# Separated Allocator Metadata in Disaggregated In-Memory Databases: Friend or Foe?


Marcel Weisgut
*Hasso Plattner Institute*
*University of Potsdam*
Potsdam, Germany
marcel.weisgut@hpi.de

Daniel Ritter
*Hasso Plattner Institute*
*University of Potsdam*
Potsdam, Germany
daniel.ritter@hpi.uni-potsdam.de

Martin Boissier
*Hasso Plattner Institute*
*University of Potsdam*
Potsdam, Germany
martin.boissier@hpi.de

Michael Perscheid
*Hasso Plattner Institute*
*University of Potsdam*
Potsdam, Germany
michael.perscheid@hpi.de



***Abstract*—Memory allocation has a significant impact on the performance of in-memory databases. While state-of-the-art memory allocators work well in DRAM-only setups, some of their design decisions might no longer yield efficiency if data is tiered to disaggregated memory or secondary memory tiers.**

**In this work, we study the performance impact of metadata in memory allocators and their tiering to disaggregated memory in the context of in-memory databases for the first time. We show how to separate metadata and application data by the example of jemalloc, which is widely used for data-intensive applications, and study performance effects for different workloads.**



***Index Terms*—Disaggregated Memory, Memory Allocators, Metadata Separation, In-Memory Databases**


## I. Introduction

In-memory databases (IMDBs) store most of their data in main memory to reach low latencies and high throughput. However, main memory becomes a limiting factor for IMDBs due to growing data [1] and the necessary co-location of resource types, such as compute and memory (e. g., [2, 3]), limiting their independent scalability and leading to resource over-provisioning [4, 5]. To overcome these challenges, the next hardware generation will physically separate computation and memory resources (called memory disaggregation) [6, 7], which allows IMDBs to prevent memory limitations of dedicated servers. While these technological advances are highly relevant for IMDBs, academic and industrial database research only recently picked up disaggregated memory (DM) hardware (e. g., [1, 8]). Besides the challenges around increased data access latency [9], transparent access to this new data tier without changing the design of the database management system (DBMS) is crucial. While non-uniform memory access (NUMA) denotes an established memory interface to leverage DM, specific memory resources with dynamic memory allocations are simpler and less invasive [10]. However, in both cases, memory allocation and location significantly impact the query processing performance of a DBMS [11, 12]. Especially, the combined storage of data and metadata in current memory allocators might be inefficient [13].

In this work, we study the impact of metadata separation on disaggregated in-memory DBMSs in terms of (1) query performance, (2) the duration of migrating data from CPU-local main memory to DM (tiering), and (3) compare performance impact for data stored on a remote NUMA node. As an exemplary memory allocator, we use *jemalloc*, which is widely used, e. g., for data-intensive applications (cf. [11, 12]) and in well-known DBMSs, such as Hyrise [14], Umbra [12], and RocksDB [13]. As DBMS, we use the open-source, columnar, in-memory system Hyrise. We created a new tiering plugin, allowing for flexible assignment of memory resources via jemalloc to migrate application data and metadata from local memory to secondary memory tiers, such as SSD and a DM appliance without changing the DBMS.

In particular, we investigate whether co-location of a memory allocator's application data and metadata is a performance bottleneck and whether its separation can improve the performance in a DM scenario. We make the following contributions:

- *Tiering Concept.* We present an approach that allows separating jemalloc's metadata from the tiered application data on block-level devices.
- *Tiering Prototype.* We extend an existing IMDB and enable efficient tiering with minimal code changes using C++'s polymorphic allocators and memory resources.
- *Metadata Separation Evaluation.* We evaluate the performance of different separation strategies for analytical database workloads (TPC-H, TPC-DS) on (i) block-level disaggregated memory and (ii) remote NUMA nodes.

In summary, we identified that co-locating an allocator's metadata and application data neither constitutes a performance bottleneck on external storage, such as DM and SSDs, nor shows significant performance improvements. While the NUMA results suggest slight performance improvements, the biggest effects were found for specific allocator and user-space page handler configurations. These insights are valuable when integrating DM into applications, such as IMDBs.

## II. Background and Related Work

This section introduces the jemalloc memory allocator, *UMap*, a user-space page management library used in our data migration experiments, and discusses related work.

