Adaptive Query Compilation in Graph Databases

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

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

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

2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW) | 978-1-6654-4890-1/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDEW53142.2021.00027

***Abstract*—Compiling database queries into compact and efficient machine code has proven to be a great technique to improve query performance and to exploit characteristics of modern hardware. Furthermore, compilation frameworks like LLVM provide powerful optimization techniques and support different backends. However, the time for generating machine code becomes an issue for short-running queries or queries which could produce early results quickly. In this work, we present an adaptive approach integrating graph query interpretation and compilation. While query compilation and code generation are running in the background, the query execution starts using the interpreter. As soon as the code generation is finished, the execution switches to the compiled code. Our evaluation shows that autonomously switching execution modes helps to hide compilation times.**

***Index Terms*—persistent memory, graph databases, adaptive query compilation, query interpretation**

I. Introduction

Query processing is one of the main tasks of any Database Management System (DBMS), along with transaction processing, data storage and management. For decades, relational databases have been the measure of all things when it came to performing these tasks. However, in recent years, there have been developments towards providing data models that extend, complement or even replace the relational model. Graphs are one of such data models. They are specifically designed to process and analyze relationships between data. Unlike relational DBMSs which rely heavily on join operations to capture relationships between data, Graph DBMSs store and access them inherently. This has some advantages, particularly with regards to performance of query processing.

When processing graph patterns in queries, it is noticeable that they often involve the processing of similar or same operators, possibly even in the same order. Further, the code to be executed may contain duplicate or dead code. A query interpreter would execute all of these statements one after the other, even if some of them do not contribute to the query result. The solution to this problem is query compilation. However, this also comes with a problem: the compilation time. While compilation is not an issue for long-running queries, it can be longer than the actual query runtime for short-running queries. Adaptive query compilation is one technique to solve these problems altogether. Essentially, the idea behind this technique is to integrate query interpretation and compilation into the processing. An interpreter is used at the beginning while the compilation is carried out in the background. When the compilation completes, the execution switches to the compiled code [1].

Our work is centered around Poseidon[1] [2], a hybrid transactional/analytical processing (HTAP) graph database that enables transactional graph processing on Persistent Memory (PMem) based on a property graph model. The design of its storage architecture is tailored to the characteristics of PMem, one of which is its higher latency and lower bandwidth relative to DRAM. In Poseidon, nodes, relationships, and properties are stored on PMem to offer similar advantages to an in-memory database while also guaranteeing persistence. The underlying data structures are persistent vectors, which are organized as linked lists of fixed-size arrays or chunks.

In this paper, we focus on the utilization of efficient techniques to hide both query compilation times and PMem access latency in graph query processing. Our contributions are as follows:

- We present an approach to generate efficient machine code from graph algebra expressions.
- As query compilation times and PMem access latency adversely affect performance, we demonstrate a technique that autonomously switches execution modes after compilation in order to hide compilation times and PMem latency.
- To provide a robust query engine and to avoid recurring compilation time overhead, we introduce the usage of a PMem query code cache.

II. Related Work

Query compilation is an actively researched technique to speed up query processing. There exist two basic approaches for databases to compile queries: high-level template-based and low-level intermediate representation (IR)-based compilation.

The high-level template-based approach, also referred to as template expansion, fills operator templates with the appropriate query arguments. A high-level compiler, e.g., GCC or clang, is used to transform the templates into machine-code. Hekaton is a database engine for Microsoft's SQL Server [3]. It provides a query compiler that transforms algebra plans through several optimization steps into high-level C code. An external C code compiler transforms this code into an executable format which is called to process the actual query. LegoBase provides another high-level compiler [4]. This query

[1]https://github.com/dbis-ilm/poseidon_core

978-1-6654-4890-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDEW53142.2021.00027

compiler also transforms the query plan via multiple steps, where declarative elements of the query are replaced with imperative high-level code. It is based on the Lightweight Modular Staging framework which converts Scala code to a graph-like IR. The code goes through multiple transformations into low-level and optimized C code by replacing the graph nodes with instructions or data structures used for query processing. LB2 is an extension of LegoBase, which uses Futamura Projections to combine an interpreter with a compiler [5].

Compiling high-level code introduces additional compilation time that reduces the resulting performance. The low-level IR-based approach avoids high-level compiler and generates IR code instead. [6] provides a query compiler using the LLVM compiler framework for the HyPer database. The key to the success of this work is to process tuples as long as possible in the CPU registers. Based on this work, [7] proposed an approach to mask the compilation time with an adaptive approach. A special virtual machine that mimics the LLVM IR instructions is used to interpret the query while the compilation runs. This reduces the waiting time for compilation and can improve the handling of short-running queries whose compilation time would be longer than their execution time [5]. The ideas from this work build the fundament of our approach for graph database query processing.

