Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA

Tim Gubner
CWI
tim.gubner@cwi.nl

Peter Boncz
CWI
boncz@cwi.nl

ABSTRACT

In recent years, hardware has become increasingly diverse, in terms of features as well as performance. This poses a problem for complex software in general and database systems in particular. To achieve top-notch performance, we need to exploit hardware features, but do not fully know which behave best on the current, and more-so future, machines. Specializing query execution methods for many diverse hardware platforms will significantly increase database software complexity and also poses a physical query optimization problem that cannot be solved robustly with static cost models.

In this paper, we propose a new query execution architecture addressing these problems. Based on the flexible domain-specific language VOILA, it can generate thousands of different flavors from a single code-base. As an abstraction, a virtual machine (VM) allows hiding physical execution details, such that the VM can transparently switch between different execution tactics within each query, applied at a fine granularity. We show rules to describe a search space for good tactics, and describe efficient search strategies, that limit the overhead of adaptive JIT code generation and compilation. The VM starts executing each query in full vectorized code style, but adaptively replaces (parts of) query pipelines by code fragments compiled using different execution flavors, exploring this search space and exploiting the best tactics found, casting adaptive query execution into a Multi-Armed Bandit (MAB) problem. Excalibur, our prototype, outperforms open-source systems by up to 28× and the state-of-the-art system Umbra by up to 1.8×. In specific queries Excalibur performs up to 2× faster than static flavors.

PVLDB Reference Format:
Tim Gubner and Peter Boncz. Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA. PVLDB, 16(4): 829 - 841, 2022.
doi:10.14778/3574245.3574266

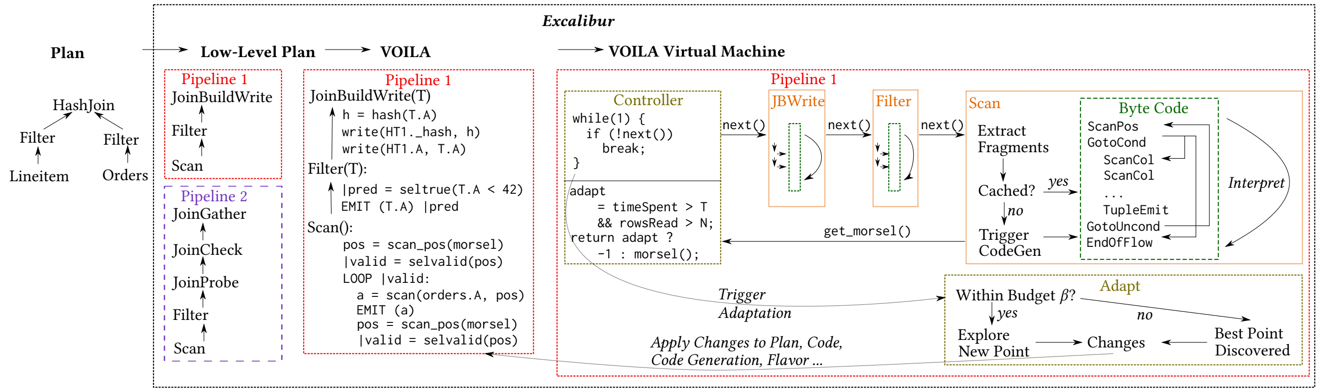
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/t1mm3/db_excalibur.

1 INTRODUCTION

Analytical query performance is an important driver of growth in data science applications, yet significant software challenges lie immediately ahead. One trend is in stalling CPU improvement, as the limits of Moore's law and Dennard scaling are causing classical performance boosting techniques, such as increased clock-speed, increased core-counts and wider-issue CPUs, to dry up. In response, hardware is rapidly diversifying, with computer architects seeking improved performance through specialization, typically targeting high-growth applications such as gaming, video processing and deep learning, but not necessarily SQL systems. The trend towards a heterogeneous hardware landscape is hastened by a computing market split between consumer hardware, now focused on mobile devices, versus very large cloud companies now creating their own specialized server hardware, whereas the standardized on-premise server market is quickly diminishing in importance. Concretely, we observe that the decades-old hegemony of X86 CPUs is challenged by a variety of new options using ARM variants or RISC-V; in conjunction with specific hardware extensions regarding persistent memory, SIMD, encryption, compression and network communication. For database software, this hardware fragmentation poses optimization and maintainability challenges. We have shown that on different hardware, different styles of analytical query engine design prevail [21]; so the question is how one can create future-proof database systems? Database systems are of the most complex software systems and take many person-years to develop, and trying to optimize for this diversifying hardware landscape would cause a significant growth in code size, that in the long run will turn into technical debt and a robustness and maintenance liability.

