Towards a Future of Fully Self-Optimizing Query Engines

Paul Blockhaus[1], Gabriel Campero Durand[1], David Broneske[2] and Gunter Saake[1]

[1]*Otto-von-Guericke University, Magdeburg, Germany*

[2]*German Centre for Higher Education Research and Science Studies, Hannover, Germany*

Abstract

With the ever-increasing heterogeneity of hardware, the database community is tasked with adapting to the new reality of diverse ecosystems. The traditional workflow of hand-tuning query engine implementations to the underlying hardware might become untenable for an ever-growing variety of hardware with different performance characteristics. Systems like Micro-Adaptivity in Vectorwise or HAWK have been studied as adaptive solutions, but their adoption remains limited. Envisioning a solution simplified for adoption, we propose a practical take on adaptive reprogramming using the domain-specific language Voila and the MLIR compiler framework. We identify five main challenges in the area, and demonstrate how we tackle the first challenges. To show the feasibility of our approach, we include a brief evaluation of its performance on TPC-H; comparing 120 generated variants from a small subspace of potential optimizations.

Keywords

Adaptive Reprogramming, Micro-Adaptivity, Query Engines, Compiled Query Execution, MLIR, Voila

1. Introduction

Emerging trends in hardware development show a paradigm shift away from a single instruction set architecture (x86) towards a more heterogeneous ecosystem (e.g., RISC-V, ARM). While the complete consequences for the performance and design of database systems in this new era continue to be fleshed out [1], it is certain that engineers will continue to be tasked with adapting storage and query engines to new hardware. This trend has already been visible in the last few years with database systems adopting heterogeneous hardware [2] and finding benefits by exploiting new processing capabilities such as SIMD, non-volatile RAM, among others. This increased heterogeneity opens-up entire new sets of improvements for query engines. In this circumstance, the choice of a physical algorithm is likely to increasingly hinge less on the characteristics of queries, but rather on those of the underlying hardware. Broneske et al., among others, showed that with the increasing heterogeneity, not only the choice of algorithm and operator placement matters for query processing, but the implementation variant of the algorithms is crucial[3]. In numerous instances, promising variants are based only on so-called micro-optimizations commonly used by developers, as well as compilers, to optimize high-performance algorithms (e.g., branch predication, loop unrolling).

Over the last decade, research has cemented that these micro-optimizations have a considerable influence on query performance. Hence, several solutions have been proposed to identify and use the best combinations of micro-optimizations to reach peak-performance in hardware-sensitive algorithms over specific heterogeneous hardware [4, 5]. However, to the best of our knowledge, none of these approaches has led to a fully adaptive reprogramming system that can automatically generate, choose and maintain the best query execution plan and strategy, applying micro-optimizations to the query on the fly before executing it. Consequently, the goal of a flexible and highly adaptive approach to peak-performance and to relieve the burden for ongoing hardware-aware optimization and specialization, seems to arguably remain only partly realized. To help achieve this goal of adaptive reprogramming, we outline a vision for a fully adaptive, self-optimizing hardware-tuned query engine. We identify five main challenges that need to be addressed on the way and integrate the components into a single system. In order to have a sustainable platform for our vision, we choose to link upfront some solutions in compiler development. This enables us to directly benefit from the early adoption of new features into compilers and unifies the infrastructure for better maintenance and efficient development. To demonstrate a first step towards realizing our vision, we develop a prototypical open-source approach[1] that allows for adaptivity in the entire query execution through the use of a domain-specific language (DSL).

Our implementation is based on Voila [6], a DSL for query execution able to model various strategies. For our work, we implemented a new compilation backend for Voilas's execution. The backend is based on the MLIR framework, which enabled us to implement micro-optimizations of entire query plans at a per-operator level. To show the potential of our approach, in this work we

34th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), June 7-9, 2023, Hirsau, Germany

blockhau@ovgu.de (P. Blockhaus); campero@ovgu.de (G. C. Durand); broneske@dzhw.eu (D. Broneske); saake@ovgu.de (G. Saake)

[1]https://github.com/SuperFoo42/voila_mlir/tree/v0.1

empirically evaluate the performance of the DSL and its variants on an adapted subset of the TPC-H benchmark.

