Integrating a Variable-size Page Buffer Manager in Kúzu

Fadhil Abubaker

ABSTRACT

The Kúzu graph database management system (GDBMS) uses a buffer manager with two statically allocated buffer pools. Each such buffer pool holds pages of a fixed size for loading database pages or for storing intermediate query processing results. A disadvantage of such a static allocation is the inability of the buffer pools to fully utilize the entire buffer memory space. As such, if one buffer pool is at full capacity, it cannot utilize vacant memory present in the other buffer pool to allocate new pages. To achieve better buffer memory utilization, an `mmap`-based buffer manager can be used, similar to the one employed in the Umbra disk-based DBMS [8]. This paper describes the integration of an `mmap`-based buffer manager within Kúzu. Benchmarks show that the `mmap`-based buffer manager has comparable performance to the existing buffer manager, with improved memory utilization.

1 INTRODUCTION

Disk-based database management systems (DBMSs) traditionally employ a buffer manager with fixed-size pages. While such a buffer manager is easy to implement, storing DBMS objects such as large strings or hash tables larger than the page size becomes difficult. This involves splitting the objects into page-size chunks and spreading them across multiple pages. Complex logic is then required to assemble and disassemble objects that span many pages [8].

Instead of using fixed-size pages, the buffer manager can allocate variable-size pages for storing large objects directly. One approach for implementing such a buffer manager is to have a single buffer pool that hosts pages of different sizes. However, such a buffer pool is prone to external fragmentation: when pages of different sizes are allocated and de-allocated successively, it can create holes in the buffer pool. As a result, while the buffer pool may overall have x bytes of free space, there is no single block that can accommodate a page of size x, leading to wasted space. Figure 1 illustrates this problem, where the buffer pool has enough space to allocate a 4KB page but is unable to do so due to fragmentation.

To overcome external fragmentation, multiple buffer pools can be allocated that each hold pages of a single size. However, this requires memory to be individually configured per buffer pool. If one buffer pool uses up its allocated memory, there is no mechanism to request more memory from another buffer pool. This leads to under-utilization of buffer memory. Figure 2 shows this problem, where a new 4KB page cannot be allocated even though the 2KB buffer pool is completely unused.

Kúzu's buffer manager uses the multiple buffer pool approach, where it has two internal buffer pools with page sizes of 4KB and 256KB. The amount of memory is configured separately for each buffer pool. This is reflected in the constructor for `BufferManager`, which takes two arguments: the size of the 4KB buffer pool and the size of the 256KB buffer pool. As a result, Kúzu's buffer manager is prone to under-utilization, since buffer pools are unable to steal memory from each other once they reach their static memory limits.

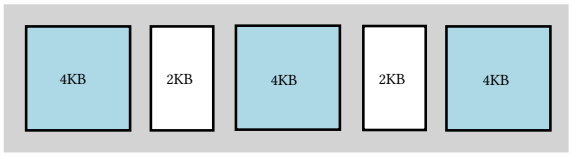

Figure 1: External fragmentation in a buffer pool that contains variable-size pages. Pages in blue are pinned, pages in white are unpinned. While there is enough space for a 4KB page (by evicting the two 2KB pages), there is no continuous region of memory to load the 4KB page.

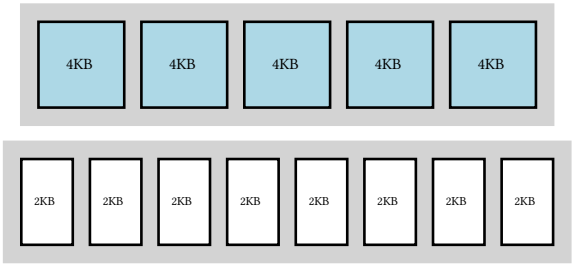

Figure 2: A buffer manager containing two separate buffer pools that hold pages of 4KB and 2KB. Pages in blue are pinned, pages in white are unpinned. The buffer manager is unable to allocate another 4KB page, even though there is unused memory in the 2KB buffer pool.