In response to this database systems software challenge, we proposed VOILA [20], that introduces a domain-specific language (DSL) to encode relational operators, and from there we can generate thousands of different *flavors* of execution styles. The gist of its DSL is that it logically encodes the operations to be performed on data as well as the layout of the data structures, yet it does not specify the order or execution-style of these operations, and instead tries to create independence of these operations; thus providing freedom to execute them in different ways. The challenge of determining the best execution flavor for a query plan, out of the thousands of possibilities, was left as future work in [20], and is the topic of this paper. The classic approach would have been to construct detailed physical cost models for VOILA, and solve the problem of finding the best flavor statically, before query execution starts, as a physical query optimization problem. However, such physical cost models would need to be created for every hardware type that exists, as we found that flavor performance varies strongly among different hardware [21], and would never be future-proof. Instead, we pursue a *micro-adaptive* approach [42]. A crucial property we exploit is that all flavors generated from a VOILA program operate on the same data structures, which allows to switch flavors *in-flight* and at *fine granularity*: switching flavors can be done for the whole query, or only for one pipeline, or just a fragment of a pipeline.

Proceedings of the VLDB Endowment, Vol. 16, No. 4 ISSN 2150-8097.
doi:10.14778/3574245.3574266



```
JoinBuildWrite(T)
    h = hash(T.A)
    write(HT1._hash, h)
    write(HT1.A, T.A)
Filter(T):
    |pred = seltrue(T.A < 42)
    EMIT (T.A) |pred
Scan():
    pos = scan_pos(morsel)
    |valid = selvalid(pos)
    LOOP |valid:
      a = scan(orders.A, pos)
      EMIT (a)
      pos = scan_pos(morsel)
      |valid = selvalid(pos)
```

```
while(1) {
  if (!next())
    break;
}
```

```
adapt
    = timeSpent > T
    && rowsRead > N;
return adapt ?
    -1 : morsel();
```

```
ScanPos
GotoCond
  ScanCol
  ScanCol
  ...
  TupleEmit
GotoUncond
EndOfFlow
```

Figure 1: Excalibur Architecture. Excalibur uses multiple layers to generate code and allows adaptive re-optimization based on runtime feedback. Instead of full compilation and re-compilation, Excalibur allows reusing already compiled fragments.

In this paper we describe Excalibur, an adaptive query execution engine, that at its core runs a VOILA Virtual Machine (VM) and automatically generates flavors and exploits those that run fastest, given observed performance (an interaction of query and hardware, data distributions and concurrent workload).

Contributions. A novel architecture for analytic engines that:

(a) uses a high-level DSL (VOILA [20]) to flexibly combine vectorized, data-centric and other execution *flavors* in a single code-base. Importantly, VOILA flavors use the same state and data-structures, so execution can seamlessly switch flavors even in the middle of executing a query.

(b) integrates *adaptivity* into a JIT-compiled system, using a Virtual Machine (VM) architecture that can execute a query *combining* different execution flavors, generating JIT-compiled fragments on the granularity of an execution pipeline or even parts of a pipeline, and explores and exploits these combinations (*tactics*) on-the-fly.

(c) addresses the problem of finding good execution tactics, by defining a search space for tactics using Mutation Rules, and sparsely exploring this space with efficient Search Strategies (e.g. a heuristic-based strategy or MCTS) that limit the amount of code fragments that need to be compiled and tested, casting this into a Multi-Armed Bandit (MAB) problem [32].

Structure. The following section briefly introduces preliminary concepts. In Section 3 we explain query execution in Excalibur, and in Section 4 describe two basic code generation flavors (vectorized [12] and data-centric [40]) that Excalibur can mix into execution tactics. Section 5 explains micro-adaptive execution in Excalibur and in Section 6, we present multiple stretagies for exploring the large design space, followed by an experimental evaluation in Section 7. Finally, we discuss related work and conclusions.

2 BACKGROUND

This section introduces preliminary concepts, starting from the domain-specific language VOILA, the Multi-Armed Bandit problem and the Upper Confidence Bound algorithm.

