On-chip Acceleration for Log-Structured GDBMS

Alexander Baumstark
TU Ilmenau, Germany
alexander.baumstark@tu-ilmenau.de

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

***Abstract*—Emerging memory technologies like HBM or CXL-attached memory provide not only new opportunities but also introduce additional challenges for DBMSs due to their different characteristics compared to usual memory. One of them is additional data movements due to limited capacity or the higher latency of the underlying memory technology, requiring hybrid or adaptive approaches to compensate it effectively. Vast data movements reduce the overall performance of a DBMS since CPU resources are blocked until data transfer is completed. Unoptimized data locality in memory and storage, a known problem for GDBMS, further worsens the problem of data movements and hinders leveraging the full offered performance of underlying memory technology since it results in high random access. The Intel Data Streaming Accelerator, a CPU-integrated, DMA-based accelerator for memory operations, addresses this challenge by offloading data movement operations and freeing CPU resources. Unlike usual DMA engines, it offers a controllable interface to organize and offload memory movements. In this work, we leverage the memory operation accelerator, first to maintain data locality in a graph DBMS (GDBMS) effectively using an LSM-based memory layout approach, and second to provide a hybrid DRAM-HBM solution to accommodate both transactional and analytical workloads.**

***Index Terms*—LSM-Tree, Compaction, Intel DSA, HBM, HTAP, Graph Processing**

I. Introduction

Achieving data locality in DBMS is a key driver of data movements in memory and storage. For high performance, related, frequently accessed, or similar records must be stored close together in memory, storage, or near the CPU, e.g., for NUMA locality, or in distributed settings. Data locality in memory enables leveraging sequential memory access, improving processing with optimal data prefetching from memory, and improving I/O utilization when fetching from storage. An example where data locality is crucial for high performance is graph DBMSs (GDBMS). The resulting performance when processing navigation-like operations is highly dependent on the memory and storage location of topologically neighboring nodes and relationships. However, optimizing data locality is challenging when considering updates and requires additional data movement, often through sorting, to place related data in the neighboring memory regions. When considering new technologies that expand the memory hierarchy like High Bandwidth Memory (HBM), or Compute eXpress Link (CXL) - attached memory, data movements will become additionally a limiting factor for leveraging maximal possible performance since these technologies are either limited in capacity (HBM) which requires data re-organization, or their characteristics favor sequential access due to the higher access latencies (CXL). These memory movements block computation resources and decrease workload processing performance, due to the required on-the-fly data re-distribution.

A hardware-based solution to diminish these data movement costs is offered by Intel's *Data Streaming Accelerator* (DSA) - a CPU-on-chip accelerator for memory operations [1], [2]. This accelerator-like CPU feature can be mainly used to offload simple memory operations such as *memcpy* to transfer data without CPU interference. It can be used to provide the required data beforehand for the CPU, fill up caches to improve processing, or (re-)organize data in the background. We see opportunities for the DSA to support the CPU for data access when querying, restoring data locality, or fulfilling data organization DBMS tasks in the background.

Graphs are typically stored as an adjacency list [3]–[7] to provide fast updates by appending relationships of a node to an appropriate list. In previous work, we showed the applicability of adjacency lists for Persistent Memory [3] but also for Processing-In-Memory [8]. However, data locality is hard to maintain with adjacency lists and updates. The relationships belonging to a node may be distributed across memory, resulting in highly random access when queried. To benefit from the high bandwidth of emerging memory technologies, especially HBM, the DBMS must exhibit **(1)** high parallelism, which requires data organized sequentially and optimal distributed among CPU threads (data parallelism), and **(2)** amortize access latency penalties, e.g. with sequential access. Achieving this in a hybrid transactional and analytical (HTAP) DBMS requires extensive data movements.

The compressed-sparse-row format (CSR) has been considered the optimal structure for graph analytics due to the compact representation of the graph with locality-preserving features [9], but incorporating updates requires rebuilding the whole structure or applying delta approaches [10]. However, rebuilding the CSR periodically and applying deltas can often be achieved by sorting the relationships, similar to sorting and flushing buffers of a log-structured merge (LSM) tree. LSM trees were initially designed to overcome the issue of high write amplification accompanied by updates in DBMS; a similar processing can be adapted to graph storage, as proposed by [11]. We therefore adopt a CSR data organization into the memory and storage layout of a GDBMS by using an LSM-based memory and storage structure. Yet, the sorting and the introduced data movements due to flushing and compaction in an LSM tree are performance-limiting factors increasing latency at processing and blocking computation resources.

Further, analytical graph processing on HBM is advantageous, but transactional workloads would suffer from higher HBM latencies. Hybrid on-the-fly data distribution would result in high data movements between DRAM and HBM.