The Umbra disk-based DBMS [8] uses a buffer manager that transparently handles variable-size pages using a single buffer pool. Umbra's buffer manager organizes database pages into size classes, where each class contains pages of a fixed size. It allocates a virtual memory region for each size class using the `mmap` syscall, where each region can theoretically hold enough pages to fill the entire buffer pool. There is no fragmentation of the virtual memory region, since there are only fixed-size pages within a size class. In practice, however, the underlying physical memory will be fragmented, but this will be managed by the OS.

This project implements an `mmap`-based buffer manager in Kúzu inspired by the buffer manager in Umbra. The implementation seamlessly integrates into Kúzu's existing `BufferManager` class without modifying its external interface. Evaluation on the LDBC benchmarks shows that the `mmap`-based buffer manager has minimal overhead and can process more workloads with limited memory compared to the existing Kúzu buffer manager. Source code to the modified version of Kúzu can be found at https://github.com/fabubaker/kuzu/tree/umbra-bm.

2 UMBRA'S BUFFER MANAGER

The variable-size page buffer manager in Umbra organizes database pages into different size classes, each holding pages of a fixed size. Size class 0 contains pages of the smallest size (chosen as 64KB in

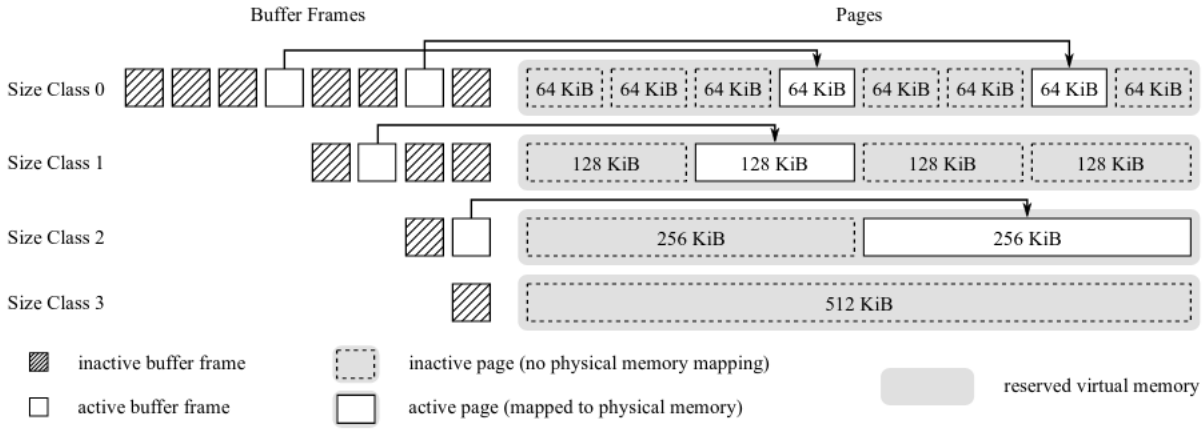

Figure 3: Illustrates the buffer manager in Umbra with 512KiB of buffer pool memory. Image obtained from [8].

Umbra, a multiple of the OS page size), and subsequent size classes contain pages with exponentially growing size. Size class *n* contains a single page that is as large as the entire buffer pool. As an example, with a buffer pool size of 512KiB, the sizes classes range from 0 to 3, with the page sizes in each class being 64KiB, 128KiB, 256KiB and 512KiB respectively. Figure 3 illustrates the example buffer manager with its different size classes.

When the DBMS is initialized, each size class is allocated virtual memory equal to the size of the entire buffer manager using `mmap`. Physical memory is only used when a page needs to be loaded into a buffer frame. This is guaranteed by the OS's demand paging behavior. This is reflected in figure 3, where only the active pages consume any physical memory. Since virtual memory is practically unlimited in a 64-bit OS, allocating more virtual memory than what is physically available does not cause any memory issues. However, the buffer manager will need to keep track of the total physical memory used by active pages to prevent it from exceeding the maximum size of the buffer pool. This is done by evicting pages to disk as physical memory usage approaches the buffer pool limit.