VOILA. Excalibur builds on top of the domain-specific language VOILA [20]. VOILA allows describing operators in a way that exposes data-parallelism and, thus, implicitly allows synthesizing SIMDized code or vectorized execution. Using VOILA, different back-ends can generate *very* different code styles and query execution paradigms [20]. We have shown that the VOILA synthesis framework covers a large design space [20].

Multi-Armed Bandits (MAB). In practice, we often have choices but we do not know which one is best. Instead we want to find the best choice at runtime (online learning). We can either explore (try new or re-try old choices) or exploit (use the best choice found, so far). This is commonly abstracted using the MAB problem. The MAB problem can be imagined as a row of slot machines and we want to maximize our possible reward by using the machine most favorable to us. To achieve this, we need to observe the distributions of all slot machines (exploration). Once we are confident about the distributions, we can pull the lever on the most favorable slot machine (exploitation) and pocket the rewards. The goal is to solve this problem with a low, hopefully sub-linear, regret (loss compared to best possible choice).

Upper Confidence Bound (UCB). One algorithm that solves MAB optimally is the Upper Confidence Bound algorithm (UCB) [6]. UCB tends to be an elegant and effective solution to the MAB problem with an attractive sub-linear regret. For each arm i, we define a score $ucb_i(T)$ and we always choose the arm with the highest score i.e. $argmax_i\ ucb_i(T)$, at a time-step T (number of calls to algorithm), For an arm i, let N_i be the number of samples collected so far, X_i the empirical mean of rewards, and a independent constant c. The score is defined as:

$$ucb_i(T) = \begin{cases} \infty & \text{if } N_i = 0 \\ X_i + c * \sqrt{\frac{log(T)}{N_i}} & \text{otherwise} \end{cases} \tag{1}$$

3 EXCALIBUR

Excalibur is a system prototype[1] intending to make query execution flexible & dynamic. It allows trying and exploiting many different execution styles (flavors), while executing the query. **We now walk through its architecture in Figure 1.**

To execute a query, its query plan is handed over to Excalibur, along with readers that allow scanning the base tables involved in the query. From there on, Excalibur translates the plan into its own plan representation (Low-Level Plan). In the Low-Level Plan

[1] Source code and scripts can be found under https://github.com/t1mm3/db_excalibur

representation, the query is split into pipelines (Pipeline 1 and Pipeline 2) with simple operator chains inside each pipeline. These operator chains can be and are pipelined to minimize the size of intermediates. Afterwards, we expand plan operators into code in the domain-specific language VOILA (shown for Pipeline 1). The VOILA program is used to generate byte code which can efficiently be interpreted. This step also involves generating the required code fragments that are invoked by the byte code. These fragments could already be cached and then do not require compilation. After the code generation has finished, the pipeline is evaluated by interpreting each operator in a (vectorized) iterator-based fashion, i.e. calling a `next()` method that returns batches of tuples (typically 1024) produced by the operator. Inside the `next()` method, the byte code interpreter calls compiled code fragments for each byte code.

Instead of fully evaluating the pipeline, we can interrupt execution after a certain number of tuples or CPU cycles. This is handled by the Controller, which triggers the evaluation of the top-most operator (in the chain) and suspends evaluation by choking the scan (`get_morsel()` returning 0 tuples). This interrupt allows making changes to the current execution flavor ("Trigger Adaptation" in Figure 1). Whether Excalibur can explore new points in the design space (flavors) or rather exploit the already explored points is decided through a Budget β. If there is enough budget, new flavors are explored, otherwise the best discovered flavor will be run. After Trigger Adaptation, any Changes are applied to the VOILA byte code and execution of the pipeline is resumed. For example, to switch to executing a query in data-centric flavor, we inline all operators into the top-most operator, compile this into one code fragment and reconfigure the byte code of the top operator, de-activating the other operators.

3.1 Execution Model

Excalibur uses two levels of relational operators: (a) high-level operators like `HashJoin` and (b) low-level operators that encode which operations a high-level operator , e.g. `HashJoin`, must perform.

High-Level operators are rather a logical construct than a part of physical query execution. They own the state shared by the low-level operators (most notably data structures) and provide high-level features such as progress estimation (needed later).