The remainder of the paper is structured as follows: First, we introduce the tools upon which our prototype is built, with an overview of previous work. Next, we introduce the five main challenges we identified for a fully adaptive query engine. We follow with an overview of the architecture of our prototype, that tackles the first two challenges. Finally, we present an early empirical evaluation.

2. Background & Related Work

In this section, we discuss the MLIR framework and the Voila DSL, which are the basis for our prototype; enabling efficient and adaptive generation of operator variants. Afterward, we give a brief overview of previous work.

2.1. MLIR

MLIR [7] is a compiler framework relying on LLVM [8] to quickly build DSLs. MLIR offers a generic intermediate representation providing a unified syntax and support for commonly needed functionalities of DSL compilers, such as type inference, lowering to executable code, interfaces for common code transformations, and many more. The two core components of MLIR are dialects and transformations, where dialects are different DSLs, tailored to specific use cases, e.g., the *scf*-dialect represents structured control flow elements, the *memref*-dialect represents memory references and access, and the *gpu*-dialect allows expressing instructions executed by GPUs, as well as the communication between CPU and GPU.

The structuring of MLIR in dialects and the possibility of mixing multiple dialects into this single representation allows for an efficient interoperation between dialects and simplifies their implementation. Other than dialects, transformations are the central component of MLIR. They convert between different dialects, while also supporting transformation patterns within a dialect, for example an optimization or canonicalization. As dialects are designed to represent problems in their domain efficiently, transformations can also be handled efficiently. This leads to better optimization times, and simpler rules, compared to general-purpose compilers such as LLVM. Additionally, transformations can be used in a plug-and-play fashion by conversion to the particular dialect. These concepts show, to the best of our knowledge, a great potential for a compiled query engine capable to utilize heterogeneous hardware and are unique for compiler frameworks.

2.2. Voila

Voila [6] is a DSL designed for efficient query execution, independent of the execution paradigm. To this end, the language is decoupled from the execution backend, which can flexibly run Voila programs in a vectorized or compiled fashion. To achieve this independence, Voila is designed with vector data types as first class citizens. The size of the vector determines the execution style, where a size of ∞ corresponds to operator-at-a-time execution and a size of 1 being a tuple-at-a-time execution. To have a language that allows this degree of flexibility, the granularity of the operations is set to execute on entire vectors, but the instructions are kept general purpose to be able to write different algorithms without a large re-implementation effort. The resulting granularity is somewhere between relational algebra operators, e.g., MAL [9], and virtual CPU instructions, e.g., LLVM IR.

2.3. Previous Approaches

In recent years, several approaches for adaptivity in database query execution have been proposed. One of the first approaches achieved micro-adaptivity in Vectorwise [4] through operator variants that were precompiled in libraries and could be replaced during runtime by dynamic library loading.

A new approach on micro-adaptivity was proposed with Excalibur. Excalibur is a virtual machine for adaptive query execution [10]. It utilizes Voila to generate queries that are executed in a vectorized fashion and replaces parts of the execution pipeline with compiled variants to search for an optimal execution plan variant.

The HAWK framework [5] used the C preprocessor and OpenCL to achieve adaptivity of the query algorithms on heterogeneous hardware. Looking beyond the variant generation, both solutions also considered the selection of the optimal variant from the variant pool, using machine learning. This was necessary as the spanned search space is extremely large, and the performance of each on a given hardware-query combination is inherently hard to predict. While the micro-adaptivity approach used an online multi-armed bandit to find the optimal variants, HAWK used offline-learning with a heuristic to test in the optimization space for the most efficient variants.

Micro-adaptivity and HAWK suffer from a rather restricted set of adaptivity due to pre-compiled variants using a template mechanism on top of a high-level language. In addition, HAWK doesn't provide any run-time adaptivity and has to be trained for a workload in advance. Excalibur is able to overcome the restrictions of high-level language templating, but still suffers from the problem of having to implement each variant transformation.

3. Challenges for Adaptive Query Execution Frameworks

Challenge 1 – Granularity of Adaptivity

Finding the right granularity at which adaptive reprogramming is introduced is a key factor for the efficiency in several aspects. The granularity determines the complexity of the system, as well as the resulting possibilities to reach peak-performance. A coarse granularity at the level of, e.g., pipeline programs will only allow for a global level of flexibility, such as optimizing compiler flags, and their influence will overall not lead to large benefits compared to the choice of algorithms and access paths used for the query. Though compiler flags could improve usage of hardware capabilities, the approach is still limited in optimizing queries individually with regard to the query characteristics.

