g., 128 B)
M	The whole buffer size (e.g., 128 MB)

Table 2. Theoretical comparison between LRU and AMG-Buffer.

Method	Hot Tuples in the Buffer	Hit Ratio
LRU	$\frac{0.2 \cdot M}{T}$	$\frac{M}{T} (M < T)$
AMG-Buffer	$\frac{(0.12+0.32) \cdot M}{T}$	$\frac{1.9 \cdot M}{D} (M < \frac{5 \cdot D}{11})$

4. Performance Evaluation

In this section, we conduct extensive experiments to evaluate AMG-Buffer using various workloads to demonstrate our proposal's superiority.

4.1. Settings

We conduct all experiments on a PC powered with a processor of Intel Core i5-6200U 2.30 GHz, 8 GB DDR4 DRAM, and an SSD (Samsung 860 EVO) of 500 GB. The operating system is Ubuntu 8.4. We use the direct I/O mode in Linux to eliminate the influence of the file-system cache. We use a dataset of 1 GB and a buffer pool of 128 MB (10% of the data set) by default for all experiments. Additionally, we use the tuple size of 128 B and the page size of 4 KB by default.

Traditional buffer managements are usually evaluated using page-grained requests, such as <*page id, request type*>. In this paper, we need to use tuple-grained requests consisting of a tuple id and a request type. As there is no open-source workload satisfying our experiments' need, we generate tailor-made traces to verify the performance of AMG-Buffer.

In the following experiments, we evaluate the effectiveness of AMG-Buffer by comparing it with various buffer management schemes, including LRU [1], LIRS [2], CFLRU [4], CFDC [5], and MG-Buffer [7]. For emulating various real circumstances, we change workloads with different read/write ratios and hybrid ratios. At last, we evaluate AMG-Buffer under dynamic workloads and a large data set. Note that for SSD-based databases, the overhead of disk I/Os costs is still the main bottleneck. Thus, we focus on I/O costs and ignore the overhead of migration and merging operations.

4.2. Workloads

The target trace file has a sequence of read-write operations on tuples. We divide the storage device pages into two types, namely, hot pages and cold pages. The size of hot pages accounts for 20%, which will absorb 80% of query operations, meaning most queries will concentrate on the hot pages. We define two types of queries, namely, narrow-queries

(*NQs*) and wide-queries (*WQs*). Every *NQ* reads or writes one tuple of the page; every *WQ* reads or writes several tuples of the page. For simplification, we define the ratio of *NQs* among all queries (*NQs*+*WQs*) as *hybrid ratio*. We build various workloads by configuring different hybrid ratios and different read-write ratios, as shown in Table 3. In the trace names, the character *R* represents the read ratio and the symbol *H* represents the hybrid ratio. For example, in the "R20-H50" workload, "R20" means that the read ratio is 20% among all requests and "H50" means that the hybrid ratio is 50% (*NQs* account for 50%).

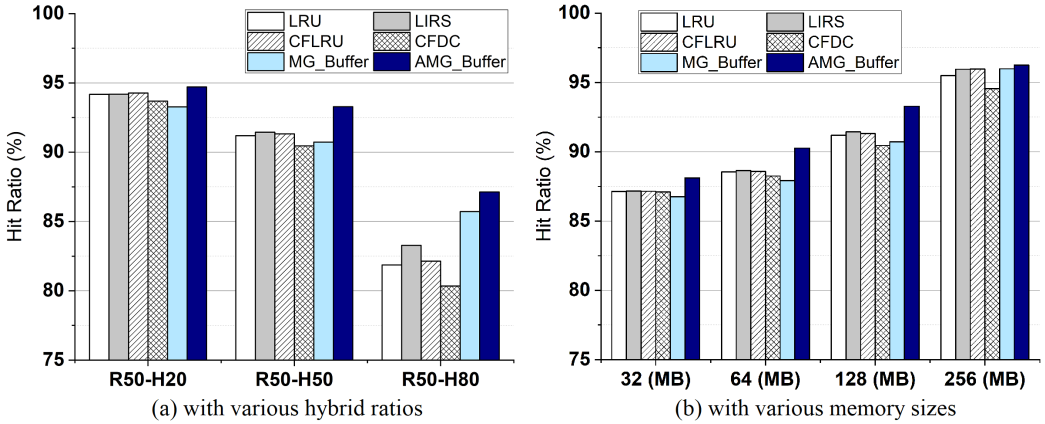
To demonstrate the effectiveness of AMG-Buffer, we conduct experiments under various workloads and settings, including different memory sizes, hybrid ratios, and read/write ratios. In addition, we also evaluate the adaptivity of AMG-Buffer using dynamically changing workloads. Finally, we test the performance of AMG-Buffer using a large data set to see whether AMG-Buffer can adapt to large data sets.