Low-level operators specify the physical implementation of a corresponding high-level operator. Each low-level operator uses the vectorized Volcano model i.e. it is an iterator with a `next()` method that returns multiple tuples stored as an array of columnar vectors. While our low-level operators share some similarity with LOLEPOPs [37], they are different from DB2 BLU [43] or Starburst [23]. Specifically, we further decompose the join into sub-operators.

Instead of a monolithic `HashJoin` operator, we use a sequence of `JoinProbe`, `JoinCheck` and `JoinGather`, of course after building the hash table using `JoinBuild`. Consequently, our joins can be easily extended in the future, e.g. `JoinProbe` can be replaced by using a perfect hash [7, 11, 19]. Low-level operators are the physical unit of query execution. An operator can be white-box (expressed in our domain-specific language VOILA) or black-box, which allows integrating operators for which no representation in VOILA exists (e.g. the `Output` operator that materializes the query result). White-box operators expose VOILA code and, therefore, qualify for compilation and interpretation-compilation hybrids.

Table 1: Byte Code instructions. Certain instructions are not strictly necessary but exist for performance-purpose, these instructions are marked with an asterisk (*).

Byte Code Instruction	Description
`GotoCond`	If condition == constant: Goto "line"
`GotoUncond`	Goto "line"
`EndOfFlow`	Signal end of stream
`End`	End of program
`Copy`	Copy value/vector
`Emit`	Returns tuples from operator
`ScanPos`	Allocates a position for reading a table
`ScanCol`	Reads a column chunk from `ScanPos`
`SelNum`	Turns position inside table into predicate
`WritePos`	Allocate a position for writing a table
`CompiledFragment`	Call compiled VOILA fragment
`BucketInsert`*	*(Complex) VOILA operation*
`SelUnion`*	*(Complex) VOILA operation*

Almost all relational operators in Excalibur are implemented using VOILA (white-box). Notable black-box exceptions are `Output`, which produces the query result, and `JoinBuild`, that builds the hash table of a join, after the inner relation has been materialized (resembling the Morsel-driven parallel hash join with a shared hash table [35]). Excaliburis a interpreter (VM) that evaluates VOILA plans, while being able to leverage VOILA's flexibility.

3.2 Interpretation

Excalibur executes query plans as block-based pull iterators (i.e. vectorized execution [12]), exploiting the fact that VOILA programs can always be executed as vectorized primitives, which provides low-latency efficient interpreted execution as a starting point.

Vectorized Byte Code. The VOILA code is translated into an easily and efficiently interpretable representation (byte code). Our byte code encodes auxiliary operations required to execute (vectorized) VOILA code, while keeping VOILA code mostly encapsulated in fragments. VOILA fragments are invoked via `CompiledFragment`. The supported instructions are shown in Table 1.

Generating Byte Code. While generating the byte code, we check for fragments that need to be compiled. This could be atomic operations (e.g. adding two columns) or complex fragments (e.g. gathering a multi-column join probe result). For each, we generate a corresponding `CompiledFragment` instruction and trigger compilation.

3.3 Compilation into Vectorized Primitives

VOILA fragments are compiled into machine code with LLVM [33], a widely used framework for building compilers. Especially for short-running queries, compilation is quite costly ($10 - 100$ ms).

Caching. Fortunately, compilation can often be omitted be caching frequently used fragments. Especially for simple code fragments (e.g. consisting of 1-2 VOILA operations) this method is quite effective, as small fragments can often be re-used. Re-use can happen inside the same pipeline, query, or across queries. Essentially, this caching mechanism approximates vectorized execution (very simple cached fragment with only one operation = vectorized primitive) while still allowing complex custom-tailored fragments.

Parallel Compilation. Besides reducing compilation time (thanks to caching), code fragments also provide a means to parallelize

