Distributed and Parallel Databases (2023) 41:359–386
https://doi.org/10.1007/s10619-023-07430-4

Adaptive query compilation in graph databases

Alexander Baumstark[1] · Muhammad Attahir Jibril[1] · Kai-Uwe Sattler[1]

Accepted: 20 April 2023 / Published online: 12 May 2023

Abstract
Compiling database queries into compact and efficient machine code has proven to be a great technique to improve query performance and exploit characteristics of modern hardware. Particularly for graph database queries, which often execute the exact instructions for processing, this technique can lead to an improvement. Furthermore, compilation frameworks like LLVM provide powerful optimization techniques and support different backends. However, the time for generating and optimizing 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. When the code generation is finished, the execution switches to the compiled code. Our evaluation of the approach using short-running and complex queries show that autonomously switching execution modes helps to improve the runtime of all types of queries and additionally to hide compilation times and the additional latencies of the underlying storage.

Keywords Adaptive query compilation · Persistent memory · Graph databases · Query execution modes · Adaptive code switching

1 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 performing

✉ Alexander Baumstark
alexander.baumstark@tu-ilmenau.de

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

Kai-Uwe Sattler
kus@tu-ilmenau.de

1 Technische Universität Ilmenau, Ilmenau, Germany

these tasks. However, in recent years, there have been developments toward providing data models that extend, complement, or even replace the relational model. Graphs are one such data model. 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 regard to the performance of query processing. Furthermore, nowadays the actual data of a DBMS is stored on different storage types. Today's memory hierarchy of systems on which the DBMSs are executed is characterized by different access speeds and bandwidths at each level. This has an impact on the execution of queries, which may be much slower than on traditional systems. Persistent Memory (PMem) and different types of SSDs are some of these additional storage types. The other characteristics of these storage types require adapting requirements for modern DBMSs. Through the work presented here, we show a possibility around the higher latencies of these memory types. Techniques to hide latencies range from adapting the data structures to the characteristics to interleaving the execution of a query with other processes. Another technique to achieve the low latency requirement is query compilation, which enables this by producing highly optimized code for the execution of queries. Further, when processing graph queries, it is noticeable that they often involve the processing of similar or the same operators, possibly even in the same order. The code to be executed may contain duplicate or dead code. A query interpreter would run all of these instructions one after the other, even if some of them do not contribute to the query result. This problem can be evaded with query compilation. However, this also comes with a further 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 initially while the compilation is carried out in the background. When the compilation completes, the execution switches to the compiled code [9]. Our work is centered around Poseidon[1] [8], a hybrid transactional/analytical processing (HTAP) graph database that enables transactional graph processing on PMem and DRAM based on a property graph model. The design of its storage architecture is mainly tailored to the characteristics of PMem, which comes with higher latency and lower bandwidth compared 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.

[1] https://dbgit.prakinf.tu-ilmenau.de/code/poseidon_core.

– 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.

Our experiments show that executing JIT-compiled code is always faster than ahead-of-time (AOT) compiled code. The adaptive approach shows promising results when executing short-running queries and provides at least the same performance as AOT-compiled code. Further, adaptive code compilation is a suitable technique to hide additional latencies introduced by PMem. For several queries, the runtime of the queries on PMem is the same as the runtime on DRAM when using adaptive query compilation.

The remainder of this work is structured as follows. First, the work that has been considered as a basis for this thesis is explained. Then the storage architecture and the query processing of the graph database Poseidon are shown. The next section examines the general design decisions of the query compiler. After that, the adaptive approach is presented and techniques are used. In the last part of this work, the shown techniques are evaluated in terms of their performance by using standard benchmarks.

2 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.

2.1 High-level code generation

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 [15]. This query 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 [17].

2.2 Low-level intermediate representations

Compiling high-level code introduces additional compilation time that reduces the resulting performance. The low-level IR-based approach avoids a high-level compiler and generates IR code instead. Neumann and Leis [13] 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, Kohn et al. [11] 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 [17]. The ideas from this work are the foundation of our approach to 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. Funke et al. [5] 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 [14]. Query compilation is also employed in existing commercial graph DBMS 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 [2].

2.3 Adaptive query compilation

Further developments for the compilation of queries consider the compilation times of LLVM. They are considered too high and therefore LLVM is replaced by self-developed frameworks. Funke and Teubner [4] developed a compiler with an alternative to the LLVM, which consists of Flounder IR and ReSQL, which compiles SQL to machine code with very low latency. Kersten et al. [10] developed a system that bypasses the slow startup time of a query interpreter and directly generates fast code for a query. Among other things, this makes it possible to work without an interpreter entirely. Nevertheless, the alternative to LLVM for compiling code with the lowest possible latency is to develop a custom IR that is adapted to the DBMS' requirements. Unfortunately, this approach increases the development effort of the database, since a complex compiler has to be developed in addition to the actual system. Furthermore, the presented system uses multiple execution modes to generate low-latency code for each case. Furthermore, the only fast alternative to JIT compilation with LLVM is the direct generation of assembly code with a framework such as ASMJit, which also increases the effort for the development and maintenance of

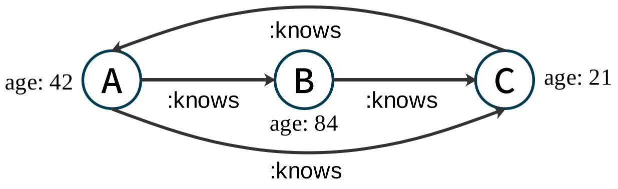


Fig. 1 Example property graph of three Person-nodes connected via relationships with the label knows

the DBMS. However, to our knowledge, there exists no graph DBMS that exploits adaptive query compilation to enhance query processing on PMem.

3 Poseidon graph database

Poseidon is a native graph database based on the property graph data model. An example of a property graph with labels is shown in Fig. 1. In the example, individual nodes are connected via relationships, representing a graph with persons knowing each other. The properties of the nodes are assigned directly. The formal definition of a property graph a given in the following definition.

Definition 1 A graph G consists of nodes N and directed relationships $R \in N \times N$, denoted by $G = (N, R)$. A node $n \in N$ is identified by a unique identifier $id : N \rightarrow ID$. From the set of labels L, a label is assigned to each node and relationship using the label function $l : (N \cup R) \rightarrow L$. Further, a property is a key-value pair $(k, v) \in P$. The properties P are $P = K \times D$, where K is the set of property names and D is the property values. Properties can be assigned using $p : (N \cup R) \rightarrow \mathcal{P}(P)$, using the powerset $\mathcal{P}$.

In Poseidon, 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, which are organized as linked lists of fixed-size arrays of object records, referred to as chunked vectors. The chunked vector is optimized for sequential access to fully utilize PMem. The corresponding data structure design for nodes and relationships is illustrated in Fig. 2. Further details of the data structure are given in [8]. At the same time, we also provide an implementation for persistent storage on disk as well as a non-persistent variant for direct storage on DRAM. The choice of storage medium must be specified accordingly before compilation. For storage on disk, we developed a buffered vector that stores the respective data from disk for processing in DRAM in the same format as a chunked vector for PMem. This allows the storage of transactional graph data on all possible storage media. Nevertheless, all data structures were optimized for the exploitation of PMem. However, these methods can also be applied to all other storage media and yield a similar result (Table 1).