In this work, we tackle the data movement problems of an LSM-based GDBMS by leveraging Intel DSA technology. The **contributions** of this paper are:

1) We analyze LSM-based memory and storage layout for graph relationships with a fast flushing and compaction strategy leveraging Intel DSA-based data movements.
2) We propose a hybrid DRAM-HBM placement and access strategy for low-latency transactional workloads and high bandwidth analytical workloads, leveraging HBM and Intel DSA-based data movements.

We center our work around Poseidon[1], an HTAP GDBMS optimized for modern memory and storage hierarchies [3].

II. Background & Related Work

Intel DSA. Intel's Data Streaming Accelerator (DSA) is a CPU-integrated memory operation accelerator introduced with Sapphire Rapids. It is integrated into the SoC near the memory controller and LLC and provides its own instruction set [2]. There is up to one instance per NUMA node in an SNC configuration, and it communicates with the host via PCIe. DSA operates through work queues (WQs), which are hardware-limited but can be served by multiple instances in parallel. To execute an operation, a descriptor is allocated and submitted via shared virtual memory (SVM) or memory-mapped I/O (MMIO), avoiding OS kernel overhead. DSA's processing engines handle address translation, data movement, and execution, minimizing CPU load.

DSA Operations. Intel DSA primarily supports memory copy operations, moving data from a source address to a destination, similar to `memcpy`. A dualcast operation extends this by transferring data from a single source to two distinct destinations. Memory filling enables sequentially populating a memory region with an 8-byte pattern. Memory compare operations can directly compare two regions or a region against a fixed pattern. Additional operations handle data integrity fields, CRC checksums, cache flushing, and computing memory region deltas. Operations and their parameters are specified in a previously allocated descriptor. Each of these operations can be processed either synchronously or asynchronously. The descriptors are dispatched to WQs, where the DMA engine executes them. Furthermore, multiple operations can be batched and submitted to the WQs together. The Intel DML library[2] for data operations can be used with Intel DSA accelerators.

DSA Opportunities. Offloading memory operations to Intel DSA presents significant advantages for DBMS in heterogeneous memory environments. By leveraging DMA engines alongside CPU-based transfers, memory bus bandwidth can be fully utilized, boosting data movement throughput. Since data transfers operate independently, they enable asynchronous and parallel execution, reducing cache pollution compared to CPU-only operations [12]. Additionally, DSA facilitates hiding high-latency memory accesses, such as those in HBM or CXL memory, through prefetching. Previous works [12], [13] show that DSA achieves higher throughput with fewer threads, making it well-suited for tasks like LSM compaction and workload-aware query processing. However, challenges remain in thread-DSA coordination, NUMA optimization for optimal performance, limited DSA resources, and operations limited to memory copy or comparison. Despite this, its integration into modern Intel CPUs and performance benefits make it a valuable asset for DBMS optimization.

Graph Locality. HTAP GDBMSs employ adjacency lists to handle read and write workloads by appending new data directly to the corresponding lists [3], [4]. Navigational queries, such as graph traversals, require only traversing a node's relationship list. In our graph engine, Poseidon [3], we implement adjacency lists using relationship identifiers for linkage. With such a list, each node stores the identifiers of its first ingoing and outgoing relationships, while each relationship record maintains pointers to the next and previous relationships, forming a linked list per node. Traversing these lists incurs random access overhead since updates scatter belonging data, posing challenges for memory technologies with higher access latencies, and requiring efficient approaches to maintain topological graph locality with updates.

LSM-Tree. LSM-Trees were originally designed to optimize write-heavy transactional workloads [14] and often in most key-value DBMS [15]. Instead of in-place updates, writes are first buffered in a log or another fast, append-friendly structure. These buffered writes are periodically sorted and flushed to storage, referred to as a sorted run. Over time, multiple sorted runs are merged through a process called compaction, reducing fragmentation and optimizing read performance. Bloom filters are an efficient technique to identify positions where records are stored using multiple hash functions, but are prone to false-positive results [16].

Related Work. CSR-ordered layout improves locality but requires costly full-storage reorganization or adaptive approaches for updates [10], [17]–[19]. Packed Memory Arrays (PMAs) enable adaptive CSRs but suffer from high storage overhead due to graph sparsity [20], [21]. LSM-based graph storage has been proposed to address this, e.g., in [11], [22]. An LSM-based approach in [11] maintains a CSR-like structure using a multi-level index instead of Bloom filters to minimize false-positive lookups, an idea we adopt in our work. [22] uses RocksDB to enhance update efficiency with adaptive strategies for graph storage. Comprehensive evaluations and initial approaches with Intel DSA were conducted in [12], [13], [23], [24]. Hardware-accelerated LSM compaction has been addressed with FPGAs [25], [26], GPUs [27], or disaggregation [28]. We use an LSM-based approach to improve the locality in a GDBMS while leveraging the high bandwidth of HBM and on-chip acceleration with Intel DSA for memory movements, especially for sorting and compaction.