Apart from LLVM, there exist other approaches that introduce their own low-level IR to compile queries. One of the critical factors when (just-in-time) compiling code is register allocation. [8] showed an optimized approach by providing a lightweight intermediate representation that estimates value lifetimes before code generation. The Voodoo IR is a declarative algebra especially for many-core architecture that provides a set of vectorization instructions to generate OpenCL code [9]. Query compilation is also employed in existing commercial graph database systems like Neo4j and TigerGraph. Neo4j introduced the pipelined (known in older versions as compiled) runtime that transforms Cypher queries into Java bytecode. This bytecode is then executed and optimized at runtime by the HotSpot Java virtual machine. Besides a query interpreter, TigerGraph also supports a compiler that transforms GSQL queries into C++ code. The generated C++ code is then compiled into a dynamic library and linked with the database instance [10]. However, to our knowledge, there exists no graph DBMS that exploits adaptive query compilation to enhance query processing on PMem.

III. Poseidon Graph Database

Poseidon is a native graph database based on the property graph data model. Nodes, relationships, and properties are stored in PMem to offer similar advantages to an in-memory database while also guaranteeing persistence. The nodes, relationships, and properties are stored in separate persistent tables (vector of chunks), which are organized as linked lists of fixed-size arrays of object records. The corresponding data structure design for nodes is illustrated in Fig. 1.

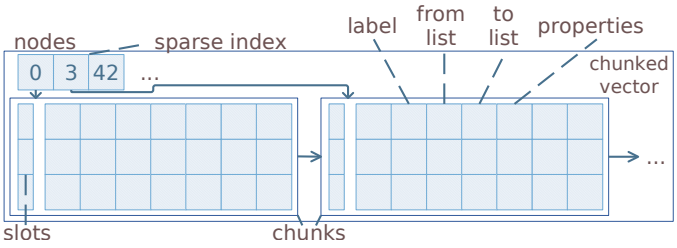

Fig. 1. Graph Data Structure

The chunk vector is optimized for sequential access to fully utilize PMem. Each row in a chunk represents a slot for storing a node record. For efficient reclamation of deleted entries, a bitmap is used in each chunk to mark empty and used slots. The chunks are linked by persistent pointers, forming a linked list. A scan over all nodes in a graph is achieved by traversing the linked chunks. On top of this, a sparse index is used which maps the offsets (identifiers) of the first records of each chunk to their memory location.

Each node record stores the id (offset) of the node, its label, the offsets of its first outgoing and its first incoming relationship, and the offset of its properties. The offsets are used to retrieve the node's relationships and properties from the corresponding chunk vectors. A relationship record consists of the relationship's id (offset), its label, the offset of its properties, the offsets of its source and destination nodes, as well as the offsets of the next relationships of its source and destination nodes. This way, it is possible to traverse all relationships of all nodes, i.e., traversing the entire graph. Furthermore, properties of nodes and relationships are stored in a separate chunk vector as key-value pairs.

IV. Query Processing

The query processing engine of Poseidon provides different approaches to execute queries expressed in graph algebra. We refer to them as *execution modes*. As the Poseidon graph database storage layout aims to exploit PMem and the read access on PMem is slower than on DRAM, it requires the hiding of additional latency by efficient cache utilization and multithreading. Additionally, we use the execution modes for further latency hiding. We now describe the query engine of Poseidon.

A. Graph Algebra

Graph Algebra is the starting point of Poseidon's query engine. We adopted the graph algebra proposed by [11], which extends the relational algebra by two main graph operators:

- GetNodes scans the nodes of the graph.
- Expand visits neighbouring nodes by traversing incoming and outgoing relationships.

Since this algebra also extends the concept of a relation – a graph relation, the usual operators from relational algebra can be further applied to the results of the graph operators. We provide these graph algebra operators as well as the

traditional relational algebra operators in our query engine. In addition, we implement a simple query language based on graph algebra. This is mainly intended to simplify the parser development. An example query in this language is

```
Expand(OUT, "Person",
  ForeachRelationship(FROM, ":knows",
    NodeScan("Person")))
```

which finds all pairs of friends by traversing all the outgoing relationships of type `":knows"` for each person node to retrieve the person nodes connected to it.

The implementation of Cypher or the GQL standard is planned for future work.

B. Push-based Query Processing

We scan the underlying persistent storage sequentially in order to retrieve nodes or relationships from a graph. As PMem is not block-oriented but rather byte-addressable, there is need for optimized sequential access. Therefore, we opt for a multi-threaded push-based query engine. Besides the advantage that the data flow corresponds to the control flow, in contrast to the classic iterator model, this choice also has advantages for code generation, as shown by [12].