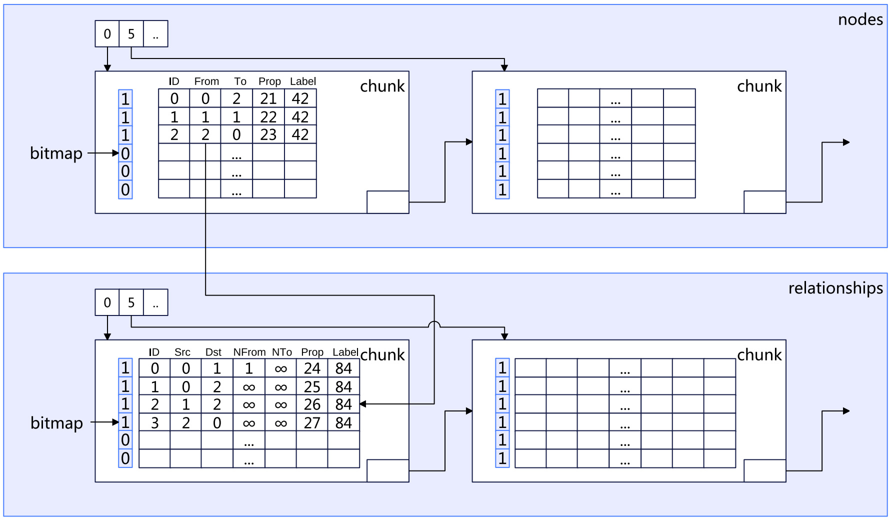


Fig. 2 Data structure for storage of graph data

In a chunked vector, each row in a chunk represents a slot for storing a node, relationship, or property 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. The internal representation of the example in Fig. 2 is shown in Sect. 3. Not existing links are represented with the maximum possible value of the field (∞). Further, transaction data stored in records are omitted. 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 chunked 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. The linkage between properties and nodes (and relationships) is achieved in the same way as demonstrated for the linkage of nodes and relationships.

3.1 Query processing

The query processing engine of Poseidon provides different ways to execute queries expressed in graph algebra. We refer to them as *execution modes*. As the storage layout of Poseidon aims to exploit PMem and the read access on PMem is slower than on DRAM, it requires hiding latencies by efficient cache utilization and

Table 1 Node and relationship table layout

NodeID	FromID	ToID	PropertyID	Label
0	0	2	21	42
1	1	1	22	42
2	2	0	23	42

RshipID	SrcNode	DstNode	NextFrom	NextTo	PropertyID	Label
0	0	1	1	∞	24	84
1	0	2	∞	∞	25	84
2	1	2	∞	∞	26	84
3	2	0	∞	∞	27	84

multi-threading. Additionally, we use the execution modes for further latency hiding. We now describe the query engine of Poseidon.

3.1.1 Graph algebra

Graph algebra is the starting point of the query engine of Poseidon. We adopted the graph algebra proposed by [7], which extends the relational algebra by two main graph operators. The main reason why this algebra was chosen in Poseidon to answer graph queries is that this algebra is based on relational algebra, which requires only an additional structure (graph relation). Furthermore, this algebra provides only two additional operators (GetNodes, Expand), which simplifies the implementation. The basic idea of the operators was adopted for Poseidon. Only the functionality and naming were changed to the be conform with internal structures. The two additional operators for processing graph relations are:

– GetNodes: scans the nodes of the graph.
– Expand: visits neighboring 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. For reasons of clean separation of the operators' functionality, we split the original Expand operator of [7] into two operators: ForeachRelationship and Expand. The ForeachRelationship operator traverses over all relationships of a node of the given direction (to, from). With the new Expand operator, we get the source or the destination of an edge. These two operators enable to traverse through a connected graph without the usage of join operators with the nodes and relationship table. In addition, we implement a simple query language based on this algebra. However, this query language is used for internal processing. Based on these operators other query languages like Cypher or GQL can be implemented. An example query using the described graph algebra language for the graph given in Fig. 1 is shown in the following. The shown indentation of the operators of the query is optional and serves only for the purpose of adequate visualization of the example.

```
Expand(OUT, εPersonε,
    ForeachRelationship(FROM, ε : knowsε,
        NodeScan(εPersonε)))
```

This query tries to find 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 individual operators in this language are formulated in the same order as they are processing, starting in reversed order in order to be compliant with the push-based query processing. For the execution of multiple sub-queries, which are merged with a join and a predicate, the queries can be defined in a similar way. For this, the respective sub-queries are passed in as input parameters of the join with an additional join predicate. The join algorithm can be selected directly as an operator. At the time of writing, Poseidon supports cross join, nested loop join, hash join, and left outer join. The following example combines the result of the previous query with all person nodes of the graph using a cross-product.

```
CrossJoin(
  Expand(OUT, εPersonε,
        ForeachRelationship(FROM, ε : knowsε,
            NodeScan(εPersonε))),
  NodeScan(εPersonε)
)
```

This set of operators is sufficient to answer queries on a graph, which can be written without effort and are similar to the well-known relational algebra.

3.1.2 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 a 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 [16]. The access paths of algebra queries are scans (NodeScan, RelationshipScan, IndexScan) and create (CreateNode, CreateRship) operations. Figure 3 shows the structure of push-based query processing. 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

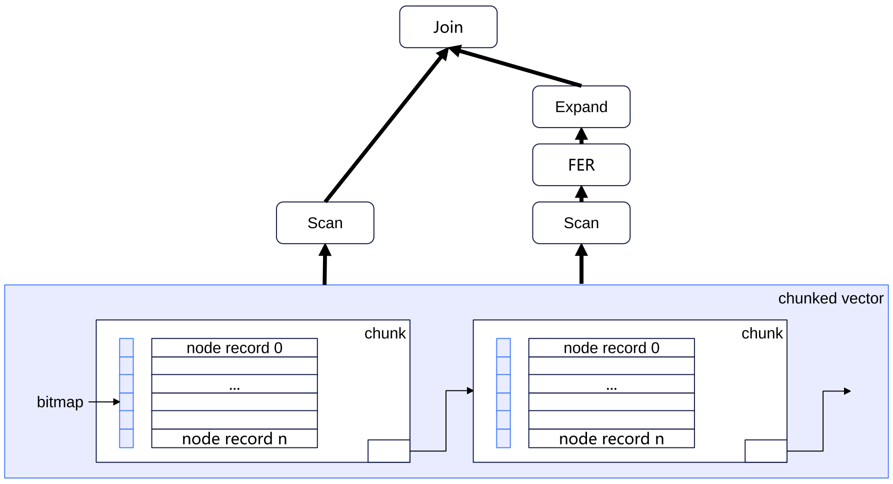


Fig. 3 Push-based query engine for the execution of graph algebra queries

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 that stores the tuples from the registers in (intermediate) storage. We have implemented all graph algebra operators in C++ and refer to these functions as our 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 the optimization flag for the compiler). This code builds the basic blocks for the interpretation execution mode.

3.1.3 Parallelism

In order to utilize modern multicore systems, the engine is able to process queries in parallel. 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 [12], whereby chunks (Morsels) are assigned to 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 task, 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 task. The last step of the query merges all results from the workers and returns them to the caller. The actual execution of the query can be done with different execution modes described in the following.