[1] https://github.com/dbis-ilm/poseidon_core
[2] https://intel.github.io/DML/

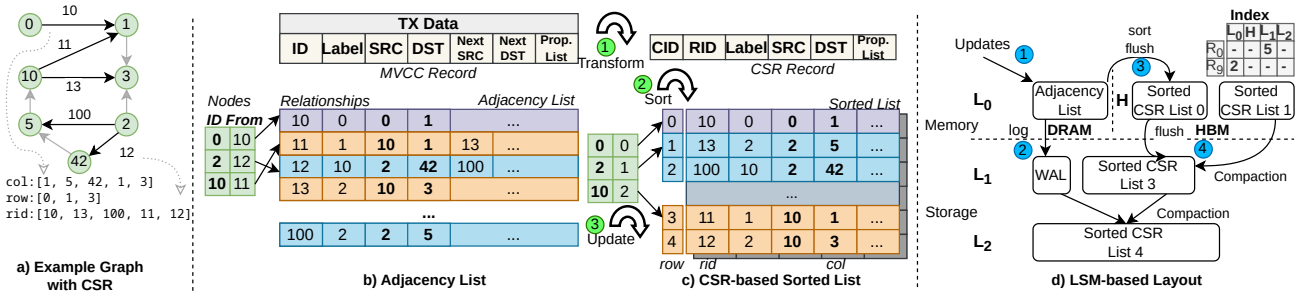

Fig. 1. Architecture for storing relationships in an LSM-based format. An adjacency list is used as the first layer to buffer updates fast with a WAL. A flush sorts the data into a sorted list, corresponding to a CSR-ordered list. Sorted lists are buffered in HBM to enable fast analytics and compaction.

III. LSM-BASED GRAPH STORAGE

We adopt a CSR-ordered approach to optimize data locality by sorting the relationships according to the order of their belonging nodes, as shown in Figure 1a. Relationships are represented as a matrix connecting source and destination nodes. The row list stores the starting offsets of each node's relationships, while the column list indicates the positions of non-null entries. This approach minimizes random access by requiring only a single (randomly accessed) lookup for the first relationship of a node, while the remaining data can be efficiently sequentially accessed. To overcome extensive data sorting for restoring the CSR layout after updates, we further employ an LSM-based approach. The core idea is to structure memory and storage into hierarchical layers, with each level accommodating more data and containing a CSR-sorted order. Then, periodically, updates to the CSR are applied. Initially, updates are inserted into the first layer, residing in the main memory. Once a threshold is reached, updates (as pages or memory chunks) are sorted and flushed to persistent layers.

Since these layers may contain multiple previously flushed pages with duplicate or deleted records, a periodically scheduled compaction uses merge-sort techniques to merge data. Compaction binds resources and affects query processing when running in parallel. Furthermore, since each layer can be optimized for the underlying hardware, we see an opportunity to integrate HBM as an additional layer for fast compaction and query processing due to the higher offered bandwidth.

A. Structural Design

We structure our LSM design in different layers where the first layers are stored in memory (DRAM and HBM) and the remaining in SSD storage (cf. Figure 1d). Additionally, to improve multi-version concurrency control (MVCC), we incorporate the following design decisions:

1) Records in the volatile layer use MVCC to track dirty lists, applying changes upon commit. New records are inserted using a WAL approach for durability.
2) Persistent layers are not directly updatable and require additional relationship entries, carried out at compaction.
3) Deletes are processed logically with tombstoned records.

The first layer L_0 is used to buffer insertions and updates, which are new relationships (cf. Figure 1d (1)).

We use pre-allocated and fixed-sized memory chunks of 1 MB ($\approx$ 10K records). Records in the first layer (L_0) represent relationships in an adjacency list. Belonging relationships (of a node) are linked by their IDs, representing a linked list of relationship records per node. Since new relationships are only appended to the list, it enables fast insertions. For every update, a WAL log entry will be written (2). Whenever a chunk is full (3), the content will be sorted and flushed to the next layer. We introduce an HBM layer (**H**) for fast query access and data transfer when compaction (4), logically between DRAM and storage. We sort and flush data from the adjacency list before flushing it to storage, leveraging high bandwidth, and using it for compaction and query processing.

Sorting & Flushing. When a chunk in the adjacency list exceeds its capacity, it is flushed to the next layer (cf. Figure 1b, c). Before flushing, the relationship list is sorted. Since sorted relationships inherently follow a CSR order, explicit linkage between relationships can be discarded (1).

