JIT happens: Transactional Graph Processing in Persistent Memory meets Just-In-Time Compilation

Muhammad Attahir Jibril
TU Ilmenau
Germany
muhammad-attahir.jibril@tu-ilmenau.de

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

Philipp Götze
TU Ilmenau
Germany
philipp.goetze@tu-ilmenau.de

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

ABSTRACT

Graph databases are used for different applications like analyzing large networks, representing and querying knowledge graphs, and managing master data and complex data structures. Besides graph analytics, the transactional processing of concurrent updates and queries represents a challenging data management task. In this paper, we investigate the usage of persistent memory as a very promising technology for graph processing. We present a novel architecture for transactional processing of queries and updates on a property graph model that exploits and addresses the specific characteristics of persistent memory by hybrid storage and memory management as well as a just-in-time query compilation approach. Our experimental evaluation on interactive short read and update query workloads show that PMem-based systems that are well-designed to exploit PMem characteristics outperform traditional disk-based systems significantly and have only a small overhead compared to DRAM-only systems. Moreover, the evaluation shows that JIT compilation brings performance benefits especially when an adaptive compilation approach is leveraged to hide the overhead of compilation as well as the latency of PMem.

1 INTRODUCTION

Graph databases represent an important class of NoSQL systems with numerous flavors, including systems for analyzing large graphs, systems for querying knowledge bases, and systems supporting updates on graphs and navigational queries. They are designed for different graph data models ranging from RDF triples to property graph models, as well as different processing models from database query processing to approaches like the bulk synchronous parallel (BSP) model.

The numerous available systems mainly adopt the typical architectures of database systems, i.e., traditional disk-based architecture, in-memory architecture or scalable, distributed solutions. Graph data are either stored in disk-based data structures and loaded into memory for processing or kept directly in in-memory structures (without requiring to load data during startup) while using techniques like logging to allow for persistent updates.

In this work, we present a novel architecture for graph databases based on persistent memory (PMem). PMem – also known as non-volatile memory (NVM) or storage-class memory (SCM) – is one of the most promising trends in hardware development which have the potential to hugely impact database system architectures. Characteristics such as byte-addressability, read latency close to DRAM but with read-write asymmetry, and inherent persistence open up new opportunities for database systems. Specifically, Intel's Optane DC Persistent Memory Modules (DCPMMs) are already available on the market and supported by the Persistent Memory Development Kit (PMDK) [17]. Several studies, as well as our experiments, have identified the following characteristics of this technology (we elaborate these in more detail in Section 3):

(C1) PMem has a higher latency and lower bandwidth than DRAM.
(C2) Reads and writes on PMem behave asymmetrically.
(C3) DCPMMs internally work on 256-byte blocks.
(C4) Failure atomicity is only guaranteed for 8-byte aligned writes.

The focus of our work is an architecture for hybrid transactional/analytical processing (HTAP) on a property graph model. Transaction support covers insert/update/delete operations on nodes, relationships, and their properties with ACID guarantees. Furthermore, we support Cypher-like navigational queries. In this paper, we particularly focus on data structures and techniques for query and transaction processing in graph databases exploiting PMem and addressing the characteristics (C1)-(C4) mentioned above. Although we aim for HTAP, we do not consider graph analytics in this paper yet. Exploiting PMem for graph analytics is discussed by other researchers, e.g., in [13]. Our contributions are as follows:

- We present the architecture of an HTAP graph engine with storage structures designed for PMem, primarily taking (C1)-(C3) into account.
- We discuss the implementation of a timestamp ordering-based multiversion concurrency control (MVTO) protocol optimized for PMem addressing (C4).
- We describe our just-in-time (JIT) query compilation approach for compiling graph queries into machine code to hide the higher latency of PMem as described in (C1).

Thus, the novelty of our work lies in the design, adaptation as well as evaluation of transaction and query processing techniques to leverage the idiosyncrasies of persistent memory for graph databases.

2 RELATED WORK

Several of the approaches presented in this paper are based on insights from previous work. In particular, the lessons learned regarding the new concepts of data structures for PMem had a

major impact on the design decisions for our graph engine. There is, to our knowledge, no transactional graph system or JIT query compilation approach targeting persistent memory, yet. Hence, we approach the subject from three directions: general graph management, PMem-aware data structures and storage engines as well as query compilation.

Graph Management. For graph data management, numerous data models and systems have been proposed in the past. Among the several database models for graph data [3], RDF for the Semantic Web and property graph models are the most prominent. On top of these, query languages like SPARQL for RDF triple data, diverse SQL dialects, and dedicated languages like Cypher[1] and Gremlin [36] have been developed. The SQL standardization committee is currently working on standardizing the graph query language GQL.

Depending on the supported data model and query language, graph database systems are either special-purpose systems such as triple stores for RDF like Virtuoso, native stores for property graphs e.g., Neo4j, relationally-backed approaches such as DB2RDF [7] and EmptyHeaded [1]; or extensions of SQL systems like Grail [11] and SAP HANA [37]. Here, standard DBMS implementation techniques are used for data storage, indexing, transaction management, and query processing. Particularly, traversal operations [32], as well as support for graph analytics [29, 38], play an important role. However, only very few approaches try to support HTAP workloads (TigerGraph, Neo4j) and to our best knowledge, no established graph system is utilizing PMem yet [6].

Recently, however, Gill et al. [13] investigated the application of DCPMMs in Memory mode for running graph analytics. They evaluated large scale data sets on existing graph frameworks and demonstrated that their NUMA-aware algorithms on cheaper single machine setups with DCPMMs can outperform more expensive DRAM-only cluster setups. With Sage [9], the authors have shown that the AppDirect mode of DCPMMs in combination with sophisticated algorithms can even achieve a better performance than an unmodified in-memory graph database used on PMem in Memory mode. They especially address the asymmetry of PMem by introducing the parallel semi-asymmetric model. Here, the entire graph is stored as a read-only copy in PMem and a smaller mutable part in DRAM. A volatile auxiliary structure keeps track of deleted edges for graph filtering. Since the focus is on parallel analytical queries, we assume that no transactional updates are possible. In this paper, we want to make the appropriate contribution in this regard.

PMem-aware Storage Designs. Researchers recently started adapting existing data structures to PMem. This includes several variants of the B$^+$-Tree [8, 43], hybrid variants like the FPTree [31], the LB$^+$-Tree [27], DPTree [49], and HiKV [47], as well as LSM-Tree variations [19]. There are also latch-free B$^+$-Tree variants targeting modern hardware, such as the Bw-Tree [26, 45] and the BzTree [4]. While the former addresses multi-core systems with flash storage, the BzTree is explicitly designed for PMem. Apart from the individual data structures, some approaches for PMem-based storage engines have been proposed. SOFORT [30] is a columnar transactional storage engine leveraging PMem by minimizing logging and updating data in place, aiming for mixed OLAP and OLTP workloads. Peloton [33] is another relational DBMS engine already considering PMem by applying write-behind logging [5]. The basic idea is to write and flush all changed entries in-place to PMem during commit. A more recent proposal is the key-value store RStore [25]. It opts for a log-structured design with an index. It utilizes linked and fix-sized append-only blocks in PMem. Once a block is full, it is considered immutable and indexed in a volatile tree which is rebuilt during recovery. Additionally, RStore employs partitions that are owned by only one thread at a time, each having its own log to parallelize recovery.