4 Adaptive query compilation

The database system described so far is able to process queries using AOT-compiled code defined in C++. In this section, we describe the design of the query compiler with different execution modes and the adaptive mode combining the execution modes to provide suitable performance for all workloads and to hide latencies introduced by PMem.

4.1 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 called. For this purpose, we use the *visitor pattern* [6]. For each operator, we provide at least one AOT-compiled function which executes the operator with given parameters. These AOT-compiled operator functions are can be linked together, forming a cascade of functions that executes the actual query. The linkage is done through the function arguments of the operator functions, i.e., the following operator is the given argument of the current operator functions. A downside of this approach is the additional effort in the implementation of each query operator. Further, each possible tuple element type must be handled explicitly in the code, which increases the development effort and maintenance of the database code. The resulting code is heavily template-based and requires runtime type information (RTTI).

4.2 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.

4.2.1 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 architecture-independent optimizations to improve the resulting code performance by executing optimization passes on the IR code. Optimization passes optimize IR code using well-known code compiler optimization techniques like dead code elimination, loop unrolling, and instruction combining. The resulting

[2] https://github.com/libfirm/libfirm.

[3] https://www.cs.tufts.edu/~nr/c--/.

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 require efficient solutions [1]. Lastly, LLVM provides a backend for several architectures like X86, ARM, or GPU. Using LLVM enables to provide the compiler for other architectures with low effort. 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 Kersten et al. [9] and Kohn et al. [11], and our experiments with different code generation and compiler setups.

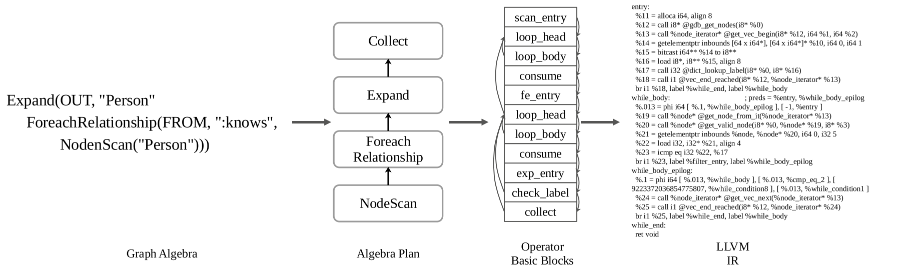
(**RQ1**) Processing of tuple results as long as possible in registers

(**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 Kersten et al. [9] shows that this approach requires fewer instructions, thereby enhancing performance with respect to query runtimes. (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 with (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.



```
entry:
  %11 = alloca i64, align 8
  %12 = call i8* @gdb_get_nodes(i8* %0)
  %13 = call %node_iterator* @get_vec_begin(i8* %12, i64 %1, i64 %2)
  %14 = getelementptr inbounds [64 x i64*], [64 x i64*]* %10, i64 0, i64 1
  %15 = bitcast i64** %14 to i8**
  %16 = load i8*, i8** %15, align 8
  %17 = call i32 @dict_lookup_label(i8* %0, i8* %16)
  %18 = call i1 @vec_end_reached(i8* %12, %node_iterator* %13)
  br i1 %18, label %while_end, label %while_body
while_body:                                   ; preds = %entry, %while_body_epilog
  %.013 = phi i64 [ %.1, %while_body_epilog ], [ -1, %entry ]
  %19 = call %node* @get_node_from_it(%node_iterator* %13)
  %20 = call %node* @get_valid_node(i8* %0, %node* %19, i8* %3)
  %21 = getelementptr inbounds %node, %node* %20, i64 0, i32 5
  %22 = load i32, i32* %21, align 4
  %23 = icmp eq i32 %22, %17
  br i1 %23, label %filter_entry, label %while_body_epilog
while_body_epilog:
  %.1 = phi i64 [ %.013, %while_body ], [ %.013, %cmp_eq_2 ], [
9223372036854775807, %while_condition8 ], [ %.013, %while_condition1 ]
  %24 = call %node_iterator* @get_vec_next(%node_iterator* %13)
  %25 = call i1 @vec_end_reached(i8* %12, %node_iterator* %24)
  br i1 %25, label %while_end, label %while_body
while_end:
  ret void
```

Fig. 4 Transformation steps from graph algebra to LLVM IR code

Therefore, there is not much room for further optimization steps. Hence, it can be implemented in C++ and called from the generated code. However, this requires the generated code to be fully compatible with the AOT-compiled query engine, as outlined in (RQ4) and is also beneficial when switching between execution modes. For the implementation of transactions, Poseidon uses an MVCC protocol. To hide the latencies as much as possible, this was implemented using a hybrid implementation. For example, changed nodes or relationships are managed in a dirty list, which is managed in DRAM memory to achieve the lowest possible latencies. Whenever a transaction is complete, the records of the dirty list are propagated to the appropriate persistent storage. The code is constant for most operations, which simplifies the implementation of the operators since AOT-compiled code can be used, which also reduces the compilation time. These AOT-compiled functions are called by function calls when they are needed by an operator.

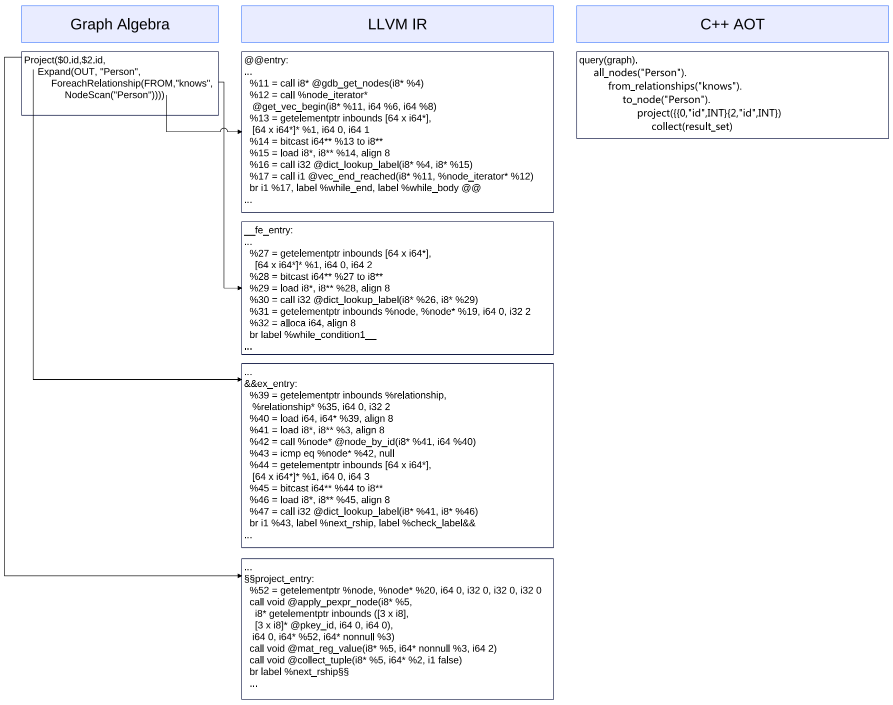
4.2.2 IR code generation

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. Figure 4 illustrates the whole transformation process from a given graph algebra query into LLVM IR. In the LLVM IR, each function comprises multiple basic blocks that contain the instructions to be executed. The last instruction of a basic block is a terminator that either branch 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 flows (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 the 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. 4. 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 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.

```
BasicBlock *PContext::while_loop(Function *parent,
                                 FunctionCallee get_begin,
                                 FunctionCallee get_next,
                                 FunctionCallee reached_end,
                                 Value *it_alloca, Value *cond_param,
                                 BasicBlock *nextBB,
                                 BodyFunction &loop_body,
                                 BasicBlock *loop_body_condition)
```