In a sorted relationship list, traversing a node's relationships only requires iterating until a new source or destination node ID appears in the next record. For the sorted layer, we allocate a fixed-size, chunk-based memory area (cf. Figure 1c). Sorting is performed without an explicit sorting algorithm; instead, the adjacency list is traversed, and relationships belonging together are buffered (2). After all belonging relationships are found, they are inserted directly into the sorted list. Traversing and buffering enable sorting the adjacency list with multiple threads. As a result, the relationship list aligns with the nodes table order, effectively achieving a CSR-like structure (compare Figure 1a and b). For the flushing of already sorted lists, the data can be directly copied from memory to a previously allocated page and flushed into persistent storage. The sorted list contains multiple blocks of belonging relationships, effectively requiring a merge-sort operation. For this, we again use multiple threads and buffer belonging relationship blocks in memory before moving all to the allocated page. Finally, the changed identifiers of the relationships have to be applied to the appropriate node records (3) in the nodes table. For fast access to the relationships across the layers, we employ multiple hash indexes for each layer to access the desired relationships directly. The hash indexes are updated after sorting and flushing the records into the next layer.

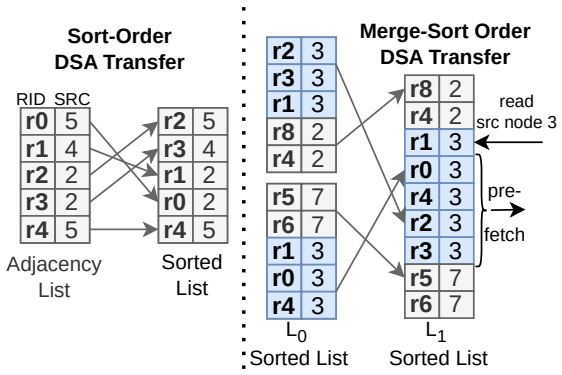

Fig. 2. Intel DSA-based sort and merge data transfers.

Indexing. When accessing a relationship, it must be determined which layers store them. Typical LSM-based approaches use Bloom filters, where multiple hash functions indicate relevant layers. However, false positives can degrade performance and increase access latency in transactional workloads. To address this, we employ a multi-index approach similar to [11]. Each layer maintains a hash index mapping nodes to the positions of their stored relationships. When retrieving relationships from the adjacency list, a volatile (dirty) version is allocated for fast updates, following our MVCC approach [3]. Dirty versions are applied to the adjacency list records upon transaction commit. Updates occur only in the adjacency list, generating dirty versions of flushed records. To reflect changes, the modifications must be propagated through flushing and compaction, as flushed records are read-only. In the meantime, updated versions are indicated through the index and checked for validity using MVCC. For transactional consistency, reading another transaction's changes requires accessing the dirty version via the adjacency list using the MVCC protocol. Reading from the adjacency list follows the entries in the belonging relationship list. For the sorted CSR-based list, only the first relationship of a node is fetched; subsequent relationships are retrieved sequentially and checked against the index for newer versions, until a different node's relationship is encountered. Employing this approach comes with vast memory movements due to sorting and compaction. We, therefore, see opportunities to improve this with on-chip acceleration offered by Intel DSA.

B. On-Chip Acceleration

We use Intel DSA's on-chip accelerator to improve the compaction of the LSM-based memory and storage approach, where most of the performance-critical data movements occur. Typically, the compaction strategy is scheduled in a background thread, like in RocksDB [15].

Placement. HBM's high bandwidth makes it well-suited for analytical processing, but its higher access latency hinders transactional workloads.

Additionally, analytical workloads on HBM require high parallelism to utilize memory bandwidth fully. As previous research suggests, incorporating HBM in DSA-based data transfers improves throughput [13]. To address these challenges, we propose a hybrid DRAM-HBM data placement strategy for relationships, optimizing performance across workload types. First, we allocate the adjacency lists in DRAM in NUMA interleaved mode. Then, we allocate every sorted list HBM for maximum throughput for sorting, compaction, but also query processing for analytics. We allocate the sorted lists in HBM NUMA interleaved mode to increase DSA transfer throughput using multiple DSA engines. The remaining layers are in storage. The compaction load sorted lists from storage into HBM.

Sorting & Flushing. Flushing and sort-merging incur high data movement, leading to significant CPU resource binding. Thus, we adapted the Intel DSA engine for data movements to sort the records in CSR order. For the actual sorting process (cf. Figure 2), we employ the following steps:

1) Sorted list allocation in HBM.
2) Sort order identification scans the adjacency list to map source to new destination positions in the sorted chunk.
3) Asynchronously assign sort order to batched DSA-descriptors and start the transfer.
4) Interleave with processing until all descriptors finish

For maximal throughput, we use one DSA engine per NUMA node (if available), which leverages NUMA-local transfers of the NUMA interleaved lists. Since allocating a DSA descriptor increases submission latency, we pre-allocate multiple descriptors and manage them using a freelist. Whenever a job is finished, its descriptor can be reset and reused by another job.

