Cooperative Memory Management for Table and Temporary Data

Robert Lasch
TU Ilmenau, SAP SE
Germany, Walldorf
robert.lasch@tu-ilmenau.de

Thomas Legler
SAP SE
Germany, Walldorf
thomas.legler@sap.com

Norman May
SAP SE
Germany, Walldorf
norman.may@sap.com

Bernhard Scheirle
SAP SE
Germany, Walldorf
bernhard.scheirle@sap.com

Kai-Uwe Sattler
TU Ilmenau
Germany, Ilmenau
kus@tu-ilmenau.de

ABSTRACT

The traditional paradigm for managing memory in database management systems (DBMS) treats memory used for caching table data and memory for temporary data as separate entities. This leads to inefficient utilization of the available memory capacity for mixed workloads. With memory being a significant factor in the costs of operating a DBMS, utilizing memory as efficiently as possible is highly desirable. As an alternative to the traditional paradigm, we propose managing the entire available memory in a cooperative manner to achieve better memory utilization and consequently higher cost–effectiveness for DBMSs. Initial experimental evaluation of cooperative memory management using a prototype implementation shows promising results and leads to several interesting further research directions.

ACM Reference Format:
Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. 2023. Cooperative Memory Management for Table and Temporary Data. In *1st Workshop on Simplicity in Management of Data (SiMoD '23), June 23, 2023, Bellevue, WA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3596225.3596230

1 INTRODUCTION

Main memory requirements are a significant factor for the cost of operating a DBMS. In systems running analytical or mixed workloads, while many individual consumers require memory, the majority of the memory capacity is required by two consumers: First, *table data* needs to be loaded and held in memory to read and update it. Second, running complex analytical queries requires memory for *temporary data*, e.g., join hash tables or sort buffers.

As pure in–memory systems are uneconomical nowadays [6], we focus on disk-based systems. Traditionally, disk-based systems manage the memory used for table data separately from all other memory. They load table data from storage into memory through a buffer manager, which manages a pool of memory to cache table data. The pool's size is user–configured and memory allocated to it cannot be used for other memory consumers like temporary data.

SiMoD '23, June 23, 2023, Bellevue, WA, USA

ACM ISBN 979-8-4007-0783-4/23/06.
https://doi.org/10.1145/3596225.3596230

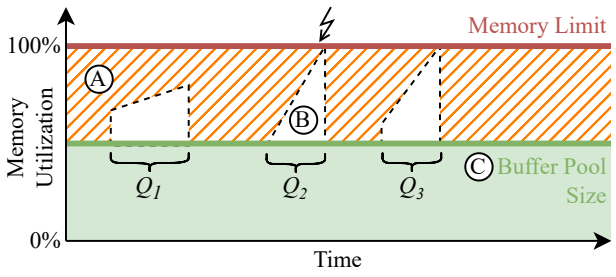

Figure 1: Traditional memory management Ⓐ leaves memory unused, Ⓑ can cause unnecessary query failures or spilling to disk if configured sub–optimally, and Ⓒ requires the user to configure an appropriate buffer pool size.

As shown in Figure 1, the traditional paradigm for managing memory leads to several issues. As the buffer pool size needs to be configured such that enough memory capacity is left to hold the peak amount of temporary data created by any query in the workload, significant parts of the memory capacity left for temporary data will remain unused Ⓐ in the times when smaller or no queries are executing. This is problematic since the unused capacity could be used to cache more table data, improving throughput for the transactional part of the workload. Furthermore, configuring an adequate size for the buffer pool can be time–consuming and error–prone Ⓒ and result in queries failing or having to spill to disk Ⓑ if too little memory capacity is left for temporary data.

In this paper we propose a simple idea to solve these problems: **DBMSs should manage memory used for table and temporary data in a cooperative manner, with flexible amounts of memory assigned to either area.** We call this paradigm *cooperative memory management*, and introduce and evaluate it after first discussing traditional memory management and related work.