Listing 1: Example JIT-ed vectorized primitive computing $-x$ and $x * y$. Only the selective scalar path is required, the other paths can be omitted to decrease compilation time.

```
void jit_1(PrimArg* arg) {
  int i=0;

  // Deserialize inputs and outputs
  int* sel = arg->sources[0]->first;
  int num = arg->sources[0]->num;
  long* in_val1 = arg->sources[1]->first;
  long* in_val2 = arg->sources[2]->first;
  long* out_val1 = arg->sinks[0]->first;
  long* out_val2 = arg->sinks[1]->first;

  if (ignore_selvector(sel, num, true, 2*64, 2)) { // optional
    // Optional unrolling
    for (; i+16<num; i+=16) { /* ... */ }

    for (; i<num; i++) {
      out_val1[i] = -in_val1[i];
      out_val2[i] = in_val1[i]*in_val2[i];
    }
  } else { // Use selection vector, mandatory
    // Optional unrolling
    for (; i+16<num; i+=16) { /* ... */ }

    for (; i<num; i++) {
      out_val1[sel[i]] = -in_val1[sel[i]];
      out_val2[sel[i]] = in_val1[sel[i]]*in_val2[sel[i]];
} } }
```

Listing 2: Decision to ignore selection vector. Ignore selection vector for dense predicates without filtered out tuples, average bits per VOILA node are above a certain limit.

```
bool ignore_selvector(int* sel, int& num, bool can_full_eval,
    double sum_bits, double num_nodes) {
  if (!num && !can_full_eval) return false;

  double score = sum_bits / num_nodes / SCORE_DIVISOR;
  double min_size = (scope * VECTOR_SIZE) / (score + 1.0);
  return num > min_size;
}
```

compilation, even inside a single pipeline. Code fragments are independent pieces of VOILA code that are glued together by the surrounding byte code. Therefore, code fragments can be compiled independently of each other which allows parallelizing compilation.

Compiling Vectorized Primitives. We generate vectorized primitives, functions that operate on columnar chunks of data. Note that data-centric compilation fits into the vectorized model (e.g. Hyper uses morsels [35], table chunks like vectors, delivered by its vectorized scan operator that decompresses DataBlocks [31]).

The basic function template is illustrated in Listing 1. It iterates over the input predicate (selection vector, which in our system always exists and then evaluate the VOILA code value-at-a-time. Furthermore, this generic template allows interesting variations:

- We can choose to ignore the predicate. This, however, is not always possible (e.g. for example operations that can raise an error), but can lead to better SIMD performance [19]. Choosing only requires a quick density check on the selection vector, like illustrated in Listing 2.
- Important code paths can be unrolled. This means splitting the loop into the unrolled loop that processes N values (e.g. 16 using SIMD) at once and a residual loop that processes the remainder.
- Code can be annotated to enable/disable SIMDization of the code, or define different SIMD widths (e.g. triggering AVX2 instead of AVX-512 to prevent down-clocking on some processors [2, 30]).

Later, we expand this template for specific flavors such as vectorized and data-centric execution. Note that the performance of these variations is hard to predict. Therefore, Excalibur will chose the best one dynamically at query runtime.

3.4 Code Cache

The idea is to *fingerprint* code fragments and look the fingerprint up in the cache. For this cache, however, lookup performance under updates is crucial. Therefore, we use an asynchronous eviction process that does not require write latches during lookups.

Asynchronous Eviction. Instead of replacing during lookups, we have an asynchronous processes that cleans up excess fragments in the cache. Therefore, during lookups we just need to update a reference counter and a last-updated timestamp, using atomics. This only requires a shared latch to prevent concurrent updates.

Eviction. Periodically cleanup is triggered. We mark the N least recently used fragments evictable. When eviction is triggered again and if they have not been touched in between, we safely evict them.

Adapting N. Typically, we aim for a constant cache size ($\leq T$ fragments) with a margin for new fragments (say 10%). Let F be the current number of fragments in the cache. To stay within the bounds, we have to evict $T - F$ fragments. However, we cannot guarantee that our eviction process will clean up $N = T - F$ fragments, because they might have been updated/used in between (there is a time lag). Therefore, we measure the number of fragments, we were able to evict, calculate the eviction rate (out of X, we evicted Y) and over-allocate the number of eviction candidates by the corresponding factor ($\frac{X}{Y}$) during the next iteration.

Footprint per Fragment. Ideally all code fragments are cached and do not incur JIT-overhead. Practically, however, this is not feasible. The important question is, how many fragments can realistically be cached i.e. what is their memory footprint.