The access paths of algebra queries are scans (NODESCAN, RELATIONSHIPSCAN, INDEXSCAN) and create (CREATENODE, CREATERSHIP) operations. Scan operations initiate graph traversals by scanning the node or relationship tables. Each node (or relationship) that matches the given label and satisfies the property filter predicate will be forwarded separately to the subsequent operator. In the case of a create operation, the newly created node or relationship will be forwarded to the next operator. Subsequent operators append their result (i.e., generated tuple element) to the existing tuple, forming a list of tuple elements. An operator can access any element in the tuple from the previous operator. The tuples are pushed until a pipeline-breaker is reached. A pipeline-breaker is the last operator in a query pipeline which stores the tuples from the registers into (intermediate) storage.

We have implemented all graph algebra operators in C++ and refer to these functions as our ahead-of-time (AOT)-compiled query engine. This makes it convenient to run queries directly with high performance, as the AOT-compiled code is optimized (using -O3 as optimization flag for the compiler). This code builds the basic blocks for the interpretation execution mode.

C. Query Interpretation

Query interpretation is a straightforward way to execute queries, whereby for each operator of a given query, the appropriate AOT-compiled function is simply called. For this purpose, we use the *visitor pattern* [13]. Each operator is linked to the appropriate AOT-compiled function. The single-operator functions are then linked together, forming a cascade of functions that executes the actual query. A downside of this approach is the additional effort in the implementation of each query operator. Each possible tuple element type must be handled explicitly in the code. The resulting code is heavily template-based and requires runtime type information (RTTI).

D. Query Compilation

Query compilation is a well-known technique to speed up query processing. However, it comes with various accompanying factors that must be treated appropriately in order to achieve this speedup. These include the selection of the compiler backend, intermediate representation (IR) language and the general control flow of the program. The following describes the query compiler engine of Poseidon and the handling of these factors.

1) Compiler Design: We chose LLVM as the compiler backend for generating code for queries just-in-time (JIT). LLVM has several advantages over other compiler backends like libFirm[2] or C–[3]. First, it provides a low-level IR language for code generation. The compilation of low-level code is faster than high-level code like C++. Second, LLVM enables further optimization steps to improve the resulting code performance by executing optimization passes on the IR code. The actual passes optimize IR code using well-known code optimization techniques like dead code elimination, loop unrolling, and instruction combining. The resulting code exhibits higher performance as it uses fewer and more efficient instructions. Third, it provides tools for implementing a JIT compiler. Generating and compiling code at run-time is the major requirement of a query compiler. Implementing these tools is a challenging task because problems like register allocation and instruction selection/scheduling need efficient solutions [14]. Lastly, LLVM provides a backend for several architectures and GPUs. With this solution, we can port the query compiler to another architecture with ease.

Based on the LLVM backend, we outline four requirements that must be fulfilled in order to generate high-performance query code. These requirements are derived from the work of [1], [7] and our experiments with different code generation and compiler setups.

- **(RQ1)** Processing of tuple results as long as possible in registers
- **(RQ2)** Pre-processing required initializations and space
- **(RQ3)** Tuple element type handling at code generation
- **(RQ4)** Full compatibility with AOT-compiled code

To meet (RQ1), all instructions of the generated code have to be inlined in a single function. This is crucial to processing the tuples as long as possible in registers without materialization. As the push-based approach processes one tuple result at a time, the actual tuples can be dematerialized directly into registers and only materialized at pipeline-breakers. Besides, the work of [1] shows that this approach requires fewer instructions, thereby enhancing performance with respect to query runtimes.

[2]https://github.com/libfirm/libfirm
[3]https://www.cs.tufts.edu/~nr/c--/

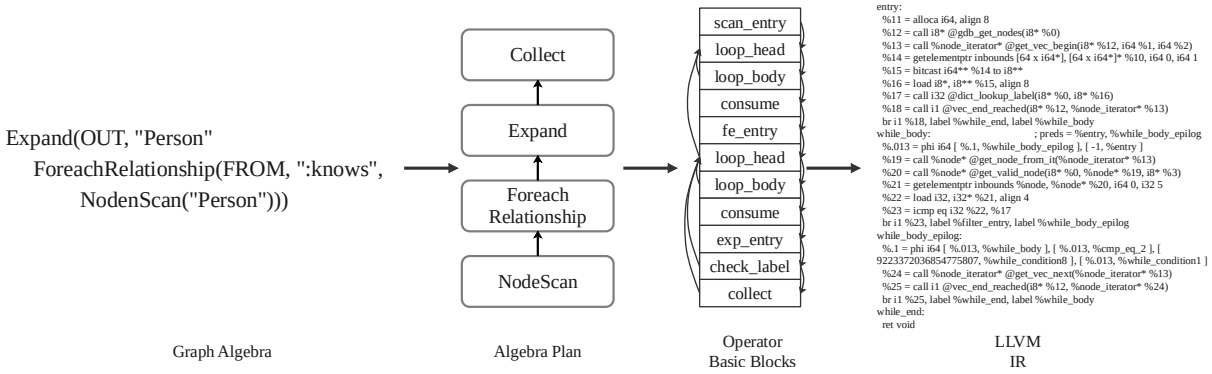

Fig. 2. Transformation steps from Graph Algebra to LLVM IR Code

(RQ2) specifies memory allocations in initialization to be done outside of the generated code. Allocating memory is a costly process and can hinder the resulting performance of the code. Keeping allocations out of the generated code reduces execution times. The necessary space to process a query is known before code generation and can be obtained by analyzing the query, e.g., by the number of projections or tuple element types.

