A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries

Immanuel Haffner
Saarland Informatics Campus
immanuel.haffner@bigdata.uni-saarland.de

Jens Dittrich
Saarland Informatics Campus
jens.dittrich@bigdata.uni-saarland.de

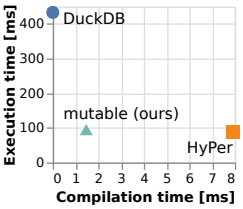


Figure 1: Design space of query execution engines, based on TPC-H Q1 benchmark results. The compilation time is the time to translate a QEP to machine code. The execution time is the time to execute the machine code and does not include the compilation time.

ABSTRACT

Query compilation is crucial to efficiently execute query plans. In the past decade, we have witnessed considerable progress in this field, including compilation with LLVM, adaptively switching from interpretation to compiled code, as well as adaptively switching from non-optimized to optimized code. All of these ideas aim to reduce latency and/or increase throughput. However, these approaches require immense engineering effort, a considerable part of which includes reengineering very fundamental techniques from the compiler construction community, like register allocation or machine code generation – techniques studied in this field for decades.

In this paper, we argue that we should design compiling query engines conceptually very differently: rather than racing against the compiler construction community – a race we cannot win in the long run – we argue that code compilation and execution techniques should be fully delegated to an existing engine rather than being reinvented by database architects. By carefully choosing a suitable code compilation and execution engine we are able to get just-in-time code compilation (including the full range from non-optimized to fully optimized code) as well as adaptive execution in the sense of dynamically replacing code at runtime – for free! Moreover, as we rely on the vibrant compiler construction community, it is foreseeable that we will easily benefit from future improvements without any additional engineering effort. We propose this conceptual architecture using WebAssembly and V8 as an example. In addition, we implement this architecture as part of a real database system: mu*t*able. We provide an extensive experimental study using TPC-H data and queries. Our results show that we are able to match or even outperform state-of-the-art systems like HyPer.

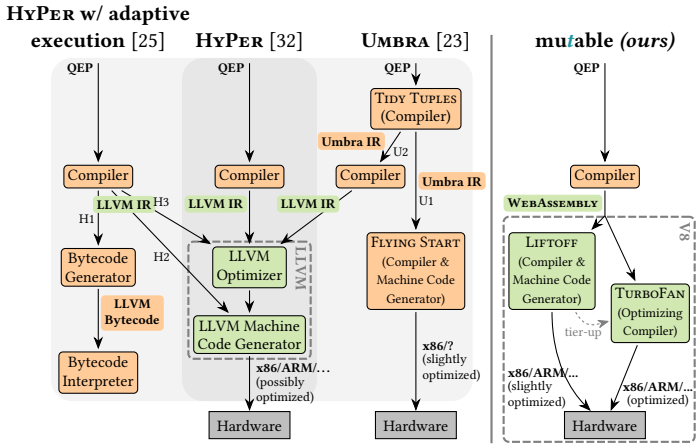
1 INTRODUCTION

To execute SQL queries, database systems must determine for each query a *query execution plan* (QEP) that defines how to execute the query. The QEP is then executed by either interpretation or compilation. Many early database systems used an interpreter for query execution, as it is easy to maintain and portable [24]. The Volcano model presented a generic and extensible design, adopted by many database systems that followed [15]. The induced overhead of interpretation was dwarfed by the high costs for data accesses in disk-based systems [8, 22, 32]. However, in modern main memory systems data accesses are significantly faster and the interpretation overhead suddenly takes a large share in query execution costs [3, 32]. Therefore, main memory systems must keep any overheads during query execution at a minimum to achieve peak performance. This development was the reason for an extensive body of work on query interpretation and compilation techniques and sparked a seemingly endless debate which of the two approaches to prefer [22–24, 27, 32, 38, 42]. Recently, Kohn et al. proposed an adaptive approach to query execution, where the database system can seamlessly transition from interpreted to compiled query execution [25]. This approach requires *both* a query interpreter *and* a query compiler that must be interoperable, which is achieved by a particular execution mode named *morsel-wise execution* [28]. Kersten et al. followed up on this work and present adaptive execution by switching from non-optimized to optimized code during query processing [23]. Despite the promising results of both works, we believe that implementing either approach requires expertise in interpreter and compiler design and poses an immense development effort, ultimately preventing wide-spread adoption.

In this work, we propose a new architecture for query execution engines of database systems. Rather than reengineering compiler technology, we suggest to employ a suitable and – most importantly – existing execution engine that takes care of *just-in-time* (JIT) compilation and adaptive execution. We dramatically reduce the complexity of the system by relying on existing infrastructure. By translating the QEP to an interchange format and delegating execution to an underlying engine, we are able to drastically reduce compilation times while maintaining competitive execution performance, as exemplified in Figure 1.

Contributions

(1) We present a new, simplified, conceptual architecture of a query execution engine that allows us to delegate JIT compilation, optimization, and adaptive execution to an underlying engine. Like that, we avoid reengineering techniques researched and developed by the compiler community for decades.
(2) We demonstrate how to implement this architecture in a real database system: mu*t*able. We use WebAssembly as intermediate representation and Google's V8 as backend. However, any other backend with similar properties as V8 conceptually works as well. mu*t*able supports the full pipeline of compiling SQL queries to executable code.
(3) We discuss in detail the pros and cons over compiling with LLVM, adaptive compilation, and vectorized execution.
(4) We discuss current limitations of our approach and how they will get resolved in the (near) future.
(5) We provide an extensive experimental study, showcasing that even though we use an architecturally much simpler



(a) Architectural overview of compiling query engines.

	HyPer w/ adaptive execution [25]	HyPer [32]	Umbra [23]	mutable *(ours)*
Interpretation	✓	✗	✗	✓
Fast JIT Compilation	✗	✗	✓	✓
Optimizing Compilation	✓	✓	✓	✓
Adaptive Execution	✓	✗	✓	✓
Different HW (x86, ARM, etc.)	✓	✓	✓(?)	✓

(b) Feature matrix for the architectures in Figure 2a.

Figure 2: Orange ○ means (potentially re-) implemented by the system itself, green ○ means used off the shelf, and red ○ means desirable but lacking.

approach than state-of-the-art, we are able to match or even outperform state-of-the-art query compilers like HyPer.

Outline — In Section 2 we present our architecture and compare to other compiling query engines. We further investigate the strengths and weaknesses of our architecture. In Section 3 we introduce WebAssembly. In Section 4 we elaborate how we compile QEPs to WebAssembly. In Section 5, we present our ad-hoc generation of specialized library code. We explain how we execute a compiled query within a WebAssembly engine in Section 6. We conduct a comparison to related work in Section 7 and present our experimental evaluation in Section 8. Section 9 concludes our work.

2 A NEW ARCHITECTURE FOR COMPILING QUERY ENGINES

We begin by motivating the need for an architectural simplification of query engines. We then propose our architecture and discuss pros and cons. In Figure 2a we present an overview of the architectures of prominent compiling query engines.

2.1 Other Architectures

HyPer — Although the very first relational database system, System R, already compiled queries to machine code [10], compiling queries only really became maintainable with the use of a compilation framework, such as LLVM used in HyPer [32]. The original architecture of HyPer is shown in the second column of Figure 2a. HyPer translates the QEP with its own compiler to LLVM IR, the *intermediate representation* (IR) of LLVM. LLVM provides a large set of optimization passes that can be applied to the IR, potentially transforming the IR and increasing program efficiency. HyPer applies a fixed, handpicked subset of LLVM's optimization passes [32]. This subset was chosen such that optimization time is balanced with optimization gain. After applying the optimization passes to the IR, HyPer runs LLVM's machine code generation to obtain executable code. HyPer then runs this code on the hardware.

HyPer w/ adaptive execution — Potentially long-running compilation with LLVM delays query execution. Therefore, Kohn et al. propose in a follow-up work to extend the compilation pipeline by interpretation and adaptively switching from interpreted to compiled execution as soon as compilation completes [25]. This architecture is shown in the first column of Figure 2a and is an extension of the original architecture of HyPer. It uses the same compiler to translate QEPs to LLVM IR. At this point, there are three paths to proceed with. The first path H1 translates the LLVM IR with their bytecode generator to LLVM bytecode, an IR developed by the authors that is similar to LLVM IR yet optimized for interpretation. This LLVM bytecode is then interpreted by their bytecode interpreter. The second and third path, H2 and H3, both rely on the original architecture of HyPer: H2 directly translates the LLVM IR to machine code, producing an *"O0"* executable. H3 incorporates LLVM optimizations, eventually producing an *"O2"* executable. While the query is being executed by interpretation of the LLVM bytecode, the LLVM IR is optimized and compiled to machine code in the background. Once this process completes (and the query has not yet terminated), the system switches from interpreted to compiled execution. Switching is enabled by morsel-wise execution [28].