2 TRADITIONAL MEMORY MANAGEMENT

The traditional paradigm for memory management in DBMSs designates separate memory regions for table and temporary data. For mixed workloads, where the transactional part of the workload relies on and benefits from caching table data in memory, and the analytical part temporarily requires memory for large intermediate results, this results in two problems:

Inefficient memory utilization. First, the available memory capacity is not used efficiently. This manifests in different cases, as shown in Figure 1, assuming a mixed workload of transactional queries that benefit from cached table data but require little or no memory for temporary data, and analytical queries that require significant memory capacity for temporary data: Ⓐ Capacity remains unused when no or only small analytical queries like Q_1 are running. Using this unused capacity for caching table data could improve overall system throughput. Ⓑ If the memory capacity is not provisioned optimally for the workload, or the size of the buffer pool is not well configured, analytical queries may run out of memory and have to spill data to disk or fail outright, as Q_2 illustrates. If the required memory could instead be temporarily taken from the buffer pool, the level of service of the system could be improved, at the cost of temporarily reducing transactional throughput. With the traditional paradigm, the available memory is ever only fully utilized if one or multiple analytical queries require exactly the amount of memory that is not already used by the buffer pool, as Q_3 shows.

Configuring memory usage. Second, even if inefficient memory utilization is acceptable for a system, another closely related problem is that the traditional paradigm Ⓒ always requires configuring the memory usage of the system in some form. In the general case, manual tuning of the memory allocated to different DBMS components is often trial–and–error in nature and thus can take significant amounts of time and risk interruptions in business operations. Additionally, the memory demands of workloads may also change over time — both on a short sub–second, but also on larger timescales. Consequently, no single configuration can provide optimal performance. Although automated approaches [8] exist that attempt to solve these problems by trying to find the optimal configuration autonomously, they add additional complexity to the system, can still result in sub–optimal configurations, and are prone to require user intervention. Ultimately such approaches also rely on tuning the memory configuration *before* an allocation happens — if tuning has not happened before, the allocation will still fail. We envision allocations to change the memory configuration on-demand as, e.g., query operators allocate memory.

3 RELATED WORK

Our work is motivated by ongoing efforts to improve the cost-efficiency of data management systems by moving away from pure in-memory systems back to disk-based caching systems [6]. While there has been significant progress made in bringing the performance of disk-based systems close to in-memory systems by reducing the overhead of buffer management for table data [4, 5, 7], we find that so far handling the conflicting requirements for table and temporary data has been largely ignored.

Dageville et al. [2] propose a method to assign limited working memory to concurrently executing operators to maximize throughput for analytical queries in Oracle9i, but assume a fixed buffer pool size. Storm et al. [8] adaptively tune the memory allocated to different system components including the buffer pool and temporary memory in IBM DB2, but this ultimately does not allow for on–demand temporary allocations, but requires tuning decisions to be made beforehand, as discussed in Section 2. Both approaches do not prevent underutilization of certain memory areas.

Our prototype implementation of cooperative memory management utilizes the low–overhead nature of the `vmcache` [4] approach to extend the memory managed through the buffer pool from just cached table data to also include intermediate results, breaking down the barrier between these memory consumers that exists in traditional systems. It manages all memory in a way that leaves no memory unused and enables on–demand allocation of temporary data close or up to the memory limit.

4 COOPERATIVE MEMORY MANAGEMENT

The simple solution we propose in this paper to solve the problems discussed in Section 2 is to do away with making the memory usage of different DBMS components configurable altogether. Instead, the system should indifferently serve memory requests for caching table data or for temporary data until the memory limit is reached. The memory limit may be the available DRAM capacity or a user–configured limit. When the limit is reached and new requests arrive, the system should evict cached table data in order to free enough memory to satisfy the request. This way, with enough memory demand, memory is always fully utilized for caching table data when it is not needed for temporary data, but memory for temporary data can still be allocated at any point in time up to the memory limit. This takes the burden of configuring memory areas, e.g., buffer pool size, off of system users, and maximizes the utility gained from the available memory capacity by always using memory not currently used for other purposes to cache table data.

We envision cooperative memory management to be enabled by a central *memory manager* in the system that keeps track of the amount of memory allocated to caching table data and to temporary data, and handles requests for either type of memory. Conceptually, the memory manager combines the functionality of the buffer manager in a traditional system with the ability to also serve memory requests for temporary data. It handles all allocations and deallocations in the system and ensures that the total memory allocated both to caching table data and to temporary data never exceeds the memory limit. In essence, it provides the following interface to the rest of the system:

- *pin(pid)*: Pin a table data page, load the page from storage if not already in memory.
- *unpin(pid)*: Unpin a table data page.
- *allocateTemporary(size)*: Allocate memory for temporary data.
- *freeTemporary(pointer, size)*: Free memory previously used for temporary data, which can be reused for caching table data.