JIT Query Compilation. Similarly, there are numerous works on query compilation techniques. Neumann [28] presented a query compiler architecture using the LLVM framework[2] to generate and compile code for queries in the HyPer database. Based on this, Kohn et al. [21] proposed an approach to mask the compilation time by compiling the query in the background while interpreting it. They further improve the efficiency by using different execution modes depending on the query type. There are also works that try to provide a lightweight approach, apart from LLVM. An alternative approach, LegoBase, provides a query compiler that generates high-level code in multiple steps, where each step replaces declarative components of the query with imperative code [41]. Funke et al. [12] proposed a lightweight intermediate representation (IR) to reduce compilation times for queries by estimating value lifetimes before code generation. The Voodoo IR [35] is a declarative algebra for utilizing many-core architectures and GPUs by generating OpenCL code. Although these approaches are designed for relational DBMSs, query compilation is also applied in several graph DBMSs, like TigerGraph and Neo4j. However, while JIT query compilation is a broad research topic, there is presently no system that utilizes it to hide the memory access latency of PMem.

3 PERSISTENT MEMORY SPECIFIC DESIGN GOALS

This section aims to summarize the observations of several studies as well as our experiments regarding the characteristics and challenges introduced by PMem – in particular, Intel's Optane DCPMMs. Subsequently, we derive general design goals for systems trying to integrate PMem in their hardware landscape. We hope they help others to avoid common pitfalls when conceiving new efficient systems for modern storage hierarchies.

3.1 Characteristics and Challenges

The first three items presented are specific characteristics of Intel's DCPMM technology, while the remaining are explicit challenges that mostly result from PMem and other system peculiarities.

- **(C1) PMem has a higher latency and lower bandwidth than DRAM.** Random access read latency and the read bandwidth of PMem is worse than DRAM by a factor of about three. Persistent writes are also slower than writes to DRAM. PMem bandwidth is about 7 × lower than that of DRAM [42, 48].
- **(C2) Reads and writes on PMem behave asymmetrically.** This concerns several aspects, namely performance, energy consumption, and cell wear. Asymmetrically slower writes cost more energy and lead to wear.
- **(C3) DCPMMs internally work on 256-byte blocks.** They utilize a write combining buffer that is used to reduce

[1] https://www.opencypher.org

[2] http://llvm.org/

write load by trying to combine four cache lines into one 256-byte block write. Interestingly, read operations also benefit when a multiple of the block size is used [42, 48].

(C4) Failure atomicity is only guaranteed for 8-byte aligned writes. The largest failure-atomic store instruction covers only 8 bytes of data, aligned on an 8-byte boundary. Anything larger has to be implemented in software. This means that inconsistencies of data structures due to partial changes in case of system failures and reordering of instructions by the compiler or the CPU have to be avoided.

(C5) PMem allocations are expensive. Compared to DRAM allocations, PMem allocators such as the PMDK allocator need significantly more time [14, 15, 24]. This is mainly due to the necessity of cache line flushes and recovery measures. In conjunction with the higher latencies of PMem, allocations can be –depending on the number of threads– up to 8× slower than on DRAM [24].

(C6) Dereferencing persistent pointers can prevent optimizations. A persistent pointer is a 16-byte structure consisting of a pool identifier (similar to a file path) and an offset in this pool. It was introduced in PMDK and keeps its validity across application restarts. Since this concept of persistent pointers is not integrated into compilers (yet), their handling cannot be automatically optimized as it is the case for volatile pointers [39].

3.2 Design Goals

From the above characteristics and challenges, we can more or less directly formulate corresponding general design goals as follows. Apart from the generic usability of these goals, we will also use them as a foundation for the design decisions in the next section.

(DG1) Algorithmically save writes (C1 & C2). This was one of the first common goals when PMem came up. The idea is to reduce the number of writes by trading them off for more reads. Furthermore, certain intermediate results can be kept in DRAM instead of PMem. In practice, it has been shown that not the number of writes but rather the number of flushed cache lines is decisive.

(DG2) Opt for a DRAM/PMem hybrid storage design (C1 & C2). It has been shown that a pure PMem-only architecture causes too much performance degradation compared to its DRAM counterpart. A hybrid DRAM/PMem approach is therefore highly recommended when seeking the best performance and still requiring persistence [14, 15].

(DG3) Optimize the access granularity to 256 bytes (C3). Besides, the data structures should be aligned to cache lines. Only then a sequential pattern and correspondingly the peak bandwidth can be reached. Everything else can be considered as random access.

(DG4) Prefer failure-atomic writes over logging or shadowing (C4). For this purpose, flushing of cache lines via the `clwb` (cache line write back) instruction and barriers such as `sfence` (store fence) have to be used. However, the number of such barriers should be minimized for best performance. PMDK transactions can be used to simply and universally achieve failure atomicity. However, for performance-critical sections, the underlying logging and snapshotting approach can lead to excessive overhead. Thus, in the long run, an individual realization of failure atomicity with optimally arranged 8-byte stores, `clwb` instructions, and barriers should be preferred.

(DG5) Use group allocations and reuse blocks of memory instead of deallocating (C5). Not every new record in a system should be associated with an allocation. The less frequent allocation of larger blocks or groups can amortize the overhead. Deallocating can also be replaced by suitable free space management. Since they increase the number of allocations, copy-on-write techniques should be replaced by in-place updates or reuse a pre-allocated space.

(DG6) Avoid dereferencing of persistent pointers (C6). Persistent pointers should preferably only be used during application (re)start for initialization. Afterward, the current valid virtual pointer or application-specific offsets should rather be used. Alternatively, the external location could be converted to a virtual reference once before using it multiple times. In addition, pointer chasing should be avoided as well, as shown in [14, 15].

4 STORAGE MODEL

Essentially, there are two classes of graph data models, namely RDF triple stores and property graphs. RDF stores express everything as triples (subject-predicate-object) which link two nodes or a node to a property value (also called resources and literals). Predicates can therefore be relationships or property keys. Property graphs, on the other hand, consist of explicit node, relationship, and property structures where the properties are directly assigned to a node or relationship. The RDF model creates a lot of redundancies, which could lead to additional write load, which in turn will most likely have a negative impact on PMem performance. Therefore we decided to opt for the property graph model, which is more compact, more expressive, and more efficient to query.

Data Model Definition. In the following, we adopt a property graph model where a graph $G = (N, R)$ consists of nodes N and directed relationships $R \subseteq N \times N$. Each node $n \in N$ is identified by a unique identifier $id : N \rightarrow ID$. Furthermore, a label (used, e.g., as a type descriptor) is assigned to each node and each relationship via a labeling function $l : \{N \cup R\} \rightarrow L$ where L is the set of labels.

Properties are represented as key-value pairs $(k, v) \in P$ with $P = K \times D$ where K denotes the set of property keys and D the set of possible values including numbers, strings, etc. To each node and relationship, a set of properties can be associated via $p : \{N \cup R\} \rightarrow \mathcal{P}(P)$ where $\mathcal{P}(P)$ denotes the power set of P.

4.1 Design Decisions

The above data model is implemented by storing the graph in node, relationship and property tables maintained in persistent memory. For efficient data access, the specific characteristics of current PMem technology as mentioned in Sections 1 and 3 have to be taken into account. The application of our derived design goals led to the following key design decisions:

(DD1) Each of the tables is managed as a **linked list of chunks** where a chunk is a fixed-sized array (cache-line aligned and a multiple of 256 bytes) of records. To reuse the

space of deleted records, standard free space management using a persistent list is implemented. This way, tables can dynamically and efficiently grow or shrink for updates, by allocating/deallocating chunks (DG3, DG5).

(DD2) A chunk stores **equally-sized records** of the same type (nodes, relationships, properties). Thus, records can be **addressed via their offsets**. Similar to a sparse index, an additional persistent lookup table allows efficient access to chunks based on the record offset (DG1, DG6). Note that we use array offsets because they can be represented as 8-byte integers instead of 16-byte persistent pointers. This not only saves space but also allows for **failure-atomic stores** and avoids costly dereferencing (DG1, DG3, DG6).

(DD3) In order to represent nodes and relationships as equally-sized records, **properties are outsourced** to a separate table. Furthermore, all variable-length values (e.g., strings) are **dictionary encoded**. Both lead to a reduced number of write operations (DG1).