Umbra — Although the architecture of HyPer with adaptive execution reduces query latency without sacrificing performance for long-running queries, initial interpretation is still slow and compilation with LLVM takes relatively long. This observation lead to another follow-up work by Kersten et al., in which the authors drop interpretation entirely in favor of fast JIT compilation [23]. This architecture, shown in the third column of Figure 2a, is implemented in Umbra. In this architecture, the QEP is first translated by Tidy Tuples into their own Umbra IR. Umbra provides two compilation paths, U1 and U2, for this IR. U1 translates the IR directly to machine code with their JIT compiler Flying Start. This compiler performs only a fixed amount of passes over the IR, employs only a few fast optimizations, and generates slightly optimized *"O1"* machine code. Its purpose is to produce machine code fast while exploiting some potential for optimization. U2 translates the Umbra IR further to LLVM IR and follows the LLVM compilation pipeline as in HyPer, eventually producing a fully optimized *"O2"* executable. Similar to HyPer with adaptive execution, Umbra uses morsel-wise execution to switch from the code produced by Flying Start to the fully optimized code produced by LLVM.

Criticism. The original design of HyPer is clean and simple: use an existing compilation framework to compile QEPs to efficient machine code. However, the choice for LLVM introduces the deficiency of long compilation times delaying execution. Both HyPer with adaptive execution as well as Umbra work around this deficiency by introducing an alternative, much faster path to begin query execution and combine this with adaptively switching to optimized code when available. By inspecting our overview in Figure 2a, we can observe the sheer engineering effort that both systems undertake to enable this alternative path. We argue that neither approach will find wide-spread adoption as both require expert knowledge in interpreter and compiler design as

well as long development times that prevent wide-spread adoption. We therefore present a new architecture, that is as clean and simple as the original architecture of HyPer, yet brings the same benefits as Umbra.

2.2 Our Architecture

Requirements — To make justified decisions for our architecture, we first establish a common notion of our requirements: (1) We want to minimize the latency of query execution. (2) At the same time, we want to maximize the throughput of long-running queries. (3) Any kind of optimization should not add to the latency, meaning that optimization must be interweaved with execution. (4) Rather than solving (1)-(3) ourselves, we want to build on existing infrastructure.

Towards a solution — (1) To minimize latency of query execution, we can use interpretation or fast compilation of QEPs. (2) To increase throughput, we can apply crucial optimizations when compiling, e.g. register allocation. (3) To avoid optimization delaying query execution, optimization and query execution can happen in parallel. Query execution should switch to execution of the optimized code as soon as it becomes available. To increase adaptivity, optimizations should be applied on a fine granule: rather than waiting for the entire QEP to be optimized, we can compile and optimize individual pipelines and immediately make use of the optimized code. (4) Existing infrastructure providing the desired traits comes in the shape of JIT compilation frameworks or entire engines, controlling compilation, execution, and re-optimization.

Implementation – With a suitable JIT infrastructure at hand, the architecture of the query engine becomes surprisingly simple: translate the QEP to the interchange format and submit it to the infrastructure implementing (1)-(3) for execution. To our satisfaction, there is a plethora of projects implementing requirements (1)-(3) in an off-the-shelf engine. We give an overview of available projects and potential interchange formats in Section 7. Our choice for implementing this architecture is as follows: We translate QEPs to WebAssembly and delegate execution to the V8 engine. V8 is Google's JavaScript and WebAssembly engine and it fulfills all our aforementioned requirements. The fourth column of Figure 2a shows how V8 embeds into our proposed architecture. V8 provides two compilation tiers: fast compilation with Liftoff [19] and optimizing compilation with TurboFan [6]. Although initially WebAssembly is compiled with Liftoff to quickly start execution, V8 gradually replaces code during execution by optimized code produced by TurboFan as soon as it becomes available [19, 34]. V8 hence not only compiles WebAssembly but also takes care of adaptive execution. V8's Liftoff fulfills the same purpose as Umbra's Flying Start while V8's TurboFan can be seen as an optimizing compiler like LLVM with optimization passes, yet it is designed for a JIT environment and hence much faster. While Umbra has to implement and steer switching from non-optimized to optimized code, we can rely on V8 gradually optimizing the code during execution. Further, V8 provides fine-granular control over which optimizations to perform, whether to optimize adaptively during execution, and whether to enable the Liftoff compiler. One more benefit particular to V8 is that it compiles WebAssembly, which is an excellent interchange format between QEP and V8 as we elaborate in Section 3. We provide a summarized feature comparison in Figure 2b.

3 WEBASSEMBLY

Having a fast JIT compiler is inevitable to reducing latency in a compiling query engine, but it is certainly not enough. A (JIT) compiler takes as input the program, encoded in text or some kind of bytecode. We hence must translate the QEP to a suitable format accepted by the compiler. This step adds to the overall compilation time. It is therefore necessary to choose a fitting interchange format to enable fast translation of QEPs.

WebAssembly, or short *Wasm*, is "a low-level assembly-like language with a compact binary format that runs with near-native performance" [13]. Among the many high-level goals of WebAssembly, we see three key features that make it the instrument of choice for JIT compiling QEPs. The first key feature is that WebAssembly is size and load-time efficient, allowing for fast code generation, fast JIT compilation to machine code, and resource-friendly caching of already compiled code [17, 21]. Second, WebAssembly is a *virtual instruction set architecture* (ISA) and therefore hardware independent and embeddable in many environments. Third, WebAssembly can be compiled to execute at near native speed [21] and make use of modern hardware capabilities, e.g. SIMD [6]. Many WebAssembly engines offer debugging interfaces. The V8 engine provides an interface using the Chrome DevTools protocol over web socket. A developer can launch Google Chrome and connect to the V8 instance. The developer then has access to a wide range of debugging tools, including breakpoints, watchpoints, and memory inspection.

Although the name "WebAssembly" suggests that it was developed for the web, WebAssembly is primarily a virtual ISA that can be embedded in an execution environment. We highly recommend to the curious reader the work of Haas et al. [17], where the design of WebAssembly is elaborated in great detail and advantages over other low-level IRs are discussed.

3.1 Embedding WebAssembly

Despite its many benefits, WebAssembly comes with two significant limitations. The first limitation is that WebAssembly does not provide a standard library. Data structures like hash tables, algorithms like sorting, and even basic routines such as `memcpy` are not available out of the box. The second limitation is that WebAssembly does not support generic programming. Hence, we cannot simply implement a library with generic algorithms and data structures ourselves. However, we shall work around these limitations by building on the ability to rapidly generate and compile WebAssembly. We solve the entire problem of not having a library by doing ad-hoc code generation: *Every algorithm and data structure required by a QEP is generated during compilation.* We do this in such a way, that we provide the concrete types of generic components, as required in the QEP, to the code generation process, which directly produces *monomorphic* code. Our approach allows us to rapidly generate code that is already fully inlined and specialized for the data types used in the QEP. We are able to achieve performance improvements that, in some cases, can have a tremendous impact. We elaborate our approach of ad-hoc library code generation in Section 5.

4 COMPILING SQL TO WEBASSEMBLY

In this section, we elaborate how to compile QEPs of SQL queries to WebAssembly. We dissect a QEP into pipelines, for which we generate code in topological order. We briefly revisit the pipeline model in Section 4.1. In Section 4.2 we sketch how we compile simple relational operators to WebAssembly. In Section 4.3 we

Listing 1 Example query to demonstrate the pipeline model.

```
SELECT R.x, MIN(S.x)
FROM R, S
WHERE R.x < 42 AND R.id = S.rid
GROUP BY R.x;
```

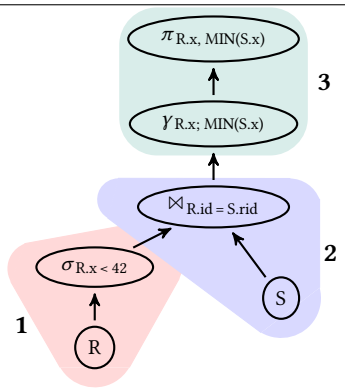


Figure 3: A QEP for the query in Listing 1, containing three pipelines enumerated in topological order.

explain how we compile complex operators without relying on an existing library by integrating ad-hoc generation of algorithms and data structures into the compilation process.

4.1 Pipeline Model

A QEP is – in its most essential form – a tree with tables or indexes at the leaves and relational operators at the inner nodes.[1] Figure 3 shows a QEP for the query in Listing 1. The edges between nodes of the tree point in the direction of data flow.

The tree structure of a QEP can be dissected into *pipelines* [9]. A pipeline is a linear sequence of operators that does not require materialization of tuples. To identify the pipelines of a QEP, we hence must identify all operators that require materialization, named *pipeline breakers* [32]. The most common pipeline breakers are grouping, join, and sorting; table scan, index seek, selection, and projection are not pipeline breakers.