Compaction. Compaction merges multiple sorted lists into a single structure and flushes it to the next layer. This process also removes deleted records marked by tombstones. For each layer in persistent storage, we use a single file whose size (number of pages) grows logarithmically per layer. Chunks (= pages) of sorted lists from storage are loaded to HBM for compaction. Compaction involves sorting and merging sorted lists into the next layer, following the same approach described for merging adjacency lists but on multiple relationships belonging to the same node, improving transfer throughput (cf. Figure 2 right). Since the lists are already sorted, the process requires fewer data transfers. Throughput efficiency increases, since data transfer sizes increase, which is perfectly suitable for DSA-based transfers. Sort order analysis and DSA descriptor assignment are processed with multiple threads in parallel. Records marked for deletions are ignored, while duplicates are merged into a single record. The required chunks for the new sorted list are allocated in HBM, source and destination ranges are assigned to DSA descriptors asynchronously, and data is transferred in batches, further optimizing efficiency. With this, HBM bandwidth can be efficiently utilized. Compaction is mainly processed in the background. As soon as all the sorted lists are compacted into a single list, the index is updated and the list is flushed.

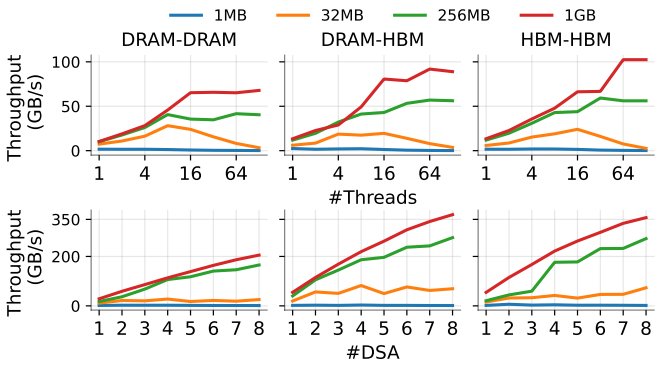

Fig. 3. Throughput comparison of multi-threaded CPU vs. Intel DSA.

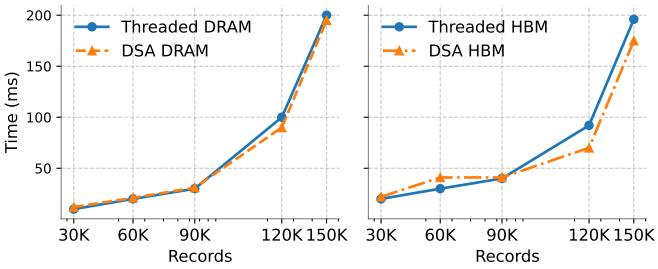

Fig. 4. Time for sorting and flushing from adjacency list to sorted CSR list.

Query Processing. To mitigate higher HBM latencies, we leverage Intel DSA transfers and implement basic prefetching for records stored in HBM. Since a node's relationships are sorted and stored contiguously in memory, we can prefetch them into DRAM for low-latency access. High-throughput data transfers through HBM help hide latency effectively. Additionally, the asynchronous nature of DSA-based transfers allows prefetching to run predominantly in the background, minimizing its impact on query processing.

In Poseidon, the main operator that accesses the relationships is the `MATCH` operator [3]. With the adjacency list approach, this operator iterates over a node's relationship list by following the `next_src` field of each relationship record until the end of the list is reached, to find a matching destination node. With the LSM-based approach, only the first relationship must be fetched directly, as subsequent relationships belong to the same node. For high-bandwidth processing, the sorted lists in HBM allow efficient exploitation of sequential access, which is well-suited for graph analytics. We now support basic aggregation operations for analytics, joins, and sort operations for supporting query operators like aggregations or joins. The underlying implementation of these algorithms uses SIMD instructions such as AVX512 for fast execution [29] and executes the workloads in parallel on the individual chunks of the sorted lists by all available threads. Whenever the data needed is not in memory, it will be read from disk into HBM. The limited capacity of HBM is another drawback (64GB per socket). To overcome this, we employ an LRU-based strategy to evict sorted lists from HBM to DRAM whenever there is no space left in HBM.

For transactions, a node's relationships are prefetched into DRAM and interleaved with the processing of preceding operators (cf. Figure 6 right), to hide higher access latencies. This is achieved by leveraging the push-based query engine of Poseidon [3]. When traversing the graph, the nodes table is scanned first. Then, the relationship list of each node is accessed with the first relationship fetched from HBM, while the remaining relationships are prefetched into DRAM asynchronously. Nodes are scanned one by one, followed by processing of their prefetched relationships.

IV. Evaluation

Our system has 2× Intel Xeon Max 9462 CPUs (32 cores, 64 GB HBM2E each), 384 GB DDR5, and 8 DSA engines. It runs Debian 12.8 with Linux 6.10, configured as SNC4 with HBM in flat mode. We use the GTPC benchmark [30], which adapts TPC-C and TPC-H for HTAP GDBMS, with 5 warehouses as the scaling factor (5M nodes, 10M relationships, 10 GB total). The approaches were implemented in Poseidon GDBMS with a buffer sized to hold half the data. Each benchmark was run 10 times, and results were averaged.