As access to data pages has to be synchronized among threads, the memory manager provides at least two variants of the *pin()* and *unpin()* calls, one for shared read-only locking, and one for exclusive write locking, which can be implemented with low overhead [4].

When serving requests for memory (i.e., *pin()* or *allocateTemporary()* calls), the memory manager can simply allocate new memory using the system allocator as long as the total memory use does not exceed the limit. Once the limit is reached, it has to evict — using some eviction policy — table data pages and free their memory until there is enough space for the new allocation.

The only case where an *allocateTemporary()* call cannot be handled occurs when the combination of the new request, already allocated temporary memory, and pinned data pages exceed the

memory limit. However, this happens far later with the cooperative approach than in a traditional system, where data pages cannot be evicted to make room for temporary data.

The required functionality can be implemented on top of an existing buffer manager by pinning pages exclusively for *allocateTemporary()* calls, and unpinning those pages for *freeTemporary()* calls. This makes it so that buffer frames used for temporary data can be reused for table data after use. Thus, the main obstacles for implementing cooperative memory management in existing systems with a buffer manager are to turn the buffer manager into a memory manager, and then modifying the system to perform — at least — all large temporary allocations through the memory manager. With such an implementation, allocation granularity for temporary data has to be a multiple of the buffer manager's page size. However, as we show in Section 5, more efficient and flexible implementations are possible using virtual memory–based buffer management.

5 EVALUATION

We evaluate cooperative memory management using a prototype[1] that implements a buffer manager similar to the `vmcache` design proposed by Leis et al. [4]. The prototype implements the vision outlined in Section 4 by tracking all memory allocations through the buffer manager. If temporary allocation results in a violation of the memory limit, it simply evicts table data pages until enough memory for the allocation is available before allocating the memory through `jemalloc`. This is possible as page eviction in the `vmcache` approach returns the memory to the OS and thus enables performing other allocations without running out of memory. Likewise, the memory manager tracks the total allocated size for temporary data and makes sure to keep the memory footprint of cached table data combined with the temporary allocation size below the memory limit. Our buffer manager opens the database file using `O_DIRECT`, bypassing the Linux page cache. It uses the clock replacement algorithm to select table data pages for eviction. Data is stored in columnar format and (possibly composite) primary key indexes are supported using B^+–trees, using a page size of 4 KiB.

We also experimented with serving temporary allocations by pinning `vmcache` pages. This involved finding contiguous ranges of unused or evicted pages in `vmcache`'s virtual memory area. We found that our naïve implementation for finding these ranges added too much overhead to each allocation to be feasible. Therefore we instead allocate temporary pages using `jemalloc` in the current prototype. However, exploring temporary allocation by pinning buffer pool pages further is interesting future work as it may allow the buffer manager to also evict temporary data pages and thus result in more flexible memory management.

Setup. We run experiments on a dual–socket system with 38–core/76–thread Intel Xeon Platinum 8368 CPUs and 128 GiB of DDR4–3200 DRAM. The operating system is SLES 15 SP2 (Linux kernel 5.3.18). For storage, we use a 750 GB Intel Optane DC 4800X NVMe SSD, and a 1.92 TB Intel D3–S4620 SATA SSD. We use `numactl` to limit execution to a single socket to avoid NUMA effects.

[1]Code at https://github.com/dbis-ilm/cooperative-memory-management

To compare cooperative memory management to the traditional paradigm, we run benchmarks in two scenarios: First using cooperative memory management, and second emulating a traditional system in the prototype by partitioning memory into fixed–size regions for table and temporary data each taking 50 % of the memory limit. To evaluate the impact of cooperative memory management with respect to disk performance, we repeat all experiments with the database placed either on the SATA or NVMe SSD.