Listing 1 Code Abstractions for Loops in IR Code

Nevertheless, generating efficient IR code requires more effort than implementing the operators in a high-level language like C++. We implemented different abstractions to facilitate the implementation of the operators in the 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 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++ function where further IR code will be generated. Listing 1 shows the signature of a code abstraction to generate a head-driven loop in IR code. This function generates each of the basic branching statements of a loop as well as checks a given loop condition. The actual body, which will be executed only if the given condition matches, is passed as a C++ function. This allows the specification of a function in C++ which generates loop constructs in low-level IR code. Overall, in spite of these abstractions bringing the code style closer to high-level, there is still operator-dependent code that must



```
Project($0.id,$2.id,
  Expand(OUT, "Person",
    ForeachRelationship(FROM,"knows",
      NodeScan("Person"))))
```

```
@@entry:
...
  %11 = call i8* @gdb_get_nodes(i8* %4)
  %12 = call %node_iterator*
   @get_vec_begin(i8* %11, i64 %6, i64 %8)
  %13 = getelementptr inbounds [64 x i64*],
   [64 x i64*]* %1, i64 0, i64 1
  %14 = bitcast i64** %13 to i8**
  %15 = load i8*, i8** %14, align 8
  %16 = call i32 @dict_lookup_label(i8* %4, i8* %15)
  %17 = call i1 @vec_end_reached(i8* %11, %node_iterator* %12)
  br i1 %17, label %while_end, label %while_body @@
...
```

```
__fe_entry:
...
  %27 = getelementptr inbounds [64 x i64*],
    [64 x i64*]* %1, i64 0, i64 2
  %28 = bitcast i64** %27 to i8**
  %29 = load i8*, i8** %28, align 8
  %30 = call i32 @dict_lookup_label(i8* %26, i8* %29)
  %31 = getelementptr inbounds %node, %node* %19, i64 0, i32 2
  %32 = alloca i64, align 8
  br label %while_condition1__
...
```

```
...
&&ex_entry:
  %39 = getelementptr inbounds %relationship,
   %relationship* %35, i64 0, i32 2
  %40 = load i64, i64* %39, align 8
  %41 = load i8*, i8** %3, align 8
  %42 = call %node* @node_by_id(i8* %41, i64 %40)
  %43 = icmp eq %node* %42, null
  %44 = getelementptr inbounds [64 x i64*],
   [64 x i64*]* %1, i64 0, i64 3
  %45 = bitcast i64** %44 to i8**
  %46 = load i8*, i8** %45, align 8
  %47 = call i32 @dict_lookup_label(i8* %41, i8* %46)
  br i1 %43, label %next_rship, label %check_label&&
...
```

```
...
§§project_entry:
  %52 = getelementptr %node, %node* %20, i64 0, i32 0, i32 0, i32 0
  call void @apply_pexpr_node(i8* %5,
    i8* getelementptr inbounds ([3 x i8],
    [3 x i8]* @pkey_id, i64 0, i64 0),
   i64 0, i64* %52, i64* nonnull %3)
  call void @mat_reg_value(i8* %5, i64* nonnull %3, i64 2)
  call void @collect_tuple(i8* %5, i64* %2, i1 false)
  br label %next_rship§§
  ...
```

```
query(graph).
  all_nodes("Person").
    from_relationships("knows").
      to_node("Person").
        project({{0,"id",INT}{2,"id",INT})
          collect(result_set)
```

Fig. 5 LLVM IR code generation for graph algebra expressions

be handled directly in low-level IR code. Pointer arithmetic is another common construct that occurs in query code. To get the fields of the records like IDs, labels, or offsets, the address to the corresponding memory area must be obtained. In LLVM IR, this can be accomplished by using the *GetElementPointer* (GEP) instruction. Since this is an often-used instruction when processing tuples, we implemented further abstractions for this purpose to ease the development. For each occurring record field, we provide an abstraction that executes the GEP instruction to retrieve the pointer to the field value. The value of the field can be loaded into a register, by using a load instruction with the pointer as an argument. Figure 5 shows generated IR code generated from a graph algebra expression. To start each query, the corresponding storage is scanned, which can be on DRAM, disk, or PMem. The calls to the respective storage are performed by external function calls, similar to the calls for the management of transactions. This again leads to less effort for the generation of IR code. The nodes (or relationships) are loaded individually into registers. Subsequent operators, such as the ForeachRelationship operator, then access the respective register. Since the registers of a CPU can be accessed very fast, this improves the runtime of the query processing.

4.3 Code optimization

The main advantage of query compilers over query interpreters is the execution of efficiently generated code. For example, when executing code for a selection operator, an interpreter cannot recognize the connection between instructions. Thus, it can happen that some instructions are needlessly executed several times. The execution of unoptimized IR code leads to similar results as the execution of interpreted code. Therefore, optimization of IR code using optimization passes is needed. This will have a similar effect to compiling C++ code using the -O3 optimization level, rather than -O0. Generated IR code contains multiple branches, which may be unused because the condition is never fulfilled. By applying LLVM optimizations (passes), this code can be improved, the memory usage reduced, and in general, the execution of a query accelerated. Combining, deleting, or replacing instructions improves the code efficiency and the resulting query processing performance. We use the following cascade of optimization passes to improve the code efficiency regarding the query performance and size:

- *Promoting memory to register* Changes stack allocated memory for variables to memory registers which are faster to access because registers are near to the CPU.
- *Control flow simplification* Merges conditional branches when necessary.
- *Dead-code elimination* Deletes unused code which reduces the compilation time and size of the code.
- *Instruction combining* Combines instructions that are similar or share the same conditions.
- *Dead-store elimination* Removes unused store operations for unused memory.

After executing these passes, the resulting query code contains only the instructions needed to process the query.

4.4 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 persistently. It can be stored on disk or other persistent storage like 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 will be stored in the persistent map with the

Table 2 Compilation times for a graph query finding paths with different lengths (hops)

No. of hops	Compilation time (ms)
1	112
2	153
3	183
4	208

already obtained key. Only the generated machine code is cached persistently, allowing it to retrieve and call it with different parameters, i.e., different labels or properties for each operator. Compiled code is stored directly in memory in its binary form with all additional references (symbols, external functions, data). The overall workflow using cached code is the same as with code compilation, except that the code is available immediately. It can be executed directly by the functions provided by the LLVM framework.

4.5 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.

Table 2 shows the overhead of compiling graph queries with different lengths, which correspond to the number of hops through the graph. As the query length increases, the effort required to compile the query also increases. During this time, without further adaption, the processing would have to wait for the compilation to be completed, although the results could already be available or the query could have been processed already. 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.