Table 3. Characteristics of the workloads.

Type	Number of Queries	R/W Ratio	Hybrid Ratio
R50-H20	50,000,000	50/50	20/80
R50-H50	50,000,000	50/50	50/50
R50-H80	50,000,000	50/50	80/20
R20-H50	50,000,000	20/80	50/50
R80-H50	50,000,000	80/20	50/50

4.3. Hit Ratio

Figure 3a shows the hit ratios of all compared algorithms, where the hybrid ratio varies from 20% to 80%. Note that, in our workloads, a *WQ* query consists of several tuple accesses within a page. Thus, for a *WQ* query, even if the first tuple misses, the other tuples can hit in memory, which only causes one I/O operation. This leads to high hit ratios for all algorithms. Along with the hybrid ratio increasing, the hit ratios of all algorithms decreases. Compared with traditional ones, due to the tuple-grained buffer, MG-Buffer and AMG-Buffer can saves more hot tuples in memory. Furthermore, AMG-Buffer saves the benefits of spatial locality. Hence, AMG-Buffer can achieve a higher hit ratio than other algorithms, especially for "R50-H80". In Figure 3b, we set the hybrid ratio to 50% (H50) and increase the memory size from 0.25 to 2 times the default size. AMG-Buffer still achieves a slightly higher hit ratio than others.



(a) with various hybrid ratios (b) with various memory sizes

Figure 3. Comparison of hit ratios.