(DD4) The **connections** between nodes and their relationships as well as their properties are represented via array **offsets** instead of (persistent) pointers. Because relationships are directed, each node refers to its list of both outgoing and incoming relationships, also via offsets.

(DD5) The **storage model** is designed **hybrid** both for secondary indexes and for transaction management (DG2). Further details are provided below.

4.2 Key Data Structures

In the following, we give an overview of the key data structures to represent the property graph model and further structures necessary to realize our design decisions and achieve a great performance.

Nodes, Relationships, and Properties. Fig. 1 illustrates the primary storage structures of a persistent graph which we have implemented using Intel's PMDK [17]. The highlighted row illustrates a respective node or relationship record. On top of both the node and relationship table, an additional sparse index is used which maps the identifiers of the first record of each chunk to their corresponding memory location. For each chunk there is a bitmap to indicate free and occupied record slots, enabling an efficient reclamation of deleted entries. The chunks are linked by a persistent pointer to allow the scanning of all data. Node records consist of a label, the offset of the first incoming and first outgoing relationship, as well as the offset to their properties. Relationship records also have a label as well as the offset to their properties. Furthermore, they store the location of the source and destination nodes that they connect. Optionally, relationship records hold offsets to the next relationships of their source and destination node. Note that the records for nodes and relationships contain a few additional fields needed for transaction processing which are described in Section 5. In total, this results in a record size for nodes and relationships of 56 and 72 bytes respectively.

The properties are stored in a separate chunked table as key-value pairs. These are grouped in batches, each belonging to a single node or relationship, to obtain cache-line-sized records. In order to allow variable-length key-value pairs, string types are stored as dictionary codes. If there are more properties for a single node or relationship, the property record links to the next entry. These data structures resemble the typical storage layout

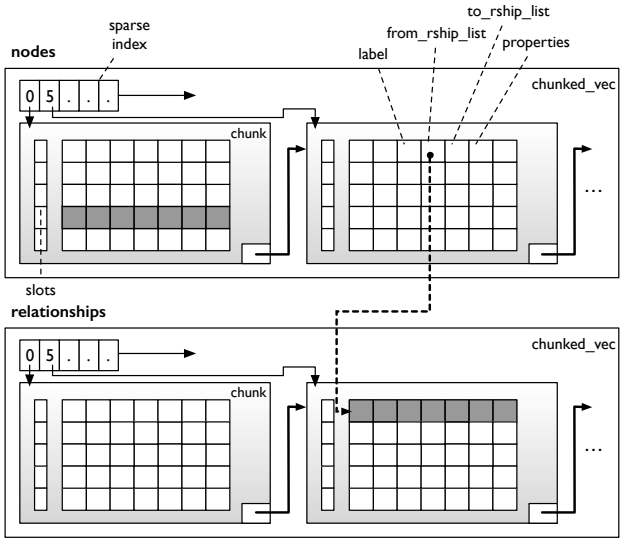

Figure 1: Graph data structures

of disk-based, table-oriented systems such as SQL databases or even graph databases like Neo4j. However, in our case, table chunks are not copied between disk and memory but instead accessed directly in PMem. In addition, nodes, relationships, and associated properties can be addressed individually via their identifiers/offsets.

Dictionary. As mentioned before, to allow for variable-length labels, property keys, and values, a dictionary is used. This compresses strings and, thus, reduces space and write overhead as well as ensures that records remain addressable by offset. Furthermore, the comparison of codes instead of strings speeds up operators such as filters. The dictionary consists of two hash tables for bi-directional translation to make lookups fast. These must be kept persistent, in case of failure, since the codes and strings are not stored elsewhere. An alternative could be to only store one of the hash tables in PMem and rebuild the other DRAM-resident part. Depending on the workload (either more inserts or more queries), the more frequently used table should be kept in DRAM.

Hybrid Indexes. The table-based storage model is useful for lookups on physical node/relationship identifiers (which represent array offsets) as well as scan-intensive processing where large parts of the nodes or relationships are visited. However, for lookup queries on node/relationship properties, scans are too expensive. In order to accelerate these queries, we additionally provide B^+-Tree indexes. An index can be constructed on nodes with a given label and for a property. The values of these properties are used as keys in the index. Since the indexes are secondary data structures that can be rebuilt in the event of a failure, they do not have to be completely persistent. To still have a good compromise between recovery and query performance, we opted for a DRAM/PMem hybrid approach (selective persistence) similar to [18, 31, 47]. In particular, this means that the leaf nodes are stored in PMem and the inner nodes in DRAM, resulting in a maximum of one PMem-resident node being read per lookup (if not already cached by the CPU) and significantly reduced recovery time. This has an additional economic advantage since less DRAM is used, which we expect to be more expensive than

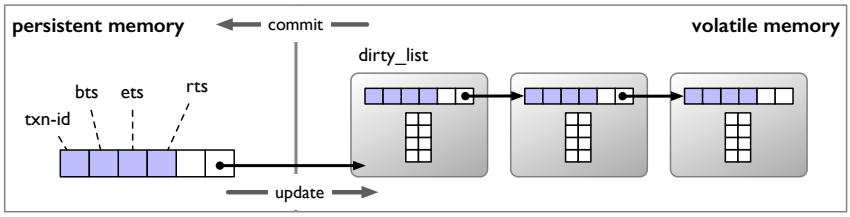

Figure 2: Structure of transactional data

PMem in the near future. In accordance with DG3, all nodes on PMem are cache-line-aligned and a multiple of 256 bytes. For analytical queries, multi-dimensional index structures optimized for PMem could also be used where properties represent the dimensions [18].

5 TRANSACTION PROCESSING

An HTAP architecture requires high-performance concurrency control mechanisms. Several studies in the past [34, 46] have shown that DBMSs with multi-version concurrency control (MVCC) exhibit higher concurrency than their single-version counterparts. Here, transactions can be concurrently executed on different versions of the same object, thus increasing the overall transaction throughput especially when the transactions are long-running and contention is high [22]. This also allows for scalability and efficient utilization of modern multi-core CPUs. MVCC is implemented differently by different DBMSs, each making certain design decisions in order to optimize for its target workloads. The interplay between these design decisions ultimately results in computation and storage overhead trade-offs. Below, we discuss how we implemented these MVCC design decisions in a PMem setting to achieve our design goals presented earlier in Section 3.

5.1 Concurrency Control Protocol

Existing concurrency control (CC) protocols such as two-phase locking (2PL), optimistic concurrency control (OCC), or timestamp ordering (TO) can essentially be used in a multi-version setting. We chose MVTO as our CC protocol. With our MVTO implementation, we support updates of an arbitrary number of objects within a single transaction and achieve snapshot isolation guarantees. Note, that we use MVTO here mainly as an example to evaluate how an MVCC protocol implementation can exploit and address the specifics of PMem. However, in principle, the main concepts should apply to implementations of other protocols too.

There is a transaction identifier (timestamp), `txn-id`, given to each transaction at the beginning of the transaction that uniquely identifies it. Each data object maintains meta-data fields for concurrency control purposes. To this end, we extend the data structures of nodes and relationships, as shown in Fig. 2, by additional *persistent* fields – `txn-id`, begin timestamp `bts`, end timestamp `ets` and read timestamp `rts` – and a volatile field – pointer. The `txn-id`-field is used for write-locking, by way of coordinating which versions are valid for which write-transactions. By default, it is set to zero except if the object is locked by a write-transaction, where it is set to the transaction's `txn-id` using a CaS instruction [22]. The begin timestamp and end timestamp fields mark the validity of an object for access by a read-transaction, while the read timestamp indicates the latest transaction that read it. The pointer field stores a *volatile* pointer to a list of dirty objects (i.e., in DRAM) to address (DG1) and (DG2). Alternatively, the `bts`, `ets`, and `rts` fields and perhaps also the `txn-id` of the current version could be moved to DRAM in order to reduce the persistent record size. These fields could then be re-initialized during recovery (or during the first access after a failure). However, this could also be disadvantageous because the transaction information of the current version would always have to be retrieved with another random read in DRAM.