In contrast, fine-grained adaptivity incurs additional overhead during variant generation, and high complexity on the system for optimization due to the large variant space. For example, introducing reprogramming into HyPer's operators [11] would lead to a huge variant space, as the operators can be written in LLVM IR, but it also adds the complexity of an entire compiler that has to be able to exploit this variability, entirely disregarding the complexity of implementing algorithms directly in LLVM IR. The use of mainstream compilers for this level of variability is rather limited, as current compilers only concentrate on generating the best code for the common case, or work with profiles to optimize for a pre-profiled use-case [12]. To our knowledge, there is no approach that instructs the compiler to transform LLVM IR or similar low-level intermediate languages to optimize the code at such fine granularity.

Therefore, current approaches choose the granularity of pipeline-operators for variability, as a compromise between complexity and variability. These operators are usually implemented in a high-level general-purpose programming language, which limits the degree of variability through its language constructs and mapping to hardware instructions. In the presence of heterogeneous systems, the language in which the operators are implemented has to support all target platforms, which either leads to using a hardware-oblivious language, e.g., OpenCL, or a hardware-sensitive alternative, mixing multiple languages. Both approaches come with their own disadvantages, they might not be performance portable, or add a lot of complexity through re-implementation of algorithms to optimize for hardware characteristics.

In order to overcome the above-mentioned problems of the individual approaches, Broneske et al. proposed the use of DSLs [3]. While this does not directly solve the problem of choosing the right granularity, it allows designing the DSL to fit an arbitrary granularity, or multiple levels of granularity with different language constructs, as required. Additionally, the DSL can be designed to efficiently support various types of hardware efficiently and even handle work distribution scenarios [13]. Unfortunately, to our knowledge, there are no established approaches that utilize a DSL towards adaptive reprogramming in data management.

Challenge 2 – Adaptivity Mechanism

Closely linked to the choice of granularity is the choice of the adaptivity mechanism. This refers to the component responsible for creating variants on the chosen granularity, and therefore also (partially) defining the variant space available. This component will depend on the granularity and query execution paradigm. Current approaches introduce adaptivity through an additional abstraction layer, producing the variants in a preprocessing step before compilation, i.e., using the C preprocessor for variant templates. These variants are then used either directly in a compiled execution engine [5] or loaded dynamically from libraries [4].

While this approach can be easily implemented, and is shown to be very effective, it has some drawbacks, such as limited adaptivity according to the expressiveness of the underlying language and specification of the preprocessor. Additionally, using a high-level language for adaptivity does not guarantee that the optimizations are actually applied because state-of-the-art compilers apply their own heuristics on the code that may undo the applied optimizations or optimize differently. We argue that a direct integration of the adaptivity mechanism into the compiler will help to overcome these problems, as the compilers already support many micro-optimizations by default; they just have to be applied. This eliminates redundancy in the design and also gives full control over the compilation process, which also allows restricting the optimizations or heuristics that the compiler uses, to not interfere with manually adapted code. Another advantage of this approach is that it combines nicely with our proposed solution for the first challenge, as modifications of the compiler to support a DSL can already require the design of a compilation pipeline. In our prototype, we are able to utilize the different granularities of different DSLs that MLIR offers to adaptively optimize the query in MLIR's own compilation pipeline.

Challenge 3 – Learning Variants

With the first two challenges, we described the problem of variability, but without a good variant selection mechanism, such variability might be essentially useless. The main problem for finding the best variants is the extensive search space that increases exponentially with every added optimization. The state-of-the-art approaches, we described in subsection 2.3, leverage machine learning techniques to learn and predict the best variants for a certain workload. The HAWK-framework uses an offline-learning approach that is fed with example queries and, in this way, learns the best flavors for the underlying hardware [5]. To reduce the learning time, additional heuristics are used to prune the exploration space. This approach focuses solely on optimization for a mostly static workload and best working parameters for optimal hardware utilization. Another approach used by micro-adaptivity is an online-learning approach that can also adapt to changing workloads [4]. This is achieved by formulating the variant selection as a multi-armed bandit learning problem. This continuously relearning system is also efficiently usable without previous learning.