### A. The jemalloc Memory Allocator

jemalloc is an open-source[1] `malloc(3)` implementation especially designed for scalable concurrency and fragmentation avoidance [15], which has experimentally been shown to

[1] jemalloc on GitHub, visited 3/22: https://github.com/jemalloc/jemalloc

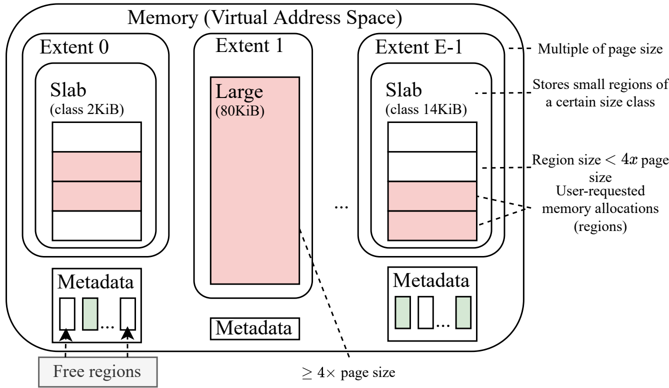



Fig. 1: jemalloc's memory layout

have the best performance for IMDBs [12]. jemalloc splits memory into extents, which are aligned to a multiple of the page size [16]. User-requested memory allocations are called *regions*. Regions are categorized into *small* and *large* size classes. Small classes are smaller than, and large classes greater than or equal than four times the page size. One large region is stored in one extent. Contiguous small regions of the same size form a slab, which resides in a single extent. jemalloc's memory layout is exemplified in Figure 1.

With memory being split into extents, an arena manages a set of extents. Arenas are self-contained memory allocators [15] (i. e., managing extents as mutual exclusive memory regions) and for each arena, jemalloc allows to optionally specify custom extent management functions, so-called extent hooks [16]. Those extent hooks include, for example, functions to allocate and deallocate memory for a given extent. jemalloc uses multiple arenas to reduce lock contention [16].

### B. UMap: User-Space Page Fault Handling

*UMap* is a user-space page management library based on the `userfaultdf` mechanism [17]. It comes with its own page cache and provides various configuration parameters, which allow optimizing the page management to match the needs of applications' data access patterns. Such parameters are, for example, the size of the internal page cache, the page size, and the number of workers to fetch and evict pages (fillers and evictors). For further details about UMap, we refer the interested reader to the work of Peng et al. [17, 18].

### C. Related Work

To the best of our knowledge, no work addresses metadata separation for in-memory databases with disaggregated storage. Nevertheless, some papers cover one or two of these aspects, which we discuss subsequently.

The memory allocator *Makalu* [19] ensures persistence of metadata but does not cover the idea of separating metadata from application data.

The memory allocator *WAlloc* [20] designed for non-volatile random access memory (NVRAM) decouples metadata from application data and stores volatile metadata in local DRAM. This reduces the number of NVRAM writes and, thus, increases the lifetime of NVRAM with its limited endurance. Nevertheless, NVRAM is still located in the same machine and not related to DM like in our work.

*PAllocator* [21] is a persistent allocator for storage class memory (SCM). The allocator internally uses two allocators: one for small, the other for big allocations. The latter uses hybrid SCM-DRAM trees to persist metadata. This tree persists the leaf nodes as a linked list in SCM and stores inner nodes in DRAM for performance reasons.

Zonouz et al. leveraged jemalloc's internal separation of metadata and application data to relocate the metadata to another position of the process memory space for data protection reasons [22]. Rather than optimizing data access latencies, they store metadata inside a memory enclave to ensure that heap metadata cannot be affected during heap buffer overflows.

## III. System Overview & Architecture

This section introduces our prototypical extension of the Hyrise in-memory database for DM, which we use in our experimental evaluation.

### A. Hyrise In-Memory Database

Hyrise is an open-source[2], relational, in-memory DBMS that can be non-invasively extended for our purpose through its built-in plugin mechanism [14]. Data is stored in a columnar layout and each table is horizontally partitioned into fixed-sized *chunks*. The resulting column fractions stored in a chunk are called *segments*. Data is inserted into the most recent chunk, which is mutable. Using MVCC [23] concurrency control, updates are executed by appending the new tuple and invalidating the previous version (similar to deletes). Once a chunk reaches its limit, the chunk is marked as immutable and a new mutable chunk is appended to the table. The size of chunks, usually between a few to dozens of megabytes, and their immutability make chunks and their segments suitable units for data movement and placement on slower storage tiers.