One disadvantage of using query interpreters is the explicit type handling of tuple elements during query processing. For each possible tuple element type, there must exist the appropriate function to process it according to the given query. The type of the tuple element must be checked to call the corresponding function. This introduces additional control flow and increases the processing time of the query. Query compilation can eliminate this behavior by generating code only for the needed tuple element type, in conformity to (RQ3). The information of each tuple element type in the query is known before code generation, e.g., a projection of the id property of a node is of integer type. This can be used to generate only the code necessary for the tuple processing without explicit type checking.

It is not necessary to generate query code completely in LLVM IR. For example, the aggregation operator processes the same operations in every query. Therefore, there is not much room for further optimization steps. Hence, it can be implemented in C++ and called from the generated code. This however requires the generated code to be fully compatible with AOT-compiled query engine, as outlined in (RQ4), and is also beneficial when switching between execution modes.

2) IR Code Generation: The starting point of our query compiler is a query expressed in graph algebra. To fulfill the requirement (RQ4), the query engine provides a data-centric code generation approach, where each graph algebra operator uses the (inlined) push-based interface. We similarly make use of the visitor pattern to generate the appropriate IR code for each operator. (RQ1) requires that the complete query pipeline is inlined into a single (IR) function. Therefore, operators are handled differently from linking the callback functions in the AOT/interpretation approach.

Fig. 2 illustrates the whole transformation process from a given graph algebra query into LLVM IR. In the LLVM IR, each function comprises of multiple basic blocks that contain the instructions to be executed. The last instruction of a basic block is a terminator that either branches to another basic block or returns to the caller. In our approach, an operator consists of at least two basic blocks. The first basic block is the entry point of the operator and it executes the actual work. A complex operator with different control flow (conditions, loops) introduces additional basic blocks. The last basic block of an operator is the consume block. An emitted tuple result of an operator can only reach this basic block if it should be pushed to the next operator, e.g., when the predicate of filter operator is fulfilled for this tuple. The purpose of the consume block is to branch to the next operator via its entry basic block. However, whenever a condition is not fulfilled, the control flow branches back to the previous operator, which is defined as the re-entry point and is usually the condition block (loop head) of the previous operator. Ultimately, this forms a chain of operators that represents the given query in IR. An example basic block chain from a query plan is shown is Fig. 2.

The chain of basic blocks needs to be adapted when the query pipeline contains multiple sub-pipelines, by way of pipeline-breakers like join or aggregation operators. When generating code for joins, we choose the right side of the pipeline to be processed first, for simplicity. In other words, the tuples of the right pipeline are materialized before the tuples of the left pipeline. LLVM makes the conversion of this inline pipeline into IR code easy. The right side of the pipeline can be processed independently from the left side. Only the concatenation of the entry basic block and the pipeline-breaker must be adapted. For this purpose, the entry point of the function is fixed to the entry basic block on the right side and the finish basic block is linked to the entry basic block on the left side. This approach allows for code generation for a query pipeline with multiple sub-pipelines in one inlined function.

Generating IR code is more complex than implementing the operators in a high-level language like C++. We implemented different abstractions to facilitate the implementation of the

operators in LLVM IR code. Loops are an often-used control flow pattern in query operator code. Therefore, we provide several loop abstractions that help to write IR code for query operators. An IR loop consists of at least two basic blocks: the loop head where the condition will be evaluated, and the body where the instructions of the loop are executed. There are basically two types of loops in the generated query code: a *for*-loop with a loop counter and a head controlled *while*-loop. For both types, we provide a high-level interface to generate the appropriate code. The loop body can be passed as a C++ lambda function where further abstractions or IR low-level code are used. This enables to write low-level code similarly to high-level code and generally facilitates the implementation. Overall, in spite of these abstractions bringing the code style closer to high-level, there is still operator-dependent code that must be handled directly in low-level IR code.

Pointer arithmetic is another common construct that occurs in query code. To get the corresponding ids and offsets, pointers of structures must be followed. This can be done in LLVM IR using the *GetElementPointer* (GEP) instruction. Since operators quite often access fields of a structure, e.g., to get the id of a node or the offset of the next relationship, we implemented further abstractions for this purpose to ease the development. For each occurring structure field, we provide an abstraction that executes the GEP to retrieve the field value followed by a load instruction to move the value into a register.