We argue that an online-learning approach is, in general, preferable over offline learning, as it is designed to adapt to changes in the data management workloads by itself, whereas offline learning would need re-training over new representative workloads, which nowadays can change quickly and therefore is not easily predictable. A solution combining offline-learning to establish the starting models for online-learning might combine the best of both approaches.

We envision to not only use models to learn the best optimizations that can be used to instruct the compiler to apply certain optimizations on chosen places, but also to generate parameters for each optimization, and to guide the compiler regarding the number of optimization passes for applying the optimizations. Finally, we envision that the selected solution should integrate well with an MLOps framework, considering the lifecycle, evaluation, and maintenance of the models used.

Challenge 4 – Integration Into High-Level Optimization

To take a step from a pure query execution engine towards a more widely usable database engine, an adaptive reprogramming framework must be compatible with other high-level optimizations such as algebraic optimization, index and algorithm selection. To achieve this goal, one of two challenges has to be accomplished: Either the reprogramming framework needs to be integrated into an existing DBMS, or the missing components are integrated into the framework. Neither of these two solutions has been successfully applied yet. To the best of our knowledge, there only exist approaches that integrate parts of traditional optimizations into the adaptivity pipeline. The HAWK-framework, for instance, has limited support for hash table algorithm selection besides the operator-level variant granularity. Furthermore, Jungmair et al. showed a first example of how to integrate algebraic optimization successfully into MLIR, from which on further high-level optimizations can be introduced [14]. Even though both versions are possible, we plan to adapt the modeling of relational algebra in MLIR as it is a more flexible approach; with the added possibility of combining it with the learning mechanism mentioned before.

Challenge 5 – Adaptivity for OLTP

Current approaches in code generation for data processing focus heavily on OLAP queries, which is reasonable due to their complexity and high latencies. This focus neglects OLTP queries, which, we believe, might still provide room for improvement through code generation, with a focus on optimizing together sets of concurrent queries or templates of them. We envision, at this stage, that OLTP queries could be served in at least three main ways. First, recent work has shown promising results by tackling the challenge of learning an adaptive concurrency control mechanism for a mix of pre-defined query templates at different contention scenarios [15]. We consider that different mixes of these queries and the variable concurrency control mechanism might be able to leverage a code generation framework for a higher flexibility and adaptivity, in the search for high throughput. To achieve this, we consider that multi-query optimization might be necessary, opening the door for further kinds of improvements, such as operator sharing. Secondly, we consider that interpretation of MLIR for faster execution without compile overhead, and a switch between interpretation and compilation through coroutines could be relevant for workload mixes that could be categorized as HTAP. Finally, we argue that the flexibility of MLIR allows for the study of a new dialect focusing on robust processing over adjacent data versions (as used in MVCC or in Delta Lakes), which might be able to provide a competitive support for high-contention scenarios.

4. A Transformation-Based Approach for Adaptivity

In the following, we present our prototype that seeks to tackle, at this stage, the first two challenges described in the previous section. We start with the architecture of our approach (cf. Figure 1).

To have a more fine-grained granularity with a larger variant space while only adding a rather small overhead, we adapted Voila [6] as a DSL with sub-operator granularity (Challenge 1), and the additional feature of being agnostic to the query execution strategy. The design of Voila also allows for simple adoption to be efficiently executed on heterogeneous processors, as similar designs already demonstrated [16]. Instead of explicit predication, we rely on a separate compilation pass to automatically choose when to use predication and when to materialize selections, which allows for more flexibility in the

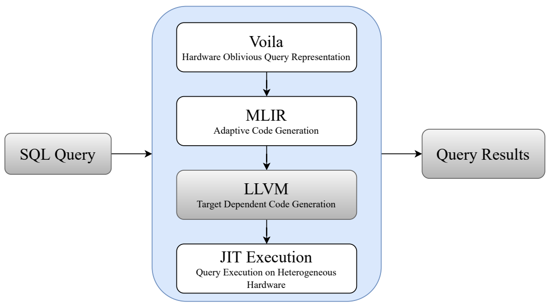

Voila
Hardware Oblivious Query Representation
MLIR
Adaptive Code Generation
LLVM
Target Dependent Code Generation
JIT Execution
Query Execution on Heterogeneous Hardware
SQL Query
Query Results

Figure 1: High-level architectural schema of the framework

optimization while also simplifying the language.