### B. Experimental Disaggregated Memory

In our experiments, we run a Hyrise instance on a server that is connected to storage and memory as depicted in Figure 2. The server is connected to an NVMe-based SSD and to a Gen-Z-based[3], block-level DM appliance. Both the SSD and the DM appliance are connected via PCIe Gen 3.0. The DM appliance consists of a Gen-Z bridge, a Gen-Z switch, and Gen-Z memory modules. The server is connected to the bridge, which presents the DM appliance's memory as NVMe namespace and translates NVMe I/O to Gen-Z memory transactions. The bridge is connect to the switch, which connects the compute node (i. e., server) with Gen-Z memory modules utilizing the 3.0" form factor per SFF-TA-1008 specification. Furthermore, the server is attached to 256 GB DDR4 memory.

[2]Hyrise on GitHub, visited 3/22: https://git.io/hyrise
[3]GenZ specification, visited 3/22: https://genzconsortium.org/specifications/

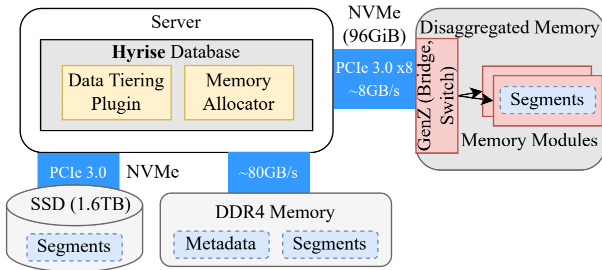



Fig. 2: Hyrise setup with disaggregated memory

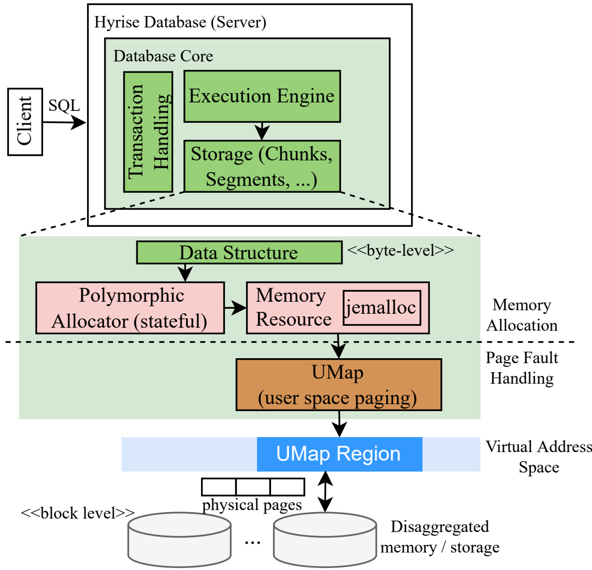



Fig. 3: Hyrise tiering approach

### C. Hyrise Tiering Approach

Hyrise uses C++17's polymorphic allocators to decouple the data storage and data access. Initially started as a pure in-memory database, Hyrise's codebase uses numerous data structures of C++'s standard library (e.g., `std::vector`). Using the C++ polymorphic allocator interface and polymorphic memory resources, data structures can be dynamically placed on different memory tiers with only minimal changes to the codebase (cf. [24]). Creating an SSD-resident integer vector can be accomplished with `std::pmr::vector<int>(std::pmr::polymorphic_allocator{&ssd_memory_resource})`.

We extended Hyrise through a data tiering plugin, in which we use memory resources to migrate segments between storage tiers (e. g., disaggregated memory or SSD). Figure 3 depicts how we use Hyrise's polymorphic allocators and jemalloc to migrate segments to secondary storage tiers.

To memory-map files from secondary storage tiers, we decided against `mmap`. Besides various short comings of `mmap` (cf. [25]), the main issue for an IMDB with `mmap` is that memory-mapped pages reside in the kernel's page cache, whose size cannot be restricted by the IMDB. This can be prevented by using page fault handlers such as UMap, which provides its own size-configurable page cache and performs the page fault handling in the user-space. Hence, we added

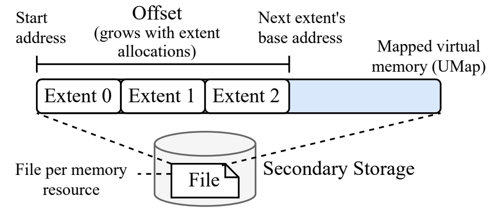



Fig. 4: jemalloc memory resource

UMap (cf. Section II-B) to Hyrise for our tiering approach.

### D. Separating Metadata and Application Data

Since recently, jemalloc has allowed handling metadata allocations differently than application data allocations [26]. For an arena (cf. Section II-A), this allows controlling whether custom extent hooks are used for metadata allocations. This feature is in an experimental state and not fully exposed to the public API. To leverage it, we implemented a custom extent hook and use a jemalloc-internal data structure in the tiering plugin of Hyrise. Our extension allows to store metadata on DRAM while application data (e. g., segments) can be stored on different devices, such as DM and SSD (cf. Figure 2).

