On the Impact of Memory Allocation on 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 analysis on 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.

ACM Reference Format:
Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3329785.3329918

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 can be implemented in many different ways [2, 10, 15, 16]. As a result, a large number of short living memory allocations of varying size are requested. 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. Because most multi-node machines rely on a non-uniform memory access (NUMA) model, requesting memory from a remote node is particularly expensive.

In this paper, we perform the first comprehensive study of memory allocation in modern database systems. 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]. 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 [4, 13, 14]. However, most 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.

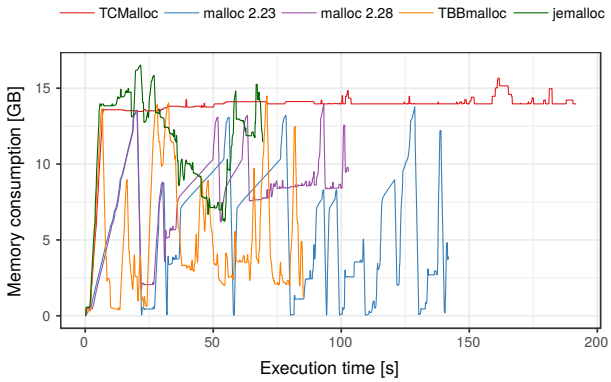

Figure 1: Execution of a given query set on TPC-DS (SF 100) with different allocators.

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. On the other hand, `jemalloc` has the highest memory consumption and does not release memory directly after execution of the query. Consequently, the allocation strategy is crucial to the performance and memory consumption behavior of in-memory database systems.

2 EXPERIMENTAL ANALYSIS

The full version of the experimental analysis can be found in [3].

For the experimental evaluation, we use a database system that uses pre-aggregation hash tables to perform multi-threaded group bys and joins [10]. Our DBMS has a custom transaction-local chunk allocator to speed up small allocations of less than 32KB. Since only small allocations are stored within medium-sized chunks, the memory efficiency footprint of these small object chunks is marginal. Additionally, the memory needed for tuple materialization is acquired in chunks that grow as more tuples are materialized. Thus, we already reduce the stress on the allocator.

To simulate a realistic workload, we use an exponentially distributed workload to determine query arrival times. We sample

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

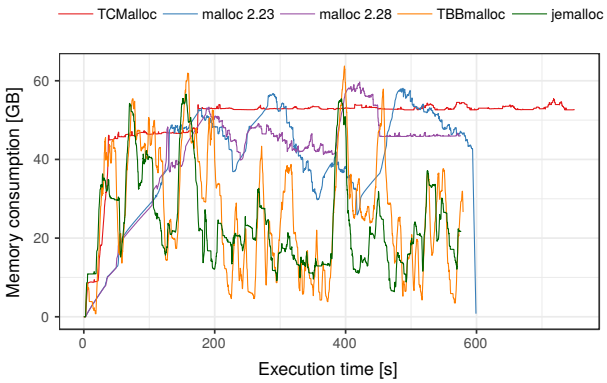

Figure 2: Memory consumption over time (4-socket Xeon, $\lambda = 1.25$ q/s, SF 100).

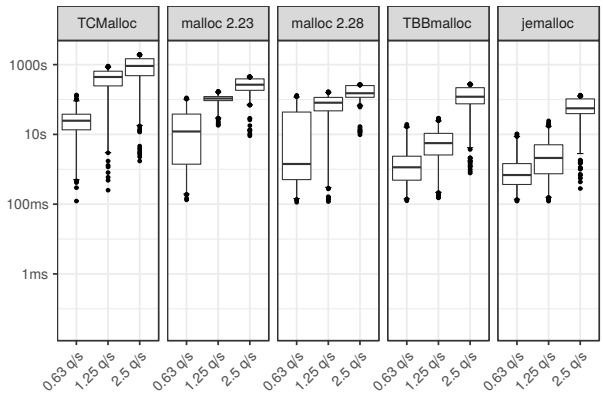

Figure 3: Query latency distributions for different query rates (4-socket Xeon, SF 100).

from the distribution to calculate the time between two events. An independent constant average rate λ defines the waiting time of the distribution. The executed queries of TPC-DS are uniformly distributed among the start events. Our 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.

Figure 2 shows the memory consumption over time for TCP-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` [8, 9] and `jemalloc` [5, 6] release most of their memory after query execution. Both `malloc` implementations [11, 12] hold a minimum level of memory which increases over time. `TCMalloc` [7] releases its memory accurately with `MADV_FREE` which is not visible by tracking the system provided resident memory.

We analyze two additional workloads that use the rates $\lambda = 0.63$ and $\lambda = 2.5$ queries per second. Figure 3 shows the query latencies of the three workloads. The allocators 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

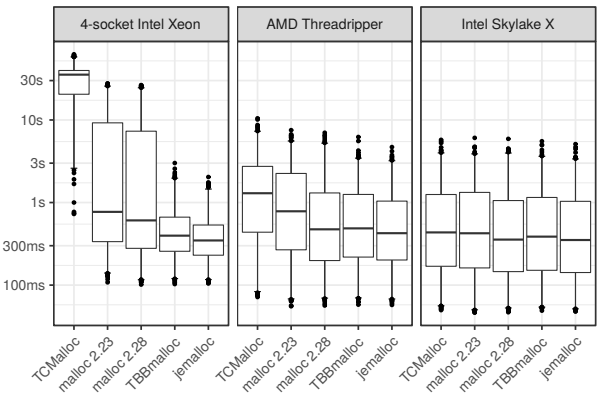

Figure 4: Query latencies ($\lambda = 6$ q/s, SF 10).

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.

In Figure 4, we show the latencies for the $\lambda = 6$ q/s workload on different hardware architectures. 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 advanced placement strategy to obtain fast accesses. In particular, `TCMalloc` and `malloc 2.23` without a thread-local cache have a reduced performance. Yet, the most interesting behavior is introduced by the multi-socket Intel Xeon. `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.

3 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 use `jemalloc` as the standard allocator for our DBMS.

REFERENCES

[1] Raja Appuswamy, Angelos Anadiotis, Danica Porobic, Mustafa Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads. *PVLDB* 11, 2 (2017), 121–134.

[2] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. 37–48.

[3] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. *CoRR* (2019). https://arxiv.org/pdf/1905.01135

[4] Dominik Durner and Thomas Neumann. 2019. No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System. In *ICDE*.

[5] Jason Evans. 2015. Tick Tock, Malloc Needs a Clock [Talk]. https://dl.acm.org/citation.cfm?id=2742807. In *ACM Applicative*.

[6] Jason Evans. 2018. jemalloc ChangeLog. https://github.com/jemalloc/jemalloc/blob/dev/ChangeLog. (2018).

[7] Google. 2007. TCMalloc Documentation. https://gperftools.github.io/gperftools/tcmalloc.html. (2007).

[8] Intel. 2017. Threading Building Blocks Repository. https://github.com/01org/tbb/tree/tbb_2017. (2017).

[9] Alexey Kukanov and Michael J Voss. 2007. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal* 11, 4 (2007).

[10] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*. 743–754.

[11] GNU C Library. 2017. The GNU C Library version 2.26 is now available. https://sourceware.org/ml/libc-alpha/2017-08/msg00010.html. (2017).

[12] GNU C Library. 2018. Malloc Internals: Overview of Malloc. https://sourceware.org/glibc/wiki/MallocInternals. (2018).

[13] Radu Stoica and Anastasia Ailamaki. 2013. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*. 7.

[14] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. 18–32.

[15] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. 2019. Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities. In *CIDR*.

[16] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. 362–373.

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).