Implementing those types of adaptivity directly into the DSL has the great advantage that it makes the DSL adaptive by design (Challenge 2). In contrast to other approaches, there is no longer the need to provide an additional callback to pre-compiled algorithms or to change the query plan. Instead, the query engine directly calls the generated query code, which is adapted during JIT-compilation by the supplied transformation and optimization passes.

The DSL itself is built upon the MLIR and LLVM frameworks, which enable simple definitions of micro-optimizations as transformations. Some commonly used transformations, such as loop unrolling, vectorization, parallelization, etc., are already implemented and can be used. Transformations in MLIR are applied in passes, and additional filters even allow applying transformations selectively on certain instructions, or supplying certain parameters to the passes. In addition to the pre-existing passes, we implemented a selection dematerialization pass to compensate for the missing predication by replacing the materialized result of the selection with a predicate, indicating the selected tuples, when possible. The dematerialization pass uses a forward data flow analysis of selection results to decide whether to materialize the result or keep the predication. Currently, we materialize the results any time, where pipeline breakers are encountered, but the pass can be extended with further options and more complex scenarios on when to replace materialization with predication and vice versa.

As MLIR is built upon LLVM, it also comes with native support for heterogeneous hardware. Hence, we are confident that our DSL is able to adjust its behavior to fit processing paradigms on heterogeneous hardware with only minimal extensions. However, in the following w.l.o.g., we focus on the extensible CPU-only part. Therefore, we use vectorization to utilize SIMD capabilities of modern CPUs, as well as multithreading with built-in LLVM coroutines, or the OpenMP runtime.

In order to translate Voila to executable machine code, we added a backend to Voila that translates Voila to a Voila-MLIR dialect. From there on, we use the MLIR framework and its dialects to introduce adaptivity into the code and lower it to LLVM, which is then JIT-compiled and executed.

The usual compilation pipeline of our framework looks as follows, once Voila has been translated to Voila-MLIR, which includes type resolution and transformation of columns to the tensor data type, aggressive function inlining is performed to achieve redundant sub-query removal through canonicalization and common sub-expression elimination, as well as allowing for better optimizations by eliminating function call boundaries.

Afterward, the dematerialization pass is performed and the resulting plan is lowered to MLIR's internal dialects, either `linalg`, or `affine`. The `linalg` dialect describes loops using the concept of iterators. Each `linalg` operation consists of a set of input tensors and generates a set of output tensors. The body of `linalg` operations has to be largely side effect free, and aside of regular reductions doesn't allow for data or control flow dependencies between the iterations. Therefore, we resort to the `affine` dialect for operations that need broader constraints. The `affine` dialect models range-based for loops and allows direct memory manipulations, only constrained by the loop induction variables.

Once, Voila-MLIR is translated to these high-level internal dialects of MLIR, we apply loop optimization passes such as parallelization, tiling, unrolling and loop fusion. Afterward, we continue lowering towards more lower-level dialects such as the `vector` dialect that represents virtual, hardware-independent vectors and instructions that can directly be mapped to the hardware's SIMD capabilities. Other translations in this phase are lowering of parallel loops to the `async` dialect, which represents multi-threading through LLVM coroutines, or to the `openmp` dialect. In this step, it is also possible to transform parallel loops for execution on further coprocessors, such as GPU, e.g. by using the `gpu` dialect. This dialect models the GPU execution model and also handles communication between CPU and GPU. For a concrete execution on GPU, the dialect can further be lowered to `SPIR-V` or AMD and NVIDIA specific dialects.

In the next step, the low-level MLIR is lowered to LLVM and finally compiled to executable byte code and then the query can be executed.