DSA Throughput. We conducted a throughput comparison between CPU (parallel `memcpy`) and DSA across multiple threads, as previous work lacked a direct evaluation. Data was distributed across threads, with source and destination memory allocated locally on each NUMA node. Threads were bound using `mbind`. As shown in Figure 3, DSA achieves higher throughput for higher transfer sizes with just one engine compared to 1–4 CPU threads. With increasing DSA engines, the throughput surpasses even highly parallel CPU transfers. When HBM is involved as either the source or the destination, throughput increases further, reaching an average of 50 GB/s with the CPU and 130 GB/s with DSA. This improvement is due to lower access latency and DMA-based operations. For lower transfer sizes, both approaches are comparable. At larger transfer sizes, throughput drops with more threads due to page faults and TLB misses. DSA achieves higher throughput on large data as its DMA transfers avoid TLB misses.

Sorting, Flushing & Compaction. We benchmark the individual stages: sorting and flushing from the adjacency list to a sorted list (DRAM vs. Hybrid), and compaction from a sorted list to persistent storage. We evaluate only the memory part (sorting and merging). The performance of sorting and flushing from the adjacency list in different sizes is shown in Figure 4. Sorting makes only a small percentage of the overall percentage with a few milliseconds. We compare a threaded implementation with 32 threads with the DSA-based implementation. There is mostly no difference in DRAM for smaller sizes, up to 90 K. With increasing size, DSA-based data movement is slightly faster. Similarly, for the HBM placement of the sorted list. As data size increases, HBM transfers become significantly faster, amplifying performance gains. DSA further enhances efficiency by reducing CPU load and freeing resources for other tasks.

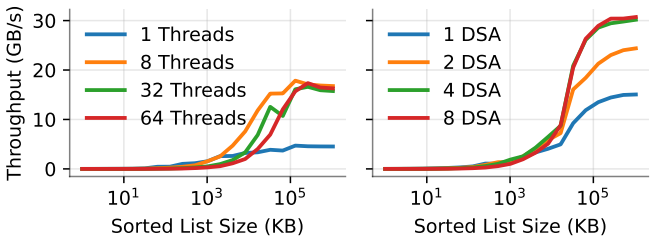

Fig. 5. Sorted list compaction with CPU and DSA-based approach.

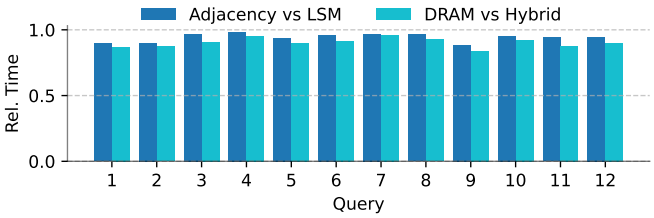

Fig. 6. Improvement of OLTP query execution using GTPC OLTP queries.

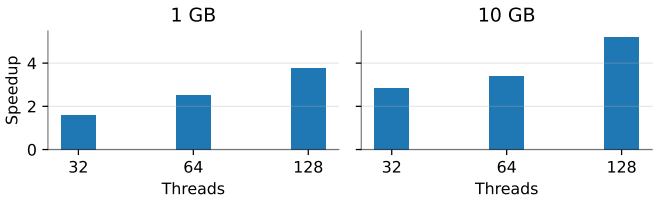

Fig. 7. Aggregation speedup in HBM with Sorted Lists.

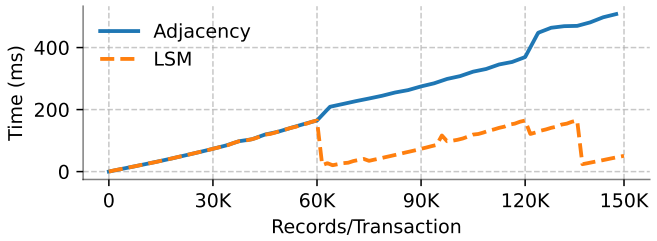

Fig. 8. Update execution time comparison between adjacency list and CSR.

Figure 5 compares compaction using the CPU-based approach against the accelerated approach with Intel DSA. We created different sorted list sizes (up to 1 GB) and inserted random relationship blocks ranging from 1 KB to 512 KB. We measure only the memory-related part. Flushing to storage adds additional execution time. DSA-based compactions are executed asynchronously, using one thread per DSA engine. In contrast, for CPU-based compaction, we utilize multiple threads and process sorting and merging in parallel. For smaller sorted list sizes (a few KBs), there is no noticeable difference in throughput due to the small transfer sizes and counts. However, as the list size increases, the throughput of DSA-based transfers improves significantly, reaching up to 3× that of CPU-based transfers, even when compared to higher thread counts. A single DSA engine can achieve higher throughput for larger list sizes and transfer counts than 64 CPU threads, primarily due to its efficient asynchronous processing. Consequently, DSA-based transfers enable higher compaction throughput while consuming fewer CPU resources, freeing them for other tasks. However, for smaller list sizes and transfers, the performance difference remains negligible.