A notable advantage of the buffer manager is its lack of external fragmentation within the virtual memory region. A single buffer pool that holds variable-size pages will require complex page consolidation and reclamation algorithms to reduce external fragmentation [10]. Since each size class holds pages of a fixed size, there is no external fragmentation across the buffer pool. Note that the underlying physical memory will be fragmented, however the Linux kernel already contains mechanisms for managing this, such as memory compaction [1]. Instead of implementing mechanisms to solve external fragmentation within the buffer manager, Umbra leverages the existing algorithms in the OS through `mmap`.

3 KÚZU IMPLEMENTATION

We implement a slightly modified version of Umbra's buffer manager into Kúzu: namely, the implementation only supports two size classes of 4KB and 256KB. This makes it easy to design the `mmap`-based buffer manager as a drop-in replacement for the existing buffer manager, which has existing buffer pools of 4KB and 256KB. Finally, we replace the existing GClock eviction policy in Kúzu with the queue-based strategy in Umbra that works well with variable-size pages.

3.1 Buffer Manager

This section gives a brief overview of the existing buffer manager in Kúzu. It then outlines the `mmap` syscall and the implementation of the new `mmap`-based buffer manager.

3.1.1 Kúzu's malloc-based buffer manager. At startup, Kúzu's current buffer manager constructs two buffer pools, one for 4KB pages and one for 256KB pages. The 4KB buffer pool is used to load database files while the 256KB buffer pool is used for storing intermediate query processing results. By default, 75% of configured memory is alloted to the 4KB buffer pool and 25% to the 256KB buffer pool.

A `Frame` is the unit of buffer space in Kúzu. The constructor for `Frame` takes as argument the number of bytes to allocate for that `Frame`. Internally, the constructor calls the C++ `make_unique` method to allocate a byte array of the specified size. `make_unique` constructs an object using the `new` method, which in turn calls `malloc` underneath to allocate heap memory for the object.

On initialization, the `BufferPool` class creates multiple `Frame` objects with either 4KB or 256KB arguments. These `Frame` objects are placed in an array held by the `BufferPool`. As a result of calling `malloc` to allocate buffer memory, Kúzu immediately consumes the configured amount of physical memory on startup.

The `FileHandle` class is used to manage I/O operations on files within Kúzu. Each `FileHandle` instance has flags that denote whether it is used to store 4KB or 256KB pages. The methods exposed by the `BufferManager` all take a a `FileHandle` object as argument. Internally, the `BufferManager` cases on the page size flag in the `FileHandle` object to determine the correct buffer pool to call the operation on. This can be seen in `buffer_manager.cpp`, where each method executes a conditional check to see which buffer pool to use.

3.1.2 mmap syscall. The `mmap` syscall is used to create a new virtual memory mapping in Unix systems [6]. Given the amount of bytes to allocate, the memory protection and visibility flags, it allocates a region of virtual memory with the desired properties and returns a pointer to it. Due to demand-paging [11], the allocated virtual memory does not consume any physical memory when `mmap` returns. Only when a specific address within the virtual region memory region is accessed does the OS allocate physical memory. Furthermore, physical memory allocations are always performed in 4KB chunks, which is the OS page size.

The `madvise` [5] call is a companion syscall to `mmap`. It can be used to provide hints to the kernel about the usage patterns of a specific memory region allocated through `mmap`. Most of these hints are related to improving the performance of the application using `mmap` and do not affect any run-time semantics. An exception to this is the `MADV_DONTNEED` hint, which indicates to the OS that the region of memory will not be used in the near future and it can free the underlying physical memory. Subsequent reads of the virtual memory region will return zero-filled pages to the application.

