

# Experimental Study of Memory Allocation for High-Performance Query Processing

Dominik Durner  
Technische Universität  
München  
dominik.durner@tum.de

Viktor Leis  
Friedrich-Schiller-Universität  
Jena  
viktor.leis@uni-jena.de

Thomas Neumann  
Technische Universität  
München  
thomas.neumann@tum.de

## ABSTRACT

Somewhat surprisingly, the behavior of analytical query engines is crucially affected by the dynamic memory allocator used. Memory allocators highly influence performance, scalability, memory efficiency and memory fairness to other processes. In this work, we provide the first comprehensive experimental study that analyzes and explains the impact of memory allocation for high-performance query engines. We test five state-of-the-art dynamic memory allocators and discuss their strengths and weaknesses within our DBMS. The right allocator can increase the performance of TPC-DS (SF 100) by 2.7x on a 4-socket Intel Xeon server.

## 1. INTRODUCTION

Modern high-performance query engines are orders of magnitude faster than traditional database systems. As a result, components that hitherto were not crucial for performance may become a performance bottleneck. One such component is memory allocation. Most modern query engines are highly parallel and heavily rely on temporary hash-tables for query processing which results in a large number of short living memory allocations of varying size. Memory allocators therefore need to be scalable and be able to handle myriads of small and medium sized allocations as well as several huge allocations simultaneously. As we show in this paper, memory allocation has become a large factor in overall query processing performance.

New hardware trends exacerbate the allocation issues. The development of multi- and many-core server architectures with up to hundred general purpose cores is a distinct challenge for memory allocation strategies. Due to the increased number of pure computation power, more active queries are possible. Furthermore, multi-threaded data structure implementations lead to dense and simultaneous access patterns. Because most multi-node machines rely on a non-uniform memory access (NUMA) model, requesting memory from a remote node is particularly expensive.

This article is published under a Creative Commons Attribution License (<http://creativecommons.org/licenses/by/3.0/>), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and ADMS 2019. *10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'19)*, August 26, 2019, Los Angeles, California, CA, USA.

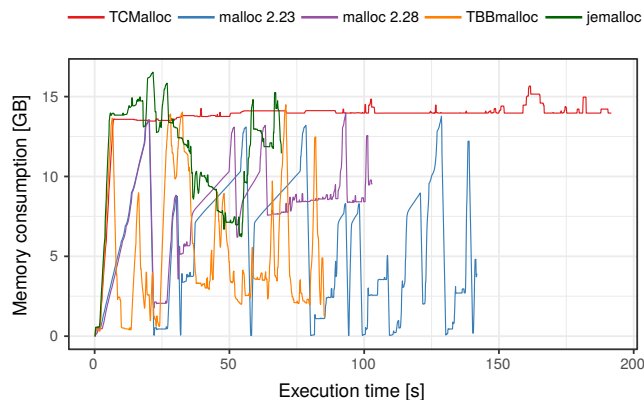


Figure 1: Execution of a given query set on TPC-DS (SF 100) with different allocators.

Therefore, the following goals should be accomplished by a dynamic memory allocator:

*Performance:* Minimize the overhead for malloc and free.

*Scalability:* Reduce overhead for multi-threaded allocs.

*Memory Fairness:* Give freed memory back to the OS.

*Memory Efficiency:* Reduce memory consumption.

In this paper, we perform the first comprehensive analysis that highlights and explains the impact of memory allocation in modern database systems. We evaluate different approaches to the aforementioned dynamic memory allocator requirements. Although memory allocation is on the critical path of query processing, no empirical study on different dynamic memory allocators for in-memory database systems has been conducted [1].

Figure 1 shows the effects of different allocation strategies on TPC-DS with scale factor 100. We measure memory consumption and execution time with our multi-threaded database system on a 4-socket Intel Xeon server. In this experiment, our DBMS executes the query set sequentially using all available cores. Even this relatively simple workload already results in significant performance and memory usage differences. Our database linked with jemalloc can reduce the execution time to  $\frac{1}{2}$  in comparison to linking it with the standard malloc of glibc 2.23. Moreover, the used average and peak memory consumption of the allocators vary highly. Although the resident memory consumption seems high for TCMalloc, it already gives back the memory to the operating

system lazily. Consequently, the allocation strategy is crucial to the performance and memory consumption behavior of in-memory database systems.

The remainder of this paper is structured as follows: After discussing related work in Section 2, we describe the used allocators and their most important design details in Section 3. Section 4 highlights important properties of our research DBMS “umbra” and analyzes the executed workload according to its allocation pattern. Our comprehensive experimental study is evaluated in Section 5. Section 6 summarizes our findings.

## 2. RELATED WORK

Ferreira et al. [9] analyzed dynamic memory allocators for a variety of multi-threaded workloads. However, the study considers only up to 4 cores. Therefore, it is hard to predict the scalability for today’s many-core systems.

In-memory DBMS and analytical query engines, such as HyPer [15], SAP HANA [20], and Quickstep [23] are built to utilize as many cores as possible to speed up query processing. Because these systems rely on allocation-heavy operators (e.g., hash joins, aggregations), a revised experimental analysis on the scalability of the state-of-the-art allocators is needed. In-memory hash joins and aggregations can be implemented in many different ways which can influence the allocation pattern heavily [2, 3, 26, 17].