In Figure 3, we have colored and enumerated the three pipelines of the QEP. Pipeline 1 scans table `R`, selects all tuples where `R.x < 42`, and inserts all qualifying tuples into a hash table for the join. Pipeline 2 scans table `S` and probes all tuples against the hash table constructed by pipeline 1. Every pair of tuples from `R` and `S` that satisfies the condition `R.id = S.rid` is joined and inserted into another hash table where groups of `R.x` are formed. Pipeline 3 iterates over these groups and performs the final projection.

After dissecting the QEP into pipelines, each pipeline is compiled separately. However, we must order the pipelines such that all data dependencies of the QEP are satisfied. For example, pipeline 3 iterates over all groups produced by grouping. Hence, pipeline 2 that forms those groups must be executed *before* pipeline 3. By topologically sorting the pipelines we compute an order that satisfies all data dependencies in the QEP.

The pipeline model allows us to dissect a QEP into linear sequences of operators that process tuples without need for intermediate materialization. The pipeline model does *not* dictate whether to push or pull tuples, whether to process tuples one at a time or in bulk, or whether to execute the QEP by compilation or interpretation. In this work, we compile the pipelines of a QEP such that a single tuple is pushed at a time through the entire pipeline until it is materialized in memory.

[1]The authors are aware that a QEP need not strictly be a tree and in some situations a representation as directed acyclic graph is desirable [33].

4.2 Compiling Simple Operators

To compile simple operators to WebAssembly, we follow the approach of Neumann [32] , i.e. we generate data-centric code. We do not yet implement advanced code generation techniques, such as relaxed operator fusion [30] or access-aware code generation [12]. However, the approach of Neumann [32] does not work for complex operators, as we will outline in Section 4.3. Generating WebAssembly code is very similar to generating LLVM code. In the following, we briefly sketch how we compile simple operators of a QEP.

Table scan, index seek, and pipeline breakers — The start of a pipeline – which is either a table scan, an index seek, or a pipeline breaker – is translated to a loop construct. For a table scan, we emit code to access all tuples of the respective table. For an index seek, we emit code to iterate over all qualifying entries in the respective index. For a pipeline breaker, e.g. grouping, we emit code to iterate over all materialized tuples, e.g. groups. The remainder of the pipeline is compiled into the loop's body.

Selection — A selection is compiled to a conditional branch. It is debatable whether to prefer short-circuit evaluation. For "simple" predicates, short-circuit evaluation is likely a bad choice: it introduces a conditional branch that unnecessarily stresses branch prediction [42]. It may further lead to a conditional load from memory, which may negatively impact prefetching [22]. For "complex" predicates, short-circuit evaluation likely pays off: a conditional branch can bypass costly evaluation of the right hand side of a logical conjunction or disjunction [39]. This transformation is a part of *if-conversion* [4]. We perform the aforementioned transformations during query optimization and *before* compilation. This optimization relies on domain-specific knowledge, e.g. predicate selectivities, that is inaccessible to the Wasm compiler. We do not implement predication in this work: every selection is compiled into one or more conditional branches.

Projection — The projection of an attribute or aggregate does not require an explicit operation. The code necessary to access the attribute's or aggregate's value has already been generated when compiling the beginning of the pipeline. To compile the projection of an expression, we compile the expression and assign the result to a fresh local variable. In contrast to interpretation, projecting attributes *away* is performed implicitly and requires no further code. Because the attribute that is projected away is not used further up in the QEP, no code using the attribute is generated. The register or local variable holding the attribute's value is automatically reclaimed during compilation to machine code [7].

4.3 Compiling Complex Operators

Compiling complex operators that need sophisticated algorithms and data structures is particularly difficult in our setting, as we cannot rely on an existing standard library providing generic implementations. We solve this deficiency by the ad-hoc generation of required algorithms and data structures during compilation of the QEP. We will explain this in detail in Section 5.

Hash-based Grouping & Aggregation — Hash-based grouping is a pipeline breaker: the incoming pipeline to the grouping operator assembles the groups in a hash table and updates the group's aggregates. The pipeline starting at the grouping operator iterates over all assembled groups as explained above.

An important distinction between our work and previous work is how inserts and updates to the hash table are performed. Previous work – including both interpretation- and compilation-based execution – relies on the existence of a pre-compiled library that provides a hash table implementation [25, 27, 32]. There, operations on the hash table must use a type-agnostic interface, ruling out effective implementations of certain has table designs. The major issue is that the type of a hash table entry is unknown at the time when the library is compiled. To lookup a key, the key's hash is required. The hash can be computed *outside* of the library and the computed hash value can be passed through the hash table's interface, as done by Neumann [32]. However, hash collisions must be resolved and duplicates must be detected. Because of the type-agnostic interface, the hash table has no means to compare two keys. Hence, a callback function for pair-wise comparison is passed to the hash table's lookup function. Note, that looking up n keys requires *at least* n such callbacks! The situation gets worse if the hash table must be able to grow dynamically. To grow a hash table, all elements of the table must be rehashed. Again, because the hash table is type-agnostic, it has no means to compute the hashes. Hence, a callback function for hashing must be provided in addition to the comparison callback or the computed hash values must be stored within the hash table. Another downside of using a pre-compiled library is that calls to the library cannot be inlined at the callsite: every access to the hash table requires a separate function call.

We resolve these issues by generating and JIT compiling the code for the hash table during compilation of the QEP. Although sounding expensive and prohibitive, we show in Section 8 that generating and compiling WebAssembly is affordable at running time. We explain the generation of library code in detail in Section 5.

Simple Hash Join — A simple hash join is a pipeline breaker for one of its inputs: the incoming pipeline, by convention the left subtree of the join, inserts tuples into a hash table. The pipeline of the join probes its tuples against that hash table to find all join partners. The same distinction between our work and previous work as for Hash-based Grouping & Aggregation applies here. To avoid artificial constraints on hash table design and to avoid issuing a function call per access to a hash table, we generate and JIT compile the required hash table code during compilation of the QEP. This approach is elaborated in Section 5.

Sorting — Sorting is a pipeline breaker and very similar to Grouping & Aggregation. Before the sorting operator can produce any results, all tuples of the incoming pipeline must be produced and materialized. After the incoming pipeline has been processed entirely, the sorting operator can output tuples in the specified order.

We implement the sorting operator by collecting all tuples from the incoming pipeline in an array and sorting the array with Quicksort. The way we integrate sorting into the compiled QEP is an important distinction between our work and previous work. In previous work that performs compilation, a sorting algorithm already exists as part of a pre-compiled library that is invoked to sort the array. The interface to this sorting algorithm is type-agnostic, i.e. the sorting algorithm does not know what it is sorting. In order to compare and move elements in the array, additional information must be provided when invoking the sorting algorithm. For comparison-based sorting, the size of an element in the array and a function that computes the order of two elements must be provided. This is very well exemplified by `qsort` from libc. This design leads to two severe performance issues. First, because the size of the elements to sort is not known when the library code is compiled, a generic routine such as `memcpy` must be used to move elements in the array. This may result in suboptimal code to move elements or even an additional function call per move. Additionally, values cannot be passed through registers and must always be read from and written to memory, obstructing optimization by the compiler. Second, to compute the order of two elements an external function must be invoked. This means, for every comparison of two elements the sorting algorithm must issue a separate function call. (To sort n elements, at least $\Theta(n \log n)$ such calls are necessary!)

When the QEP is being interpreted, e.g. in the vectorized execution model, similar problems emerge. Although tuples need not be moved if an additional array of indices is used, the sorting algorithm must delegate the comparison of two tuples to the interpreter, where the predicate to order by is dissected into atomic terms that are evaluated separately. This leads to significant interpretation overhead at the core of the sorting algorithm.

We resolve the aforementioned issues by generating and JIT compiling the library code during compilation of the QEP. Our generated sorting algorithm is precisely tuned to the elements to sort and the order to sort them by. In particular, the comparison of two elements is fully inlined into the sorting algorithm. We explain this approach in detail in Section 5.

5 AD-HOC LIBRARY CODE GENERATION

In Section 3.1 and Section 4.3 we already motivated ad-hoc generation of specialized library code during compilation of a QEP. In this section, we elaborate our technique along the example of generating specialized Quicksort. While other building blocks of QEPs, e.g. hash join, would also suit as interesting example of our approach, we choose Quicksort for two reasons: (1) Quicksort is a recursive algorithm. We demonstrate that our ad-hoc generation is not limited to mere code fragments but can generate entire recursive functions ad-hoc. (2) At its core, Quicksort repeatedly performs pair-wise comparison of elements when partitioning the data set. This part of the algorithm benefits most from specialization to a particular element type and sort order, demonstrating the significant impact specialization can have on performance (cf. Section 8.2). We begin with partitioning and inlined comparison of elements before we explain how we generate Quicksort. Note that we need not generate code for an entire library (e.g. libc or STL) but we generate code for only those routines required by the QEP. Therefore, ad-hoc generation must be implemented only for those parts of libraries used by QEPs.