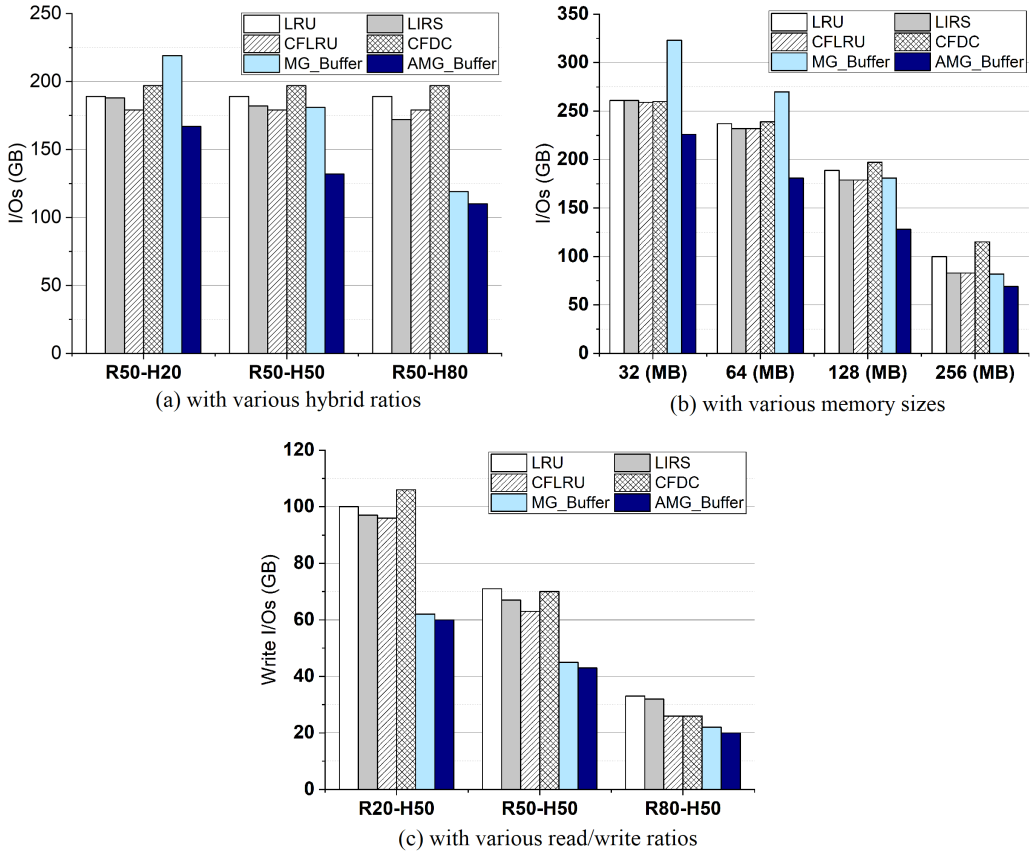


Figure 4. Comparison of I/Os.

4.4. I/Os

Figure 4a shows the I/Os for all algorithms with a varying hybrid ratio. First, MG-Buffer causes more I/Os when meeting many *WQs*, compared with page-grained buffers (e.g., LRU, LIRS, CFLRU, and CFDC). This is because MG-Buffer loses the benefits of spatial locality. Furthermore, MG-Buffer incurs more additional I/Os when writing incomplete dirty pages. Compared to MG-Buffer, AMG-Buffer can distinguish pages by clustering rates to choose the page-grained buffer or tuple-grained buffer, which alleviates these problems. For different hybrid ratios, compared with others, AMG-Buffer can achieve the least I/Os. Figure 4b shows the I/Os under different buffer sizes. When the buffer size is less than 128 MB, MG-Buffer incurs more I/Os than traditional ones and AMG-Buffer reduces I/O costs by up to 13% (32 MB) and 23% (64 MB) than others. When the buffer size is larger than or equals 128 MB, both AMG-Buffer and MG-Buffer have fewer I/Os than LRU, CFLRU and CFDC due to caching more hot tuples, but AMG-Buffer reduces I/Os by 29% (128 MB) and 15% (256 MB) compared to MG-Buffer. Figure 4c shows the write I/Os of all algorithms under different read/write ratios. Both AMG-Buffer and MG-Buffer have fewer write I/Os than LRU, CFLRU, and CFDC and are friendly to SSDs. When we set the read ratio to 20% (R20), 50% (R50), and 80% (R80), AMG-Buffer reduces write I/Os by up to 39%, 37% and 38%, respectively, compared to traditional ones. All results demonstrate that AMG-Buffer can retain the benefit of both the tuple-grained and the page-grained buffer management policy.

4.5. Workload Adaptivity

In this experiment, we aim to verify the adaptivity of AMG-Buffer for dynamic workloads. First, we run all algorithms with a balanced workload R50-H50. As shown in Figure 5a, AMG-Buffer reduces both write and read operations. Compared to traditional algorithms and MG-Buffer, AMG-Buffer reduces up to 29% and 27% overall I/Os, respectively, yielding a high-performance improvement on the balanced workload. We also

report the time performance in Figure 5b, which confirms that AMG-Buffer has the best time performance among all algorithms. In particular, AMG-Buffer can save more dirty tuples in the buffer and has fewer write I/Os than traditional ones. Moreover, AMG-Buffer avoids the additional read I/Os when writing incomplete dirty pages, resulting in higher performance than MG-Buffer.