5. Evaluation

5.1. Dataset & Modifications

To get an overview of the real-world behavior of our framework, we use a subset of the TPC-H benchmark. More precisely, we choose the queries Q1, Q3, Q6, Q9 and Q18 as a representative sample for typical OLAP queries [17]. Since our framework does not yet support

Table 1
Benchmark machine specifications

	Processor	RAM	SIMD
Machine 1	Intel Xeon E-2286M@2.4GHz	128GB	AVX2
Machine 2	Intel Xeon Gold 6130@2.1GHz	346GB	AVX-512
Machine 3	Intel Xeon E5-2630 v3@2.4GHz	1008GB	AVX2

joins efficiently and has no support for string data types, we modify the TPC-H dataset slightly to still be able to run the queries.

We use order-preserving dictionary compression for string columns and denormalize tables that need to be joined in order to replace joins by selections that are simpler to implement efficiently. We argue that the dictionary compression has no significant impact on the results of our benchmarks, as strings in TPC-H are mostly only used as payloads [18] and since we are already using column-oriented query processing, the strings would commonly not be loaded at all for most of the time. The only exception here is Q9, where a string pattern matching takes place, which has to be translated in the case of dictionary compression. We choose to work around the matching by using a large IN-clause, checking for existence of each tuple using a hash table. For the replacement of joins with simple selections on denormalized tables, Li and Patel suggest that it has no large influence on the query performance and leads to comparable results [19].

5.2. Setup

In order to get a first overview of the performance characteristics of our experiments on different platforms, we use three different machines listed in Table 1. All three machines use Intel processors, but from different generations and with slightly different architectures and clock-rates, which should already show varying results.

To keep performance differences limited to the hardware, we created a unified build tool chain with all machines using the same bootstrapped LLVM version 13.0.1 to build the experiments. Additionally, we supplied the following compiler flags for optimization: `-O3 -flto -march=native` . Where `-O3` instructs the compiler to apply more aggressive, but also expensive (in terms of compile-time) optimization of the code, `-flto` enables link-time optimizations, such as de-virtualization of functions and `-march=native` tells the compiler to enable instructions that are available for the architecture on which the code is compiled. Such instructions are usually SSE4 and AVX, among others. While more advanced instructions increase the overall latency of our framework to optimize and run code, this also makes the binary largely not portable to other architectures. Overall, the optimizations only have an effect on the framework compile-time of the query, but not on the query execution time, as these flags are not applied to the query compiler. To get more robust timing results, we disabled software-based frequency scaling and ran each benchmark 100 times. The reported times were averaged using the median.

5.3. Adaptivity Behavior

In order to get an overview of the effect of our optimizations on the runtime of the queries, we only generate a subset of the variant space that we are able to generate. To restrict the number of variants to a maintainable size, we only create all configurations of the following five major optimizations and configurations:

loop unrolling: copying the loop body multiple times within the loop to reduce control-flow overhead of the loop condition check

loop tiling: splitting the loop body into an inner and outer loop, effectively creating a blocking of memory access to increase cache performance

selection dematerialization: replacing materializations of intermediate selection results by bitvectors indicating selected tuples to reduce memory usage and increase vectorization capabilities

parallelization: splitting independent loop iterations to multiple cores using LLVM coroutines or OpenMP

vectorization: transform instructions in loop bodies to SIMD counterparts for data-parallel execution

In addition, we used some minor, permanently enabled optimizations such as CSE, inlining and buffer optimization passes to obtain both better optimizable and optimized code.

Furthermore, we restricted the configuration of these optimizations to a per-program level instead of generating variants for each operation. With these prerequisites, we generated 120 variants for each machine and show the results of the pure execution times without compilation as heatmaps in Figure 2. To keep the time for running the configurations acceptable, we used the TPC-H dataset with scale factor 1.

Each column of a heatmap represents a different vector-size, while the rows describe combinations of unrolling, tiling and selection dematerialization. The different parallelization techniques are divided by Figure 2a, Figure 2b and Figure 2c.