### E. Segment Migration and Extent Allocation

Three steps are executed to migrate a segment to another storage tier: (i) the segment is copied from one memory resource to the destination resource by calling the segment's data structures' copy constructors with a polymorphic allocator as the allocator parameter (cf. Section III-C), (ii) the pointer within the segment's chunk is atomically switched, and (iii) the previous allocation is freed. The memory resource internally uses jemalloc calls for memory allocations. Therefore, we refer to it as *jemalloc memory resource*. Its initialization maps a large memory address range to a file stored on the DM using UMap. Furthermore, it creates a new jemalloc arena, for which metadata separation can be controlled as explained in Section III-D. For this arena, we specify a custom extent allocation function via extent hooks (cf. Section II-A). All extents allocated with this function are allocated linearly in a file-backed address range as shown in Figure 4. The first extent's base address equals the mapped address range's start address. Each extent allocation increases an offset. The sum of the start address and the offset is the next extent allocation's base address. For region allocations (cf. Section II-A), the jemalloc memory resource uses `mallocx` [16].

## IV. Experimental Evaluation

With the described tiering approach, we experimentally evaluated the performance impact of metadata separation in jemalloc, in the following two scenarios: (i) both metadata and application data is stored on the block-level device (i. e., DM or SSD); (ii) metadata is stored on CPU-local DRAM whereas application data is stored on the block-level device (DM or SSD). To study jemalloc isolated from UMap and our disaggregated memory appliance for scenarios (i) and (ii), we replaced both by a remote NUMA node.

## A. Setup

The experiments were executed on a machine with two CPUs (2.5 GHz base, 3.1 GHz turbo), on which each socket serves as a NUMA node. Each NUMA node has 128 GiB of DDR4-3200 memory, distributed over two DIMMs. In addition to the local DRAM, the machine is connected to the DM appliance described in Section III-B. The system runs on Ubuntu 20.04 with kernel version 5.11.0-16-generic. We used GCC 9.3, UMap version 2.1.0 (Git commit `a06f1a3`), and jemalloc version 5.2.1 (Git commit `011449f1`).

For experiments on the extended Hyrise database, we use the analytical database benchmarks TPC-H and TPC-DS[4] with no further compression for the datasets. For both benchmarks, a *scale factor* (SF) can be specified, which approximately determines the size of the generated data in GB. When data is stored on DM or SSD, Hyrise generates segments and migrates all of them to the target device (cf. Section III-E).

Except for the number of UMap evictors and fillers, we used UMap's default configuration. Since optimizing the UMap configuration is not focus of this work, we set the number of fillers and evictors to one each, and executed the benchmarks single-threaded to reduce potential noise introduced by multiple running threads. Initial experiments showed that the overhead of UMap with a page cache large enough so that all data is stored in that DRAM-local page cache is marginal: with UMap, query latencies in Hyrise are increased by only about one percent according to the geometric mean for the TPC-H and TPC-DS benchmarks with scale factor 8.

## B. Disabling jemalloc's Retain Option

By default on 64-bit Linux systems, jemalloc does not discard unused virtual memory but retains it for later reuse [16]. This, however, results in using more virtual memory address space. With the DM's capacity of 96 GiB and using the extent allocation method (cf. Section III-E), we exceeded the file-backed memory address range with (TPC-H/DS) scale factors larger than one. This can be avoided by disabling the retain option as shown in the following experiment.

*Setup.* We performed the TPC-H benchmark with scale factor one for various UMap page sizes. The UMap page cache was set to 20% of the SF's data size (i. e., 200 MB). The set of queries was executed over 20 minutes in random order. For the DRAM-only baseline, we enabled the retain option (default).

*Results.* Figure 5 shows the resulting runtime with and without retaining unused memory with data segments stored on DM normalized to the runtime of segments stored on DRAM. Besides avoiding virtual memory exhaustion, it can be seen that disabling jemalloc's retain option also improves the runtime, except for the page size of 4 KiB. Furthermore, the figure shows that a page size of 256 KiB led to the lowest total runtimes. We also made this observation with SF 10 (not shown). It can be seen that the runtime increases with decreasing and increasing page sizes. This could be explained due to higher page load events for smaller page sizes and

[4]As of March 2022, Hyrise supports 47 TPC-DS queries.