Write transaction. A transaction T always updates the latest version of an object o. It creates a new version o_{i+1} of the object if no other transaction has a lock on o_i and o_i has not been read by a more recent transaction (i.e., the transaction identifier $id(T) > \texttt{rts}(o_i)$). Otherwise, T aborts. The `txn-id` field of o_{i+1} is set to $id(T)$. In case of an update, o_{i+1} is kept in the dirty list in volatile memory until commit. If the transaction inserts a new object, this object is already stored in the persistent array (i.e., in PMem), but still locked until the end of the transaction.

Read transaction. A transaction T reads version o_i of an object for which $id(T)$ is between the `bts` and `ets`, i.e., $\texttt{bts}(o_i) \leq id(T) < \texttt{ets}(o_i)$, and which is not locked by another active transaction. Thus, the object is accessed in PMem first (representing the most recent committed version) and if this is not the version valid for T then the dirty list in volatile memory is traversed to retrieve the correct version. In case of a lock held by another transaction, the transaction is aborted. Upon reading o_i, the `rts` field is updated to $id(T)$ unless $\texttt{rts}(o_i) \geq id(T)$. In this case, the transaction reads an older version without updating `rts`.

Commit. For commit of a transaction T, the timestamp fields of the updated object version o_{i+1} are set accordingly: `bts` to $id(T)$ and `ets` to `INF` and for the previous version o_i, the field `ets` is set to $id(T)$. In the case of delete, `ets` of the deleted version o_{i+1} is set to $id(T)$ instead. If the object was newly created, however, it is simply unlocked (i.e., resetting `txn-id` to 0). Otherwise, o_{i+1} has to be copied back to PMem. In order to guarantee failure atomicity, this memory copy has to be performed atomically. This can be implemented in different ways. One approach is to rely on the solution provided by the Intel PMDK to atomically update and persist data that is larger than the power-fail atomic size or portions of data that are non-contiguous. PMDK uses transactional operations for memory allocation, freeing, and setting. Internally, these transactions are implemented via redo logging to ensure the atomicity of memory allocations and undo logging for transactional snapshots [40]. Other approaches are, e.g., using Multi-Word CaS instructions such as PMwCAS [44] which allows atomically changing multiple 8-byte words on PMem. In our current implementation, we use the PMDK solution for the sake of simplicity (DG4). However, this comes with a small overhead.

5.2 Version Storage

A transaction updating an object version o_i creates a new version o_{i+1} by making a copy of o_i and appending it to the front of the list of dirty versions (i.e., *version chain*). It then performs all updates on o_{i+1} in DRAM until commit. Keeping all uncommitted data in volatile memory is a design decision we made in order to minimize the number of writes to PMem (DG1, DG2). This hybrid DRAM/PMem approach allows for the creation of all versions by transactions to be a volatile copy instead of the more expensive copy to PMem, and also allows for all the write operations that occur during the lifetime of a transaction to be performed at DRAM latency until the transaction is to commit when the updates are finally persisted in PMem. Note that a dirty object has the same structure as its committed version but with a different validity interval (as specified by the range [`bts`, `ets`]).

5.3 Garbage Collection

In our current implementation, we use Transaction-level Garbage Collection (GC), where storage space occupied by dirty versions that are not going to be used anymore is reclaimed at transaction-level granularity [46]. A node or a relationship maintains a volatile dirty list, only if there is a valid dirty version of it. A dirty version is not used anymore if it becomes invalid (i.e., the transaction that created it aborts) or if it is no longer *visible* to any active transaction (i.e., its `ets` $< id(T)$ of the oldest active transaction T). All empty or unused dirty lists are discarded during a commit. If the storage space to be reclaimed is in PMem, either because a committed transaction deleted the object or the object was inserted by an aborted transaction, we do not deallocate the record slot(s). Rather, we simply mark it with a bitmap as free for later reuse (DG5).

6 QUERY PROCESSING

The characteristics of PMem have several implications for query processing. First of all, data access is no longer block-oriented and, therefore, has to be optimized for sequential access. The direct and byte-addressable access is very similar to in-memory databases. In graph databases, this is particularly useful for traversal operators. However, as mentioned above, reading from PMem is slower than from DRAM. Hiding this higher latency requires efficient cache utilization, multithreaded processing, and various execution modes.

6.1 Push-based Approach

We address these requirements by a multithreaded push-based query engine. Our engine provides a set of graph-specific algebra operators [16] such as NodeScan, RelationshipScan, and ForeachRelationship; as well as standard relational operators like Filter, Project, and several Join variants. As every operator is implemented and ahead-of-time (AOT)-compiled, i.e., available at run-time, the engine is able to interpret queries (given as graph algebra expressions) directly by calling these operators with the required parameters. Processing a typical traversal query (Match in Cypher) is initiated by scans on the node or relationship tables including filters. For each node satisfying the optional filter condition the traversal operation is applied, i.e., the NodeScan operator forwards the current node to the next operator ForeachRelationship, and so on. (Fig. 3).

Though traversals could be also implemented using joins of a standard relational query engine [11], the ForeachRelationship leverages the direct addressability of data in PMem. As described

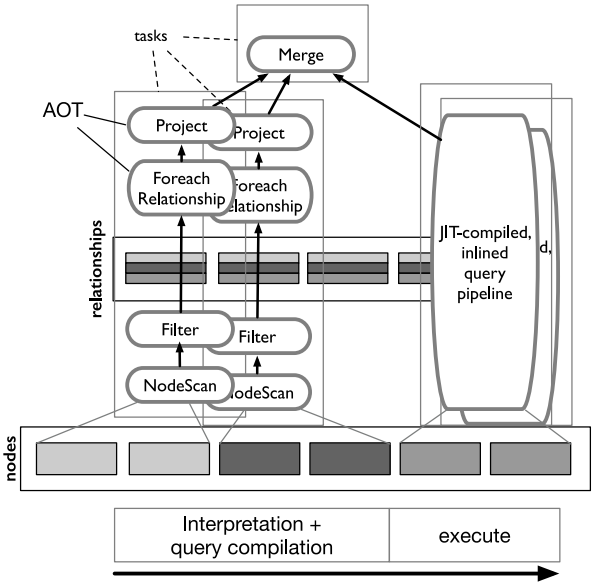

Figure 3: Query execution plan

in Sect. 4.2, node records contain the persistent addresses (offsets) of their relationships, which in turn store the address of the sibling nodes, and are used to traverse the path. This avoids the problem of join escalation during traversals.

For query parallelization, we leverage a task model. Scans are performed as parallel scans: Each task processes a range of the node/relationship tables. Thus, all subsequent operators following the scan are also performed within this task until a pipeline breaker like a join or sort operator is reached. This way, we follow the morsel-driven parallelism approach [23].

For using indexes in query processing, an appropriate IndexScan operator is provided that performs a lookup or range scan on the B$^+$-Tree, extracts the matching nodes from the node tables, and passes them to the subsequent operator pipeline.

6.2 Just-In-Time Query Compilation

Besides the AOT compiled query engine, we implement a JIT query compiler that transforms graph algebra expression into machine code. Traditional query execution engines use a query interpreter to execute query statements. This has several drawbacks that reduce the resulting performance. An interpreter relies on AOT compiled code, which means that the appropriate methods must be available for every possible occurrence of a particular tuple element type. Furthermore, a query interpreter is not able to recognize equal or redundant instructions. Particularly for operators that lead to variable cardinalities, like selections or aggregations, it introduces additional overhead. Query compilation is an approach to tackle these issues. Our approach aims to compile given graph algebra expressions into highly efficient machine code using the JIT compilation technique [20]. As the compiling framework, we chose the LLVM compiler infrastructure because it provides numerous relevant features for the JIT compilation, reliable performance, and portability to several architectures. Moreover, the LLVM IR provides an instruction set suitable for the implementation of all the abstractions needed for our graph query engine. One significant requirement for the JIT query compiler is the fulfillment and compliance with the formulated design goals (DG1-DG6). This is mainly done by reusing

(calling) AOT-compiled code, e.g., access methods to nodes or methods for transaction processing. Thereby, the code generation effort will be reduced because it is already compliant with the design goals and optimized by the AOT compiler.