4.6 Parallelism

As our adaptive query engine makes use of parallelism, we first describe how it is implemented. To switch between different modes, we make use of parallel execution of chunks, which are previously assigned to tasks and made available to workers for pooling in a task pool. The granularity of the switch is a task, which means that the tasks (or chunks) are finished in the mode in which they were started. This has the advantage that no further information has to be exchanged between two executions. Figure 6 shows the extension of the push-based and Morsel-driven query engine by the compiled mode. 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.

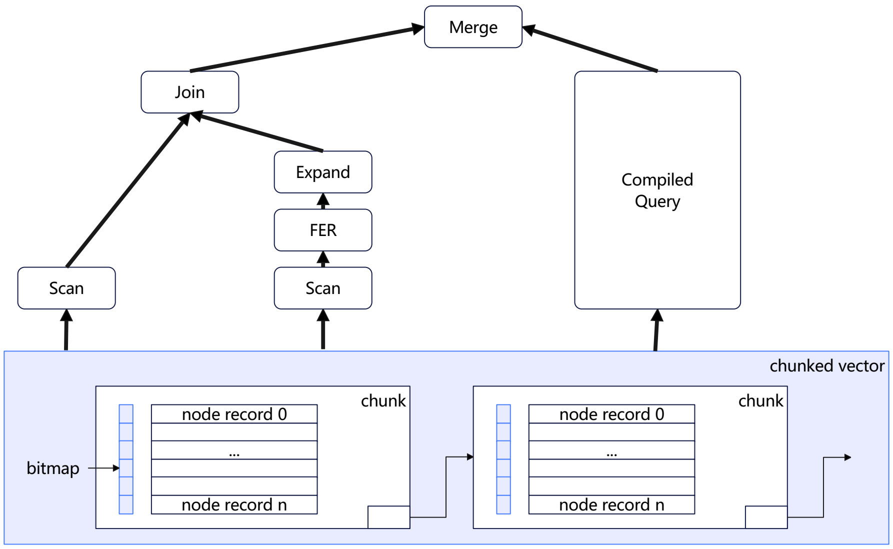


Fig. 6 Adaptive query compilation

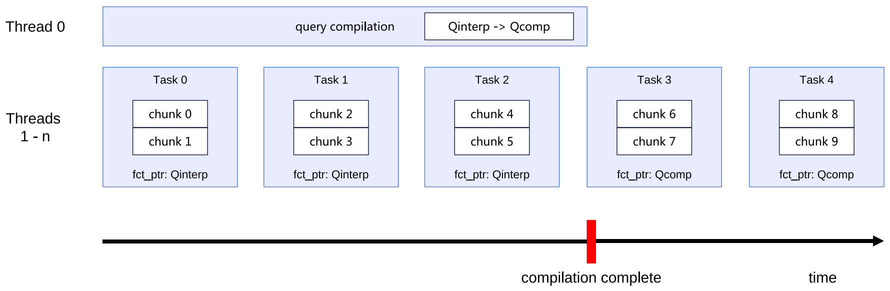


Fig. 7 Adaptive compilation flow

4.7 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 [11], 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. This approach enables hiding latencies of the underlying storage technology which improves the overall performance of the query processing. Furthermore, we make use of morsel-driven parallelism to switch the mode at query processing.

A complete overview of the adaptive query compilation flow is shown in Fig. 7. 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

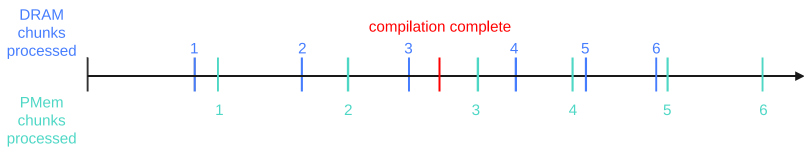


Fig. 8 Comparison of adaptive query compilation on different storage types on a timeline

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 joins as an 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.

This method is our basic idea to hide the latencies of the underlying storage medium, such as PMem, to get an approximation of DRAM runtime. If we compare the execution of a query using this method on DRAM as well as on PMem, for example, it is noticeable that the number of chunks executed on the different modes differ. Schematic comparison between the execution of adaptive query compilation on different storage types is shown in Fig. 8. It shows that when executing adaptive query compilation on a storage type with higher latencies like disk or PMem, the number of chunks executed with the optimized and compiled code is higher, than when executing the same query on DRAM. This is because processing the individual chunks requires more processing time due to the additional latencies of storage. Therefore, the processing stays at an early stage of the processing pipeline and there are more queries available to process using the faster compilation mode. With this, the execution time for processing the individual chunks is saved, which hides the latencies of Disk or PMem storages.

4.8 Code caching latency

Caching code in persistent memory is beneficial for later query execution, as no recompilation is necessary. However, retrieving code from PMem or SSD also introduces an overhead. The overhead is on the one hand the additional access latency of the storage 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 storage latencies effectively.

5 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. We compare the execution time of all queries on DRAM with the execution using PMem in order to show the benefit of the adaptive approach in hiding the additional access latency.

5.1 Environment and 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 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 12.0.1 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, the Interactive Update queries, that update node, and relationship objects, and the subset of the Business Intelligence (BI) queries for complex queries to show the full potential of our query compiler. The subset of BI queries consists of queries 1–13, for which the query compiler can generate executable code at the time of the writing of this work. However, these queries are sufficient to demonstrate the performance of the techniques shown in this paper, since they have a higher complexity than the short-read due to multiple sub-pipelines, joins, and aggregations. We ran the queries on the SNB data at scale factor 10 for the short-reads and interactive updates 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. The number of total chunks for nodes is 1602, for relationships 121,219, for the node properties 3762, and for relationship properties 2936. Each chunk contains up to 80 entries. The total size of the graph is 13.5 GB.

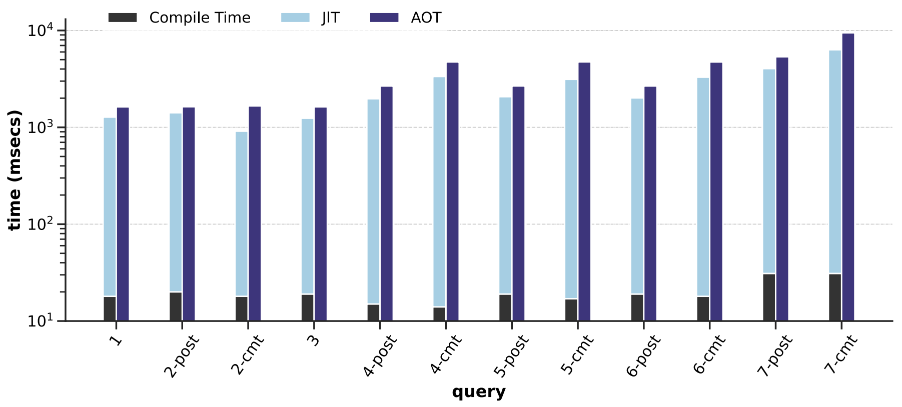


Fig. 9 SNB short read queries on PMem (single-threaded)

5.2 Benchmark results

In the following benchmarks the execution of the mentioned LDBC queries is based on the SNB dataset. We focus on single-threaded, multi-threaded, and index-based execution. The single-threaded execution is intended to show the quality of the generated code, where no adaptive switching is used. The multi-threaded execution will demonstrate adaptive switching and compare it to AOT-compiled code. The index-based execution will be executed on the SNB short reads and will compare the quality of the generated code with optimal execution times. Furthermore, one execution each on DRAM and PMem will be shown to illustrate the behavior for hiding PMem latencies.