E. Code Optimization

The compiled query code is fast even without optimization. But applying the LLVM passes can additionally improve the memory usage and execution of a query. We generate machine-code generically, which means that we create basic blocks with instructions that are not accessible (dead-code). In order to eliminate this dead-code and combine often-used instructions, we use the following cascade of optimization passes:

- Promoting Memory to Register
- Control Flow Simplification
- Dead-Code Elimination
- Instruction Combining
- Dead-Store Elimination

After executing these passes, the resulting query code contains only the instructions needed to process the tuples. Additionally, all memory access is transformed into register access, which improves the execution performance.

F. Code Caching

Compiling every query each time into machine code would be too costly. When compiling similar queries every time, i.e., queries with the same operators but different arguments like labels, the repeated compilation negatively affects performance. To solve this problem, we have implemented a cache for storing the code of compiled queries directly on PMem. For this purpose, we use a persistent map to store and retrieve compiled code. Before compilation, a query is assigned a unique key which, for simplicity, is the concatenation of the names of its constituent operators. This key is used to find existing compiled code for the query. The compiled code is used for query processing if it is present in the map. Otherwise, the new query will be compiled and the code stored in the persistent map with the already obtained key. Only the generated machine code is cached in PMem, allowing to retrieve and call it with different parameters, i.e., different labels or properties for each operator.

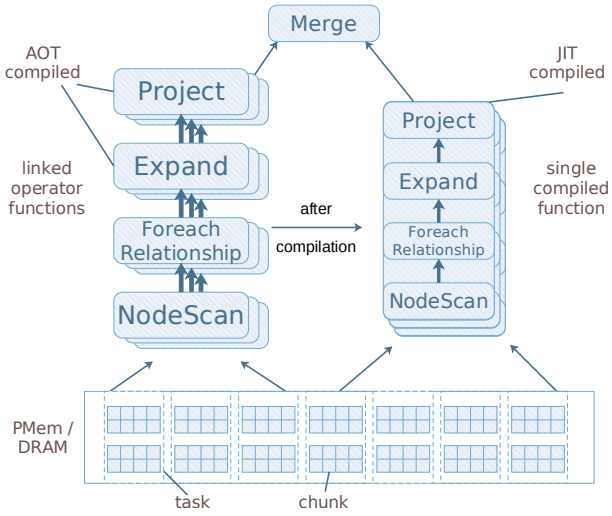

Fig. 3. Adaptive Query Compilation

V. Adaptive Query Processing

Although the compiled query code is fast, a problem arises when executing short-running queries that touch only a few tuples. The compilation process would consume more time than the actual query runtime. In this section, we show an approach to hide the compilation times and additionally the PMem access latency. The goal is to bring closer the runtimes on PMem to those on DRAM.

A. Parallelism

As our adaptive query engine makes use of parallelism, we first describe how it is implemented.

The basic principle of parallelism is adapted to the underlying storage layout of Poseidon. As mentioned before, nodes and relationships are stored in a chunk vector data structure. The number of existing chunks is always known beforehand and each chunk can be accessed individually. Thus, we implemented parallelization according to morsel-driven parallelism [15], whereby chunks are assigned to morsels, which are fine-grained task packages. The task packages are then pushed onto a task pool, where workers (threads) pull the task to perform the work, i.e., the actual query. To each morsel, the start and end chunks are assigned, which enables the processing of a range of chunks in one task. In the case of single-threaded execution, all chunks are assigned to a single morsel. The last step of the query merges all results from the morsels and returns them to the caller.