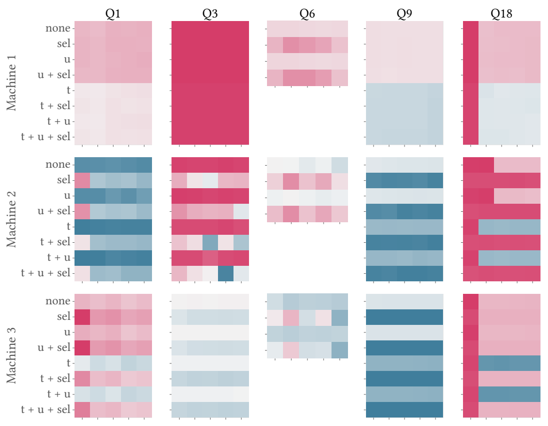
Due to an outdated OpenMP runtime on Machine 3, there are no heatmaps for OpenMP parallelization in this machine. Furthermore, combining vectorization together with tiling for Q6 was problematic. The main reason was that the generated code could not be compiled because parallel loops only allow reductions on scalar types through atomic read-modify-write operations. However, as the loop body was vectorized, the reduction would have to be done on vectors, which is not supported. Due to this limitation in MLIR, we decided to exclude variant results with tiling altogether for Q6. The key observations from this experiment are:

1. All machines show similar trends regarding variants, but different configurations turn out to be most efficient (e.g., unrolling vs. no unrolling for Q6 or OpenMP vs. async).
2. Tiling has a mostly negative influence on the query performance, with a speed-up ×1 to ×0.1 compared to no optimization.
3. For Q1 and Q6, selection dematerialization improves the performance by a factor of up to ×10, whereas Q3 and Q9 showed decreased performance by as low as ×0.25.
4. Loop unrolling had only a moderate influence when combined with vectorization for larger vector sizes.
5. Vectorization had slightly positive effects on the runtime of Q6 (×2 speed-up), for the other queries it had slightly negative effects.

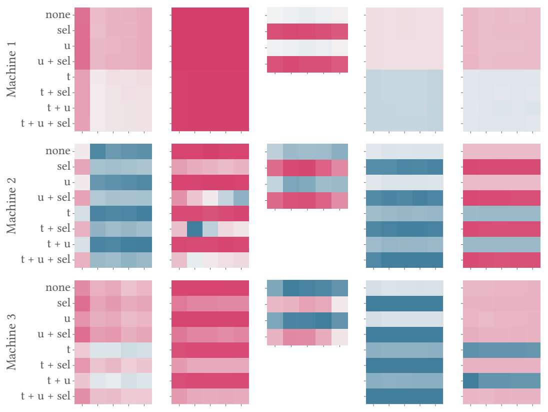
Overall, these experiments show that our approach is able to generate differently performing variants that can be used to optimize the runtime of different queries depending on the executing hardware and software characteristics – even with only a small part of the possible variant space explored. On the other hand, we also observe some unexpected behavior, such as the partially low influence of vectorization and parallelization on the runtime, especially visible for Q1, Q9, and Q18. After inspection of the generated MLIR code, we saw that these queries are often not transformed to parallel loops because the transformation pass of MLIR is too restrictive in its side effect analysis and does not parallelize/vectorize loop nests with non-affine branches instead of traversing and analyzing their memory accesses recursively.

6. Conclusion & Future Work

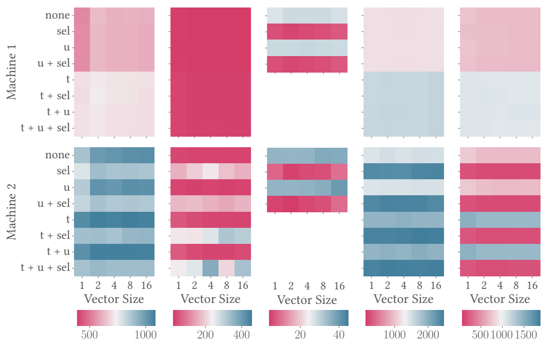
In this paper, we presented our vision of the future of fully adaptive database query execution engines and showed that support of micro-optimizations in database query engines can have a performance impact in the order of magnitudes. At first, we identified the main challenges to tackle in order to achieve a fully adaptive database engine that, despite the clear advantage of such systems, still does not exist today. We find that there are approaches that tackle each of the challenges individually, but up to now there exists no framework that takes everything into



(a) Variants performance for all machines without parallelization



(b) Variants performance for all machines with coroutines



(c) Variants performance for Machines 1 and 2 With OpenMP

Figure 2: Heatmaps showing the runtimes of the query variants with optimizations: tiling (t), unrolling (u), selection dematerialization (sel)

account to achieve an entirely adaptive solution. Therefore, we propose a simple and practical approach that leverages the Voila DSL and the MLIR compiler framework to introduce adaptivity into the query compilation pipeline and, thus, we tackle the challenges of adaptivity and granularity of the optimizations. Moreover, we highlight further pathways to tackle the remaining challenges in the context of our approach.