5.1 Conceptual Comparison

Before diving into the code generation example, let us reconsider our approach on a conceptual level and compare it with alternatives. A problem that is inherent in all query execution engines is that their supported operations must be polymorphic. Joins, grouping, sorting, etc. must be applicable to attributes of any type and size. We aim to provide this polymorphism at query compilation time by generating specialized library code. To understand how other systems solve this task, let us look at state-of-the-art solutions.

Vectorized Interpretation — In the vectorized processing model, operations are specialized for the different types of vectors. In Listing 2, we provide an example for the evaluation of a

Listing 2 Vectorized processing example. A selection vector is successively refined to compute `R.x < 42 AND R.y > 13`.

```
1 /* Create a fresh vector with indices
2  * from 0 to VECTOR_SIZE - 1. */
3 sel0 = create_selection_vector(VECTOR_SIZE);
4 /* Evaluate LHS of conjunction. */
5 sel1 = cmp_lt_i32_imm(sel0, vec_R_x, 42);
6 /* Evaluate RHS of conjunction. */
7 sel2 = cmp_gt_i64_imm(sel1, vec_R_y, 13);
```

selection with a conjunctive predicate. The initial selection vector `sel0` is successively refined by calls to vectorized comparison functions `cmp_*` and eventually `sel2` contains all indices where the selection predicate is satisfied. A vectorized query interpreter executes a QEP by calling these vectorized functions and managing the data flow between function calls. To achieve short-circuit evaluation of the condition, the selection vector `sel1` is passed to the second comparison, such that the right-hand side of the conjunctive predicate is only evaluated for elements that also satisfy the left-hand side. In a compiling setting, short-circuit evaluation is usually implemented as a conditional branch. In the vectorized processing model, that control flow is converted to data flow. Conditional control flow can benefit from branch prediction, which works well in either case when the selectivity is very high or very low. However, when the control flow is converted to data flow, the benefit on low selectivities is lost [22, 37, 42], as we exemplify in our example in Listing 2. Assume that the left-hand side of the condition is barely selective. Although the outcome of evaluating the left-hand side can be well predicted, evaluation of the right- hand side in line 7 can only start once the comparison in line 5 completes. Hence, this design completely eliminates the processors ability to predict the outcome of evaluating the left-hand side and executing the right-hand side unconditionally and out of order, as opposed to how it would be in a data-centric setting. A drawback of interpretation is that operations must be specialized and compiled ahead of time. It is infeasible to provide vectorized operations for arbitrary expressions, as there are infinitely many. Therefore, the interpreter dissects expressions into atomic terms for which a finite set of vectorized operations is pre-compiled. For our example in Listing 2, this means that the interpreter must *always* evaluate one side of the conjunction after the other and cannot evaluate both sides at once.

Linking with pre-compiled library — In a compilation-based processing model, e.g. HyPer, every operation in the QEP is compiled to a code fragment. The produced code is specific to the types of the operation's operands. Arbitrarily complex expressions are compiled directly rather than taking a detour through pre-compiled functions for expression evaluation, like in the interpretation model. Thereby, the compiler can choose to implement short-circuit evaluation by conditional control flow.

The biggest drawback of compiling QEPs is the time spent compiling. While direct compilation to machine code could be done rapidly, the produced code would certainly be of poor quality. Therefore, compilation-based systems employ compiler frameworks like LLVM to perform optimizations on the code. While these optimizations can greatly improve the performance of the code, they require costly analysis and transformation. Hence, compilation of queries can easily take more than a hundred milliseconds [25].

To reduce the amount of code to compile, recurring routines like hash table lookups or sorting are pre-compiled and shipped in a library. During compilation of a QEP, when an operation can be delegated to a pre-compiled routine, the compiler simply

Listing 3 Demonstration of a compilation-based processing model with calls to a pre-compiled library. Every insertion into the hash table requires a separate function call.

```
1  /* Initialize hash table. */
2  HT *ht = lib_HT_create();
3  /* Iterate over all rows of table R. */
4  for (auto row : tbl_R) {
5    /* Evaluate selection predicate. */
6    if (row.x < 42 and row.y > 13) {
7      /* Compute hash of R.id. */
8      auto hash = ... row.id ...;
9      /* Insert into hash table. */
10     char *ptr = lib_HT_insert(ht, hash, 8);
11     *(int*) ptr    = row.id; // key
12     *(int*)(ptr+4) = row.x;  // value
13   }
14 }
```

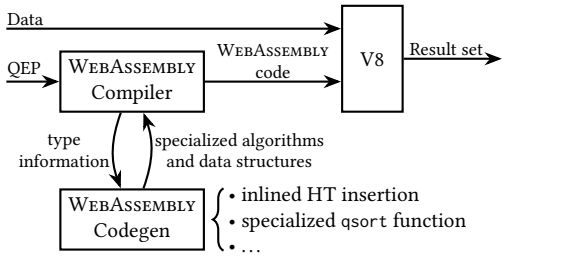


Figure 4: Compilation with on-demand code generation in mutable. QEPs are compiled to WebAssembly and dispatched to V8. Specialized code is generated on demand for algorithms and data structures required by the QEP.

produces a respective function call to the library. This is a trade-off between compilation time and running time and the biggest drawback of this approach. Function calls to a pre-compiled library prevent inlining and obstruct further optimization, thereby potentially leading to sub-optimal performance. We demonstrate this in Listing 3, where every insertion into a hash table requires a separate function call. The library code for probing the hash table can be compiled and optimized thoroughly ahead of time. Because the size of a hash table entry is unknown when the library is compiled, the size must be provided at running time when inserting an entry. In the example, the hash table must allocate 8 bytes per entry to store `R.id` and `R.x` and it is the task of the caller to assign those values to the entry.

Full compilation — In this approach, code for the entire QEP with all required algorithms and data structures is generated and compiled just in time. By generating the code just in time, it is possible to produce highly specialized code, target particular hardware features, and enable holistic optimization. One example for full compilation is template expansion, as done in the Hique system [27]. Hique provides a set of generic algorithms and data structures that are instantiated and compiled to implement the QEP. Another example is code generation via *staging*, as done in LegoBase. Here, metaprogramming is used to write a query engine in Scala LMS, that when partially evaluated on an input QEP outputs specialized C code that implements the query [24]. While full compilation can achieve the highest possible throughput, both Hique and LegoBase take considerable compilation time with hundreds of milliseconds for single TPC-H queries.

5.2 Our Approach: JIT Code Generation

Full compilation is very similar to our approach of generating required library routines just in time and JIT compiling the QEP. The key distinction is how code is generated. Previous work generates code in a high-level language. This code must then go through parsing and semantic analysis before it is translated

Listing 4 Pseudo code for the generation of specialized code that implements Hoare's partitioning.

```
1:  function Partition(order, begin, end, pivot)
2:      Emit(l ← begin)
3:      Emit(r ← end)
4:      Emit(while l < r)
5:      EmitSwap(l, r − 1)
6:      cl ← EmitCompare(order, l, pivot)
7:      cr ← EmitCompare(order, pivot, r − 1)
8:      Emit(l ← l + cl)
9:      Emit(r ← r − cr)
10:     Emit(end while)
11:     return l
12: end function
```

to a lower level IR where optimizations are performed before executable machine code is produced. Going through the entire compiler machinery takes a lot of time. Our approach, depicted in Figure 4, bypasses most of these steps. We generate specialized algorithms and data structures directly in WebAssembly. By picking a suitable WebAssembly engine, e.g. V8, we fulfill all the requirements given in Section 2.2. Our approach is able to produce highly specialized algorithms and data structures and enables holistic optimization without the drawback of long code generation and compilation times.

5.3 Code Generation by Example

To provide the reader with a better understanding of how we generate library code just in time, let us exemplify our code generation along the example of Quicksort. We build the example bottom up, beginning with code generation for partitioning and the comparison of two elements before we explain code generation of the recursive Quicksort algorithm. We use pseudo code, as it is easier to read and understand than Wasm and because our approach of ad-hoc code generation is independent of a particular language.

Hoare's partitioning scheme — Hoare's partitioning scheme creates two partitions from a sequence of elements based on a boolean predicate such that all elements in the first partition do not satisfy the predicate and all elements in the second partition satisfy the predicate. We apply Hoare's partitioning in our generated Quicksort algorithm, that in turn is used to implement sorting of tuples. In our setting, the sequence of tuples to partition is a consecutive array.