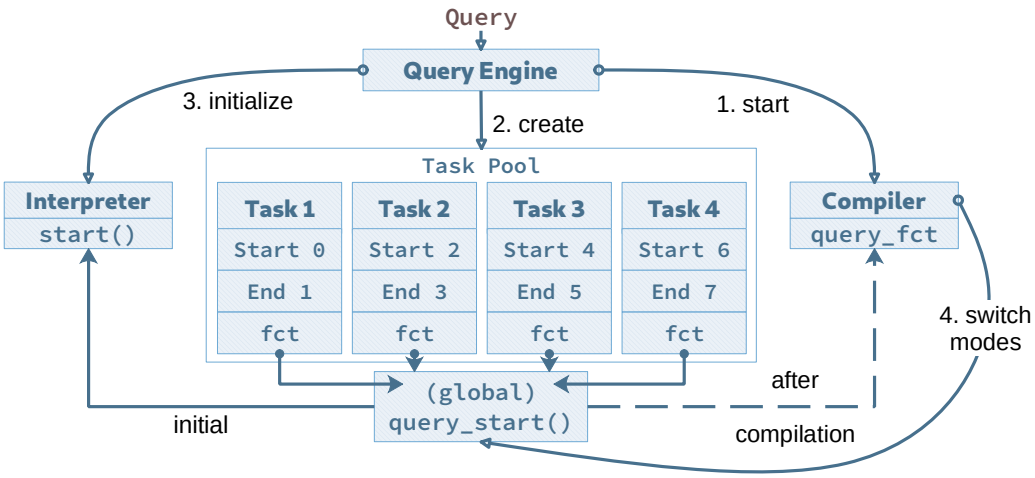

Fig. 4. Adaptive Compilation Flow

Morsel-driven parallelism can fully utilize sequential access to PMem storage. Therefore, it is well suited to fulfill our requirements. Additionally, it provides a solution to enable NUMA-locality, which also enhances the query processing performance.

B. Switching Modes

In order to fully utilize the benefits of query compilation, we provide an adaptive query compilation approach. In contrast to the approach by [7], we dispense of implementing a virtual machine but use the AOT-compiled code for interpretation. Each query is initially executed in interpretation mode while a thread working in the background compiles the query into machine code.

Furthermore, we make use of the morsel-driven parallelism to switch the mode at query processing, as shown in Fig. 3. A complete overview of the adaptive query compilation flow is shown in Figure Fig. 4. At first, a query compilation process is initiated with the given graph algebra query as input. Meanwhile, the engine initializes the task pool with the morsels to task assignments and the interpreter. Each morsel is pinned to its respective task which calls a static function that executes the given query on the morsel. Initially, the static function is set to the interpreter that concatenates the AOT-compiled operators to interpret the query. After compilation, the task function is switched to the newly compiled query code. When the workers process the next morsel the compiled query function will be executed.

Queries that introduce pipeline-breakers require more effort. Each sub-pipeline needs to be compiled into a single function that can be called by the workers. The reason for this lies in the fact that an inlined function executing multiple query pipelines cannot efficiently be scheduled. Considering join as example, the left side of the pipeline can only be executed when the right side has been completely materialized into intermediate storage. A possible solution would be to schedule the task with blocking methods that reduce the parallelization. To provide a solution without blocking we generate an individual IR function for each sub-pipeline and switch the static task function to the appropriate sub-pipeline function after materialization.

C. Code Caching Latency

Caching code in persistent memory is beneficial for later query execution, as no recompilation is necessary. However, retrieving code from PMem also introduces an overhead. The overhead is on the one hand the additional access latency of PMem and the setup time for the retrieved code, e.g., for memory management and linking. The adaptive query compilation approach treats this overhead similarly to the compilation. The query engine starts with the interpretation while the process of retrieving code from the cache starts. This hides the additional code caching overhead and the PMem latencies effectively. Additionally, this technique is also suitable to hide the latencies when storing the code on HDD/SSD.

VI. Evaluation

We now present our experimental evaluation. We show that both JIT compilation and adaptive compilation speed up query execution in comparison with AOT-compiled code. Then we showcase how the adaptive approach tackles the problem of query compilation for short-running queries, where the query compilation time could potentially be more than the actual query execution time. In addition, we also show the benefit of the adaptive approach in hiding PMem access latency.

A. Environment & Workload

We conducted our experiments on a dual-socket Intel Xeon Gold 5215. Each socket has 10 cores running at a maximum of 3.40 GHz. The system is equipped with 384 GB DRAM, 1.5 TB Intel Optane DCPMM and 4x 1.0 TB Intel SSD DC P4501 Series connected via PCIe 3.1. It runs on CentOS 7.9 with Linux Kernel 5.10.6. For directly accessing the PMem device, which is operating on AppDirect mode, we use the Intel Persistent Memory Development Kit (PMDK) version

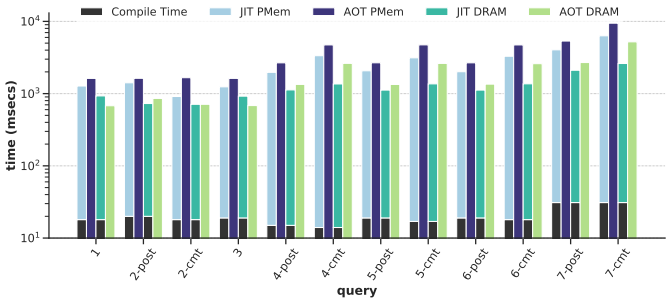

Fig. 5. SNB Short Read Queries on PMem and DRAM (Single-Threaded)