Each code fragment can be compiled in parallel, thus LLVM requires each fragment to use its own instances of `LLVMContext` and `TargetMachine`, an abstraction for hardware-specific details. This led to memory footprint of roughly 400 kB per fragment, while, for simple fragments, the machine code fits in roughly 1 kB. The extra footprint stems from LLVM which is only needed during compilation. Therefore, after compilation is done, we can safely deallocate LLVM-related objects. This, however, is a non-intended LLVM use-case and requires providing a custom memory manager, which it uses to store compiled machine code. After compilation is done, we dispose the allocated LLVM compilation utilities and just keep the machine code, which is now owned by our memory manager. This pushes the footprint of a code fragment to around 10 kB (40× smaller than the naive implementation). This currently allows roughly 100.000 fragments in 1 GB code cache.

The footprint of cached code fragments could be improved further by sharing pages between multiple fragments. Sharing pages, however, is non-trivial to do in a portable manner, because one needs to allocate physical pages, modify page flags (make writable, remove writable flag and make executable), handle concurrency (fragments compiled in parallel) and, of course, find a memory layout that satisfies the requirements of the CPU.

4 CODE GENERATION FLAVORS

Excalibur's rather generic means of query execution allows very different execution flavors. Keep in mind that there are no well-defined guidelines, rather vague rules of thumb, to decide which execution flavor is best to execute a query [21]. This makes it impossible to decide the best flavor a priori because its performance depends on the current environment (hardware at hand, #cores used ...). Therefore, we provide a bouquet of paradigms and choose the best flavor adaptively at runtime. In the following, we describe two different flavors (a) atomic fragments, resembling vectorized execution [12] and (b) fused statements which is similar to data-centric compilation [40].

4.1 Atomic Fragments (Vectorized Execution)

Our base (and fallback) flavor is to only compile the smallest possible (indivisible = atomic) fragments. For VOILA operators, this means that such fragments are basic operations in VOILA (e.g. `add`, `bucket_lookup`, `seltrue`). Interestingly, when compiling atomic fragments, the resulting strategy is basically vectorized execution very similar to MonetDB/X100 [12] that became Vectorwise and later Vector. Consequently, this (1) generates many small fragments that can be compiled in parallel and can likely be re-used (2) allows efficient memory access, inherited from vectorized execution and (3) has good chances for micro-adaptive [19, 42] optimizations like full-execution by ignoring the selection vector. Since this is the default base flavor, it is used whenever we decide to not use any other flavors, which happens for short-running queries, or when the other flavors yield worse performance.

Specialized Implementations for Complex Operations. VOILA has two complex operations: `bucket_insert`, which allocates new buckets in a hash table but can fail, and `selunion`, which `OR`s two predicates together (in the vectorized model concatenates two selection vectors). For these operations we provide specialized hard-coded implementations.

4.2 Fused Statements (Data-Centric)

Data-centric compilation [40] is the extreme of compound primitives (or fused expressions), as it inlines the whole pipeline into a single function. For, a static engine with black-box operators, this inlining process is impossible as operator borders, typically, cannot be crossed. In Excalibur, operators can be *black-box*, i.e. hard-coded with one static implementation like `Output` (delivers query results), or white-box, yielding VOILA code that can be analyzed, modified, inlined etc. Note that the (performance-wise) most impactful operators (join, group-by, filter, projection) are white-box operators, which allows us to inline them. Hence, the presence of black-box operators breaks the inlining into multiple fragments. From the inlined VOILA code, we can, then, generate data-centric code [20].

5 (MICRO-)ADAPTIVE EXECUTION

We use vectorized execution as our base execution flavor and, during query execution, try to further improve performance by generating different execution flavors and observing whether they improve performance. Note that there is both a choice of execution flavor, as well as granularity (which parts of the query plan to use it in). The combination of these two choices we call execution *tactic*.

Exploration vs. Exploitation. During execution, we attempt two different things: (a) find the best possible execution tactic (*exploration*) and (b) use the best found tactic to improve runtime (*exploitation*). Consequently, to improve the runtime, we need to spend cycles exploring potentially not very useful tactics with no clear guarantees for success, i.e. a risky bet. In addition, we want to learn good tactics and exploit them as much as possible. Such problems are typically formalized as multi-armed bandits (MAB).

A naive MAB approach would be to explore *all* possible tactics *at least once*, and then exploit the best one. Note that the set of possible tactics is very large, especially since combinations of choices (query fragmentation and flavor) get flattened into separate points in the search space (actions in the MAB formalism)[2]. In order to limit the amount of alternative code fragments that need to be compiled and tested, we focus on *sparsely searching* the design space, followed by exploiting the best point found.