Similar to the approach presented by [28], we aim to process intermediate tuple results as long as possible in the CPU registers. In order to achieve this, it is necessary to transform the complete query pipeline into a single LLVM IR function. Here it becomes apparent that a transformation from graph algebra to machine code can be easily accomplished with LLVM IR. However, to ensure reliable performance, we identified the following requirements for the IR code generation that must be met.

(1) Minimize stack allocation and avoid heap allocation.
(2) Process initializations only at the first entry point of the IR function.
(3) Process type information at (JIT) compile-time.
(4) Provide full compatibility to the AOT execution engine.

One significant advantage of query compilation over interpretation is that the tuple element type information can be handled at compile-time. The consequence of this is the absence of type conversions at run-time as code can be generated for individual types.

Starting from a graph algebra expression that forms an operator tree, each operator will be transformed into LLVM IR code. Further, each operator provides at least an *entry* and a *consume* IR basic block, representing the operator's start and end points. Complex operators comprise more basic blocks for the actual tuple processing, e.g., Join. Though, the general control flow starts at the entry basic block. After processing in further basic blocks, the control flow branches to the consume basic block to push the results to the next operator. A branching instruction links each consume basic block with the entry basic block of the succeeding operator, forming an inlined query pipeline. Fig. 4 illustrates the transformation process, starting from a query plan in the form of graph algebra. Furthermore, it shows each operator's return path, which is, for most cases, the loop header of the previous operator. The finish operator will be called after the complete scan. Depending on the query, it invokes the function return or the next query pipeline.

The query engine's current implementation provides two access paths for the query pipeline: the NodeScan and Create operator. Code generation for these operators is basically the same as for the normal operators. Both contain at least the entry and consume basic block. As an access path is always the first operator in the pipeline, it must also provide the actual generated

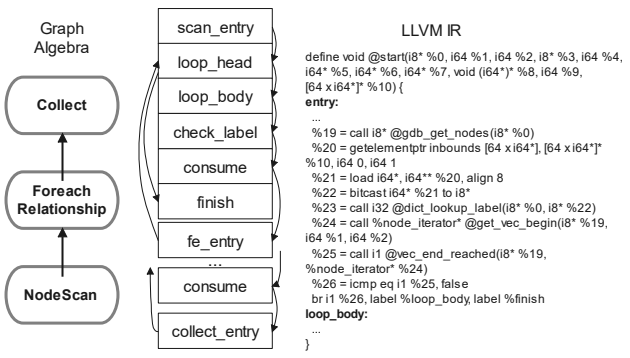

Figure 4: Graph algebra to LLVM IR transformation

function's entry point. Due to the reason that the code generator transforms the complete pipeline into a single IR function, memory allocation must be carefully handled. For this reason, the access path initializes all relevant values for the complete query pipeline, e.g., number of nodes, projection keys, or global constants.

The code generation for joins requires additional work. A join operator comprises two inputs. For now, we consider the right sub-pipeline of the join as the side which will be materialized. Consequently, it requires the prior execution of the right sub-pipeline. However, the actual code generation starts from the left sub-pipeline in order to minimize tree traverses. Whenever the code for an access path is generated, it checks if the current function is already initialized. If this holds, it swaps the function entry basic block with its own and connects the finish basic block of the second access path with the entry basic block of the previous access path. This enables the handling and execution of multiple query pipelines within a single function.

IR Code Generation. We use the *visitor design pattern* to generate the appropriate IR code for each operator. Further, this enables the extension of the query engine in future work. Each operator derives from a base class and implements a `codegen` method for code generation. The query engine calls the *visitor* to start the code generation process, which calls all the operator's `codegen` methods recursively. We implement several IR abstractions that help to generate IR code with more ease. Due to the reason that most operators rely on loops, we provide two IR loop abstractions. The `while_loop` abstraction is used for the iteration through a chunked vector. It receives the vector, the current iterator, the succeeding basic block, and the actual loop body as function arguments. The other loop abstraction is the `while_loop_condition`, used for the iteration as long as a condition is valid. Further, our abstraction set contains methods that generate code to extract label codes or property values. The operator IR code is based on these abstractions and mainly mimicking the push-based processing described previously. An additional structure is built to provide the type of tuple element at code generation and the appropriate register value. The code of the next operators is generated according to the type of the previous tuple element. For example, the projection operator uses the proper node functions if the last result tuple element is a node. This handling allows for generating code without much effort at run-time.

JIT Compilation. We extended the JIT compiler of LLVM to further features. First of all, our JIT query engine can persist already compiled code to PMem. This has the advantage that no further compilation is required for subsequent runs of a query. For that purpose, a persistent and concurrent hash map is used. The compilation output of the JIT is a binary object file that will be linked with the current database instance. Usually, this file is located in a memory buffer in the volatile memory. Before the compilation process, the query engine generates a unique query identifier that comprises the operators' identifiers, which will be used to lookup the persistent hash map for already compiled code. If the code is found, it will be linked with the current database instance. Otherwise, the compilation process of the query starts. The compiled code will be persisted in PMem after its compilation, using the query identifier as a key for the hash map.

A major advantage of JIT compilation is the ability to optimize the IR code at run-time. The LLVM framework provides a convenient approach for IR code optimization. Several LLVM

optimization passes can be used for this purpose. An analysis of the IR code reveals that it comprises mainly of loops and pointer arithmetics. Therefore, our optimization strategy focuses on these constructs. The following optimization pass cascade is used to further optimize the code at run-time:

- *Promote Memory To Register* transforms instructions that allocate stack memory into register values. This makes the IR code generation convenient and compliant with the requirement (1).
- *Control Flow Graph Simplification* merges and deletes basic blocks if they have common or no predecessors.
- *Loop Unrolling* removes the overhead of loops by explicit extraction of the body to multiple instructions.
- *Dead Code Elimination* eliminates unreachable code.
- *Instruction Combining* combines redundant instruction to form smaller and faster code with the same effect.

Additionally, the IR code is optimized with the standard C++ aggressive optimization (-O3).

Adaptive Execution. While the compiled query code itself is fast, the compilation time should also be considered. Notably, when executing short-running queries where only a small portion of data is touched, the compilation time will be longer than the actual execution time. In order to hide the compilation time as well as memory access latency of PMem, we additionally support an adaptive query processing approach, which is illustrated in Fig. 3. In contrast to the approach by [21], the adaptive execution can switch between only two modes, which is currently sufficient for our engine. The interpretation mode is always initiated first at query execution. This mode uses AOT-compiled database code to execute the query. Similarly, the visitor design pattern is used to transform the given algebra query plan into the interpret functions. These functions are then linked together, forming a cascade of functions that execute the actual query. The downside of such an approach is the additional (AOT-compiled) code overhead because every operator and its varieties must be available at compile-time. During adaptive execution, the query engine switches to the JIT mode after compilation.

We take advantage of the morsel-driven parallelism for the actual switching procedure, where morsels are pinned to a single task and pushed into a task pool. The working threads pull a task from the pool and execute the task function (the query) on the pinned morsel.

We implement the task function as a static function. As the execution always starts in the interpretation mode, it will be initialized to the appropriate function, which invokes the interpretation. While the query is executed in the interpretation mode, a background thread compiles the query plan into machine code. The compilation process emits a function that processes the query plan into machine code. As soon as the compilation is done, it redirects the static task function to the compiled function. The next pulled task from the pool will execute the compiled query function.