Some online transaction processing (OLTP) systems try to reduce the allocation overhead by managing their allocated memory in chunks to increase performance for small transactional queries [25, 24, 5]. However, many database systems process both transactional and analytical queries. Therefore, the wide variety of memory allocation patterns for analytical queries needs to be considered as well. Custom chunk memory managers help to reduce memory calls for small allocations but larger chunk sizes trade memory efficiency in favor of performance. Thus, our database system uses transaction-local chunks to speed up small allocations. Despite these optimizations, allocations are still a performance issue. Hence, the allocator choice is crucial to maximize throughput.

Preliminary results showed that memory allocation indeed has impact on the performance of query engines [4]. In this study, we analyze and explain the effects of different allocation strategies in order to understand all strengths and weaknesses of current allocators on modern hardware.

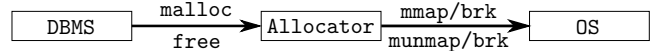
With the development of non-volatile memory (NVM), new allocation requirements were introduced. Foremost, the defragmentation and safe release of unused memory is important since all changes are persistent. New dynamic memory allocators for these novel persistent memory systems have been developed and experimentally studied [22]. However, regular allocators outperform these NVM allocators in most workloads due to fewer memory constraints.

## 3. MEMORY ALLOCATORS

In this section, we discuss the five different allocation strategies used for our experimental study. We explain the basic properties of these algorithms according to memory allocation and freeing. The tested state-of-the-art allocators are available as Ubuntu 18.10 packages. Only the glibc `malloc` 2.23 implementation is part of a previous Ubuntu

package. Nevertheless, this version is still used in many current distributions such as the stable Debian release.

Memory allocation is strongly connected with the operating system (OS). The mapping between physical and virtual memory is handled by the kernel. Allocators need to request virtual memory from the OS. Traditionally, the user program asks for memory by calling the `malloc` method of the allocator. The allocator either has memory available that is unused and suitable or needs to request new memory from the OS. For example, the Linux kernel has multiple APIs for requesting and freeing memory. `brk` calls can increase and decrease the amount of memory allocated to the data segment by changing the program break. `mmap` maps files into memory and implements demand paging such that physical pages are only allocated if used. With anonymous mappings, virtual memory that is not backed by a real file can be allocated within main memory as well. The memory allocation process is visualized below.

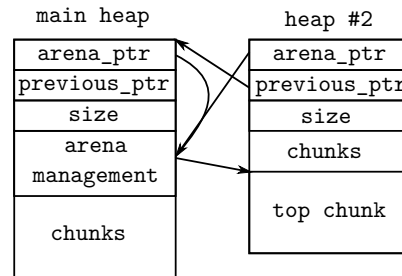


Besides freeing memory directly with the aforementioned calls, the memory allocator can opt to release memory with `MADV_FREE` (since Linux Kernel 4.5). `MADV_FREE` indicates that the kernel is allowed to reuse this memory region. However, the allocator can still access the virtual memory address and either receives the previous physical pages or the kernel provides new zeroed pages. Only if the kernel reassigns the physical pages, new ones need to be zeroed. Hence, `MADV_FREE` reduces the number of pages that require zeroing compared to regular freeing since the old pages might be reused by the same process.

### 3.1 malloc 2.23

The standard glibc `malloc` implementation is derived from `ptmalloc2` which originated from `dlmalloc` [19]. It uses chunks of various sizes that exist within a larger memory region known as the heap. `malloc` uses multiple heaps that grow within their address space.

For handling multi-threaded applications, `malloc` uses arenas that consist of multiple heaps to speed up simultaneous accesses. At program start the main arena is created and additional arenas are chained with previous arena pointers. The arena management is stored within the main heap of that arena. Additional arenas are created with `mmap` and are limited to eight times the number of CPU cores. For every allocation, an arena-wide mutex needs to be acquired. Within arenas free chunks are tracked with free-lists. Only if the top chunk (adjacent unmapped memory) is large enough, memory will be returned to the OS.



`malloc` is aware of multiple threads but no further multi-threaded optimizations, such as thread locality or NUMA

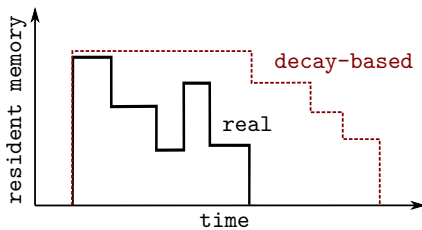
awareness, is integrated. It assumes that the kernel handles these issues.

### 3.2 malloc 2.28

A thread-local cache (tcache) was introduced with glibc v2.26 [18]. This cache requires no locks and is therefore a fast path to allocate and free memory. If there is a suitable chunk in the tcache for allocation, it is directly returned to the caller bypassing the rest of the malloc routine. The deletion of a chunk works similarly. If the tcache has a free slot, the chunk is stored within it instead of immediately freeing it.