5.2.1 Single-threaded

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.

Figures 9 and 10 show 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. Figures 11 and 12 show the results of the approach with more complex BI queries. The BI queries contain multiple sub-pipeline with joins and aggregations. The compilation time of these queries ranges between 150 and 500 ms. The runtimes of both executions are similar. However, the increase in compile time reduces the overall performance of the JIT execution, so adaptive execution is necessary for complex queries.

5.2.2 Multi-threaded

Next, we evaluate the adaptive and (non-adaptive) JIT compilation while utilizing multi-threading. Figures 13 and 14 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.

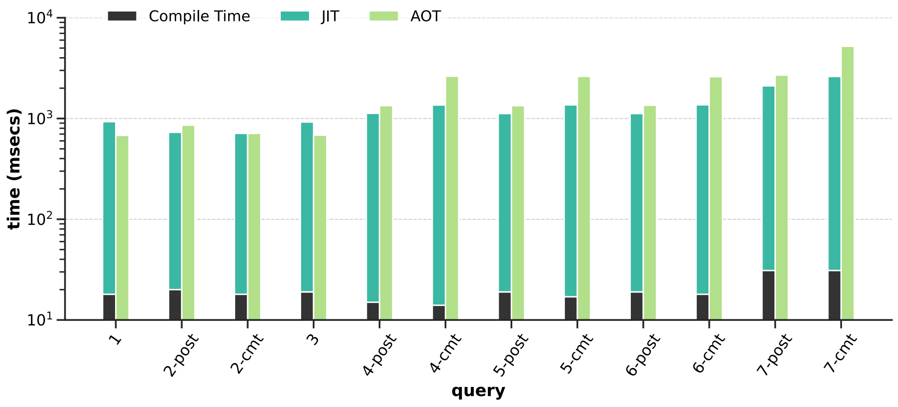


Fig. 10 SNB short read queries on DRAM (single-threaded)

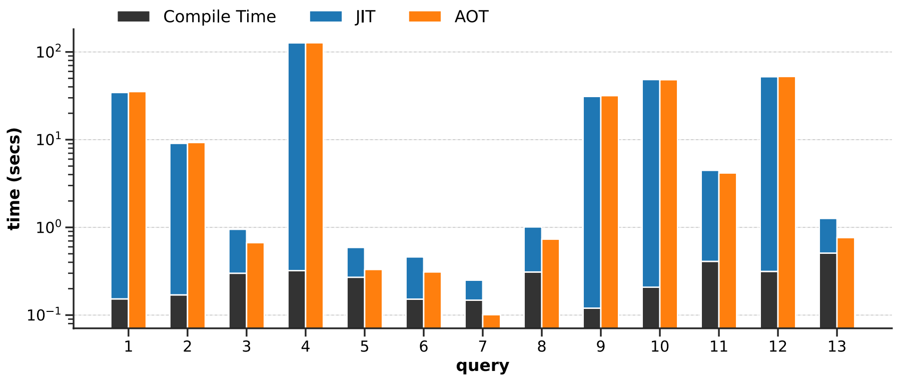


Fig. 11 SNB BI queries on PMem (single-threaded)

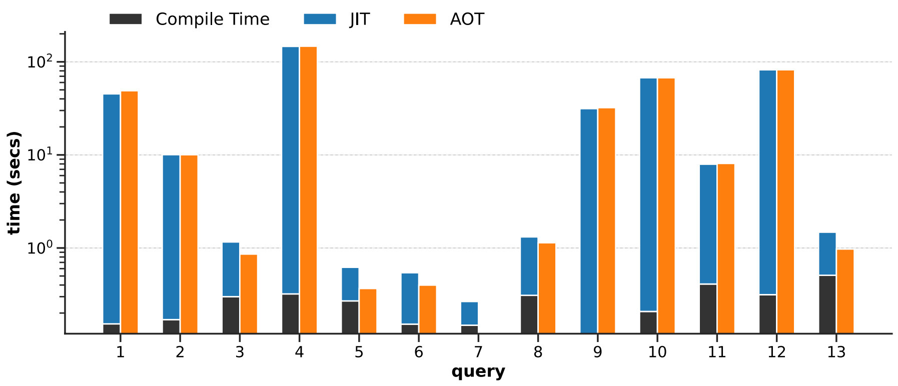


Fig. 11 SNB BI queries on DRAM (single-threaded)

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 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. Figures 15 and 16 show the multi-threaded execution of the adaptive and AOT-compiled with complex BI queries on DRAM and PMem respectively. Multi-threaded execution of these queries on both memory types solves the problems of single-threaded execution. Furthermore, for some queries on DRAM and PMem, such as queries 1, 2, 3, and 5, the runtimes are similar, which in turn can be attributed to the hiding of the PMem latencies.

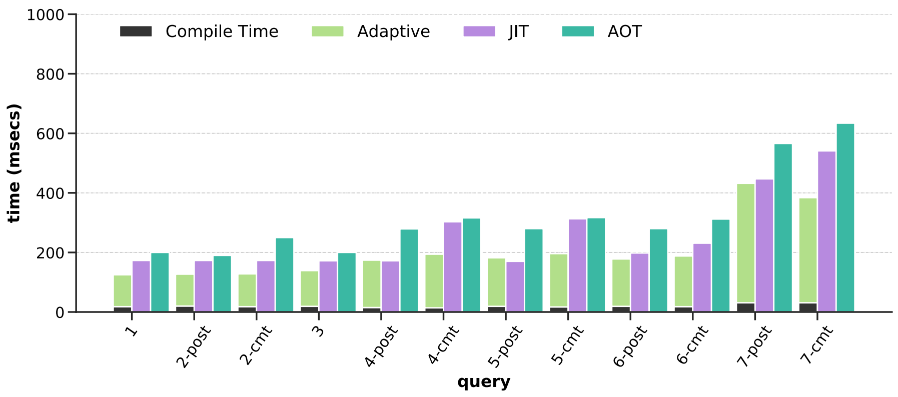


Fig. 13 SNB interactive short read queries on DRAM (multi-threaded)

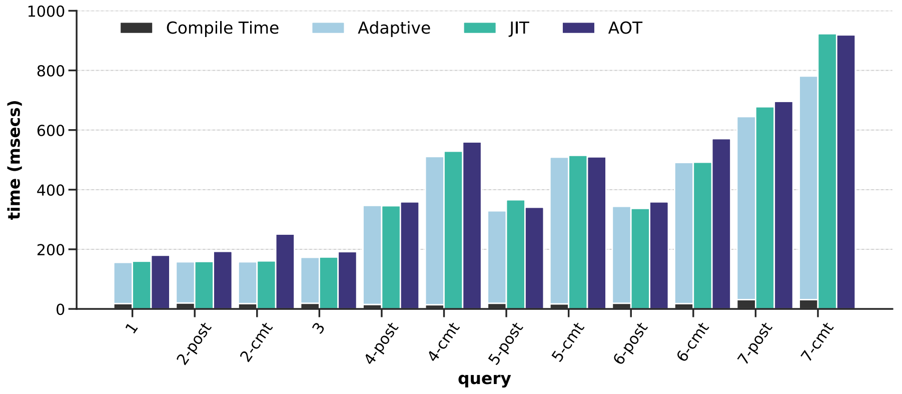


Fig. 14 SNB interactive short read queries on PMem (multi-threaded)