We provide pseudo code for the generation of specialized partitioning code in Listing 4. The function Partition takes four parameters: the *order* is a list of expressions to order by, *begin* and *end* are variables holding the address of the first respectively one after the last tuple in the array to partition, and *pivot* is a variable holding the address of the pivot to partition by. The pivot must not be in the range [*begin*, *end*). First, the algorithm copies the values of *begin* and *end* by introducing fresh variables l and r in lines 2 and 3 and then emitting code that assigns the value of *begin* to l in line 4 and the value of *end* to r in line 5. Next, in line 6, a loop header with the condition $l < r$ is emitted. The code emitted thereafter forms the loop body. In line 7, EmitSwap is called to emit code that swaps the tuples at the addresses l and $r - 1$. Note that this is a function call *during code generation*. The call will emit code directly into the loop body, as if inlined by an optimizing compiler, and there will be no function call during execution of the generated code. In lines 8 and 9, EmitCompare is called to emit code that compares the tuples at addresses l and $r - 1$ to the tuple at address *pivot* according to the order specified by *order*. Each call returns a fresh boolean variable that holds the outcome of the comparison. Just like EmitSwap, calls

Listing 5 Pseudo code for the generation of code that compares two elements based on a specified order.

```
1:  function EmitCompare(order, l, r)
2:      Emit(v ← 0)
3:      for each expr in order do
4:          vl ← Compile(expr, l)
5:          vr ← Compile(expr, r)
6:          switch type(expr) do
7:              case int
8:                  Emit(lt ← vl <int vr)
9:                  Emit(gt ← vl >int vr)
10:             case float
11:                 Emit(lt ← vl <float vt)
12:                 Emit(gt ← vl >float vt)
13:             ...                          ▷ cases for remaining types
14:         end switch
15:         Emit(v ← 2 · v + gt − lt)
16:     end for
17:     Emit(c ← v < 0)
18:     return c
19: end function
```

to EmitCompare emit code directly into the loop body without the need for a function call in the generated code. The value of variable *cl* will be true if the tuple at address l compares less than the tuple at address *pivot* w.r.t. the specified *order*. Line 10 emits code that advances l to the next tuple if *cl* is true, otherwise l is not changed. Similarly, line 11 emits code to advance r to the previous tuple if *cr* is true. This is a means of implementing branch-free partitioning. In line 12, the loop body for the loop emitted in line 6 is finished. Eventually, Partition returns the variable l, which will point to the beginning of the second partition once the loop of line 6 terminates.

The code presented in Listing 4 looks almost like a regular implementation of partitioning. However, the function *emits* code that will perform partitioning. An important part of partitioning, that we skipped in Listing 4, is how the code to compare two tuples based on a given order is generated. Therefore, we also provide pseudo code for EmitCompare in Listing 5.

First, EmitCompare creates a fresh variable v in line 2 and initializes it to 0 in line 3. Then, in line 4, the function iterates over all expressions in *order*. The call to Compile in line 5 emits code to evaluate *expr* on the tuple pointed to by l and returns a fresh variable holding the value of the expression. Analogously, line 6 evaluates *expr* on the tuple pointed to by r. Next, type-specific code to compare the values *vl* and *vr* of the evaluated expression is emitted. Because the particular code to emit depends on the type of *expr*, line 7 performs a case distinction on the type. This case distinction is performed *during code generation* and the generated code will only contain the emitted, type-specific code. In case the expression evaluates to an `int`, lines 9 and 10 emit code to perform an integer comparison of *vl* and *vr*. The cases for other types are analogous. After emitting type-specific code for the comparison of *vl* and *vr*, line 16 emits code to update v based on the outcome of the comparison. After generating code to evaluate all expressions in *order* and updating v accordingly, lines 18 and 19 introduce a fresh boolean variable c that will be set to $v < 0$, which evaluates to true if the tuple at l is strictly smaller than the tuple at r.

To put it all together, let us exercise an example. We invoke Partition with the *order* $[R.x + R.y, R.z]$, *begin* 'b', *end* 'e', and *pivot* 'p'. The generated code is given in Listing 6. Initially, in lines 1 and 2, the addresses of the first and one after the last tuple are stored in fresh variables. Then the loop in line 3 repeats as long as pointer p_l points to an address smaller than p_r. Lines 5 to 7 show the code produced by EmitSwap, that swaps two tuples using a temporary variable. In lines 9 to 20, the tuple at p_l is compared to the pivot according to the specified order. Variable

Listing 6 Generated partitioning code for the order $[R.x + R.y, R.z]$.

```
Input: b, e, p
Output: p_l
 1: var p_l ← b                          ▷Initialize pointers to the first and
 2: var p_r ← e                          ▷one after the last tuple, respectively.
 3: while p_l < p_r do
 4:     ▷ EmitSwap(p_l, p_r − 1)
 5:     var v_tmp ← *p_l                 ▷Use temporary variable
 6:     *p_l ← *(p_r − 1)                ▷to swap tuples
 7:     *(p_r − 1) ← v_tmp               ▷at p_l and p_r − 1.
 8:     ▷ EmitCompare([R.x + R.y, R.z], p_l, p)
 9:     var v_l,pivot ← 0
10:     var v_l ← p_l.x +int p_l.y       ▷Compile(R.x + R.y, p_l)
11:     var v_pivot ← p.x +int p.y       ▷Compile(R.x + R.y, p)
12:     var v_lt ← v_l <int v_pivot
13:     var v_gt ← v_l >int v_pivot
14:     v_l,pivot ← 2 · v_l,pivot + v_gt − v_lt
15:     var v_l ← p_l.z                  ▷Compile(R.z, p_l)
16:     var v_pivot ← p.z                ▷Compile(R.z, p)
17:     var v_lt ← v_l <int v_pivot
18:     var v_gt ← v_l >int v_pivot
19:     v_l,pivot ← 2 · v_l,pivot + v_gt − v_lt
20:     var v_cl ← v_l,pivot < 0
21:     ▷ EmitCompare([R.x + R.y, R.z], p, p_r − 1)
22:     ...                              ▷Code omitted for brevity.
23:     var v_cr ← v_pivot,r < 0
24:     p_l ← p_l + v_cl                 ▷Advance left cursor.
25:     p_r ← p_r − v_cr                 ▷Advance right cursor.
26: end while
```

Listing 7 Pseudo code to generate specialized Quicksort.

```
 1: function Quicksort(order)
 2:     Emit(function qsort(begin, end))
 3:     Emit(while end − begin > 2)
 4:     Emit(mid ← begin + (end − begin)/2)
 5:     m ← EmitMedianOf3(begin, mid, end − 1)
 6:     EmitSwap(begin, m)
 7:     mid ← Partition(order, begin + 1, end, begin)
 8:     EmitSwap(begin, mid − 1)
 9:     Emit(if end − mid ≥ 2)
10:     Emit(qsort(mid, end))
11:     Emit(end if)
12:     Emit(end ← mid − 1)
13:     Emit(end while)
14:     Emit(end function)
15: end function
```

v_{cl} is true if the tuple at p_l compares less than the pivot, false otherwise. Analogously, the tuple at $p_r - 1$ is compared to the pivot. To keep the example short and because the code is very similar, we omit this code and only show a place holder in line 22. At the end of the loop, in lines 24 and 25, the pointers p_l and p_r are advanced depending on the outcome of the comparisons.

The generated code will partition the range [b, e) such that the first partition contains only tuples that compare less than p and the second partition contains only tuples greater than or equal to p, w.r.t. the specified order. Note that the generated code is not a function. Instead, this code can be generated into a function where partitioning is needed. Hence, the entire code for partitioning will always be fully inlined and specialized for the order to partition by.

Quicksort — Quicksort sorts its input sequence by recursive partitioning. In our implementation of Quicksort, we compute the pivot to partition by as a *median-of-three*. With our code generation for partitioning at hand, generating Quicksort is relatively simple. We provide pseudo code in Listing 7. Line 2 defines a new function `qsort`, line 3 emits a loop that repeats as long as there are more than two elements in the range from *begin* to *end*. Inside this loop, lines 4 to 7 emit code to compute the median of three and bring the median to the front of the sequence to sort. Line 8 emits the code to partition the sequence $begin + 1$ to *end* using as pivot the median of three. After partitioning, the median must be swapped back into the partitioned sequence, which is done by line 9. Line 10 checks whether to recurse into the right partition. Line 11 emits a recursive call to sort the right partition with `qsort`. Afterwards, in line 13, code is emitted to update *end* to the end of the left partition.

We can see that by executing our Quicksort code generation, we obtain a specialized, fully inlined `qsort` function that can be called to sort a sequence by the *order* specified during code generation.

6 EXECUTING WEBASSEMBLY IN A DATABASE SYSTEM

In the preceding sections, we explained how to compile a QEP and its required libraries to WebAssembly. In this section, we elaborate how we execute WebAssembly in an embedded engine. Although this approach works with any embeddable engine, we describe the process of embedding and executing WebAssembly in V8.