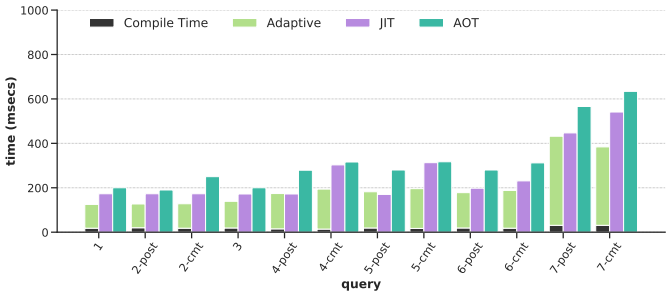

Fig. 6. SNB Interactive Short Read Queries on DRAM (Multi-Threaded)

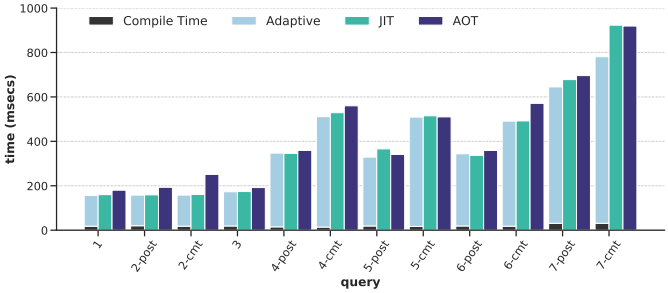

Fig. 7. SNB Interactive Short Read Queries on PMem (Multi-Threaded)

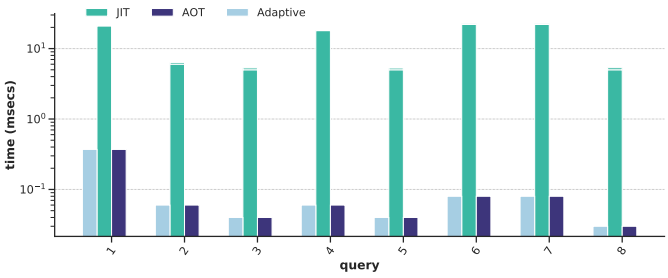

Fig. 8. SNB Interactive Update Queries on DRAM (Indexed)

1.9.1 and libpmemobj-cpp version 1.11. Furthermore, we have created an ext4 filesystem on the PMem DIMMs, mounted with the DAX option to enable direct loads and stores. We use LLVM version 11.0.1 for the compilation.

For our workload, we use the Linked Data Benchmark Council's Social Network Benchmark (LDBC-SNB). Since the benefit of the adaptive approach in hiding compilation time is more pronounced for short-running queries, we specifically target the SNB Interactive Short Read queries, which perform lookups and short graph traversals, and the Interactive Update queries, that update node and relationship objects. We ran the queries on the SNB data at scale factor 10 with different parameters and access the message class from the post and comment subclasses, referred to in the benchmark results as post and cmt respectively.

B. Benchmark Results

We first ran single-threaded executions of the SNB Interactive Short Read queries with JIT-compiled code and single-threaded. No adaptive switching between execution modes is done, and thus, the query execution waits for the compiled code. Fig. 5 shows that the JIT-compiled code always results in faster execution than AOT-compiled code on both PMem and DRAM. For most queries, it is still faster even with the compilation time combined i.e., only in the first runs of the queries. In subsequent runs of the queries, however, there is even no compilation time overhead since the cached code is executed.

Next, we evaluate the adaptive and (non-adaptive) JIT compilation while utilizing multi-theading. Fig. 6 and Fig. 7 depict the evaluation results of the queries on DRAM and PMem respectively. The results clearly show that the adaptive approach yields performance benefits both on PMem and DRAM. For most of the queries, it exhibits shorter execution times than the optimized AOT-compiled code, while in some of them, it results in at least the same execution times. Additionally, the figures also highlight how much of the total adaptive execution time is spent on the compilation. During this time, the query is executed in the interpretation execution mode before switching to the compilation execution mode. Furthermore, it can be observed from the figures that with the adaptive execution on PMem, we achieve near-DRAM execution times for queries 1, 2-CMT, 2-POST, and 3. This demonstrates the effectiveness of the adaptive approach in hiding PMem access latency. Multi-threaded execution of queries brings the execution times on PMem closer to those on DRAM, as also seen in the execution time of AOT-compiled

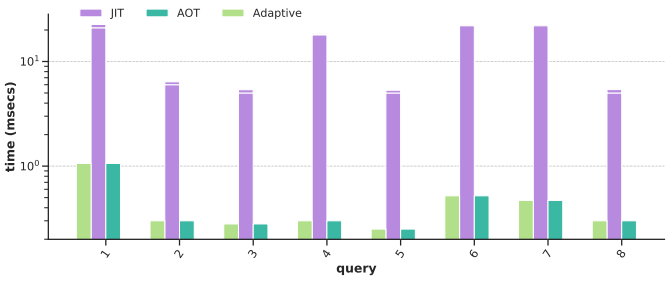

Fig. 9. SNB Interactive Update Queries on PMem (Indexed)

code. Adaptive query compilation bridges the latency gap even further, as the execution of the AOT-compiled code at the same time as the compilation also helps to hide PMem access latency. The non-adaptive execution approach yields a performance between that of the adaptive and AOT-compiled approaches. This is due to the blocking compilation process and additional overhead which arises from the management of resources (threads, memory). In general, the difference between the non-adaptive and adaptive approaches is the compilation time.