5.1 Constraints on Adaptive Execution

Suppose, we want to improve the runtime of a query fragment and we were given some method to decrease its runtime by 4× (4× speedup, $s = 4$). If this fragment only constitutes 50% of query runtime ($f = 0.5$), the overall expected speedup will drop to a disappointing 1.6×. Further suppose that we find this faster fragment not at the beginning of the query, but rather in the middle (at 50% progress, $\phi = 0.5$), then the final speedup will decrease further. In the following, we aim at finding a sweet spot for micro-adaptive optimizations which will guide the choices made by Excalibur.

Amdahl's Law. We model the impact of adaptive choices by using Amdahl's law [3]. Normally, Amdahl's law is used to compute the speedup of parallelizable computations with a sequential fraction. Here, instead parallelizing, we just accelerate the previously parallel fraction by a given factor.

We apply Amdahl's law for the progress ϕ: $S = (\phi + \frac{1-\phi}{y})^{-1}$ with y being the speedup at the specific progress (ϕ) in the query. Then, we apply Amdahl's law to determine y based on improving a fraction of the query f: $y = (1 - f + \frac{f}{s})^{-1}$, combining both yields:

$$S = \frac{1}{\phi + (1 - \phi)\left(1 - f + \frac{f}{s}\right)} \tag{2}$$

From Equation (2) we can derive that ideally we have to make good decisions (a) early and (b) on a large portion of the pipeline.

Limits on Exploration. The problem with exploring in constant intervals, as e.g. proposed by Raducanu et al. [42], is that towards the end of the query, it is still looking for better alternatives (exploring). Even though, their potential benefit cannot yield a significant improvement anymore (because it is found late).

[2] Alternatively, our problem could be modelled as a combinatorical MAB, by skipping the flattening and assuming that combinations of actions behave like the sum of its parts. This is an powerful concept that allows learning e.g. shortest paths or rankings. Solution approaches typically require an oracle to predict best actions [32], something we do not know a priori. A notable approach is the Follow-the-Perturbed-Leader algorithm [32], which introduces additional and complex tuning knobs, like well-chosen distributions for the perturbation (add random noise) to balance exploitation and exploration.

Table 2: Mutation nodes. A sequence of such nodes allows describing a specific point in the design space.

Mutation	Description
`JitFragm(begin, end, flavorMod)`	Compile fragment between `begin` and `end`, apply given `flavorMod`
`SetScope(begin, end, flavorMod)`	Set `flavorMod` for statements and expressions between `begin` and `end`
`Inline()`	Inline all VOILA operators
`SetDefault(flavorMod)`	Set default `flavorMod` for the whole pipeline
`SetConf(vectorSize, fullEval)`	Set vector size and different full evaluation threshold points
`BloomFilter(op)`	Enable Bloom filter [10, 22] at operator `op`
`SwapOps(a, b)`	Swap operators `a` and `b`

Table 3: Rules create and extend mutation sequences.

Rule	Description
`JitBiggestFragment(flavorMod, reqInline)`	JIT compiles the biggest fragment with `flavorMod` & introduces `Inline` before when `reqInline` is *true*
`ReorderFilterBySel`	Modifies plan to order filters by selectivity.
`BloomFilterMostSelJoin`	Introduces `BloomFilter` into most selective hash join.
`SetScopeFlavor(flavorMod)`	Find most expensive scope, introduces `SetScope`
`SetScopeFlavorSel(flavorMod)`	Like `SetScopeFlavor`, but scope must include VOILA's `SelTrue`, `SelFalse`
`SetScopeFlavorMem(flavorMod)`	Like `SetScopeFlavor`, but scope must include VOILA's `BucketLookup`, `BucketNext`, `BucketScatter`, `BucketGather`
`SetDefaultFlavor(flavorMod)`	Introduces `SetDefault`, if `flavorMod` is not already set.
`SetConfig(vectorSize, fullEval, scoreDiv, simdOpts)`	Introduces `SetConf`, if not already set.

To mitigate this exploration problem, we define a specific exploration budget (30% query runtime). Specifying a budget has two major advantages: (1) it forces most of the exploration to be done at the beginning of the pipeline (greedily) and (2) it limits the negative impact of over-exploring. Using a budget makes adaptivity a favorable asymmetric bet (limited loss, unbounded gain).