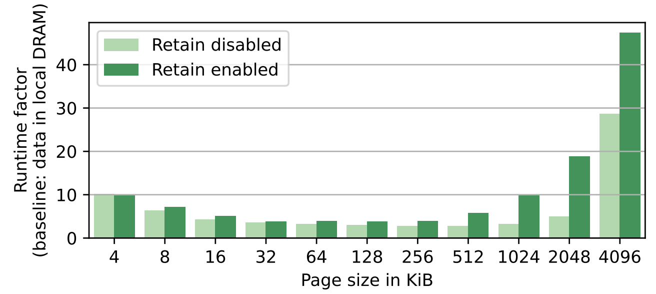



Fig. 5: Normalized total runtime of the TPC-H benchmark

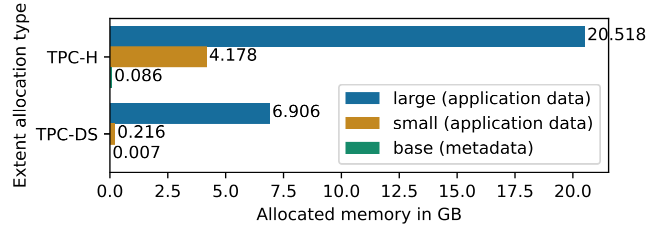



Fig. 6: Allocated memory during segment migration

higher data shipment for higher page sizes. For the TPC-DS benchmark with SF one, a page size of 32 KiB led to the lowest runtimes. In the subsequent experiments, we uniformly used a page size of 256 KiB for comparability reasons and disabled jemalloc's retain option if not mentioned otherwise.

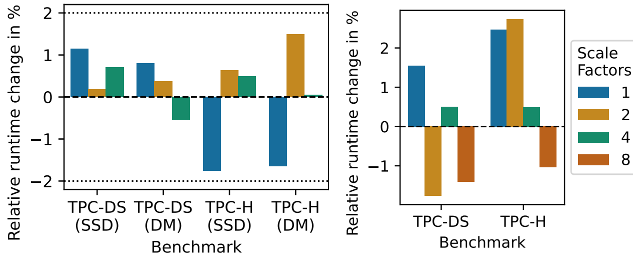
*Findings.* (1) Using our custom extent allocation strategy (cf. Section III-E), disabling jemalloc's retain option reduces the allocated virtual memory space. (2) Except for a UMap page size of 4 KiB, disabling jemalloc's retain option reduces the benchmark's total runtime. (3) UMap configurations such as page sizes are workload-specific (e. g., TPC-H) and have a large impact on the overall runtime.

## C. Metadata Memory Footprint

We assume that the more data accessed and stored on a device with increased access latencies compared to CPU-local DRAM, the higher the potential to achieve runtime improvements by storing that data on CPU-local DRAM. Therefore, we investigated the footprint of memory allocated for metadata and application data.

*Setup.* For each arena, jemalloc provides statistics about the size of small, large, and base allocations [16]. The latter is the size of allocations performed by the base allocator, from which metadata of an arena is allocated. Therefore, base extents, which are extents allocated by the base allocator, are used to store jemalloc's metadata. Using jemalloc's allocation statistics, we measured the size of application data and metadata allocated during the segment migration for the TPC-H and TPC-DS benchmarks with SF 8. For this, we measured before and after the migration and calculated the differences, which are shown in Figure 6.

*Results.* Figure 6 shows that the base extent allocations constitute only a marginal part of the allocated memory. Compared to the small and large extents, metadata only takes up a share




(a) Query processing

(b) Data migration

Fig. 7: Relative runtime change comparisons

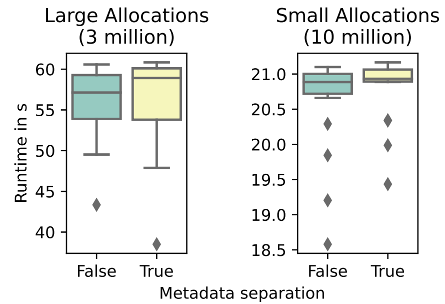



Fig. 8: Runtime for 3 M large and 10 M small allocations.

of 0.35% in the TPC-H and 0.1% in the TPC-DS benchmark. Note, the allocated application data for the TPC-H benchmark is higher than the approximately eight gigabytes specified by the scale factor due to the benchmark's string dominance [27] and C++'s memory consumption for `std::string`.
*Finding.* (4) Compared to the application data allocated for the data migration during the TPC-H and TPC-DS benchmarks, metadata takes up a marginal share of less than one percent.