The WebAssembly specification requires that each *module* – think of translation unit in C – operates on its personal memory. This memory is provided by the engine, here V8. To execute a compiled QEP inside the engine, all required data (tables, indexes, etc.) must reside in the module's memory. One way to achieve this is by copying all data from the host to the module's memory. However, this incurs an unacceptable overhead of copying potentially large amounts of data before executing the QEP. An alternative is to use callbacks from the module to the host to transfer single data items on demand. For such a purpose, V8 allows for defining functions in the embedder that can be called from the embedded code. However, such callbacks also incur a tremendous overhead, because the VM has to convert parameters and the return value from the representation in embedded code to the representation in the embedder and vice versa. At the time of writing, V8 provides no method to use pre-allocated host memory as a module's memory. Therefore, we patch V8 to add a function for exactly that purpose: `SetModuleMemory()` sets the memory of a WebAssembly module to a region of the host memory. While this function enables us to provide a single consecutive memory region from the host to the module, it is not sufficient to provide multiple tables or indexes (which need not reside in a single consecutive allocation) to a module. The problem is that WebAssembly – in its current version – only supports 32 bit addressing. Hence, we cannot simply assign the entire host memory to the module. Instead, we are limited to 4 GiB of addressable *linear memory* inside the module. In the following, we describe pagination implemented with a technique named *rewiring* to work around this limitation. However, we want to stress that 64 bit addresses in WebAssembly are on the way and pagination will become obsolete eventually.

6.1 Accessing Data by Rewiring

Rewiring [41] allows for manipulating the mapping of virtual address space to physical memory from user space. In particular, it enables us to map the same *physical* memory at two distinct *virtual* addresses. We exploit this technique to have data structures residing in distinct allocations appear consecutively in virtual address space and then use this address range as the module's memory.

We exemplify this technique in Figure 5. Assume a query accessing two tables `A` and `B`. The tables reside in completely independent memory allocations, hence there is no single 4 GiB virtual address range that contains both `A` and `B` entirely. Further, the query computes some results and we therefore allocate 1 GiB

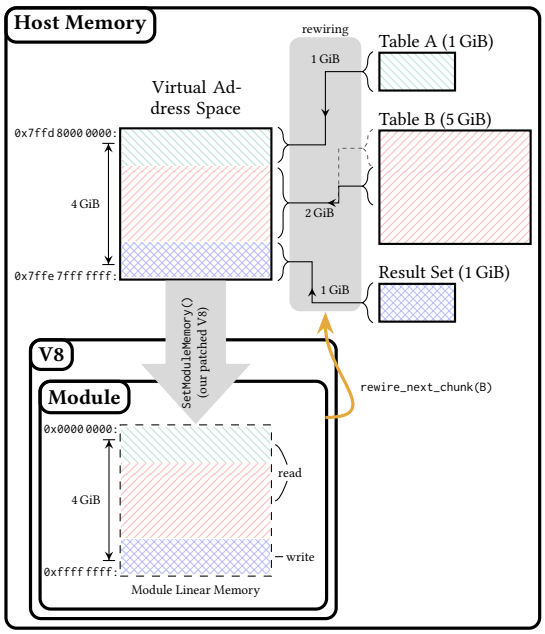


Figure 5: Example of mapping tables and output to a module's memory. The module can callback to the host to request mapping the next 2 GiB chunk of table B.

of memory to store the query's result set. To give the module access to all required memory, we first allocate consecutive 4 GiB in virtual address space. Then we *rewire* table A, a portion of table B, and the memory for the result set into the freshly allocated virtual memory. Finally, we call `SetModuleMemory()` with the freshly allocated virtual memory. The module now has access to both tables and can write its results to the memory allocated for the result set. Note that table B is 5 GiB and cannot be rewired entirely into the virtual memory for the module. To give the module access to the entire table, we install a callback `rewire_next_chunk()` that lets the host rewire the next 2 GiB chunk of table B, thereby allowing the module to iteratively process entire B.

6.2 Result Set Retrieval

Similarly to how data is made available to the module, we use rewiring to communicate the result set back to the host. As can be seen in Figure 5, the module writes the result set to a rewired allocation of 1 GiB. If the module produces a result set of more than 1 GiB, it produces the result set in chunks and issues a callback in between to have the host process the current chunk of results.

7 RELATED WORK

In addition to our motivation for using V8 and WebAssembly in Section 2.2 and Section 3, respectively, we present in Section 7.1 JIT frameworks and engines that might be used alternatively. In Section 7.2, we augment the comparison with related work that is conducted throughout Section 2.1, Section 4.3, and Section 5.1.

7.1 JIT Frameworks & Engines

The idea to build query execution on top of a JIT engine occurred as early as 1997 in the context of JIT compilation in the Java Virtual Machine (JVM) [11]. These thoughts were later implemented in Java's HotSpot VM [26] and are still being developed today in GraalVM [35]. Let us have a quick look at JIT compilers and engines we considered: MIR [29] provides an IR with an interpreter and fast JIT compiler, however it is still in early development by only a small community and does not provide adaptive execution out of the box. LibJIT [14] provides a framework for on-demand JIT compilation and reoptimization. However, it is lacking automation of adaptive execution and inlining. When we focus on executing WebAssembly, we have access to a rich set of engines. For brevity, we only mention a few examples: Wasmer [43] is a feature-rich WebAssembly runtime but does not provide efficient embedding. SpiderMonkey is Mozilla's JavaScript and WebAssembly engine and quite similar to Google's V8. We chose V8 over SpiderMonkey because of its better documentation and because it is written in C++, like our database system mutable.

7.2 Query Execution

Interpretation. — Graefe [15] proposes a unified and extensible interface for the implementation of relational operators in Volcano, named *iterator* interface. Ailamaki et al. [3] analyze query execution on modern CPUs and find that poor data and instruction locality as well as frequent branch misprediction impede the CPU from processing at peak performance. Boncz et al. [8] identify tuple-at-a-time processing as a limiting factor of the Volcano iterator design, that leads to high interpretation overheads and prohibits data parallel execution. To overcome these limitations, Boncz et al. [8] propose *vectorized* query processing, implemented in the X100 query engine within the column-oriented MonetDB system. Menon et al. [30] build on the vectorized model and introduce *stages* to dissect pipelines into sequences of operators that can be *fused*. By fusing operators, Menon et al. are able to vectorize multiple sequential relational operators. Their implementation in Peloton [36] shows that operator fusion increases the degree of inter-tuple parallelism exploited by the CPU.

Compilation. — Rao et al. [38] explore compilation of QEPs to Java and having the JVM JIT-compile and load the generated code. However, their approach sticks to the Volcano iterator model, restricting compilation from unfolding its full potential. Follow-up work explores compiling QEPs to vectorized Java code in Spark [2]. Potential compilation overheads are not discussed. Schiavio et al. [40] and Grulich et al. [16] both recently explored the support of polyglot UDFs. Both approaches build on Truffle, GraalVM's compiler-compiler. We believe the JavaScript + WebAssembly eco system could very well support polyglot programming, too. There already exists a wide range of transpilers, e.g. Transcript translates Python to JavaScript or Emscripten compiles LLVM-based languages to WebAssembly. With HIQUE, Krikellas et al. [27] propose query compilation to C++ code by dynamically instantiating operator templates in topological order. They report query compilation times in the hundreds of milliseconds. Neumann [32] presents compilation of pipelines in the QEP to tight loops in LLVM. Complex algorithms are implemented in C++ and pre-compiled, to be linked with and used by the compiled query. With the implementation in HyPer, Neumann achieves significantly reduced compilation times in the tens of milliseconds. Klonatos et al. [24] address the system complexity and the associated development effort of compiling query engines in their LegoBase system, where metaprogramming is used to write a query engine in Scala LMS that, when partially evaluated on an input QEP, yields specialized C code that implements the query. Despite the clean design, the code generation through partial evaluation as well as the compilation

of the generated code leads to compilation times in the order of seconds.

Adaptive. — A recent advancement in query execution is adaptive execution by Kohn et al. [25] and Kersten et al. [23], that we already discussed in great detail in Section 2.1.

8 EVALUATION

In this section, we want to experimentally verify that our architecture of embedding an off-the-shelf JIT engine provides competitive performance to state-of-the-art systems. We want to stress that our goal is *not* to outperform existing systems but to demonstrate that our approach enables us to achieve similar performance at much lower engineering costs.[2] We begin by evaluating the performance of QEP building blocks, then we survey TPC-H queries, and finally we examine compilation times.

8.1 Experimental Setup

We implement our approach in mu*t*able [18], a main-memory database system currently developed at our group. Although mu*t*able supports arbitrary data layouts, we conduct all experiments using a columnar layout. Since mu*t*able does not yet support multi-threading, all queries run on a single core.