### 3.3 jemalloc 5.1

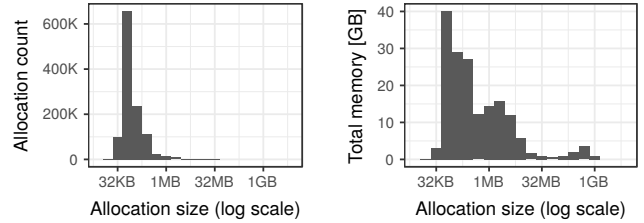
`jemalloc` was originally developed as scalable and low fragmentation standard allocator for FreeBSD. Today, it is used as the default allocator for a variety of applications such as Facebook, Cassandra and Android. It differentiates between three size categories - small ( $< 16\text{KB}$ ), large ( $< 4\text{MB}$ ) and huge. These categories are further split into different size classes. It uses arenas that act as completely independent allocators. Arenas consist of chunks that allocate multiples of 1024 pages (4MB). `jemalloc` implements low address reuse for large allocations to reduce fragmentation. Low address reuse, which basically scans for the first large enough free memory region, has similar theoretical properties as more expensive strategies such as best-fit. `jemalloc` tries to reduce zeroing of pages by deallocating pages with `MADV_FREE` instead of unmapping them. Most importantly, `jemalloc` purges dirty pages decay-based with a wall-clock (since v4.1) which leads to a high reuse of recently used dirty pages. The decay-based reclaiming frees pages that were not accessed for a certain time which is illustrated in the figure below. Consequently, the unused memory will be purged if not requested anymore to achieve memory fairness [6, 7].



### 3.4 TBBmalloc 2017 U7

Intel’s Threading Building Blocks (TBB) allocator is based on the scalable memory allocator `McRT` [12]. It differentiates between small, quite large, and huge objects. Huge objects ( $\geq 4\text{MB}$ ) are directly allocated and freed from the OS. Small and large objects are organized in thread-local heaps with chunks stored in memory blocks.

Memory blocks are memory mapped regions that are multiples of the requested object size class and inserted into the global heap of free blocks. Freed memory blocks are stored within a global heap of abandoned blocks. If a thread-local heap needs additional memory blocks, it requests the memory from one of the global heaps. Memory regions are unmapped during coalescing of freed memory allocations if no block of the region is used anymore [16, 14].



(a) By number of allocations. (b) By memory consumption.

Figure 2: Allocations in TPC-DS (SF 100, serial execution).

### 3.5 TCMalloc 2.5

`TCMalloc` is part of Google’s `gperftools`. Each thread has a local cache that is used to satisfy small ( $\leq 256\text{KB}$ ) allocations. Large objects are allocated in a central heap using 8KB pages.

`TCMalloc` uses different allocatable size classes for the small objects and stores the thread cache as a singly linked list for each of the size classes. Medium sized allocations ( $\leq 1\text{MB}$ ) use multiple pages and are handled by the central heap. If no space is available, the medium sized allocation is treated as a large allocation. For large allocations, spans of free memory pages are tracked within a red-black tree. A new allocation just searches the tree for the smallest fitting span. If no span is found, the memory is allocated from the kernel [10].

Unused memory is freed with the help of `MADV_FREE` calls. Small allocations are garbage collected if the thread-local cache exceeds a maximum size. Freed spans are immediately released since the “aggressive decommit” option was enabled (starting with version 2.3) to reduce memory fragmentation [11].

## 4. DBMS AND WORKLOAD ANALYSIS

Decision support systems rely on analytical queries that gather information from a huge dataset by joining different relations for example. In in-memory query engines joins are often scheduled physically as hash joins resulting in a large number of smaller allocations. In the following, we use a database system that uses pre-aggregation hash tables to perform multi-threaded group bys and joins [17]. Our DBMS has a custom transaction-local chunk allocator to speed up small allocations of less than 32KB. We store small allocations in chunks of medium sized memory blocks. Since only small allocations are stored within chunks, the memory efficiency footprint of these small object chunks is marginal. Additionally, the memory needed for tuple materialization is acquired in chunks. These chunks grow as more tuples are materialized. Thus, we already reduce the stress on the allocator significantly while preserving memory efficiency.

The TPC-H and TPC-DS benchmarks were developed to standardize common decision support workloads [21]. Because TPC-DS contains a larger workload of more complex queries than TPC-H, we focus on TPC-DS in the following. As a result, we expect to see a more diverse and challenging allocation pattern. TPC-DS describes a retail product supplier with different sales channels such as stores and web sales.

In the following, we statistically analyze the allocation pattern for TPC-DS executing all queries without rollup and

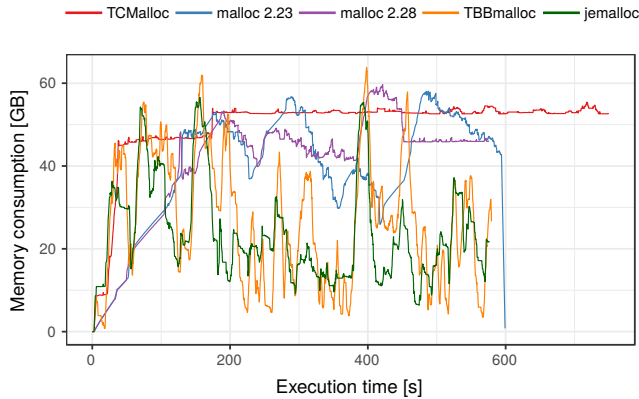


Figure 3: Memory consumption over time (4-socket Xeon,  $\lambda=1.25$  q/s, SF 100).

window functions. Note that the specific allocation pattern depends on the discussed implementation choices of the join and group by operators.

Figure 2 shows the distribution of allocations in our system for TPC-DS with scale factor 100. The most frequent allocations are in the range of 32KB to 512KB. Larger memory regions are needed to create the bucket arrays of the chaining hash tables. The huge amount of medium sized allocations are requested to materialize tuples using the aforementioned chunks.