Query Processing. Figure 6 shows OLTP query execution times for different structures and placements. We imported the data of a certain type sequentially, resulting in 10 % data in the adjacency list layer. Queries ran in parallel on 128 threads, each processing one memory chunk. Querying the graph and using the LSM as the underlying memory and storage format results in around a 10 % improvement in access. This is mostly due to the improved data locality and parallelism. However, data is imported sequentially by type, resulting in a higher locality, even with the adjacency. After certain runs of updates, the performance of the adjacency list approach decreases due to increased random access. Hybrid placement achieves faster processing for most queries due to prefetching.

We evaluated the achieved speedup when performing analytical aggregations in HBM (cf. Figure 7) with SIMD instructions. With 32 threads, the speedup is close to ×2 and can exceed ×4 with a higher thread count for HBM.

Updates. Update performance of the adjacency list and the LSM-based approach is shown in Figure 8. Only the execution time for updating the memory structure is measured, excluding sorting and flushing in the LSM-based approach, as these are handled in the background. We add an increasing number of relationships to the structures in single transactions, belonging to a small number of nodes. This leads to traversing relationship lists per node in the adjacency list to link belonging records by identifiers. Adjacency list costs increase with increasing relationship updates per transaction, mainly due to traversing the relationship list and appending. With the LSM-based approach, these costs are only in the first layer.

V. Conclusion

Intel's Data Streaming Accelerator (DSA) offers an interesting approach to offloading memory operations. Together with HBM, it can improve data transfer throughput. In this work, we have investigated how LSM-based memory for graph data can take advantage of DSA-based transfers. Our results show that a hybrid DRAM-HBM placement strategy and DSA-assisted compaction accelerate data transfers and become visible in end-to-end query execution times. For future work, we plan to optimize the LSM-based layout to capture graph properties more efficiently and incorporate other on-chip accelerators such as the Intel In-Memory Analytics Accelerator (IAA).

VI. Acknowledgment

This work was partially supported by the DFG via the "Processing-In-Memory Primitives for Data Management" (PIMPMe) project (SA 782/31) within the "Disruptive Memory Technologies" program (SPP 2377).

REFERENCES

[1] Intel Corporation, "Intel® data streaming accelerator (intel® dsa)," February 2025, accessed: 2025-02-11. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html

[2] Y. Yuan, R. Wang, N. Ranganathan, N. Rao, S. Kumar, P. Lantz, V. Sanjeepan, J. Cabrera, A. Kwatra, R. Sankaran *et al.*, "Intel accelerators ecosystem: An soc-oriented perspective: Industry product," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 848–862.

[3] M. A. Jibril, A. Baumstark, P. Götze, and K.-U. Sattler, "Jit happens: Transactional graph processing in persistent memory meets just-in-time compilation." in *EDBT*, 2021, pp. 37–48.

[4] X. Feng, G. Jin, Z. Chen, C. Liu, and S. Salihoğlu, "Kùzu graph database management system," in *CIDR*, 2023.

[5] C. Li, H. Chen, S. Zhang, Y. Hu, C. Chen, Z. Zhang, M. Li, X. Li, D. Han, X. Chen, X. Wang, H. Zhu, X. Fu, T. Wu, H. Tan, H. Ding, M. Liu, K. Wang, T. Ye, L. Li, X. Li, Y. Wang, C. Zheng, H. Yang, and J. Cheng, "Bytegraph: a high-performance distributed graph database in bytedance," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3306–3318, Aug. 2022. [Online]. Available: https://doi.org/10.14778/3554821.3554824

[6] B. Tong, Y. Zhou, C. Zhang, J. Tang, J. Tang, L. Yang, Q. Li, M. Lin, Z. Bao, J. Li, and L. Chen, "Galaxybase: A high performance native distributed graph database for htap," *Proc. VLDB Endow.*, vol. 17, no. 12, p. 3893–3905, Aug. 2024. [Online]. Available: https://doi.org/10.14778/3685800.3685814

[7] L. Zhou, Y. Rayhan, L. Xing, and W. G. Aref, "Gtx: A write-optimized latch-free graph data system with transactional support," 2024. [Online]. Available: https://arxiv.org/abs/2405.01418

[8] A. Baumstark, M. A. Jibril, and K.-U. Sattler, "Accelerating large table scan using processing-in-memory technology," *Datenbank-Spektrum*, vol. 23, no. 3, pp. 199–209, 2023.

[9] P. Fuchs, D. Margan, and J. Giceva, "Sortledton: a universal, transactional graph data structure," *Proc. VLDB Endow.*, vol. 15, no. 6, p. 1173–1186, Feb. 2022. [Online]. Available: https://doi.org/10.14778/3514061.3514065

[10] M. A. Jibril, A. Baumstark, and K.-U. Sattler, "Adaptive update handling for graph htap," *Distributed and Parallel Databases*, vol. 41, no. 3, pp. 331–357, 2023.