For the benchmark, we run a custom mixed workload on the data of the CH-benCHmark [1] with 100 warehouses and indexes on all primary keys. We configure the prototype to limit memory use to 2 GB. At this memory limit, the hot set of the transactional part of the workload fully fits into memory if all memory is used to cache table data. We run 64 concurrent OLTP streams with a simplified version of the TPC-C transaction profile that includes only the `NewOrder` and `OrderStatus` transactions. As the prototype currently does not support updates and inserts, we run `NewOrder` without inserts and emulate updates by reading any rows that would be updated by update statements. We warm up the buffer pool for 60 s running the OLTP streams, before a 120 s benchmark period. During the benchmark period, we simulate an OLAP query running concurrently to the transactional workload every 30 s. The OLAP query is simulated by allocating 50 % of the available memory capacity for temporary data and waiting for 5 s before freeing that memory again. Note that for this query, the buffer pool sizing of the traditional system represents the optimal configuration, as it leaves exactly enough space to "run" the OLAP query, while maximizing buffer pool size. During the benchmark period, we measure the OLTP throughput over time in MtpmC — million `NewOrder` executions per minute. We also report how much memory is reserved for cached table data and temporary data, respectively, as well as the temporary memory actually being used at any point in time.

Results. Figure 2 shows the results of the experiment with storage on the SATA SSD. Using traditional memory management, OLTP throughput is constant at an average of 1.42 MtpmC over the duration of the benchmark. However, it can be observed that the memory reserved for temporary data (▬) remains unused while the OLAP query (▬) is not running at the 0, 30, 60, and 90 s marks. In contrast, when using cooperative memory management, a higher OLTP throughput of 2.71 MtpmC is reached while the OLAP query is not running, as the entire available memory is used for caching table data in these periods. While the OLAP query is running, throughput is equivalent (1.33 MtpmC) to that of the traditional system, as similar amounts of memory are then used for caching table data in both settings. The larger drop in throughput when the OLAP query allocates memory before reaching the throughput level of the traditional system is a result of the OLTP workers assisting in evicting pages to make room for the requested temporary allocation. Overall this means that the average throughput with cooperative memory management (2.51 MtpmC) is significantly higher than in the traditional setting. Note that the results here are specific to the choice of workload: With less frequent, or more memory-demanding OLAP queries the advantage of cooperative memory management would be even higher, and with a more constant OLAP workload the throughput would approach the traditional setting. To reach the same transactional performance

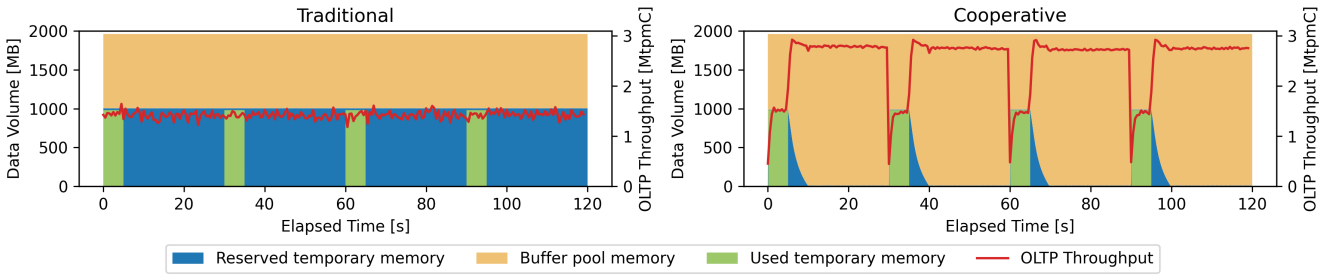

Figure 2: Running a mixed workload with traditional and cooperative memory management (storage on SATA SSD).

with the traditional paradigm as with the cooperative one in this example, the memory limit would have to be increased by nearly 50 %. This shows that cooperative memory management can use the available memory much more efficiently. If temporary variations in transactional throughput in exchange for higher peak throughput and improved memory efficiency can be accepted, it can thus reduce the operating costs of DBMSs running mixed workloads.

When using the NVMe SSD for storage, the same effects can be observed. But here the throughputs with cooperative memory management while running or not running the OLAP query are less different, namely 2.36 MtpmC and 2.75 MtpmC. This is expected as the NVMe SSD can handle the increase in page faults caused by the reduced effective buffer size for table data pages that happens while running the OLAP query more gracefully than the SATA SSD.