Note that `mmapmadvise` is restricted to POSIX-compliant Unix systems and as a result may not work on OSes like Windows.

3.1.3 mmap-based buffer manager. A new `BufferPoolMmap` class is created with the same interface as the original `BufferPool` class. The constructor for `BufferPoolMmap` calls `mmap` twice, one for the 4KB size class and one for the 256KB size class. The full memory size configured for the buffer manager is passed as argument to `mmap` in both cases. Thus, when Kúzu is configured with 1MB of buffer memory, the `mmap` calls request a total of 2MB of virtual memory on startup. Note that due to demand-paging, this allocation does not consume any physical memory as of yet.

A new constructor is implemented for the `Frame` class that takes a pointer and a page size as arguments. Additionally, the `Frame` class contains an existing `buffer` attribute wrapped in a `unique_ptr` that is used to hold the pointer to the byte array. Instead of using this, we create a raw `mmapBuffer` attribute not protected by a `unique_ptr`. This is because in order to return the `mmap` pointer as a `unique_ptr`, a custom deleter method needs to be implemented for it. Implementing such a custom deleter is non-trivial, and since `Frame` objects never get deleted during the lifecycle of the buffer manager, we leave `mmapBuffer` as a raw pointer.

As mentioned in section 2, the buffer pool needs to keep track of the total amount of physical memory consumed by all the pages. For this, the `BufferPoolMmap` class contains a 64-bit `currentMemory` integer attribute, encapsulated as a C++ `atomic`. This attribute is incremented when a page is loaded into an empty frame and decremented when a page is evicted (but not when it is replaced with another page).

When a frame is evicted, the kernel needs to be told to release the physical memory associated with that frame. This is achieved by using the `madvise` call with the `MADV_DONTNEED` flag. It can be passed a pointer and the number of bytes of memory to be relinquished by the OS. This is used to implement the `releaseMemory` method in the `Frame` class, which is called when a `Frame` is evicted (but not when it is replaced with another page, see 3.2.2 for more details).

3.2 Eviction Strategy

The existing buffer manager in Kúzu uses the GClock eviction strategy, which is meant to work with fixed-size pages. The `mmap` buffer manager instead uses an eviction queue that works seamlessly with variable-size pages. This section summarizes the motivation and design behind the queue-based page replacement strategy.

3.2.1 GClock and variable-size pages. The GClock page replacement strategy is a popular page eviction strategy used in many DBMSs [2]. GClock attempts to simulate both the Least-Recently-Used (LRU) and Least-Frequently-Used (LFU) page replacement algorithm by associating a counter with each page in the buffer pool. This counter is set to 1 once a page is loaded into the buffer pool, and is incremented every time the page is pinned and decremented when it is unpinned. On page replacement, a clock hand sweeps through the pages and replaces the first page that has its counter set to zero. The clock hand then points to the next page in the clock order, which is where it starts on the next page replacement.

However, it is not immediately obvious on how to re-purpose GClock for use with variable-size pages. This is because of the page size mismatch that can happen between the page to be evicted and the page to be loaded. As an example, while attempting to load a 256KB page GClock may choose to evict a 4KB page. Once that page is evicted, there is still not enough memory in the buffer pool to load the 256KB page. As a result, more pages will need to be evicted, which is a case the basic GClock strategy does not handle.

3.2.2 Eviction queue. Instead of using GClock, we implement the queue-based strategy used in Umbra, which in turn is borrowed from the LeanStore storage engine [4].

Pages in the buffer pool, that when unpinned, have a pin count of zero are immediately enqueued onto an eviction queue. Once the buffer manager detects that the amount of physical memory in use is close to the maximum limit, it starts evicting pages from the end of the queue. However, if at the time of eviction (after the appropriate locks are acquired) if the pin count of the page is not zero, that page is no longer considered for eviction. Instead, the buffer pool moves on to the next page at the end of queue. Pages are gradually evicted in this manner until there is enough memory to load the newly requested page.