[11] S. Yu, S. Gong, Q. Tao, S. Shen, Y. Zhang, W. Yu, P. Liu, Z. Zhang, H. Li, X. Luo, G. Yu, and J. Zhou, "Lsmgraph: A high-performance dynamic graph storage system with multi-level csr," *Proc. ACM Manag. Data*, vol. 2, no. 6, Dec. 2024. [Online]. Available: https://doi.org/10.1145/3698818

[12] R. Kuper, I. Jeong, Y. Yuan, R. Wang, N. Ranganathan, N. Rao, J. Hu, S. Kumar, P. Lantz, and N. S. Kim, "A quantitative analysis and guidelines of data streaming accelerator in modern intel xeon scalable processors," ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 37–54. [Online]. Available: https://doi.org/10.1145/3620665.3640401

[13] A. Berthold, C. Fürst, A. Obersteiner, L. Schmidt, D. Habich, W. Lehner, and H. Schirmeier, "Demystifying intel data streaming accelerator for in-memory data processing," in *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, ser. DIMES '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 9–16. [Online]. Available: https://doi.org/10.1145/3698783.3699383

[14] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[15] R. Wei, Z. Zhu, A. Kryczka, J. Zhuang, and M. Athanassoulis, "Benchmarking, analyzing, and optimizing wa of partial compaction in rocksdb," 2025.

[16] N. Dayan, M. Athanassoulis, and S. Idreos, "Optimal bloom filters and adaptive merging for lsm-trees," *ACM Trans. Database Syst.*, vol. 43, no. 4, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3276980

[17] S. Firmli, V. Trigonakis, J.-P. Lozi, I. Psaroudakis, A. Weld, D. Chiadmi, S. Hong, and H. Chafi, "Csr++: A fast, scalable, update-friendly graph data structure," in *24th International Conference on Principles of Distributed Systems (OPODIS'20)*, 2020.

[18] A. A. R. Islam and D. Dai, "Dgap: Efficient dynamic graph analysis on persistent memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607106

[19] J. Shi, B. Wang, and Y. Xu, "Spruce: a fast yet space-saving structure for dynamic graph storage," *Proc. ACM Manag. Data*, vol. 2, no. 1, Mar. 2024. [Online]. Available: https://doi.org/10.1145/3639282

[20] A. Al Raqibul Islam, D. Dai, and D. Cheng, "Vcsr: Mutable csr graph format using vertex-centric packed memory array," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 71–80.

[21] D. De Leo and P. Boncz, "Teseo and the analysis of structural dynamic graphs," *Proc. VLDB Endow.*, vol. 14, no. 6, p. 1053–1066, Feb. 2021. [Online]. Available: https://doi.org/10.14778/3447689.3447708

[22] D. Mo, J. Liu, F. Wang, and S. Luo, "Aster: Enhancing lsm-structures for scalable graph database," 2025. [Online]. Available: https://arxiv.org/abs/2501.06570

[23] J. Baik, J. Kim, C. H. Park, and J. Ahn, "Accelerating page migrations in operating systems with intel dsa," *IEEE Computer Architecture Letters*, pp. 1–4, 2025.

[24] H. Ji, M. Kim, S. Oh, D. Kim, and N. S. Kim, "Cooperative memory deduplication with intel data streaming accelerator," *IEEE Computer Architecture Letters*, 2025.

[25] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao *et al.*, "{FPGA-Accelerated} compactions for {LSM-based}{Key-Value} store," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020, pp. 225–237.

[26] D. Tang, W. Wang, Y. Mao, J. Yu, T.-W. Kuo, and C. J. Xue, "Stem: Streaming-based fpga acceleration for large-scale compactions in lsm kv," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 3893–3905.

[27] H. Sun, J. Xu, X. Jiang, G. Chen, Y. Yue, and X. Qin, "glsm: Using gpgpu to accelerate compactions in lsm-tree-based key-value stores," vol. 20, no. 1, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3633782

[28] C. Ding, J. Zhou, K. Lu, S. Li, Y. Xiong, J. Wan, and L. Zhan, "D2comp: Efficient offload of lsm-tree compaction with data processing units on disaggregated storage," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 3, Sep. 2024. [Online]. Available: https://doi.org/10.1145/3656584

[29] I. Corporation, "x86-simd-sort: High-performance sorting algorithms using x86 simd instructions," https://github.com/intel/x86-simd-sort, 2025, accessed: 2025-02-11.

[30] M. A. Jibril, A. Baumstark, and K.-U. Sattler, "Gtpc: Towards a hybrid oltp-olap graph benchmark," in *BTW 2023*. Bonn: Gesellschaft für Informatik e.V., 2023, pp. 105–117.