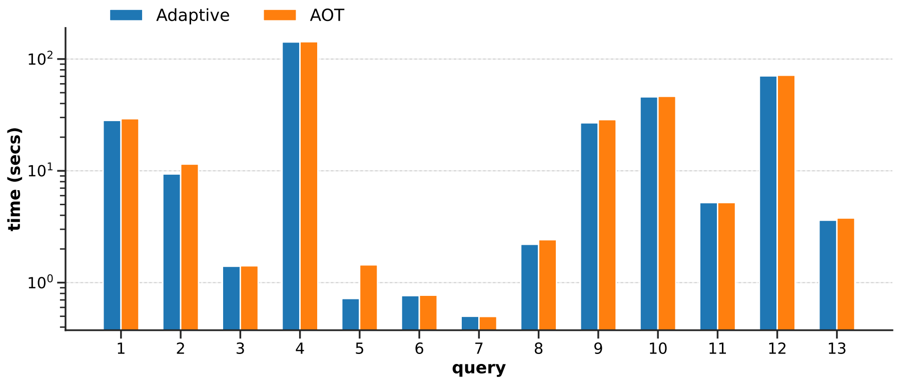


Fig. 15 SNB BI queries on DRAM (multi-threaded)

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. Figures 17 and 18 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 ms. However, as we execute the queries with indexes to retrieve the nodes with the appropriate properties, the processing time is below 1 ms. 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 ms and between 10 and 15 ms on PMem. Consequently, the time of compilation and retrieving the short queries are similar and also result in a similar performance. Though, code caching is more beneficial for queries with longer compilation times.

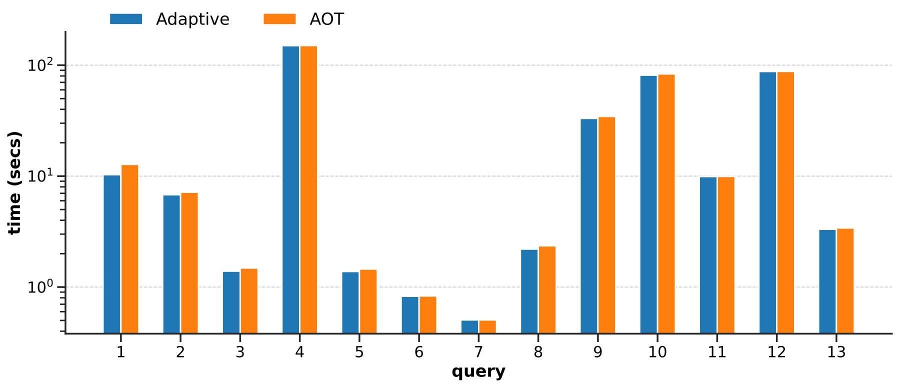


Fig. 16 SNB BI queries on PMem (multi-threaded)

5.3 Indexed execution

Figures 19 and 20 show the same queries but with the usage of an index to look up the appropriate nodes on PMem and DRAM respectively. If the compilation time is longer than the actual execution time of the query, it will be executed completely with AOT-compiled code. This results in slower execution time, as the adaptive execution would be the same as execution with AOT-compiled code. Only the processing of the limit operator is faster in compilation mode than in the execution of AOT-compiled code. The JIT-compiled code leaves the processing immediately after the limit is reached, unlike the AOT-compiled code. For some queries (4, 5, 6), the execution times of the adaptive approach are slightly higher than the execution times of the AOT code. Among other things, this is due to the additional effort required to start the adaptive compiler, such as starting multiple threads (compile thread, query thread, etc.) However, this overhead is a few nanoseconds and is negligible due to the performance gain in the other cases. The adaptive approach can provide reliable performance for most of the queries since it provides at least the same runtime as

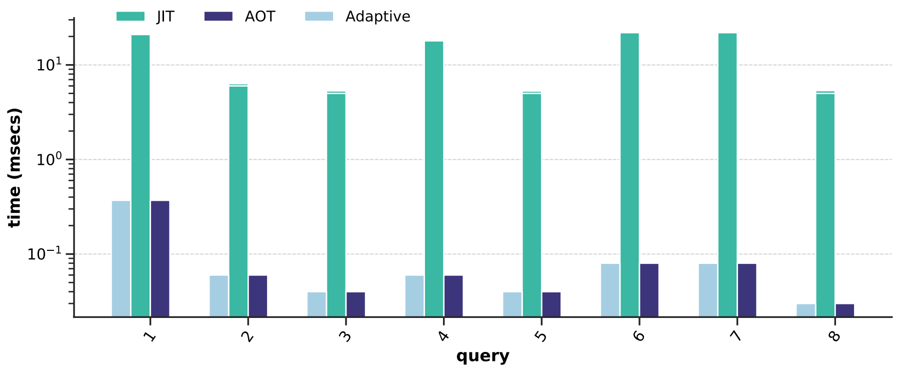


Fig. 17 SNB interactive update queries on DRAM (indexed)

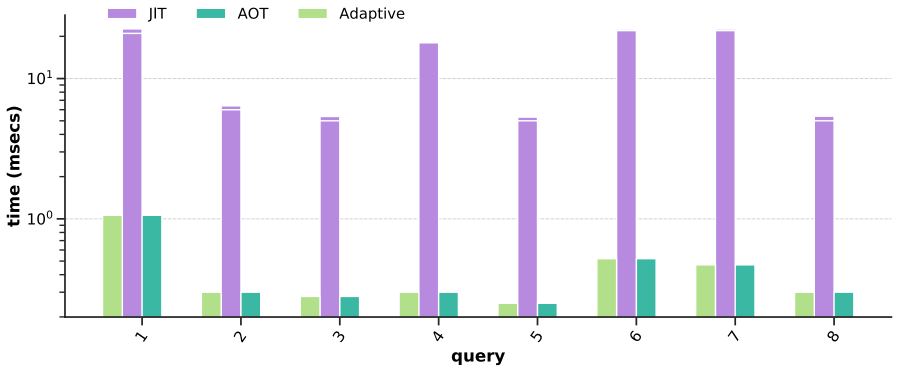


Fig. 18 SNB interactive update queries on PMem (indexed)

the AOT-compiled code for all cases. Therefore, this approach is suitable for hiding compilation time, particularly for short-running queries, in addition to hiding PMem latency.

5.4 Latency analysis

The following benchmark is intended to examine in more detail the point at which to switch to optimized and compiled code. To show this, we analyze the LDBC queries 1–4 using the SNB dataset with scale factor 10 since these benefit particularly from adaptive switching. The number of total chunks for the nodes is 1602. We compare the execution of these queries on DRAM and PMem respectively. For this, we measure only the number of executed chunks in the appropriate execution mode (interpretation and compilation mode). Table 3 shows the process of executing the chunks in interpretation mode and the time at which the optimized and JIT-compiled code is switched to. When comparing the two executions on DRAM and PMem, the different number of executed chunks in the respective modes is noticeable. Most of the chunks are executed with the compiled code. Comparing the execution with the interpretation mode on DRAM and PMem shows that on PMem fewer chunks are executed in this mode, for all of the considered queries.