Additionally, we measure which operators require the most allocations. The two main consumer are group by and join operators. The percentage of allocations per operator for a sequential execution of queries on TPC-DS (SF 100) is shown in the table below:

	Group By	Join	Set	Temp	Other
By Size	61.2%	25.7%	4.3%	8.4%	0.4%
By Count	77.9%	11.7%	8.5%	1.8%	0.1%

To simulate a realistic workload, we use an exponentially distributed workload to determine query arrival times. We sample from the exponential distribution to calculate the time between two events. An independent constant average rate  $\lambda$  defines the waiting time of the distribution. In comparison to a uniformly distributed allocation pattern, the number of concurrently active transactions varies. Thus, a more diverse and complex allocation pattern is created. The events happen within an expected time interval value of  $1/\lambda$  and variance of  $1/\lambda^2$ . The executed queries of TPC-DS are uniformly distributed among the start events. Due to the usage of the same query-set and query arrival rates, we are able to test all allocators on the same real-world alike workloads.

Our main-memory query engine allows up to 10 transactions to be active simultaneously. If more than 10 transactions are queried, the transaction is delayed by the scheduler of our DBMS until the active transaction count is decreased. Due to intra-query parallelization, all cores of the system are utilized even with a reduced number of concurrent transactions.

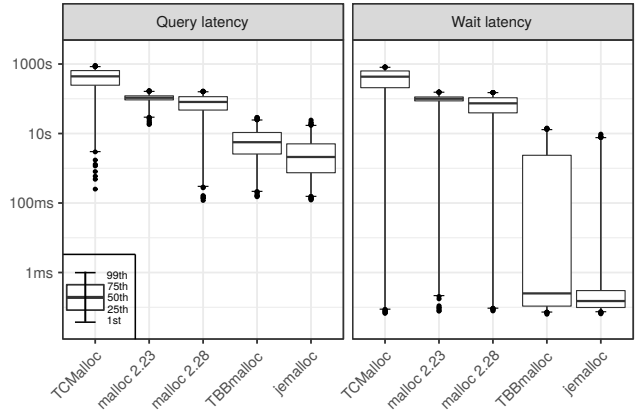


Figure 4: Total query latency and wait time (4-socket Xeon,  $\lambda=1.25$  q/s, SF 100).

## 5. EVALUATION

In this section, we evaluate the five allocators on three hardware architectures with different workloads. We show that the approaches have significant performance and scalability differences. Additionally, we compare the allocator implementations according to their memory consumption and release strategies which shows memory efficiency and memory fairness to other processes.

We test the allocators on a 4-socket Intel Xeon E7-4870 server (60 cores) with 1 TB of main memory, an AMD Threadripper 1950X (16 cores) with 64 GB main memory (32 GB connected to each die region), and a single-die Intel Core i9-7900X (10 cores) server with 128 GB main memory. All three systems support 2-way hyperthreading. These three different architectures are used to analyze the behavior in terms of the allocators' ability to scale on complex multi-socket NUMA systems. To avoid effects of loading the database into main memory, we use the second run of each workload to generate the execution graphs.

This section begins with a detailed analysis of a realistic workload on the 4-socket server. We continue our evaluation by scheduling a reduced and increased number of transactions to test the allocators' performance in varying stress scenarios. An experimental analysis on the different architectures gives insights on the scalability of the five malloc implementations. An evaluation of the memory consumption and the memory fairness to other processes concludes this section.

### 5.1 Memory Consumption and Query Latency

The first experiment measures an exponentially distributed workload to simulate a realistic query arrival pattern on the 4-socket Intel Xeon server. Figure 3 shows the memory consumption over time for TPC-DS (SF 100) and a constant query arrival rate of  $\lambda = 1.25$  q/s. Although the same workload is executed, very different memory consumption patterns are measured. TBBmalloc and jemalloc release most of their memory after query execution. Both malloc implementations hold a minimum level of memory which increases over time. TCMalloc releases its memory accurately with MADV\_FREE which is not visible by tracking the system provided resident memory of the database process. Due to huge performance degradations for tracking the lazy free-

Allocator	Local	Remote	Total	Page Fault
malloc 2.28	63B	172B	236B	41M
	100%	100%	100%	100%
jemalloc	120%	97%	103%	400%
TBBmalloc	121%	97%	103%	516%
TCMalloc	106%	105%	104%	153%
malloc 2.23	103%	100%	101%	139%

Table 1: NUMA-local and NUMA-remote DRAM accesses and OS page faults (4-socket Xeon,  $\lambda=1.25$  q/s, SF 100).

ing of memory, we show the described release behavior of TCMalloc in Section 5.4 separately. However, the overall performance is reduced due to an increased number of kernel calls.

For an in-depth performance analysis, the query and wait latencies of the individual queries are visualized in Figure 4. Although the overall runtime is similar between different allocators, the individual query statistics show that only jemalloc has minor wait latencies. TBBmalloc and jemalloc are mostly bound by the actual execution of the query. On the contrary, both glibc malloc implementations and TCMalloc are dominated by the wait latencies. Thus, our database linked with the later allocators cannot process the queries fast enough to prevent query congestion. Query congestion results from the bound number (10) of concurrently scheduled transactions that our scheduler allows to be executed simultaneously.