7 EVALUATION

In this section, we report the results of a set of experimental evaluations whose research goal is threefold:

1. We evaluate our PMem-based HTAP engine and show the effectiveness of our design decisions to exploit PMem characteristics for graph processing. In this context, we aim to investigate, on the one hand, the benefits of using persistent memory for graph processing. On the other hand, we compare our system to disk-based and DRAM-based solutions (§ 7.3).
2. We compare the speed of volatile, persistent, and DRAM/PMem hybrid B^+-Tree index lookups. We quantify the recovery overhead of our hybrid index (which we expect to be insignificant) as a trade-off for increased query performance (§ 7.4).
3. We evaluate our JIT query compilation approach. We demonstrate when and how much it enhances the performance of transactional queries. We expect the JIT compilation to yield benefits especially for long-running and more complex queries (§ 7.5).

7.1 Environment

For the experiments, we used a dual-socket Intel Xeon Gold 5215 server with 10 cores each at max. 3.4 GHz. The server is equipped with 384 GB DDR4 RAM, 1.5 TB Intel Optane DCPMM, and 4 × 1.0 TB Intel SSD DC P4501 Series connected via PCIe 3.1 x4. The server runs CentOS 7.8 (Linux 5.7.7 kernel). The operating mode of the PMem modules is set to AppDirect which allows us direct access to the devices. On the PMem DIMMs, we have created an ext4 file system and mounted it with the DAX option to enable direct loads and stores bypassing the OS cache. For accessing PMem, we used the Intel PMDK version 1.9.1 and libpmemobj-cpp[3] version 1.11. The JIT compilation was done with LLVM version 11.

7.2 Workload & Setup

The Linked Data Benchmark Council (LDBC) specifies benchmarks and benchmarking procedures and also verifies and publishes benchmark results [10]. The LDBC-Social Network Benchmark (SNB) models a social network comprising of different entity types interconnected by relationships – both with property types and property values. Activities of persons are represented as messages about topics or tags that are posted in forums moderated by unique persons. Persons like messages, have interests in tags, are members of forums, and make comments in response to posts or other comments. Message activities are the bulk of the data on the social network. There also are places and organizations to which a person is connected via residence, study, and work relationships. The LDBC-SNB defines an Interactive Workload and a Business Intelligence Workload. The Interactive Workload comprises of three classes of queries: (1) Interactive Complex Read Queries that are relatively complex and traverse a fair portion of the graph data, (2) Interactive Short Read Queries that perform lookups and short traversals within the neighborhood of a node, and (3) Transactional Interactive Update Queries that perform transactional insertions and updates of node and relationship objects [10].

We generated and used the LDBC-SNB data [2] at scale factor (SF) 10 as our benchmark data. As the focus of this paper is on transactional graph processing, not graph analytics, we selected the LDBC-SNB Interactive Short Read (SR) and the Interactive Update (IU) query sets as query workload for our experiments.

[3] https://github.com/pmem/libpmemobj-cpp

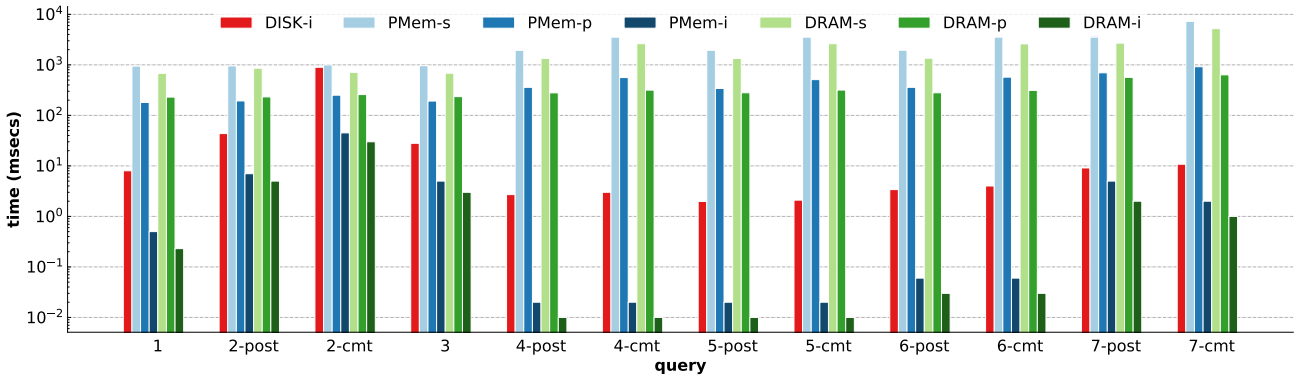

Figure 5: Results for SNB Short Reads

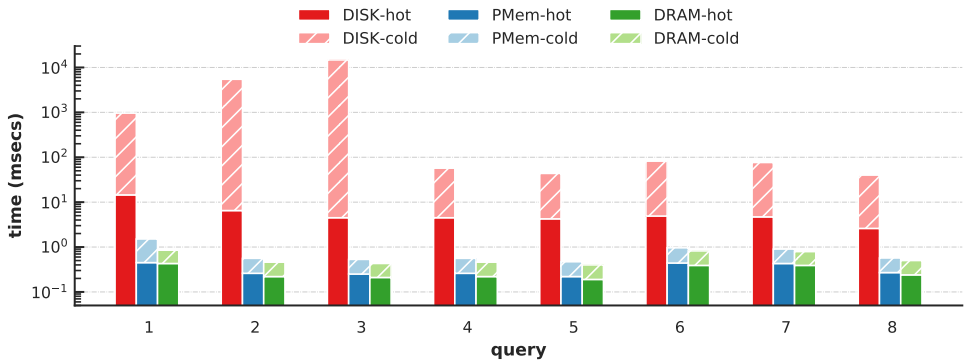

Figure 6: Results for SNB Interactive Updates

7.3 Benefit of PMem

We first want to evaluate how much the design decisions in our PMem-optimized graph engine and our implementation of transaction processing reduce the overhead of PMem in our system (denoted as PMem in the figures) compared to a pure DRAM-based in-memory implementation of it. Moreover, we want to compare the performance gains brought about by the lower access latency of PMem compared to a DISK-based system, in addition to providing persistence. To this end, we employ two baselines: A disk baseline (represented as DISK), which is an open-source native graph database where we stored all the primary data on SSD and created an additional DRAM index. For the DRAM baseline (depicted as DRAM), we adapted our system to additionally run in a pure volatile mode where we keep data entirely in DRAM. We expect our system to outperform the disk-based system. With regards to the DRAM baseline, we anticipate to bridge the performance gap with our PMem-conscious design and achieve a near-DRAM performance while providing persistence, especially for hot runs.

Interactive Short Reads. Fig. 5 shows the query execution times for the SR query set. The execution times are average times of 50 runs on hot data, each with a different input ID parameter. post and cmt (short for *post* and *comment* respectively) represent the two subclasses of a message entity. For PMem, we show the execution times without indexes for single-threaded execution (PMem-s), multi-threaded execution (PMem-p) as well as with indexing support (PMem-i). We employ similar denotations for our DRAM baseline: DRAM-s, DRAM-p, and DRAM-i. For the disk baseline, we also conducted executions with index support and report the performance numbers for hot runs (i.e., when the data is in DRAM), denoted by DISK-i. We used our hybrid DRAM/PMem implementation of the B$^+$-Tree (Section 4) for PMem, while for DRAM, we used a volatile B$^+$-Tree. We maintain the same set of indexes throughout our experiments.