We compare to three systems: (1) PostgreSQL 13.1 as representative for Volcano-style tuple-at-a-time processing, (2) DuckDB v0.2.8, implementing the vectorized model as in MonetDB/X100, and (3) HyPer, an adaptive system performing interpretation and compilation of LLVM bytecode, as provided by the `tableauhyperapi` Python package in version 0.0.11952. For PostgreSQL, we disable JIT compilation as it does not improve execution time in any of our experiments. The version of HyPer implements the adaptive approach of Kohn et al. [25] , that cannot be disabled. We therefore report HyPer's adaptive execution times composed of interpreted and compiled execution. We run all our experiments on a machine with an AMD Ryzen Threadripper 1900X CPU with 8 physical cores at 3.60 GHz and 32 GiB main memory. All data accessed in the experiments is memory resident. We repeat each experiment five times and report the median.

8.2 Performance of Query Building Blocks

With our first set of experiments, we evaluate the performance of individual query building blocks across different systems. We use a generated data set with multiple tables and 10 million rows per table. Tables contain only integer and floating-point columns, where integer values are chosen uniformly at random from the entire integer domain and floating-point values are chosen uniformly at random from the range $[0; 1]$. All data is shuffled and all columns are pairwise independent. For HyPer and mu*t*able, we report only execution times without compilation times. We further enforce compilation with the optimizing TurboFan compiler.

Selection. In our first experiment, we run several `COUNT(*)`-queries with different `WHERE`-clauses to evaluate the performance of selection. Figure 6 (a) and (b) show our measurements for selection on a 32-bit integer and a 64-bit floating-point column, respectively. We omit our findings for PostgreSQL, as the times are above 200 ms. Both mu*t*able and DuckDB implement selection by conditional branching. Therefore, both systems suffer from frequent branch misprediction at selectivities around 50% [39, 42]. The execution time of HyPer remains unaffected by varying selectivity; our educated guess is that HyPer compiles branch-free code. We can see that mu*t*able outperforms DuckDB on all selectivities and for both integer and floating-point columns. This is likely the case because DuckDB, which implements the vectorized execution model, has the overhead of maintaining a selection vector [37, 42].

We conduct two additional experiments, where we perform a selection on two independent integer columns. In the first experiment, shown in Figure 6 (c), both conditions are varied with equal selectivity. This means, the overall selectivity of the selection is the squared selectivity of either condition. Since mu*t*able does not implement short-circuit evaluation and instead evaluates the selection as a whole, a selectivity of $\sqrt{50\%} \approx 71\%$ per condition presents the worst-case for branch prediction with a time of 47 ms. DuckDB, which implements the vectorized model, must first evaluate one condition to a selection vector before evaluating the second condition on the selected rows. Because the conditions are evaluated individually, branch misprediction occurs up to twice as often and branch prediction is worst at a selectivity of 50% with an execution time around 78 ms. As the selectivity grows, the second condition must be evaluated more often. This can be seen in the slight asymmetry of the execution time curve. HyPer's execution time significantly grows with the selectivity from around 10 ms at 0% to 43 ms at 100%. We assume that HyPer again produces branch-free code. However, the second column is only accessed if the first condition is satisfied.

In the second experiment, shown in Figure 6 (d), only one condition is varied while the other is fixed to a selectivity of 1%. The overall selectivity of the selection is hence in the range from 0% to 1%. Since mu*t*able evaluates the entire selection as a whole, branch prediction works reliably well and we observe a constant execution time of around 15 ms. DuckDB evaluates the more selective condition first, resulting in a constant execution time around 24 ms.

Grouping & Aggregation. Our next experiment evaluates the performance of grouping and aggregation. We run several `COUNT(*)`-queries with different `GROUP BY`-clauses. We vary the experiment in several dimensions: the number of rows in the table, the number of distinct values in the column being grouped by, and the number of attributes to group by. Figure 7 presents our findings.

In Figure 7 (a) to (c), a lion share of execution time is spent on hash table operations. mu*t*able generates a specialized hash table implementation per query, with all hash table operations fully inlined into the query code, and is thereby able to gain an advantage over the other systems. We must note that HyPer achieves *impossible* execution times in Figure 7 (b) for 10 to 10*k* distinct values and consequently in Figure 7 (c) for a single attribute to group by (in which case there are 10k groups): HyPer answers these `COUNT(*)`-queries from internal statistics rather than executing the query.

In Figure 7 (d) we evaluate the performance of aggregation by altering the `SELECT`-clause to compute a varying number of aggregates, i.e. `MIN(T.x1), ..., MIN(T.xn)`. The time for hashing is dwarfed by the time to compute the aggregates. DuckDB computes the minimum using conditional control flow. Note that the values of each column are uniformly distributed and shuffled. Branch prediction works reliably well, because the likeliness of

[2] Please be aware that performance differences are not only due to architectural differences but – much more likely – due to different implementations of the same algebraic operations. In mu*t*able we only use text-book implementations of algebraic operations.

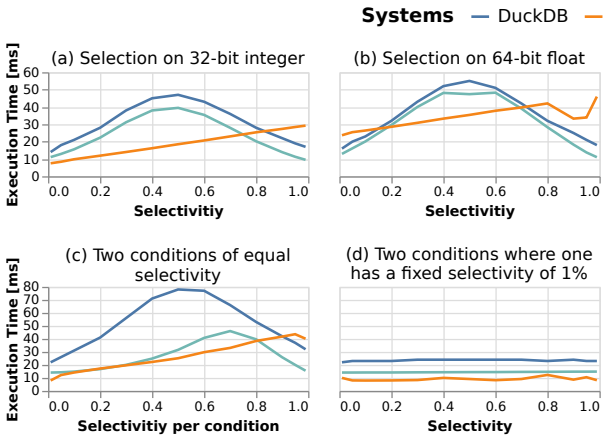


Figure 6: Evaluation of selection with one and two one-sided range predicates.

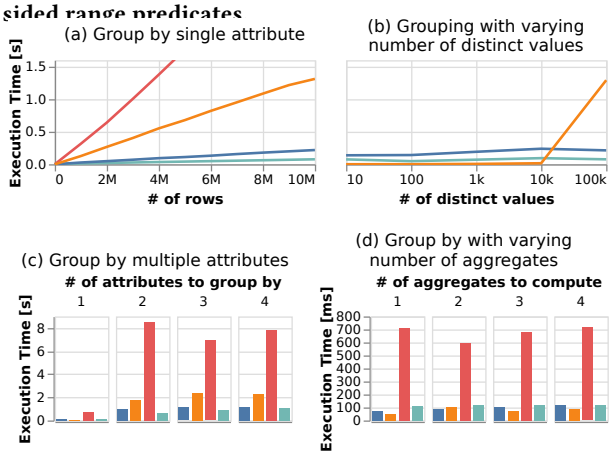


Figure 7: Evaluation of grouping & aggregation.

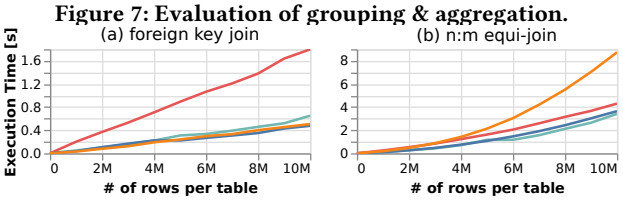


Figure 8: Evaluation of equi-join.

finding a new minimum steadily decreases as the table is further processed. mutable generates branch-free code and cannot benefit from branch prediction.

Equi-Join. In this experiment, we evaluate the performance of a foreign key join and an *n*:*m* equi-join. We perform the *n*:*m* join on non-key columns to avoid the systems using a pre-built index, since mutable does not yet support indices[3]. We fix the selectivity of the joins to 10^{-6} and vary the size of the input relations. We present our findings in Figure 8. In (a), we see the expectable linear behaviour, with all systems but PostgreSQL showing similarly good performance. In (b), we observe the expectable quadratic behaviour. Notable here is that HyPer exhibits a strong curvature and is the slowest of the systems from around 4M rows onwards. Our educated guess is that duplicates w.r.t. the join predicate lead to long collision chains in HyPer's hash table.

Sorting. Our last experiment evaluates the performance of sorting, as needed in `ORDER BY`-clauses or for merge-join. Similar to the experiment on grouping, we vary the experiment in several dimensions: the number of rows in the table, the number

[3]In particular, mutable cannot map non-consecutive data structures like indices from process memory into the WebAssembly VM. This is future work.