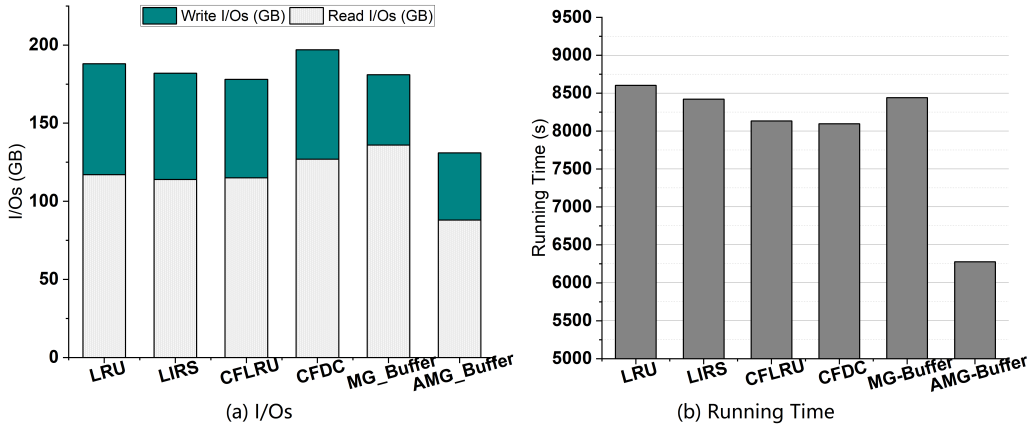


Figure 5. Performance on the balance workload.

We can see that the results in Figure 5a,b are very consistent, indicating that the I/O cost dominates the overall time performance. Hence, in the following experiments, we only report the I/O cost.

To evaluate the performance of the adaptivity of AMG-Buffer and the adjustment scheme of the P-Buffer, we simulate the change of workloads by varying the hybrid ratio (NQ/WQ) to generate a dynamic workload. Dynamic workloads are very common in real applications. For example, user queries in an E-Business platform may be changed by holidays or other events. Thus, a static buffer management policy like LRU is not suitable for dynamic workloads.

Next, we evaluate AMG-Buffer on a dynamic workload. To generate the dynamic work changing with time, we divide the queries in the trace file into epochs, each of which consists of 1 million queries. We prepare 100 epochs in the dynamic workload, i.e., the workload consists of 100 million queries in total. Then, we randomly set the hybrid ratio in each epoch from 10/90 to 90/10. Figure 6 shows the total I/Os for AMG-Buffer and its competitors. Although MG-Buffer has fewer write I/Os than LRU, LIRS, CFLRU, and CFDC, it suffers from high read I/Os. When the requested pages exhibit a high clustering rate, MG-Buffer will cause many additional read I/O operations. On the contrary, AMG-Buffer achieves the best performance on the dynamic workload due to its adaptive use of the P-Buffer and the S-Buffer.

4.6. Performance on a Large Data Set

To see the performance of AMG-Buffer on large data sets, in this experiment, we use a large data set whose size is 16 GB. The buffer size is set to 10% of the data set size. The page size and the tuple size are the same as before. We use the default balanced workload (R50-H50) in this experiment. Figure 7 shows the I/Os comparison between AMG-Buffer and the other algorithms, which presents a similar result as previous experiments. When running on the large data set, AMG-Buffer reduces the I/Os by up to 33% than traditional buffering schemes and MG-Buffer, which shows that AMG-Buffer can adapt to large databases in real scenarios.

Especially, we can see from Figure 7 that AMG-Buffer still keeps the lowest read I/Os and write I/Os among all algorithms, indicating that AMG-Buffer is not only effective at improving the overall performance of buffer management but also friendly to write operations. Such a feature is useful for write-sensitive storage devices like SSDs.

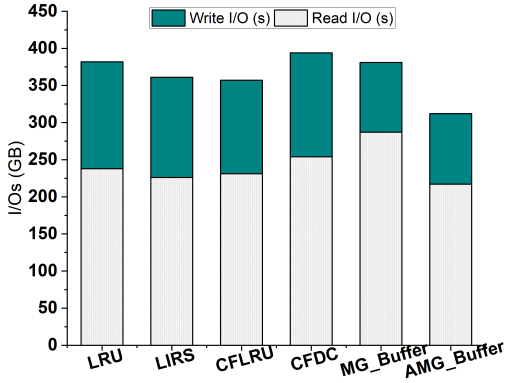


Figure 6. Performance on the dynamic workload.