The results in Fig. 5 show that exploiting PMem-specific characteristics in storage architecture and transaction processing can significantly bridge the performance gap between DRAM and PMem. It can be noted that for multi-threaded execution of some of the queries, the execution times are very close since the SR queries are short-running and the PMem latency is already hidden by the CPU caches for hot runs. An interesting research direction is thus to investigate this in the context of graph analytics, where queries are compute-intensive, long-running, and navigate across a significant portion of the graph. While the results show performance improvements of multi-threading both for DRAM and PMem, however, indexes have a stronger influence. Unlike graph analytics that significantly benefit from parallel execution, interactive queries like SR and IU benefit more from indexes, as they are essentially lookup queries whose execution time overhead comes mainly from scanning the tables of record chunks to retrieve the start node object. As a result, we compare the performance of indexed query execution both on our system and on the DISK baseline. We can see from the figure that our PMem-based system outperforms the disk-based system for indexed execution in all the queries, as we had expected.

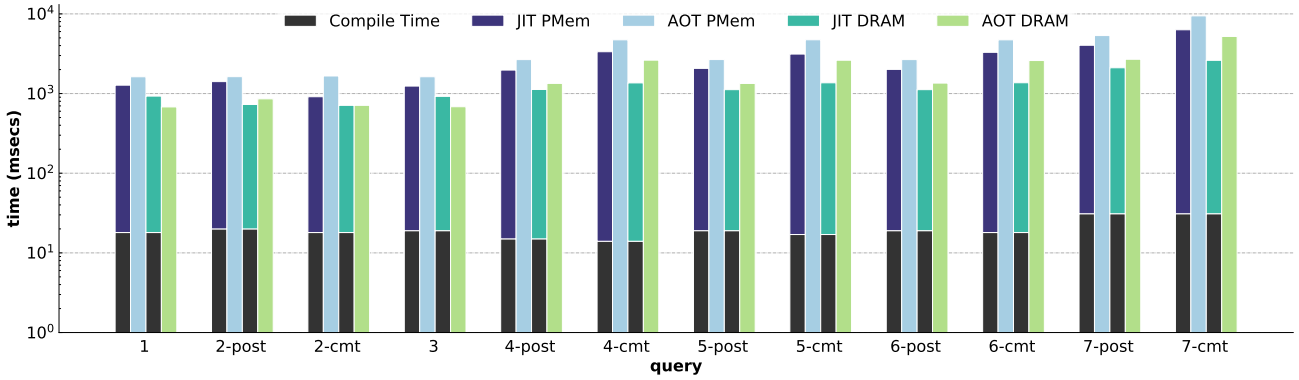

Figure 7: Results for SNB Short Reads with Single Threaded Execution

Interactive Updates. We maintain the indexed query execution and present the execution times for the IU query set with indexed support in Fig. 6. Here, we measured both the times to execute the update queries as well as times for the transactions to commit (i.e., notably, persisting the updates in PMem). Similar to the SR queries, we took the average execution time of 50 runs on hot data with varying object property values as input parameters. In addition to results on hot data, we also present the execution times for cold runs, i.e., for the first query runs. The results show our PMem-based system not only outperforms the disk-based system by an order of magnitude even for hot runs but also performs insert and update operations at near-DRAM latency. For hot data, it is even closer.

Overall, the results of Fig. 5 and Fig. 6 show that in direct comparison with the DRAM variant, our hybrid approach of MVTO implementation to address the specifics of PMem adds only a marginal overhead. This validates our MVCC design decisions of Section 5 and also obviates the need for showing the results of a pure PMem implementation which has an overhead of maintaining dirty versions on PMem.

7.4 Indexes and Recovery

We evaluated index performance and recovery by way of comparing our hybrid index that keeps inner nodes in DRAM, trading-off recovery for improved performance, against two baselines. One a volatile index that keeps all nodes in DRAM and the other a persistent index that stores all nodes on PMem. We evaluated them based on the average time for indexed scans in the SNB SR queries. To study the performance differences, we measured the time to lookup and retrieve a node ID from the appropriate index. Fig. 8 shows the average lookup time for the persistent, volatile, and hybrid indexes - denoted respectively as PMem, DRAM, and Hybrid. The lookup times are averages of all ID value lookups of nodes with the same label type (Person) in all the respective SR queries. The hybrid approach enhances the lookup performance by 2x while keeping the recovery time as low as 8 ms, in comparison to the complete volatile index build time of 671 ms. This recovery overhead would also be necessary for each index created on specific properties. Added up, the overhead of completely rebuilding the indexes in the volatile case is comparatively drastic. Therefore, we see the hybrid variant as a good compromise between runtime performance and recovery.

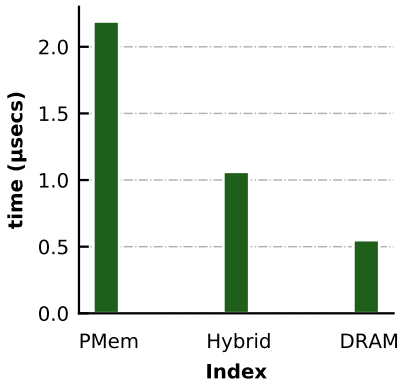

Figure 8: Average Time per Lookup of Persistent, Hybrid, and Volatile Indexes

7.5 JIT

The final part of our evaluation focuses on the JIT query compilation approach. The first two benchmarks show the capability of JIT-compiled code itself, without any mechanism to hide the compilation time. For this purpose, we execute the interactive read and update queries from the SNB. Thereafter, we examine the gain from adaptive execution. Although we expect it to be much more efficient for analytical and long-running queries, it is insightful to see the benefits on short and transactional queries in comparison to AOT-compiled code. In particular, the combination with PMem could make this approach profitable even for short-running queries.

Interactive Short Reads. Fig. 7 shows the results for the SR executed with the JIT query engine. We calculated the average execution time of 50 runs on hot data with different parameters. The queries are executed single-threaded without indexes. The compilation time of the queries is only a few milliseconds. As the number of operators increases, the compilation time increases by only a few milliseconds. However, the results show clearly that the JIT-compiled is always faster than the AOT-compiled code. The JIT-compiled code is mostly even faster when the actual compilation-time of the query is included. Especially more complex queries, like 7-POST and 7-CMT, can benefit from the JIT compilation approach.

Interactive Updates. The results for the IU executed with the JIT query engine are shown in Fig. 9. There are not many optimization possibilities for the generated IR code, as the queries are