### D. Analytical Read-Only Database Workloads

This experiment evaluated the impact of metadata separation on query runtimes for read-only, analytical workloads.
*Setup.* We performed the TPC-H and TPC-DS benchmarks and compared the query performance with data stored on the target device with and without metadata separation. For each experiment, the corresponding benchmark was executed three times. In a single execution, each query was executed five times with five previous warmup runs, resulting in 15 measured query runs. The queries were executed in a sequential order (i. e., all runs of a query were finished before the next query was started). The calculated total benchmark runtime is the sum of each query's mean execution times.
*Results.* For the TPC-H and TPC-DS benchmarks with segments stored on the DM and the SSD, respectively, Figure 7a shows the relative performance change achieved with metadata separation compared to the resulting runtimes without metadata separation. It can be seen that the performance changes for all benchmark setups is between -2% and +2%.
*Findings.* (5) Separating metadata while using the jemalloc memory resource on the DM appliance or SSD does not result in consistent latency improvements for the TPC-H and TPC-DS workloads with latency changes between -2% and +2%.

### E. Migration Time

In this experiment, we focus on write-intensive workloads in the form of data segment migrations through extended TPC-H and TPC-DS benchmarks (cf. Section III-E).
*Setup.* We measured the time required for the segment migration for the TPC-H and TPC-DS benchmarks for the SFs 1, 2, 4, and 8. For all configurations, the migration was executed ten times and the arithmetic means were used to calculate the runtime changes.
*Results.* Figure 7b shows the resulting runtime change with metadata separation compared to the runtime achieved without separation for different scale factors. Both runtime improvements and degradations can be seen for different benchmark configurations. The runtime changes between -1.8% and +2.8% (lower is better).
*Finding.* (6) Using the jemalloc memory resource for migrating data to the DM appliance, the separation of metadata shows no consistent runtime improvements in our experiments.

### F. UMap Allocation Micro Benchmark

To better understand the impact of metadata separation for write-heavy workloads, we ran sequences of memory allocations as micro-benchmarks.
*Setup.* We performed two benchmarks, one with three million sequential 16 KiB large allocations and ten million small one KiB allocations. We used the jemalloc memory resource (cf. Section III-E) with the DM appliance as target device to perform the memory allocations.

The benchmarks were performed with and without the metadata separation. Each benchmark setup was executed 20 times with jemalloc's retain option enabled, a UMap page size of 256 KiB and a UMap page cache size of 10 MiB. This cache size ensured that pages are evicted and, thus, stored on DM.
*Results.* Figure 8 shows the resulting runtime change with metadata separation compared to the runtime achieved without metadata separation. On a mean basis and compared to the runtime achieved without metadata separation, the runtime increases by 0.8% for small allocations and 0.2% for large allocations with separated metadata.
*Finding.* (7) Using the jemalloc memory resource for sequential data allocations, the separation of metadata slightly increases the runtime by less than one percent.

### G. NUMA Allocation Micro Benchmark

To study application data placements on memory with higher latencies but without user-space page handling through UMap, we stored data on a remote NUMA node. Note, the latency of a remote NUMA node is not comparable to that of DM devices as the distances can be different. Furthermore, the DM protocol, e.g., Gen-Z, adds additional latency. With this abstraction, we reduce our experiment's software and hardware stack and, thus, the potential noise.

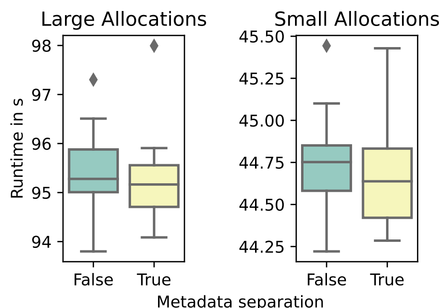



Fig. 9: Runtime for 10 million memory allocations.

*Setup.* In this experiment, we allocated data on a remote NUMA node with and without metadata separation. Using Linux's `numa(3)` library (libnuma) [28], we bound the benchmark process and its memory allocations to node one. Similar to the previous experiments, we used a polymorphic memory resource and a custom extent allocation hook to migrate data to a remote node (i. e., node zero). If metadata and application data are separated, the custom extent hook is only used for metadata allocations. When data is not separated, the hook is used for both types of data. Measurements with the *Memory Latency Checker* [29] showed an idle latency of 88 nanoseconds for a random access from node one to node one and 150 nanoseconds from node one to node zero. We enabled jemalloc's retain option (i. e., jemalloc's default setting) for this benchmark since we did not encounter virtual memory capacity problems in these micro-benchmarks as in the TPC benchmarks with the jemalloc memory resource (cf. Section IV-B). We performed two benchmarks (each 20 times), each with ten million sequential large and small allocations, respectively. In one benchmark, we performed large allocations, in the other benchmark, we performed small allocations (cf. Section II-A). We generated uniformly distributed random values between one byte and 16KiB, as well as 16KiB and 256KiB for small and large allocations, respectively.