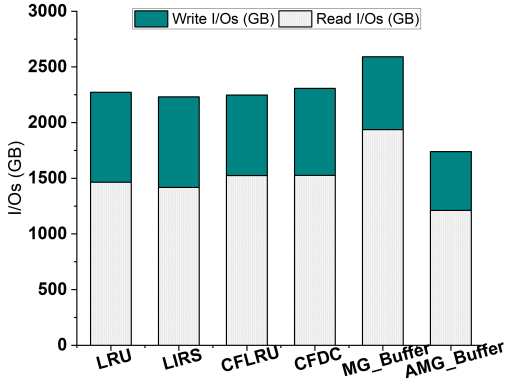


Figure 7. Performance on a large data set.

5. Conclusions

In this paper, we propose an adaptive multi-grained buffer manager called AMG-Buffer for database systems. We first notice that the page-grained buffer management schemes like LRU work poorly when users only request very few tuples within one page. Based on such an observation, we propose to use both page-grained and tuple-grained buffers to offer high efficiency for the buffer manager. In particular, we divide the entire buffer into two page-grained buffers, called F-Buffer and P-Buffer, and one tuple-grained buffer called S-Buffer. A page is first buffered in the F-Buffer. When a page is evicted out of the F-Buffer, we either move the page to the P-Buffer or move the hot or dirty tuples on the page to the tuple-grained S-Buffer. To adaptively determine whether a page-grained movement or a tuple-grained movement should be performed to replace a page from the F-Buffer, we introduce a new concept named clustering rate for every page in the F-Buffer, which is used to adaptively determine the data moving strategy when a page replacement occurs in the F-Buffer. We perform a page-grained migration policy that directly moves the pages with a high clustering rate in the F-Buffer to the P-Buffer. Moreover, we perform a tuple-grained migration policy to move the hot or dirty tuples on the pages with a low clustering rate from the F-Buffer to the S-buffer. We also develop an effective algorithm to adjust the size of the P-Buffer for dynamic workloads. We conduct extensive experiments on various kinds of workloads and settings. The results in terms of hit ratio, I/Os, and run time all suggest the efficiency and effectiveness of AMG-Buffer. As a result, the AMG-Buffer algorithm achieves the best performance in all experiments. Moreover, in the dynamic workload, AMG-Buffer still outperforms the other schemes, showing that AMG-Buffer can adapt to different kinds of workloads.

Although this work focuses on optimizing the performance of the buffer manager on HHD-based databases, some techniques are also friendly to SSD-based databases. In the future, we will improve the AMG-Buffer design to make it applicable to different kinds of storage, such as magnetic disks, SSDs, hybrid storage [31], and persistent memory [32].

Another relevant issue is investigating a learned strategy [33–35] to implement the adaptive mechanism in AMG-Buffer. It has been a hot topic in recent years to use machine learning models for optimizing database components [36]. In future work, it will be possible to consider a machine-learning-based approach to provide a more general solution to selecting the migration granularity for page replacements.

Author Contributions: Conceptualization, X.W. and P.J.; methodology, X.W.; software, X.W.; validation, X.W. and P.J.; formal analysis, X.W.; investigation, X.W.; data curation, X.W.; writing—original draft preparation, X.W.; writing—review and editing, P.J.; supervision, P.J.; funding acquisition, P.J. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Science Foundation of China (No. 62072419).