Because of these huge performance differences, we measure NUMA relevant properties to highlight advantages and disadvantages of the algorithms. Table 1 shows page faults, local and remote DRAM accesses. All measurements are normalized to the current standard glibc malloc 2.28 implementation for an easier comparison. The two fastest allocators have more local DRAM accesses and significantly more page faults, but have a reduced number of remote accesses. Note that the system requires more remote DRAM accesses due to NUMA-interleaved memory allocations of the TPC-DS base relations. Thus, the highly increased number of local accesses change the overall number of accesses only slightly. Minor page faults are not crucially critical since both jemalloc and TBBmalloc release and acquire their pages frequently. These minor page faults can be handled without disk I/O such as a request for a zeroed page [8]. Consequently, remote accesses for query processing are the major performance indicator. Because TCMalloc reuses MADV\_FREE pages and therefore avoids unnecessary zeroing of pages, the number of minor page faults remains small.

## 5.2 Performance with Varying Stress Levels

In the previous workload, only two allocators were able to efficiently handle the incoming queries. This section evaluates the effects for a varying constant rate  $\lambda$ . We analyze two additional workloads that use the rates  $\lambda = 0.63$  and  $\lambda = 2.5$  queries per second. Thus, we respectively increase and decrease the average waiting time before a new query is scheduled by a factor of 2.

Figure 5 shows the query latencies of the three workloads. The results for the reduced and increased waiting times confirm the previous observations. The allocators

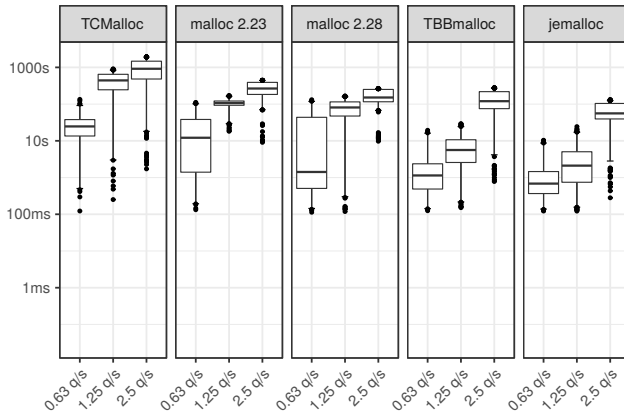


Figure 5: Query latency distributions for different query rates (4-socket Xeon, SF 100).

have the same respective latency order in all three experiments. jemalloc performs best again for all workloads, followed by TBBmalloc.

All query latencies are dominated by the wait latencies in the  $\lambda = 2.5$  workload due to frequent congestions. With an increased waiting time ( $\lambda = 0.63$ ) between queries, the glibc malloc 2.28 implementation is able to reduce the median latency to a similar level as TBBmalloc. However, the query latencies within the third quantile vary vastly. TCMalloc and malloc 2.23 are still not able to process the queries without introducing long waiting periods.

## 5.3 Scalability

After analyzing the allocators' performance on the 4-socket Intel Xeon architecture, this section focuses on the scalability of the five dynamic memory allocators. We execute an exponentially distributed workload with TPC-DS (SF 10) on the NUMA-scale 60 core Intel Xeon server, the 16 core AMD Threadripper (two die regions), and the single-socket 10 core Intel Skylake X.

Figure 6 shows the memory consumption during the workload execution. Since the AMD Threadripper has a very similar memory consumption pattern to the Intel Skylake X, we only show the 4-socket Intel Xeon and the single-socket Intel Skylake. Most notable are the differences of both glibc malloc implementations. These two allocators have a very long initialization phase on the 4-socket system, but are able to allocate their initial memory as fast as the other ones on the single-socket system. Due to more cores and the resulting different access pattern, the decay-based deallocation pattern of jemalloc differs slightly in the beginning. However, jemalloc's decay-based purging reduces the memory consumption on both architectures considerably. TCMalloc cannot process all queries in the same time frame as the other allocators on the 4-socket system whereas it finishes at the same time on Skylake.

Especially the query latencies differ vastly between the architectures. In Figure 7, we show the latencies for the  $\lambda = 6$  q/s workload. The more cores are utilized, the larger are the latency differences between the allocators. On the single-socket Skylake X, all the allocators have very similar performance. Besides having more cores, AMD's Threadripper uses two memory regions which requires a more ad-

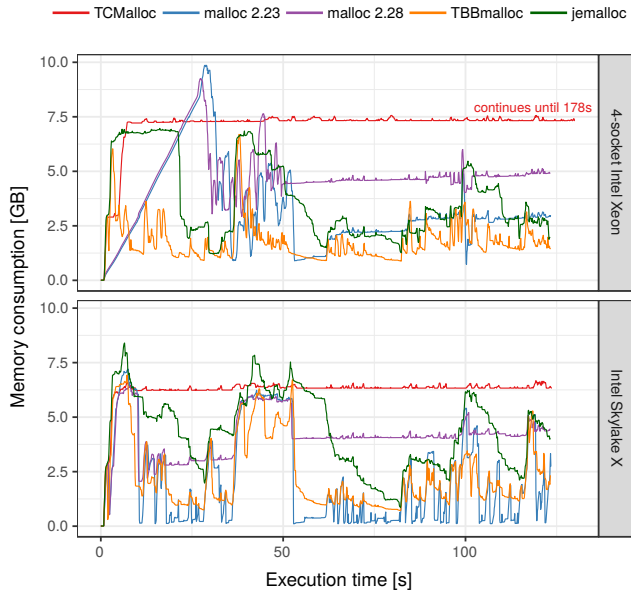


Figure 6: Memory consumption over time ( $\lambda=6$  q/s, SF 10).