*Results.* Figure 9 shows the resulting runtime change with metadata separation compared to the runtime achieved without metadata separation. On a mean basis, the runtime can be marginally reduced by 0.13% and 0.24% for small and large allocations, respectively.

*Finding.* (8) The required runtime for metadata allocations can be slightly reduced by less than one percent in a NUMA setup.

### H. Discussion

The experimental evaluation showed insightful findings, which we will briefly discuss subsequently. The findings are divided into the aspects (i) impact of configurations, (ii) impact of metadata separation on block devices, e. g., DM and SSD, with the proposed tiering approach (cf. Section III-E), and (iii) impact of metadata separation on NUMA nodes.

*(i) jemalloc and UMap Configurations.* jemalloc's retain option has significant impact on the virtual memory space consumption (cf. Finding (1)) as well as processing latency (cf. Finding (2)). The UMap page size has to be selected workload-specific due to its huge impact on query runtime (cf. Finding (3)).

*(ii) Metadata Separation on Block Devices.* In jemalloc, metadata has a marginal footprint compared to the stored application data (cf. Finding (4)). The benchmarks on analytical database workloads and the more write-heavy data migration showed no significant improvement or decay in terms of performance on external storage such as DM and SSDs, when used via the proposed tiering approach utilizing UMap (cf. Findings (5)–(6)). Those small performance changes can probably be considered noise, as even the same binary of a complex system (such as Hyrise) can vary in performance by up to 5% [30]. Moreover, through our micro-benchmarks on sequential memory allocations, we observed a marginal performance degradation for metadata separation (cf. Finding (7)). However, our experiments did not suggest a cause for the increased runtime, which requires dedicated investigations in future work.

*(iii) Metadata Separation on NUMA Nodes.* When placing application data on remote NUMA nodes, instead of external block-based devices (i. e., UMap is not used), our experiments showed a slight performance improvement for metadata separation (cf. Finding (8)).

## V. Conclusion

In this work, we studied the performance impact of separating memory allocator metadata from application data, by the example of jemalloc, for the Hyrise in-memory database on disaggregated memory and SSD as secondary memory tiers. The results on read-heavy database workloads (i. e., TPC-H, TPC-DS), write-heavy data migrations, as well as micro-benchmarks on sequential memory allocations show no significant performance impact (cf. Findings (4)–(7)).Similarly, our experiments on remote NUMA nodes showed only slight performance improvements for separate metadata (cf. Findings (8)). However, we found that the specific jemalloc configurations (cf. Findings (1)–(2)) as well as UMap parameters, e. g., the page size, buffer size, and the number of workers, used for configuring the behavior of the user-space page fault handler UMap, is crucial (cf. Finding (3)).

To better understand user-space mapping concepts such as UMap, we will study their configuration for database workloads and their interplay with memory allocators in future work. Moreover, we will explore more sophisticated migration approaches for temporary data and the impact of mixed workloads with random (de-)allocations in the context of allocator metadata separation.

## Acknowledgment

We thank Seagate Technology LLC for their contributions and support of our disaggregated memory experiments and projects. We are indebted to Markus Dreseler for his work on memory tiering in Hyrise, on which our work is built. We thank Felix Eberhardt and Andreas Grapentin for encouraging us to investigate this topic and for insightful discussions.

## REFERENCES


[1] D. Korolija, D. Koutsoukos, K. Keeton, K. Taranov, D. S. Milojicic, and G. Alonso, "Farview: Disaggregated memory with operator off-loading for database engines," in *CIDR*, 2022.

[2] Q. Zhang, Y. Cai, S. Angel, V. Liu, A. Chen, and B. T. Loo, "Rethinking data management systems for disaggregated data centers," in *CIDR*, 2020.

[3] Y. Zhang, C. Ruan, C. Li, J. Yang, W. Cao, F. Li, B. Wang, J. Fang, Y. Wang, J. Huo, and C. Bi, "Towards cost-effective and elastic cloud database deployment via memory disaggregation," *Proc. VLDB Endow.*, vol. 14, no. 10, pp. 1900–1912, 2021.