Data Availability Statement: Not applicable, the study does not report any data.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Effelsberg, W.; Härder, T. Principles of Database Buffer Management. *ACM Trans. Database Syst.* **1984**, *9*, 560–595. [CrossRef]
2. Jiang, S.; Zhang, X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Perform. Eval. Rev.* **2002**, *30*, 31–42. [CrossRef]
3. Johnson, T.; Shasha, D.E. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In Proceedings of the Twentieth International Conference on Very Large Databases, Santiago, Chile, 12–15 September 1994; pp. 439–450.
4. Park, S.; Jung, D.; Kang, J.; Kim, J.; Lee, J. CFLRU: A replacement algorithm for flash memory. In Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Seoul, Korea, 22–25 October 2006; pp. 234–241.
5. Ou, Y.; Härder, T.; Jin, P. CFDC: A Flash-Aware Buffer Management Algorithm for Database Systems. In *East European Conference on Advances in Databases and Information Systems*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 435–449.
6. Meng, Q.; Zhou, X.; Wang, S.; Huang, H.; Liu, X. A Twin-Buffer Scheme for High-Throughput Logging. In *International Conference on Database Systems for Advanced Applications*; Springer: Cham, Switzerland, 2018; pp. 725–737.
7. Wang, X.; Jin, P.; Liu, R.; Zhang, Z.; Wan, S.; Hua, B. MG-Buffer: A Read/Write-Optimized Multi-Grained Buffer Management Scheme for Database Systems. In Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), Zhangjiajie, China, 10–12 August 2019; pp. 1212–1219.
8. Tan, Y.; Wang, B.; Yan, Z.; Srisa-an, W.; Chen, X.; Liu, D. APMigration: Improving Performance of Hybrid Memory Performance via An Adaptive Page Migration Method. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 266–278. [CrossRef]
9. Ding, X.; Shan, J.; Jiang, S. A General Approach to Scalable Buffer Pool Management. *IEEE Trans. Parallel Distrib. Syst.* **2016**, *27*, 2182–2195. [CrossRef]
10. Tan, J.; Zhang, T.; Li, F.; Chen, J.; Zheng, Q.; Zhang, P.; Qiao, H.; Shi, Y.; Cao, W.; Zhang, R. iBTune: Individualized Buffer Tuning for Large-scale Cloud Databases. *Proc. VLDB Endow.* **2019**, *12*, 1221–1234. [CrossRef]
11. Jiang, Z.; Zhang, Y.; Wang, J.; Xing, C. A Cost-aware Buffer Management Policy for Flash-based Storage Devices. In *International Conference on Database Systems for Advanced Applications*; Springer: Cham, Switzerland, 2015; pp. 175–190.
12. Kallman, R.; Kimura, H.; Natkins, J.; Pavlo, A.; Rasin, A.; Zdonik, S.B.; Jones, E.P.C.; Madden, S.; Stonebraker, M.; Zhang, Y.; et al. H-store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* **2008**, *1*, 1496–1499. [CrossRef]
13. Leis, V.; Haubenschild, M.; Kemper, A.; Neumann, T. LeanStore: In-Memory Data Management beyond Main Memory. In Proceedings of the 2018 IEEE 34th International Conference on Data Engineering (ICDE), Paris, France, 16–19 April 2018; pp. 185–196.
14. Neumann, T.; Freitag, M.J. Umbra: A Disk-Based System with In-Memory Performance. In Proceedings of the CIDR, 2020. Available online: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf (accessed on 24 November 2021).
15. Diaconu, C.; Freedman, C.; Ismert, E.; Larson, P.; Mittal, P.; Stonecipher, R.; Verma, N.; Zwilling, M. Hekaton: SQL server's memory-optimized OLTP engine. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, New York, NY, USA, 22–27 June 2013; pp. 1243–1254.
16. Färber, F.; Cha, S.K.; Primsch, J.; Bornhövd, C.; Sigg, S.; Lehner, W. SAP HANA database: Data management for modern business applications. *SIGMOD Rec.* **2011**, *40*, 45–51. [CrossRef]
17. Jin, P.; Yang, C.; Wang, X.; Yue, L.; Zhang, D. SAL-Hashing: A Self-Adaptive Linear Hashing Index for SSDs. *IEEE Trans. Knowl. Data Eng.* **2020**, *32*, 519–532. [CrossRef]
18. Yang, C.; Jin, P.; Yue, L.; Yang, P. Efficient Buffer Management for Tree Indexes on Solid State Drives. *Int. J. Parallel Program.* **2016**, *44*, 5–25. [CrossRef]