The implementation of the eviction queue was mostly borrowed from DuckDB [9], since it uses the same strategy for evicting pages. A C++ implementation of a concurrent queue [7] is used to implement the eviction queue in a thread-safe manner. The `BufferPoolMmap` class is modified to hold an `evictionQueue` attribute that stores `EvictionQueueNode` objects for this purpose.

The `claimAFrame` method is modified to incorporate this new eviction strategy. When attempting to load a new page into the buffer pool, if `currentMemory` plus the size of the new page exceeds the limit, the eviction process starts. Pages from the end of the queue are evicted until there is enough memory to accommodate the new page into the buffer pool.

Note that it is possible for a page already in the queue to be pinned then unpinned again. At the time of eviction, while it may seem that the pin count is zero, the page has been recently accessed since being enqueued and hence should not be evicted.

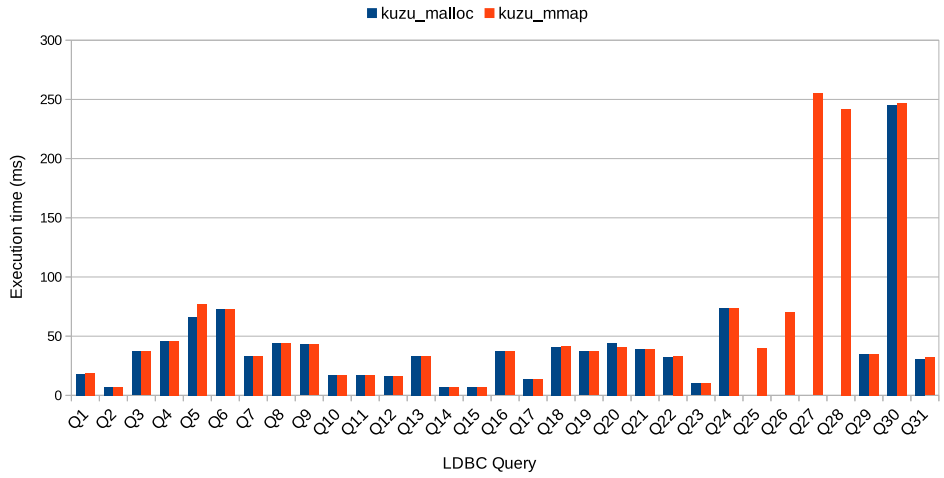

Figure 4: LDBC completion times with scale factor 0.1

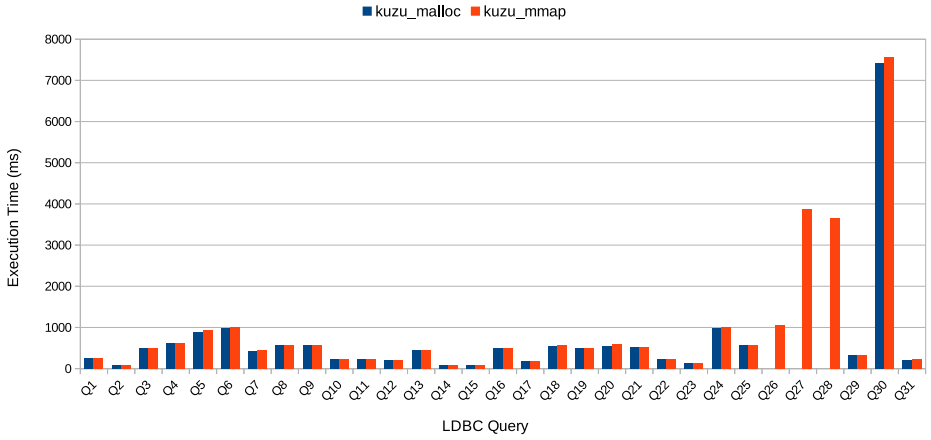

Figure 5: LDBC completion times with scale factor 1

To handle this case, both the `Frame` and the `EvictionQueueNode` class contains an `eviction_timestamp` attribute. When a `Frame` is about to be placed on the queue, the `eviction_timestamp` is incremented. The `Frame` and the current value of the timestamp are then placed into an `EvictionQueueNode` and inserted into the queue. At the time of eviction, the timestamp on the `EvictionQueueNode` is checked with the timestamp on the `Frame`. If there is a mismatch, the same `Frame` has been placed higher up on the queue and therefore should not be evicted.