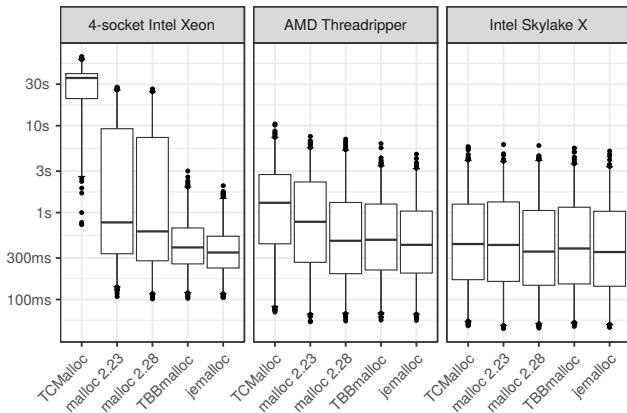


Figure 7: Query latencies ( $\lambda=6$  q/s, SF 10).

vanced placement strategy to obtain fast accesses. In particular, **TCMalloc** and **malloc 2.23** without a thread-local cache have a reduced performance. The latency variances are reduced on the Threadripper but the overall latencies are worse in comparison to the Skylake architecture.

Yet, the most interesting behavior is introduced by the multi-socket Intel Xeon. It has both the best and worst overall query performance. **jemalloc** and **TBBmalloc** execute the queries with the overall lowest latencies and smallest variance. On the other hand, **TCMalloc** is worse by more than 10x in comparison to any other allocator. Both glibc implementations have a similar median performance but incur high variance such that a reliable query time prediction is impossible.

To substantiate our findings that remote accesses and large amount of cores are the major drivers, we evaluate the queries on a single-socket of the Intel Xeon server. We use **numactl** to bind the memory to the same region as the 30

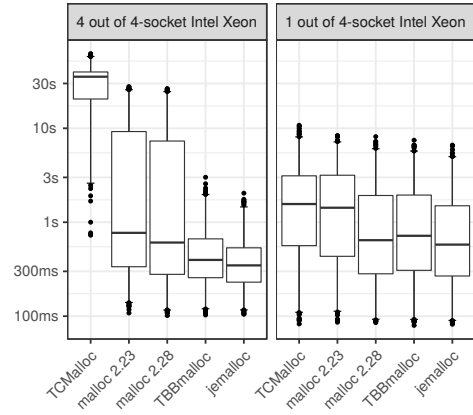


Figure 8: Query latencies ( $\lambda=6$  q/s, SF 10).

Allocator	peak total request		average total request	
	measured <sup>1</sup>	measured	measured	measured
<b>TCMalloc</b>	55.7 GB	58.1 GB	17.8 GB	53.7 GB
<b>malloc 2.23</b>	61.4 GB	61.0 GB	26.2 GB	41.3 GB
<b>malloc 2.28</b>	61.5 GB	62.6 GB	20.2 GB	42.5 GB
<b>TBBmalloc</b>	55.7 GB	55.7 GB	15.9 GB	27.9 GB
<b>jemalloc</b>	58.6 GB	59.4 GB	11.1 GB	24.7 GB

Table 2: Memory usage (4-socket Xeon,  $\lambda=1.25$  q/s, SF100).

threads used for execution. Figure 8 shows that the single-socket execution on our large system behaves similarly to the single-socket Skylake X.

The experiments show that both **jemalloc** and **TBBmalloc** are able to scale to large systems with many cores. **TCMalloc**, on the other hand, has significant performance loss on larger servers.

To validate our findings, we evaluate a subset of the queries on MonetDB 11.31.13 [13]. We observe a performance boost by using **jemalloc** on MonetDB; however, the differences are smaller because our DBMS parallelizes better and thus utilizes more cores.

## 5.4 Memory Fairness

Because DBMS often run alongside other processes on a single server, it is necessary that the query engines are fair to other processes. In particular, the memory consumption and the memory release pattern are good indicators of the allocators’ memory fairness.

Our DBMS is able to track the allocated memory regions with almost no overhead. Hence, we can compare the measured process memory consumption with the requested one. The used memory differs between the allocators due to the performance and scalability properties although we execute the same set of queries. Table 2 shows the peak and average memory consumption for the  $\lambda = 1.25$  q/s workload (SF 100) on the 4-socket Intel Xeon. We use the requested peak and average total memory consumption as the memory efficiency indicator of the allocators. The peak memory

<sup>1</sup>Due to chunk-wise allocation with unfaulted pages and measurement delays the measured amount of memory can be slightly smaller than the requested one.

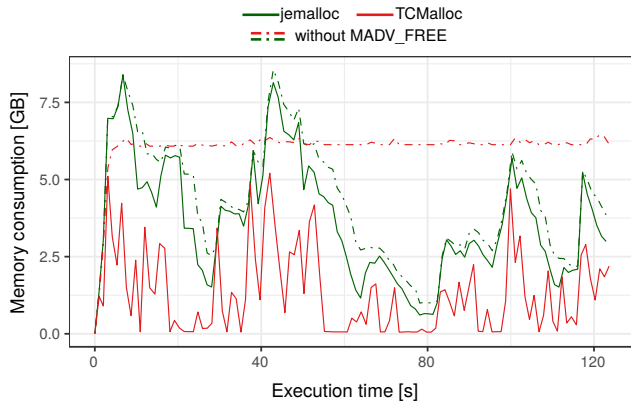


Figure 9: Memory consumption over time with subtracted `MADV_FREE` pages ( $\lambda=6$  q/s, SF 10).