19. Jin, P.; Yang, C.; Jensen, C.S.; Yang, P.; Yue, L. Read/write-optimized tree indexing for solid-state drives. *VLDB J.* **2016**, *25*, 695–717. [CrossRef]
20. Zhao, H.; Jin, P.; Yang, P.; Yue, L. BPCLC: An Efficient Write Buffer Management Scheme for Flash-Based Solid State Disks. *Int. J. Digit. Content Technol. Its Appl.* **2010**, *4*, 123–133.
21. Wang, S.; Lu, Z.; Cao, Q.; Jiang, H.; Yao, J.; Dong, Y.; Yang, P. BCW: Buffer-Controlled Writes to HDDs for SSD-HDD Hybrid Storage Server. In Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20), Santa Clara, CA, USA, 24–27 February 2020; pp. 253–266.
22. Choi, J.; Kim, K.M.; Kwak, J.W. WPA: Write Pattern Aware Hybrid Disk Buffer Management for Improving Lifespan of NAND Flash Memory. *IEEE Trans. Consum. Electron.* **2020**, *66*, 193–202. [CrossRef]
23. Li, Z.; Jin, P.; Su, X.; Cui, K.; Yue, L. CCF-LRU: A new buffer replacement algorithm for flash memory. *IEEE Trans. Consum. Electron.* **2009**, *55*, 1351–1359. [CrossRef]
24. On, S.T.; Gao, S.; He, B.; Wu, M.; Luo, Q.; Xu, J. FD-Buffer: A Cost-Based Adaptive Buffer Replacement Algorithm for FlashMemory Devices. *IEEE Trans. Comput.* **2014**, *63*, 2288–2301. [CrossRef]
25. Jin, P.; Ou, Y.; Härder, T.; Li, Z. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. *Data Knowl. Eng.* **2012**, *72*, 83–102. [CrossRef]
26. DeBrabant, J.; Pavlo, A.; Tu, S.; Stonebraker, M.; Zdonik, S.B. Anti-Caching: A New Approach to Database Management System Architecture. *Proc. VLDB Endow.* **2013**, *6*, 1942–1953. [CrossRef]
27. Zheng, S.; Shen, Y.; Zhu, Y.; Huang, L. An Adaptive Eviction Framework for Anti-caching Based In-Memory Databases. In *International Conference on Database Systems for Advanced Applications*; Springer: Cham, Switzerland, 2018; pp. 247–263.
28. Zhang, H.; Chen, G.; Ooi, B.C.; Wong, W.; Wu, S.; Xia, Y. "Anti-Caching"-based elastic memory management for Big Data. In Proceedings of the 2015 IEEE 31st International Conference on Data Engineering, Seoul, Korea, 13–17 April 2015; pp. 1268–1279.
29. Ma, L.; Ding, B.; Das, S.; Swaminathan, A. Active Learning for ML Enhanced Database Systems. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 175–191.
30. Hashemi, M.; Swersky, K.; Smith, J.A.; Ayers, G.; Litz, H.; Chang, J.; Kozyrakis, C.; Ranganathan, P. Learning Memory Access Patterns. In Proceedings of the International Conference on Machine Learning, 28–30 September 2018; pp. 1919–1928. Available online: http://proceedings.mlr.press/v80/hashemi18a/hashemi18a.pdf (accessed on 24 November 2021).
31. Jin, P.; Yang, P.; Yue, L. Optimizing B+-tree for hybrid storage systems. *Distrib. Parallel Databases* **2015**, *33*, 449–475. [CrossRef]
32. van Renen, A.; Leis, V.; Kemper, A.; Neumann, T.; Hashida, T.; Oe, K.; Doi, Y.; Harada, L.; Sato, M. Managing Non-Volatile Memory in Database Systems. In Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 1541–1555.
33. Shi, Z.; Huang, X.; Jain, A.; Lin, C. Applying Deep Learning to the Cache Replacement Problem. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2018; pp. 413–425. Available online: https://dl.acm.org/doi/10.1145/3352460.3358319 (accessed on 24 November 2021).
34. Vila, P.; Ganty, P.; Guarnieri, M.; Köpf, B. CacheQuery: Learning replacement policies from hardware caches. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, London, UK, 15–20 June 2020; pp. 519–532.
35. Yuan, Y.; Jin, P. Learned buffer management: A new frontier: Work-in-progress. In Proceedings of the 2021 International Conference on Hardware/Software Codesign and System Synthesis, Virtual Event, 10–13 October 2021; pp. 25–26.
36. Jasny, M.; Ziegler, T.; Kraska, T.; Röhm, U.; Binnig, C. DB4ML—An In-Memory Database Kernel with Machine Learning Support. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 159–173.