With the next queries, we show the worst-case of query compilation, i.e., when the actual processing time is less than the compilation time. Fig. 8 and Fig. 9 show the execution of the SNB Interactive Update queries using index scans as access paths. Using scans as access paths results in similar behavior as the Short Read queries. The compilation time of these queries lies around 5 - 10 msecs. However, as we execute the queries with indexes to retrieve the nodes with the appropriate properties, the processing time is below 1 msec. Waiting for the compiler here is not an option, as it lasts 100 times the actual processing time. The adaptive approach uses only the interpreter for these queries. Therefore, adaptive query compilation is a suitable technique to hide the compilation time for the Short Read queries.

Our experiments with code caching have shown a small overhead when retrieving the code. The overhead on DRAM is around 5 msecs and between 10-15 msecs on PMem. Consequently, the time of compilation and retrieving the short queries are similar and results also in a similar performance. Though, code caching is more beneficial for queries with longer compilation times.

VII. Conclusion

In this paper, we demonstrated an approach to transform graph algebra expressions into optimized machine code using LLVM. A common problem with query compilation is in the execution of short-running queries. The compilation time can be much higher than the actual query execution time. In this regard, we showed an approach to solve this problem with adaptive query compilation. This approach simultaneously interprets the graph query using AOT-compiled code and compiles it in the background into fast machine code. Upon completion of the compilation, it autonomously switches the execution to the compiled code. This approach inherently provides fast runtimes through efficient machine code. Moreover, it also allows for hiding PMem access latency. As a result, we achieve near-DRAM runtimes for some queries. For short queries where the compilation time would exceed the processing time, we can guarantee the performance of interpretation, which is a worthwhile tradeoff. This work has shown adaptive query compilation for the graph database Poseidon. Nonetheless, this technique is also suitable to hide the compilation time in relational databases.

For future work, we plan to push the query engine further to guarantee these runtimes for all queries. To this end, we plan to support more graph algebra operators and devise a cost model for the runtime of queries to be able to autonomously decide between interpretation and compilation before execution.

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] T. Kersten, V. Leis *et al.*, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proc. VLDB Endow.*, vol. 11, no. 13, pp. 2209–2222, 2018.

[2] 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 *24th Int. Conference on Extending Database Technology (EDBT) 2021, Nicosia, Cyprus*, 2021.

[3] C. Freedman, E. Ismert, and P. Larson, "Compilation in the microsoft SQL server hekaton engine," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 22–30, 2014.

[4] A. Shaikhha, Y. Klonatos *et al.*, "How to architect a query compiler," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1907–1922.

[5] R. Y. Tahboub, G. M. Essertel, and T. Rompf, "How to architect a query compiler, revisited," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 2018, pp. 307–322.

[6] T. Neumann and V. Leis, "Compiling database queries into machine code," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 3–11, 2014.

[7] A. Kohn, V. Leis, and T. Neumann, "Adaptive execution of compiled queries," in *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 2018, pp. 197–208.

[8] H. Funke, J. Mühlig, and J. Teubner, "Efficient generation of machine code for query compilers," in *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*. ACM, 2020, pp. 6:1–6:7.

[9] H. Pirk, O. R. Moll, M. Zaharia, and S. Madden, "Voodoo - A vector algebra for portable database performance on modern hardware," *Proc. VLDB Endow.*, vol. 9, no. 14, pp. 1707–1718, 2016.

[10] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, "Tigergraph: A native MPP graph database," *CoRR*, vol. abs/1901.08248, 2019. [Online]. Available: http://arxiv.org/abs/1901.08248

[11] J. Hölsch and M. Grossniklaus, "An algebra and equivalences to transform graph patterns in neo4j," in *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016*, 2016.

[12] A. Shaikhha, M. Dashti, and C. Koch, "Push versus pull-based loop fusion in query engines," *J. Funct. Program.*, vol. 28, p. e10, 2018.

[13] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design patterns: Abstraction and reuse of object-oriented design," in *Software Pioneers*, M. Broy and E. Denert, Eds. Springer Berlin Heidelberg, 2002, pp. 701–717. [Online]. Available: https://doi.org/10.1007/978-3-642-59412-0_40

[14] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Computer languages*, vol. 6, no. 1, pp. 47–57, 1981.

[15] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, 2014, pp. 743–754.