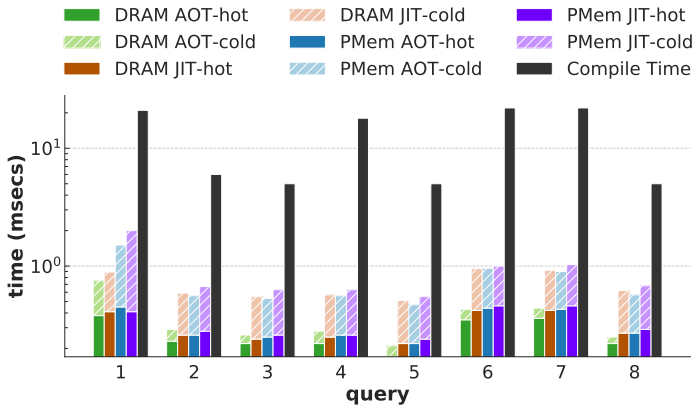

Figure 9: Results for SNB Interactive Updates executed with JIT compiled code

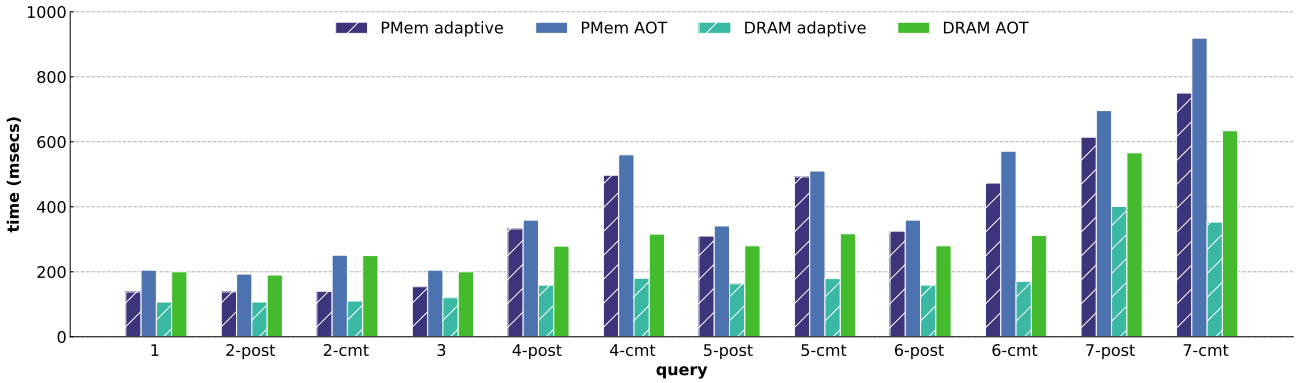

Figure 10: Results for SNB Short Reads with Adaptive Execution

short when index support is enabled. Executing these queries using scans and selections shows similar behavior to the benchmark before. However, here we focus on code for short queries, where the execution time is less than the compilation time. JIT code executed on cold data is noticeably slower, while the resulting performance on hot data is similar to the AOT code. However, executing these queries with the JIT compilation approach shows that it is not always the best option to generate code during runtime for executing a query. The compilation time for these short queries is much higher than the actual execution time. Furthermore, executing these short queries with the adaptive approach leads to the execution of the query pipeline using the AOT-compiled code entirely, which corresponds to the results of the AOT code in Fig. 9.

Adaptive Query Executions. The previous benchmarks show the capability of executing JIT-compiled queries. It is clearly visible that the JIT-compiled code itself outperforms the AOT code on DRAM and PMem. However, waiting for the compilation of the query limits the performance improvement of this approach. The adaptive query execution approach eliminates this problem by executing the AOT code while query compilation is done in the background. Additionally, this is useful to hide the memory access latency of PMem. The next benchmark compares the adaptive query execution approach using morsel-driven parallelism with multi-threaded AOT-compiled query execution. Similar to the previous benchmarks, we execute each query on DRAM and PMem. The results in Fig. 10 show that the adaptive execution is always faster than the multi-threaded AOT execution. The execution on PMem can particularly benefit from the adaptive approach. The additional latency introduced by PMem leads to an earlier execution of the fast JIT-compiled code in the query pipeline stage, which enhances the query processing. For the queries 1, 2-POST, 2-CMT, and 3, it leads to similar execution times for DRAM and PMem. The adaptive approach provides faster query execution times for most queries and in worst-case similar performance than multi-threaded AOT code. More complex queries can benefit even more from the adaptive approach as there is more space for code optimization, like for the queries 7-POST and 7-CMT.

8 CONCLUSION

Persistent memory represents a promising technology for data management solutions whose efficient use requires rethinking data structures and architectures. In this work, we have presented the first results of our PMem-based graph engine for hybrid transactional/analytical workloads. Based on the characteristics of PMem technology, we have discussed, implemented, and evaluated design choices regarding storage structure, transaction, and query processing. The promising results using the LDBC-SNB interactive short read and update query sets show that a PMem-based storage engine that is well-optimized for PMem characteristics incurs only a marginal performance overhead compared to a pure in-memory solution. The main benefits are, among others, the competitive performance without the need to

keep large parts of the data in (volatile) main memory (resulting in constant answer times both for cold and hot data) as well as near-instant recovery guarantees. Additionally, the results have shown that in comparison to AOT-compiled query execution, JIT compilation speeds up query processing when the compilation time is less than the execution time. Particularly, adaptive compilation further enhances query execution performance by hiding PMem access latency. In our ongoing work, we plan to investigate the behavior of complex graph analytics and highly concurrent updates. Moreover, there are several opportunities for further performance improvements, e.g., by employing more hybrid DRAM/PMem approaches such as for dictionaries.

ACKNOWLEDGMENTS

This work was partially funded by the German Research Foundation (DFG) in the context of the project "Hybrid Transactional/Analytical Graph Processing in Modern Memory Hierarchies (#TAG)" (SA 782/28-2) as part of the priority program "Scalable Data Management for Future Hardware" (SPP 2037) and by the Carl-Zeiss-Stiftung under the project "Memristive Materials for Neuromorphic Electronics (MemWerk)".

REFERENCES

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*. 431–446.

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).

[3] Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *ACM Comput. Surv.* 40, 1 (2008), 1:1–1:39.

[4] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018), 553–565.

[5] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *PVLDB* 10, 4 (2016), 337–348.

[6] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019).

[7] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *SIGMOD*. 121–132.

[8] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.

[9] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *PVLDB* 13, 9 (2020), 1598–1613.

[10] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD*. 619–630.

[11] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *CIDR*.

[12] Henning Funke, Jan Mühlig, and Jens Teubner. 2020. Efficient generation of machine code for query compilers. In *DaMoN @ SIGMOD*. 6:1–6:7.

[13] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *PVLDB* 13, 8 (2020), 1304–1318.

[14] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data Structure Primitives on Persistent Memory: An Evaluation. *CoRR* abs/2001.02172 (2020).

[15] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data structure primitives on persistent memory: an evaluation. In *DaMoN @ SIGMOD*. 15:1–15:3.

[16] Jürgen Hölsch and Michael Grossniklaus. 2016. An Algebra and Equivalences to Transform Graph Patterns in Neo4j. In *EDBT/ICDT*.

[17] Intel Corporation. 2020. Persistent Memory Development Kit. http://pmem.io/pmdk. Online, accessed December 2020.

[18] Muhammad Attahir Jibril, Philipp Götze, David Broneske, and Kai-Uwe Sattler. 2020. Selective Caching: A Persistent Memory Approach for Multi-Dimensional Index Structures. In *HardBD & Active @ ICDE*. 115–120.

[19] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *USENIX ATC*. 993–1005.

[20] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* 11, 13 (2018), 2209–2222.

[21] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE*. 197–208.

[22] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.

[23] 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.

[24] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *PVLDB* 13, 4 (2019), 574–587.

[25] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling Low Tail Latency on Multicore Key-Value Stores. *PVLDB* 13, 7 (2020), 1091–1104.

[26] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.

[27] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+-Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *PVLDB* 13, 7 (2020), 1078–1090.

[28] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.

[29] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SIGOPS SOSP*. 456–471.

[30] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN @ SIGMOD*. 8:1–8:7.

[31] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. 371–386.

[32] Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. 2015. GRAPHITE: an extensible graph traversal framework for relational database management systems. In *SSDBM*. 29:1–29:12.

[33] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.

[34] Andrew Pavlo and Matthew Aslett. 2016. What's Really New with NewSQL? *SIGMOD* 45, 2 (2016), 45–55.

[35] Holger Pirk, Oscar R. Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB* 9, 14 (2016), 1707–1718.

[36] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *DBPL*. 1–10.

[37] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *BTW*. 403–420.

[38] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, Muhammad Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. 2014. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*. 979–990.

[39] Steve Scargall. 2020. *PMDK Internals: Important Algorithms and Data Structures*. Apress, 313–331.

[40] Steve Scargall. 2020. *Programming Persistent Memory*. Apress.

[41] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*. 1907–1922.

[42] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *DaMoN @ SIGMOD*. 12:1–12:7.

[43] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *USENIX FAST*. 61–75.

[44] Tianzheng Wang, Justin J. Levandoski, and Per-Åke Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *ICDE*. 461–472.

[45] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.

[46] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792.

[47] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *USENIX ATC*. 349–362.

[48] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *USENIX FAST*. 169–182.

[49] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: Differential Indexing for Persistent Memory. *PVLDB* 13, 4 (2019), 421–434.