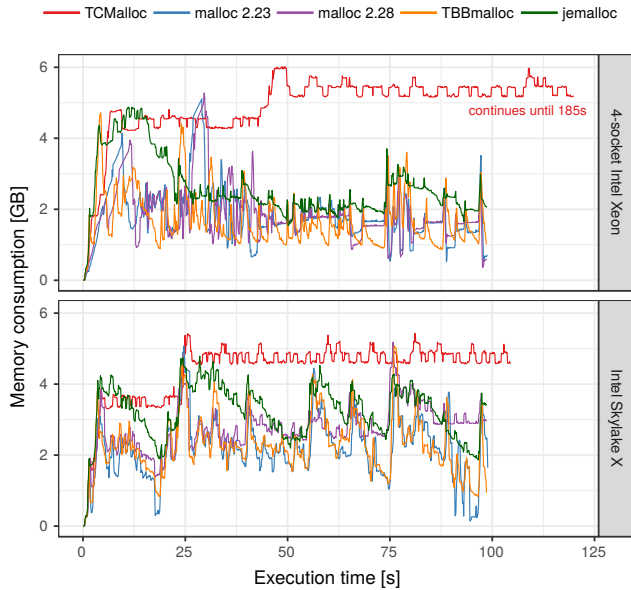


Figure 10: Memory consumption ( $\lambda=12$  q/s, TPC-H SF 10).

consumption is similar for all tested allocators. On the contrary, the average consumption is highly dependant on the used allocator. Both glibc `malloc` implementations demand a large amount of average memory. `jemalloc` requires less average memory than `TBBmalloc`. However, the DBMS requested average memory is also higher for the allocators with increased memory usage. The higher average consumption results from an overall shorter execution time. Although the consumption of `TCMalloc` seems to be higher, it actually uses less memory than the other allocators. This results from the direct memory release with `MADV_FREE`. The tracking of `MADV_FREE` calls on the 4-socket Intel Xeon is very expensive and would introduce many anomalies for both performance and memory consumption. Therefore, we analyze the madvise behavior on the single-socket Skylake X that is only affected slightly by the `MADV_FREE` tracking. The memory consumption with the  $\lambda = 6$  q/s workload (SF 10) is shown

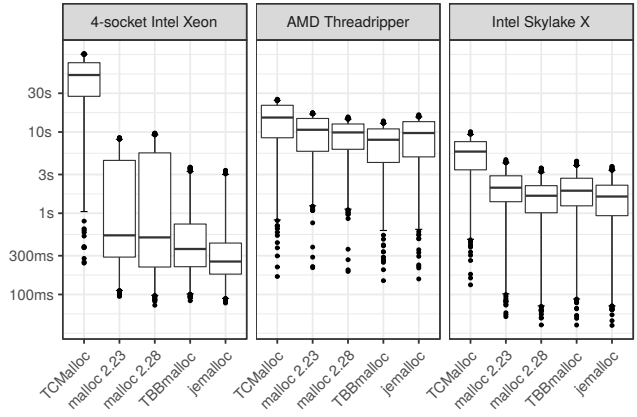


Figure 11: Query latencies ( $\lambda=12$  q/s, TPC-H SF 10).

in Figure 9. The only two allocators that use `MADV_FREE` to release memory are `jemalloc` and `TCMalloc`. The measured average memory curve of `TCMalloc` follows the DBMS required curve almost perfectly. `jemalloc` has a 15% reduced consumption if the `MADV_FREE` pages are subtracted from the used memory.

## 5.5 Scalability with TPC-H

To validate our scalability results on analytical queries, we further analyze the TPC-H benchmark. We execute an exponentially distributed TPC-H (SF 10) workload on the NUMA-scale Intel Xeon server, the AMD Threadripper, and the Intel Skylake X.

Figure 10 shows the memory consumption over time executing the TPC-H workload. Due to an increased query rate, the allocation pattern is less smooth with TPC-H than with TPC-DS. `jemalloc` purges pages according to its decay strategy and the `malloc` implementations need an initial start-up phase. Thus, the allocation pattern of TPC-H is similar to the pattern of TPC-DS.

The query latencies for TPC-H are shown in Figure 11. Similar to our previous findings, `jemalloc` and `TBBmalloc` scale best. The allocators show only on the large Intel Xeon system huge performance differences.

## 5.6 Raw Allocation

Because our DBMS uses a custom chunkwise allocator with free-lists to speed up small allocations, we also evaluate the experiments without this 2-layered allocation setup. Every allocation request gets directly forwarded to the allocator instead of using the DBMS small allocation logic.

Figure 12 shows the query latencies of the three workloads for TPC-DS (SF 100) with direct allocator usage. `jemalloc` outperforms the other allocators. In comparison to the 2-layered allocation process, `TBBmalloc` cannot efficiently process the medium sized workload anymore. The other allocators behave similar, however the query latencies are increased. Overall, the usage of an additional small allocation is beneficial to reduce query processing time.