6 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,

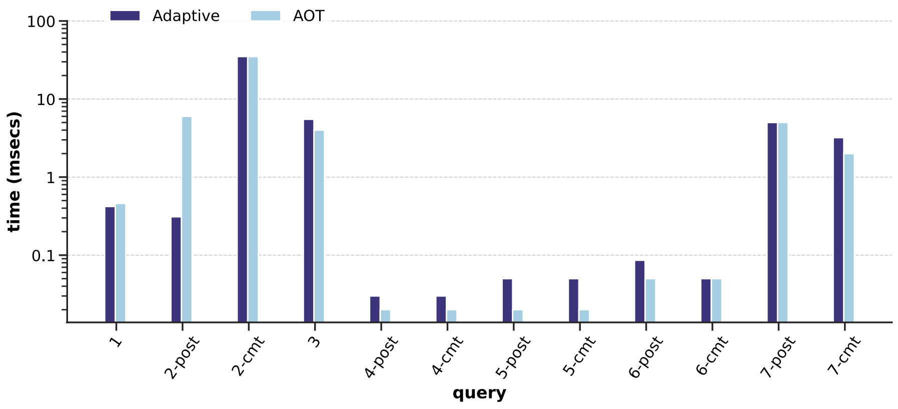


Fig. 19 SNB interactive short read queries on PMem (indexed)

it autonomously switches the execution to the compiled code. This approach inherently provides fast runtimes through efficient machine code. Moreover, this technique allows hiding access latencies of memories, such as PMem, thereby enabling similar performance to DRAM for query execution 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. The presented approach is also a tradeoff to keep the development and maintainability of the DBMS as simple as possible, but at the same time to provide the best possible performance of a query compiler. While other systems require the development of multiple query engines, such as a query interpreter and different compilers for the additional execution modes, the system presented here uses only two execution modes that serve the same purpose. This work has shown adaptive query compilation for the graph database Poseidon. Nonetheless, this technique is also suitable to hide the compilation time on other DBMSs.

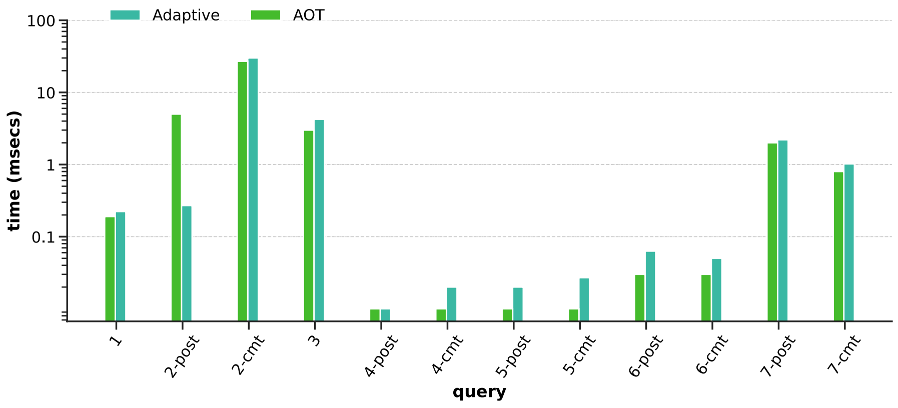


Fig. 20 SNB interactive short read queries on DRAM (indexed)

Table 3 Comparison of executed chunks with different execution modes on DRAM and PMem

Query	No. of interpreted	No. of compiled
2-post DRAM	93	1509
2-post PMem	64	1538
2-cmt DRAM	167	1435
2-cmt PMem	127	1475
3 DRAM	130	1472
3 PMem	73	1529
4-post DRAM	134	1468
4-post PMem	56	1546
4-cmt DRAM	160	1442
4-cmt PMem	52	1550

Acknowledgements 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)".

Author contributions AB and MAJ wrote the main manuscript. AB prepared all figures and the evaluation. AB wrote the extended and revised version of the main manuscript. All authors reviewed the manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Competing interests The authors declare no competing interests.

References

1. Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., Markstein, P.W.: Register allocation via coloring. Comput. Lang. **6**(1), 47–57 (1981)
2. Deutsch, A., Xu, Y., Wu, M., Lee, V.E.: Tigergraph: a native MPP graph database. CoRR. (2019). arXiv:1901.08248
3. Freedman, C., Ismert, E., Larson, P.: Compilation in the microsoft SQL server hekaton engine. IEEE Data Eng. Bull. **37**(1), 22–30 (2014)
4. Funke, H., Teubner, J.: Low-latency compilation of SQL queries to machine code. Proc. VLDB Endow. **14**(12):2691–2694 (2021). https://doi.org/10.14778/3476311.3476321

5. Funke, H., Mühlig, J., Teubner, J.: Efficient generation of machine code for query compilers. In: 16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, OR, USA, 15 June 2020. ACM, New York, pp 6:1–6:7 (2020). https://doi.org/10.1145/3399666.3399925
6. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design patterns: abstraction and reuse of object-oriented design. In: Broy, M., Denert, E. (eds.) Software Pioneers, pp. 701–717. Springer, Berlin (2002). https://doi.org/10.1007/978-3-642-59412-0_40
7. Hölsch, J., Grossniklaus, M.: 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, 15 March 2016
8. Jibril, M.A., Baumstark, A., Götze, P., Sattler, K.U.: Jit happens: transactional graph processing in persistent memory meets just-in-time compilation. In: 24th International Conference on Extending Database Technology (EDBT) 2021, Nicosia, Cyprus (2021)
9. Kersten, T., Leis, V., et al.: Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. Proc. VLDB Endow. **11**(13), 2209–2222 (2018)
10. Kersten, T., Leis, V., Neumann, T.: Tidy tuples and flying start: fast compilation and fast execution of relational queries in umbra. VLDB J. **30**(5), 883–905 (2021). https://doi.org/10.1007/s00778-020-00643-4
11. Kohn, A., Leis, V., Neumann, T.: Adaptive execution of compiled queries. In: 34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, 16–19 April 2018. IEEE Computer Society, pp 197–208 (2018)
12. Leis, V., Boncz, P.A., Kemper, A., Neumann, T.: 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, 22–27 June 2014, pp 743–754 (2014)
13. Neumann, T., Leis, V.: Compiling database queries into machine code. IEEE Data Eng. Bull. **37**(1), 3–11 (2014)
14. Pirk, H., Moll, O.R., Zaharia, M., Madden, S.: Voodoo - A vector algebra for portable database performance on modern hardware. Proc. VLDB Endow. **9**(14), 1707–1718 (2016)
15. Shaikhha, A., Klonatos, Y., et al.: How to architect a query compiler. In: Özcan, F., Koutrika, G., Madden, S. (eds.) Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, June 26–July 01, 2016. ACM, New York, pp 1907–1922 (2016). https://doi.org/10.1145/2882903.2915244
16. Shaikhha, A., Dashti, M., Koch, C.: Push versus pull-based loop fusion in query engines. J. Funct. Program. **28**, e10 (2018)
17. Tahboub, R.Y., Essertel, G.M., Rompf, T.: How to architect a query compiler, revisited. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, 10–15 June 2018. ACM, New York, pp 307–322 (2018). https://doi.org/10.1145/3183713.3196893