Allocation latencies. The cooperative paradigm's disadvantage is that allocating temporary memory can take longer than with the traditional paradigm, as table data may first have to be evicted from memory. In our example, the median allocation time for the OLAP query is 9.50 µs in the traditional setting, but 111 ms in the cooperative setting. However, profiling the allocation in the cooperative setting shows that more than 90 % of the time is spent in `madvise` calls to return physical memory to the OS after evicting table data pages. We believe that using the `exmap` kernel extension proposed by Leis et al. [4] to improve the efficiency of virtual memory manipulation could reduce allocation latencies in the cooperative setting.

While allocation latencies have been shown to have significant impact on query performance in an in–memory setting [3], their impact in a memory–constrained scenario, as we evaluate here, is unclear. To gauge the impact of the cooperative paradigm on real analytical queries, we implement CH-benCHmark *Q09* in the prototype and repeat the previous experiments with the real query instead of the simulated one. The query runs with morsel-driven parallelism using the remaining threads that are not already running the transactional workload. Median execution times with traditional and cooperative memory management respectively are 1.36 s and 1.31 s using the NVMe SSD, and 13.2 s and 12.9 s using the SATA SSD. As these results show, the execution time of the query is limited by the available disk bandwidth, which makes the effect of the increased allocation times with cooperative memory management insignificant. Execution times with cooperative memory management are even slightly faster by roughly 4 % when using the NVMe SSD. This is because more memory is available to cache data accessed by the transactional part of the workload with cooperative memory management, which results in more available disk bandwidth for the analytical query as OLTP queries need to access the disk less frequently. With the SATA SSD this effect is not visible, as the 64 OLTP threads dominate utilization of the much more limited bandwidth compared to the few threads running *Q09*.

6 CONCLUSION

In this paper we present our vision of cooperative memory management for table and temporary data in DBMS. We argue that this new paradigm allows DBMSs to use memory more effectively while also taking the burden of configuring memory areas off of system users, and evaluate the approach running a mixed workload.

While the initial results are promising, our preliminary evaluation is not without limitations and further research is required to evaluate cooperative memory management: First, we find that as a result of evicting table data pages on–demand for temporary allocations, allocation latency for larger allocations does increase in comparison to traditional memory management. Although we find that this has no observable negative effect when running a real query, further investigation is required. Second, the workload considered in our evaluation is read–only so far. Adding writes to the workload may worsen the increase in allocation latency for cooperative memory management, as dirty pages have to be written back to disk before eviction, adding latency. As most systems asynchronously write dirty pages to disk regularly for checkpointing, we do not expect this effect to be a deal–breaker for cooperative memory management, but this also remains to be evaluated.

Further research directions and open questions include:

- How can the paradigm be integrated with the service–level objectives of the DBMS? E.g., under a minimum transactional throughput requirement, it may be desirable to spill some intermediate results to disk in order to avoid evicting all cached table data.
- How do different replacement strategies compare under cooperative memory management? We expect a trade-off between the quality of policy decisions and eviction overhead — and consequently allocation latency for temporary data.
- How should memory be managed for other types of caches in DBMS, e.g., plan or result caches?
- Is it beneficial to take memory state into account for query optimizer decisions?

Finally, we believe that managing memory cooperatively can also be a useful prerequisite for fully utilizing heterogeneous memory like fabric–attached memory, as it allows the memory manager as a central instance to make informed data placement decisions.

REFERENCES

[1] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and F. Michael Waas. 2011. The mixed workload CH-benCHmark. In *DBTest'11*. ACM Press.

[2] Benoit Dageville and Mohammed Zait. 2002. SQL memory management in Oracle9i. In *VLDB'02*. VLDB Endowment, 962–973.

[3] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the impact of memory allocation on high-performance query processing. In *DaMoN'19*.

[4] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-memory assisted buffer management. In *SIGMOD'23*. To appear. https://www.cs.cit.tum.de/fileadmin/w00cfj/dis/_my_direct_uploads/vmcache.pdf

[5] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory data management beyond main memory. In *ICDE'18*. IEEE, 185–196.

[6] David Lomet. 2018. Cost/performance in modern data stores: How data caching systems succeed. In *DaMoN'18*.

[7] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with in-Memory Performance. In *CIDR'20*.

[8] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive self-tuning memory in DB2. In *VLDB'06*. VLDB Endowment, 1081–1092.