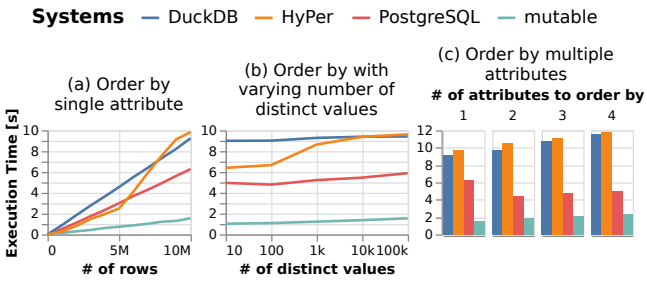
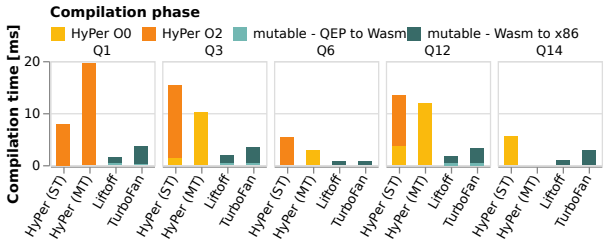
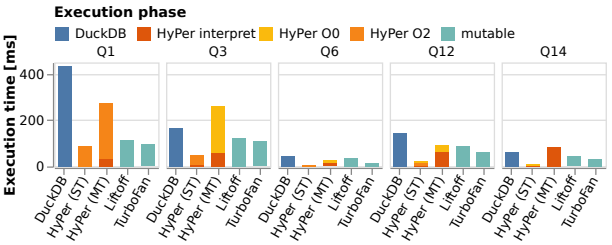


Figure 9: Evaluation of ordering.



(a) Detailed compilation times of HyPer and mutable.



(b) Detailed execution times of DuckDB, HyPer, and mutable.

Figure 10: Evaluation of TPC-H queries on a SF 1 database.

of attributes to order by, and the number of distinct values in the column to order by. Figure 9 presents our findings. We can observe a significant improvement over state of the art, that we credit to mutable's generation and consequent holistic optimization of the sorting operation, described in detail in Section 5.3. In contrast to interpretation or the use of a pre-compiled library, in Quicksort generated by mutable the pair-wise comparison of elements and the entire routine for partitioning are fully inlined and specialized to the elements' type. No callbacks are required to compare elements and no dynamic dispatches are required to determine an element's runtime type.

8.3 TPC-H

So far, our experiments only focus on individual query building blocks. Next, we conduct an experimental evaluation of TPC-H queries. By the time of writing, mutable – and in particular our WebAssembly backend – only supports a subset of SQL and hence we are only able to evaluate the TPC-H queries 1, 3, 6, 12, and 14. We are further constrained to SF-1, as for larger scale factors the data structures constructed during execution exceed the Wasm linear memory. We are currently unable to exploit the technique described in Section 6 for data structures; it is only applied to rewire single, consecutive memory regions. HyPer writes a log that includes detailed timings of the different compilation and execution phases. These timings are barely labeled and there is no documentation or otherwise explanatory information available for these values. We have discussed these values with Thomas Neumann, the original author of HyPer. With the knowledge gained from him, we assign meaning to these values to the best of our abilities. HyPer adaptively switches between

two compilation phases and three execution phases (cf. Figure 2a, H1-3), sometimes leaving out a phase. We therefore chose to stack HyPer's timings in the visualization and encode the different phases in the color. A limitation of HyPer's log is that we only know the duration of the individual phases, but not when these phases began. This is particularly unsatisfying, because compilation and execution phases may overlap, e.g. interpreted execution may overlap with O2 compilation. We evaluate HyPer with a single thread (ST) and with as many threads as we have CPU cores (MT). For mu*t*able, we provide detailed timings for the translation of the QEP to WebAssembly and the compilation and execution of WebAssembly with Liftoff and TurboFan. We present our findings in Figure 10.

With regard to compilation times, mu*t*able's optimizing compilation with TurboFan is up 6.6*x* faster than HyPer's LLVM-based optimizing compilation (Q1) and mu*t*able's fast compilation with Liftoff is up to 7.4*x* faster than HyPer's non-optimizing compilation (Q12). Note that mu*t*able's compilation times include the *generation and compilation* of required algorithms and data structures, e.g. hash table operations. At the same time, mu*t*able's execution times are competitive to HyPer's – except for Q14 where HyPer on a single core significantly outperforms all other systems. Also note that for Q14, HyPer (MT) never compiles the query and only performs interpreted execution.

9 CONCLUSION

Our goal was to simplify the architecture of query execution engines while fulfilling the high-level requirements of low latency, high throughput, and adaptive execution. We proposed to embed a suitable off-the-shelf JIT engine into the database system to delegate the execution of QEPs to. By compiling QEPs to an efficient IR and delegating execution to said engine, we have met that goal. Our architecture requires much less engineering and maintenance than previous solutions. At the same time, our experimental evaluation confirms that we achieve low latency because of fast JIT code generation and high throughput because of adaptive (re-) optimization during query execution – all fully handled by the embedded engine.

We are convinced that our approach is considerably simpler to understand and implement than current state-of-the-art. By relying on successful, battle-tested infrastructure for JIT compilation and execution, we significantly reduce the required development effort to build an adaptive yet highly efficient query execution engine. With the ongoing standardization of WebAssembly [1, 13] and the immense interest and amount of ongoing work in engines supporting this language [5, 6, 20, 31, 43], our approach provides a reliable and future-proof solution to adaptive query execution.

Further, we think that our ad-hoc generation of specialized algorithms and data structures shapes a new path for query compilation, potentially leading to much more efficient query processing than currently possible.

REFERENCES

[1] [n.d.]. *WebAssembly.* www.webassembly.org

[2] Sameer Agarwal, Davies Liu, and Reynold Xin. 2016. Apache Spark as a compiler: Joining a billion rows per second on a laptop.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie (Eds.). Morgan Kaufmann, 266–277. http://www.vldb.org/conf/1999/P28.pdf

[4] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 177–189.

[5] Bytecode Alliance. [n.d.]. *Wasmtime.* http://wasmtime.dev

[6] V8 Project Authors. 2008. *V8: Google's open source high-performance JavaScript and WebAssembly engine.* https://v8.dev/

[7] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dine hin, and Fabri e Rastello. 2008. Fast liveness checking for SSA-form programs. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 35–44.

[8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[9] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. 1996. *Dynamic load balancing in hierarchical parallel database systems.* Ph.D. Dissertation. INRIA.

[10] Donald D Chamberlin, Morton M Astrahan, Michael W Blasgen, James N Gray, W Frank King, Bruce G Lindsay, Raymond Lorie, James W Mehl, Thomas G Price, Franco Putzolu, et al. 1981. A history and evaluation of System R. *Commun. ACM* 24, 10 (1981), 632–646.

[11] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. 1997. Compiling Java Just in Time. *IEEE Micro* 17 (1997), 36–43. Issue 3.

[12] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2020. Getting swole: Generating access-aware code with predicate pullups. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1273–1284.

[13] MDN Web Docs. [n.d.]. *WebAssembly Developer Reference.* https://developer.mozilla.org/en-US/docs/WebAssembly

[14] Free Software Foundation, Inc. [n.d.]. *LibJIT.* https://www.gnu.org/software/libjit

[15] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

[16] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: efficient execution of polyglot queries. *Proceedings of the VLDB Endowment* 15, 2 (2021), 196–210.

[17] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[18] Immanuel Haffner. 2020. mu*t*able. https://bigdata.uni-saarland.de/projects/mutable

[19] Clemens Hammacher. 2018. Liftoff: a new baseline compiler for WebAssembly in V8. *V8 JavaScript engine* (2018).

[20] Apple Inc. [n.d.]. *WebKit.* webkit.org

[21] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: Analyzing the performance of webassembly vs. native code. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 107–120.

[22] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2209–2222.

[23] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* (2021), 1–23.

[24] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. *Proceedings of the VLDB Endowment* 7, 10 (2014), 853–864.

[25] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 197–208.

[26] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)* 5, 1 (2008), 1–32.

[27] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 613–624.

[28] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 743–754.

[29] Vladimir Makarov. [n.d.]. *MIR.* https://github.com/vnmakarov/mir

[30] Prashanth Menon, Todd C Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[31] Mozilla Developer Network. [n.d.]. *SpiderMonkey.* developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[32] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[33] Thomas Neumann and Alfons Kemper. 2015. Unnesting arbitrary queries. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).

[34] Tobias Nießen, Michael Dawson, Panos Patros, and Kenneth B Kent. 2020. Insights into WebAssembly: compilation performance and shared code caching in Node. js. In *EVOKE CASCON 2020*. ACM, 163–172.
[35] Oracle. [n.d.]. *OpenJDK: Graal project*. openjdk.java.net/projects/graal/
[36] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.
[37] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo-a vector algebra for portable database performance on modern hardware. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1707–1718.
[38] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. 2006. Compiled query execution engine using JVM. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 23–23.
[39] Kenneth A Ross. 2002. Conjunctive selection conditions in main memory. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 109–120.
[40] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2021. Language-agnostic integrated queries in a managed polyglot runtime. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1414–1426.
[41] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. 2016. RUMA has it: rewired user-space memory access is possible! *Proceedings of the VLDB Endowment* 9, 10 (2016), 768–779.
[42] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. ACM, 33–40.
[43] Wasmer, Inc. [n.d.]. *Wasmer*. https://wasmer.io