[4] M. Bielski, C. Pinto, D. Raho, and R. Pacalet, "Survey on memory and devices disaggregation solutions for HPC systems," in *CSE/EUC/DCABES*, 2016, pp. 197–204.

[5] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *ASPLOS*. ACM, 2021, pp. 79–92.

[6] S. Lee, Y. Yu, Y. Tang, A. Khandelwal, L. Zhong, and A. Bhattacharjee, "MIND: in-network memory management for disaggregated data centers," in *SOSP*. ACM, 2021, pp. 488–504.

[7] P. Zuo, J. Sun, L. Yang, S. Zhang, and Y. Hua, "One-sided RDMA-conscious extendible hashing for disaggregated memory," in *ATC*. USENIX Association, 2021, pp. 15–29.

[8] S. Idreos, V. Leis, K.-U. Sattler, and M. Seltzer, "Data structures for modern memory and storage hierarchies (Dagstuhl Seminar 21283)," in *Dagstuhl Reports*, vol. 11, no. 6. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[9] P. S. Rao and G. Porter, "Is memory disaggregation feasible?: A case study with Spark SQL," in *ANCS*. ACM, 2016, pp. 75–80.

[10] N. Pemberton, "Exploring the disaggregated memory interface design space," in *Workshop on Resource Disaggregation (WORD)*, 2019.

[11] D. Durner, V. Leis, and T. Neumann, "On the impact of memory allocation on high-performance query processing," in *DaMoN*, 2019, pp. 21:1–21:3.

[12] D. Durner, V. Leis, and T. Neumann, "Experimental study of memory allocation for high-performance query processing," in *ADMS@VLDB*, 2019, pp. 1–9.

[13] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "RocksDB: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, no. 4, pp. 26:1–26:32, 2021.

[14] M. Dreseler, J. Kossmann, M. Boissier, S. Klauck, M. Uflacker, and H. Plattner, "Hyrise re-engineered: An extensible database system for research in relational in-memory data management," in *EDBT*. OpenProceedings.org, 2019, pp. 313–324.

[15] J. Evans, "Tick tock, malloc needs a clock," in *Applicative 2015*, ser. Applicative 2015. Association for Computing Machinery, 2015.

[16] jemalloc manual. http://jemalloc.net/jemalloc.3.html, Accessed: 2/2022.

[17] I. B. Peng, M. McFadden, E. W. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. B. Gokhale, "UMap: Enabling application-driven optimizations for page management," in *MCHPC@SC*, 2019, pp. 71–78.

[18] I. B. Peng, M. Gokhale, K. Youssef, K. Iwabuchi, and R. Pearce, "Enabling scalable and extensible memory-mapped datastores in userspace," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[19] K. Bhandari, D. R. Chakrabarti, and H. Boehm, "Makalu: fast recoverable allocation of non-volatile memory," in *OOPSLA*, 2016, pp. 677–694.

[20] S. Yu, N. Xiao, M. Deng, F. Liu, and W. Chen, "Redesign the memory allocator for non-volatile main memory," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 49:1–49:26, 2017.

[21] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1166–1177, 2017.

[22] S. A. Zonouz, M. Zhang, P. Sun, L. Garcia, and X. Liu, "Dynamic memory protection via Intel SGX-supported heap allocation," in *DASC/PiCom/DataCom/CyberSciTech*. IEEE Computer Society, 2018, pp. 608–617.

[23] D. Schwalb, M. Faust, J. Wust, M. Grund, and H. Plattner, "Efficient transaction processing for hyrise in mixed workload environments," in *IMDM*, 2014, pp. 112–125.

[24] M. Dreseler, "Storing STL containers on NVM," in *Persistent Programming in Real Life, PIRL*, 2019.

[25] A. Crotty, V. Leis, and A. Pavlo, "Are you sure you want to use mmap in your database management system?" in *CIDR*, 2022.

[26] jemalloc patch: metadata separation. https://github.com/jemalloc/jemalloc/pull/2118, Accessed: 2/2022.

[27] L. Heinzl, B. Hurdelhey, M. Boissier, M. Perscheid, and H. Plattner, "Evaluating lightweight integer compression algorithms in column-oriented in-memory DBMS," in *ADMS@VLDB*, 2021, pp. 26–36.

[28] libnuma manual. https://man7.org/linux/man-pages/man3/numa.3.html, Accessed: 2/2022.

[29] Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html, Accessed: 2/2022.

[30] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!" in *ASPLOS*, 2009, pp. 265–276.