To ensure the correctness of the implementation, the `mmap` buffer manager was tested using the existing Kúzu test suite. All tests pass except for three: `main api_tests`, `python_api:tests` and `storage disk_array_update_test`. The first two fail since the `resize` operation is left unimplemented in the `mmap` buffer manager due to time constraints. The third test was already failing on the commit from the original Kúzu branch and is left unmodified.

4 EVALUATION

We perform experiments that showcase the performance and memory utilization efficiency of the `mmap` buffer manager. Comparisons are made with Kúzu's original `malloc` buffer manager using the LDBC business intelligence benchmark [3]. The default allocation ratio was used for the `malloc` buffer manager, where 75% of buffer memory is allocated to the 4KB buffer pool and the remaining to the 256KB pool.

Experiments were run on a machine with an Intel i7-7700HQ @ 2.80GHz CPU and 32GB of RAM with 4 physical cores and 8 logical cores. For latency measurements, we repeated each query 5 times and report the average runtime. Both buffer managers were given the same amount of buffer memory, which varies depending on the experiment. We only report results when running experiments with one thread since no major difference was found with an increasing thread count.

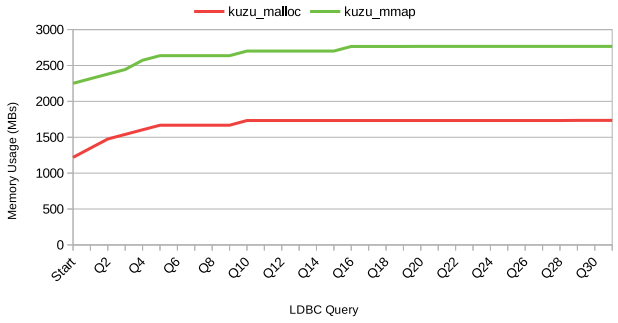

Figure 6: Virtual memory usage over LDBC query execution

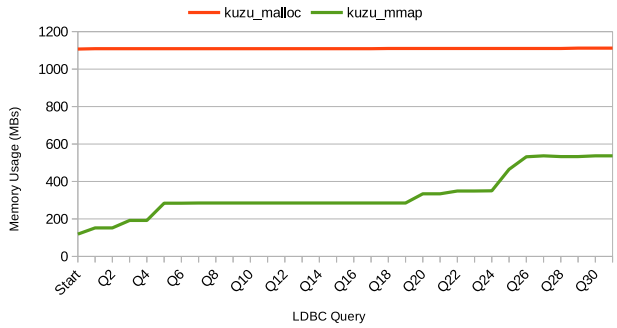

Figure 7: Physical memory usage over LDBC query execution

4.1 LDBC Completion Times

We execute all 31 LDBC queries on both the `malloc` and `mmap` buffer managers. For scale factor 0.1, both buffer managers were given 32MB of buffer memory. For scale factor 1, the buffer managers were given 512MB of memory. Results show that the `mmap` buffer manager has similar completion times to the `malloc` buffer manager on all queries. Figures 4 and 5 plot the results.

However, the `malloc` buffer manager was unable to execute three queries to completion with the given amount of buffer memory: Q26, Q27 and Q28. For scale factor 0.1, the `malloc` buffer manager was also unable to execute Q25. The execution times for these queries are omitted from the plots as a result.