We briefly evaluated the functionality and performance using a modified version of the TPC-H benchmark. We showed that our approach is able to generate diverse variants with different performance properties. While we still found some issues and the prototype is at an early stage, we are confident that when addressing the issues, our approach will be able to outperform non-adaptive systems.

Our immediate next step is the addition of support for heterogeneous hardware, as well as improvement of the prototype to be more competitive by overcoming the identified problems. Additionally, we want to extend our approach to support joins efficiently, as well as string types. For the future, we plan to extend our prototype to tackle all of our presented challenges, to become a fully adaptive reprogramming, heterogeneous database system that is able to achieve peak performance independent of the supplied workload and hardware.

References

[1] T. Gubner, P. Boncz, Highlighting the performance diversity of analytical queries using voila, in: Proc. ADMS 2021, 2021.

[2] K. Dursun, C. Binnig, U. Cetintemel, R. Petrocelli, SiliconDB: Rethinking DBMSs for Modern Heterogeneous Co-Processor Environments, in: Proc. Damon 2017, ACM, 2017.

[3] D. Broneske, Adaptive Reprogramming for Databases on Heterogeneous Processors, SIGMOD 2015 PhD Symposium, ACM, 2015, p. 51–55.

[4] B. Răducanu, P. Boncz, M. Zukowski, Micro Adaptivity in Vectorwise, in: Proc. SIGMOD 2013, ACM, 2013, p. 1231–1242.

[5] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, V. Markl, Generating custom code for efficient query execution on heterogeneous processors, VLDB Journal 27 (2018) 797–822.

[6] T. Gubner, P. Boncz, Charting the Design Space of Query Execution Using VOILA, Proc. VLDB Endow. 14 (2021) 1067–1079.

[7] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, O. Zinenko, MLIR: Scaling Compiler Infrastructure for Domain Specific Computation, in: Proc. CGO 2021, IEEE/ACM, 2021, p. 2–14.

[8] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: Proc. CGO 2004, IEEE, 2004, p. 75–86.

[9] P. A. Boncz, et al., Monet: A next-generation DBMS kernel for query-intensive applications, 2002.

[10] T. Gubner, P. Boncz, Excalibur: A virtual machine for adaptive fine-grained jit-compiled query execution based on voila, Proc. VLDB Endow. 16 (2022).

[11] T. Neumann, Efficiently Compiling Efficient Query Plans for Modern Hardware., Proc. VLDB Endow. 4 (2011) 539–550.

[12] Y. Srikant, P. Shankar, The Compiler Design Handbook: Optimizations and Machine Code Generation, CRC Press, 2003.

[13] C. Bertoni, J. Kwack, T. Applencourt, Y. Ghadar, B. Homerding, C. Knight, B. Videau, H. Zheng, V. Morozov, S. Parker, Performance portability evaluation of opencl benchmarks across intel and nvidia platforms, in: Proc. IPDPSW 2020, 2020, pp. 330–339.

[14] M. Jungmair, A. Kohn, J. Giceva, Designing an open framework for query optimization and compilation, Proc. VLDB Endow. 15 (2022) 2389–2401.

[15] J.-C. Wang, D. Ding, H. Wang, C. Christensen, Z. Wang, H. Chen, J. Li, Polyjuice: High-performance transactions via learned concurrency control., in: OSDI, 2021, pp. 198–216.

[16] H. Pirk, O. R. Moll, M. Zaharia, S. Madden, Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware, Proc. VLDB Endowment 9 (2016) 1707–1718.

[17] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, P. A. Boncz, Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask, Proc. VLDB Endow. 11 (2018) 2209–2222.

[18] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, M. Then, Get Real: How Benchmarks Fail to Represent the Real World, in: Proc. DBTest 2018, 2018, p. 1–6.

[19] Y. Li, J. M. Patel, WideTable: An Accelerator for Analytical Data Processing, Proc. VLDB Endow. 7 (2014) 907–918.