The memory consumption over time on the Intel Xeon and the Intel Skylake is shown in Figure 13. Interestingly, the release strategy of `TBBmalloc` is very different to the experiments with our additional small allocation logic. Regardless of the used system, the memory is only returned to

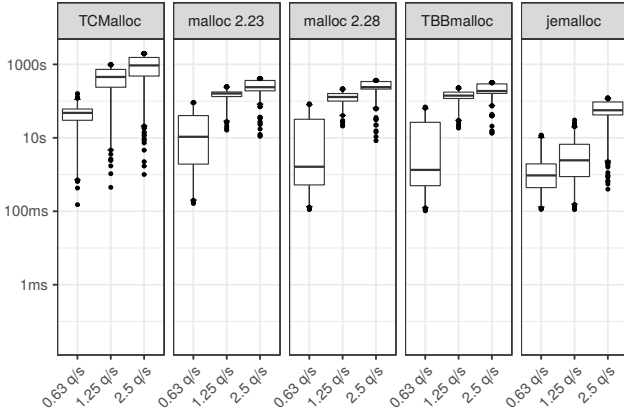


Figure 12: Query latency distributions for different query rates (4-socket Xeon, SF 100, raw allocation).

the operating system at the end of the execution. **TBBmalloc** stores small allocations in thread-local heaps which are held during the execution of the query set. The other allocators show a similar allocation pattern.


## 6. CONCLUSIONS

In this work, we provided a thorough experimental analysis and discussion on the impact of dynamic memory allocators for high-performance query processing. We highlighted the strength and weaknesses of the different state-of-the-art allocators according to scalability, performance, memory efficiency, and fairness to other processes. For our allocation pattern, which is probably not unlike to that of most high-performance query engines, we can summarize our findings as follows:

	scalable	fast	mem. fair	mem. efficient
TCMalloc	--	~	++	+
malloc 2.23	-	~	+	~
malloc 2.28	~	+	-	~
TBBmalloc	+	+	+	+
jemalloc	++	+	+	+

As a result of this work, we choose **jemalloc** as the standard allocator for our DBMS.

## 7. ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

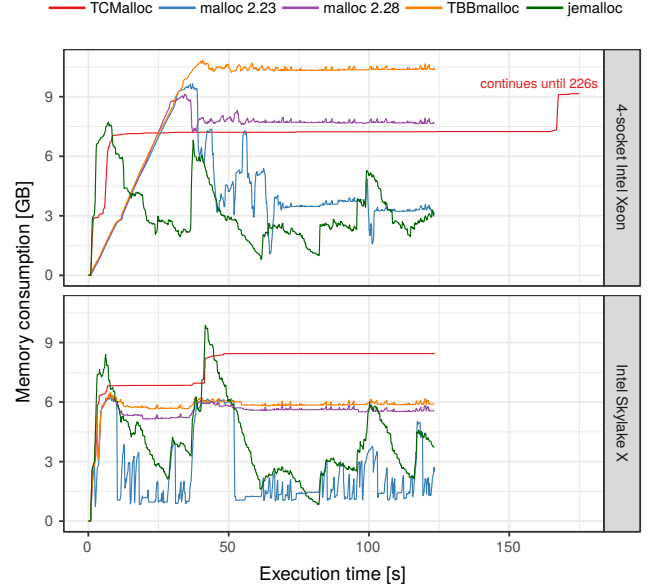


Figure 13: Memory consumption over time ( $\lambda=6$  q/s, SF 10, raw allocation).

## 8. REFERENCES

- [1] R. Appuswamy, A. Anadiotis, D. Porobic, M. Iman, and A. Ailamaki. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *PVLDB*, 11(2):121–134, 2017.
- [2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.
- [3] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48, 2011.
- [4] D. Durner, V. Leis, and T. Neumann. On the impact of memory allocation on high-performance query processing. In *DaMoN*, pages 21:1–21:3, 2019.
- [5] D. Durner and T. Neumann. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *ICDE*, 2019.
- [6] J. Evans. Tick tock, malloc needs a clock [talk]. In *ACM Applicative*, 2015.
- [7] J. Evans. jemalloc changelog. <https://github.com/jemalloc/jemalloc/blob/dev/ChangeLog>, 2018.
- [8] P. Ezolt. A study in malloc: A case of excessive minor faults. In *Linux Showcase & Conference*, 2001.
- [9] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 92–98. IEEE, 2011.
- [10] Google. Tcmalloc documentation. <https://gperftools.github.io/gperftools/tcmalloc.html>, 2007.
- [11] Google. gperftools repository. <https://github.com/>



- gperftools/gperftools/tree/gperftools-2.5.93, 2017.
- [12] R. L. Hudson, B. Saha, A. Adl-Tabatabai, and B. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM*, pages 74–83, 2006.
- [13] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.
- [14] Intel. Threading building blocks repository. [https://github.com/01org/tbb/tree/tbb\\_2017](https://github.com/01org/tbb/tree/tbb_2017), 2017.
- [15] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.
- [16] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.
- [17] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.
- [18] G. C. Library. The gnu c library version 2.26 is now available. <https://sourceware.org/ml/libc-alpha/2017-08/msg00010.html>, 2017.
- [19] G. C. Library. Malloc internals: Overview of malloc. <https://sourceware.org/glibc/wiki/MallocInternals>, 2018.
- [20] N. May, A. Böhm, and W. Lehner. SAP HANA - the evolution of an in-memory DBMS from pure OLAP processing towards mixed workloads. In *BTW*, pages 545–563, 2017.
- [21] R. O. Nambiar and M. Poess. The making of TPC-DS. In *VLDB*, pages 1049–1058, 2006.
- [22] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *PVLDB*, 10(11):1166–1177, 2017.
- [23] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.
- [24] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*, page 7, 2013.
- [25] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.
- [26] Z. Zhang, H. Deshmukh, and J. M. Patel. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*, 2019.