This is because all four queries perform full scans over the target table. As a result, these queries push the buffer manager to its limits since it has to load the entire table. Q25, Q26, Q27 perform `ORDER BY` operations and Q28 performs an aggregation over the full table. In this case, the static memory allocation of the `malloc` buffer manager prevents it from loading all the necessary pages to execute the query. Since the 4KB buffer pool is given 75% of the buffer memory, it has a smaller limit to the number of pages it can load. By contrast, the `mmap` buffer manager has no such restrictions, since the 4KB size class can effectively take up the entire buffer pool memory if required. As a result the `mmap` buffer manager is able to load all the required pages and execute the queries to completion.

4.2 Memory Usage

To study the virtual and physical memory usage of both buffer managers, we track memory usage as the LDBC queries are executed in order. For this experiment, we use LDBC with a scale factor of 1 and give both buffer managers 1GB of memory. Results are shown in figures 6 and 7. Physical and virtual memory usage is captured after executing each query using the Linux `top` utility.

At startup, the `mmap` buffer manager uses roughly twice the amount of virtual memory as the `malloc` buffer manager. This is due to both size classes in the `mmap` buffer manager being allocated the same amount of virtual memory as the entire buffer pool. In contrast, the physical memory usage of the `mmap` buffer manager starts low, while the `malloc` buffer manager completely consumes the entire 1GB of physical memory.

As queries are executed, both buffer managers consume increasing amounts of virtual memory. This increase is primarily attributed to internal objects being created and likely does not originate from the buffer managers themselves. However, physical memory consumption is static throughout the duration of the run for the `malloc` buffer manager. This is because the `malloc` buffer manager initializes all `Frame` objects in both buffer pools at startup, each of which consume physical memory on construction. By contrast, the `mmap` buffer manager is conservative with physical memory usage and by the end of the last query has only consumed roughly 550MB of physical memory.

5 CONCLUSION

In this project, we implement an `mmap`-based buffer manager that can hold variable-size pages and integrate it into Kúzu. Experimental results show minimal overhead compared to the existing `malloc`-based buffer manager with lower physical memory consumption and improved memory utilization.

However, the main advantage of such a buffer manager comes from its size classes. While there are only two size classes in this implementation, future work can look at adding more size classes. This allows for Kúzu to allocate large database objects such as strings or hash tables without having to split these across multiple pages.

Finally, the Umbra paper describes many optimizations that enhance the performance of the `mmap`-based buffer manager. For example, the paper describes the implementation of a fully-decentralized pointer swizzling mechanism. Currently, Kúzu relies on a hash table that stores mappings from pages to frames (`pageIdxToFrameMap`) to convert a page identifier to a frame and vice versa. The pointer swizzling implementation in Umbra discards the use of such hash tables and embeds page identifier information within the frame pointer itself. Further investigation is required to see what optimizations can be borrowed to improve Kúzu's query execution performance.

REFERENCES

[1] 2022. Memory compaction [LWN.net] — lwn.net. https://lwn.net/Articles/368869/. [Accessed 17-Dec-2022].

[2] Wolfgang Effelsberg and Theo Haerder. 1984. Principles of Database Buffer Management. *ACM Trans. Database Syst.* 9, 4 (dec 1984), 560–595. https://doi.org/10.1145/1994.2022

[3] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[4] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 185–196. https://doi.org/10.1109/ICDE.2018.00026

[5] madvise. 2022. madvise(2) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man2/madvise.2.html. [Accessed 17-Dec-2022].

[6] mmap. 2022. mmap(2) - Linux manual page — man7.org. https://man7.org/linux/man-pages/man2/mmap.2.html. [Accessed 16-Dec-2022].

[7] moodycamel. 2022. GitHub - cameron314/concurrentqueue: A fast multi-producer, multi-consumer lock-free concurrent queue for C++11 — github.com. https://github.com/cameron314/concurrentqueue. [Accessed 16-Dec-2022].

[8] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[9] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[10] Fridtjof Siebert. 2000. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*. 9–17.

[11] Andrew S. Tanenbaum and Albert S. Woodhull. 1997. *Operating systems: Design and implementation*. Prentice-Hall.