When running a query, we estimate the progress of the current pipeline (by tracking the data source). By estimating the progress and measuring the time spent for achieving the progress, we estimate the absolute budget used for exploration (in cycles)[3]: Absolute budget $B = (t + \frac{\phi}{t} * (1 - \phi)) * \beta$ with relative budget β (typically $0.3 = 30\%$ of query time), time t and progress $p \in [0, 1]$.

If exploration (and compilation) exceeds this budget, exploration is canceled and the residual budget is returned. Note, in case the query decelerates (starts running sub-optimally), the budget will increase, hence giving opportunity for more exploration. Further, we stop generating new tactics after 40% progress as we do not expect significant overall/net performance gains afterwards.

Other Applications. Equation (2) has many applications. For example, it can be applied to offloading work to accelerators. When an accelerator improves performance of an operation covering 40% of query performance by 10× and is triggered at the start, the best overall improvement we can possibly expect is a meagre 1.5×. If our accelerator improves performance by 100×, all else equal, we can maximally expect a rather disappointing 1.7×.

5.2 Exploitation

After the exploration budget is consumed, or the space is fully explored, our adaptive framework switches to exploiting the best points found so far. We choose the point with the lowest cost (CPU cycles per input tuple). However, during exploitation, we still maintain our performance metrics i.e. if performance of the current best choice degrades, we can still retry the already generated tactics.

5.3 Encoding the Design Space

Excalibur allows switching between tactics (i.e. different flavors applied to different fragments). Each tactic is a point in the design space. Here we discuss how Excalibur encodes points in that space.

Mutation Sequences. We define a point in the design space as a sequence of mutations that are created through rules. Currently, we have mutation nodes for (a) plan changes, (b) local configuration changes, (c) fragment JIT-ting and (d) flavor specifications. The specific nodes are listed in Table 2. Additionally, mutations can have also parameters. Most notably, `flavorMod` defines: specific unroll factors and SIMD widths for selective and, similarly, for the non-selective path of the vectorized primitive. Additionally, it allows using predicated execution using techniques described by Crotty et al. [13] or using conditional moves (`cmov`).

For instance, we can choose to combine `SwapOps` and `JitFragm`. `SwapOps` first modifies the plan and, afterwards, `JitFragm` would JIT-compile a specific fragment. Full data-centric execution can be expressed using `JitFragm` by selecting all of the pipeline.

Rule-based Generation. During exploration, we extend existing or create new mutation sequences (i.e. extending empty sequence). Table 3 shows the rule templates currently used in Excalibur. In practice, we expand the rule templates with common values for flavor and configurations. Rule-based generation provides two advantages (a) it is easily extensible and (b) we can iteratively expand the design space by applying rules onto a previous mutation sequence. Considering that over time the number of rules will likely grow, the design space will grow exponentially (assuming rules are not mutually exclusive). The rules in Table 3 were chosen to provide a minimal set of useful optimizations without inflating the design space too much. Still, materializing this large space is not practical due to compilation time and code-cache space overhead.

Therefore, how we explore the space matters.

6 EXPLORATION STRATEGIES

In this section, we present different exploration strategies, reaching from a simple randomized search, to hill-climbing with hard-coded heuristics and Monte Carlo Tree Search (MCTS).

6.1 Randomized Exploration (rand)

One could explore the space using random choices, and there are good reasons for this: randomized search is relatively easy to implement, can fully explore the space and can provide a "good" coverage of the space [15]. But for huge spaces randomized exploration might

[3] These estimation techniques have previously been used by Kohn et al. [29] and Gubner [17].

easily get "lost in the space" (i.e. not focus on interesting sub-spaces) and can take *extremely* long until the space is fully explored. This, while the budget of a short running query can only afford running a limited number of tactics.

6.2 Hard-Coded Heuristic (heur)

An intuitive approach is to try what database architects believe are good choices. This further makes the assumption that simpler choices, with shorter mutation sequences, are better (similar to Occam's razor). Therefore, we define a list of rules to apply and try:

1. Reorder filters by increasing selectivity
2. Introduce Bloom filter for selective joins
3. Heuristically JIT fragments:
   - If SelTrue/SelFalse and $\sigma < 95\%$ and $\sigma > 5\%$: Do not cross
   - If MemAccess and Cyc/Tup $> N_1$: Do not cross
4. Try fully data-centric
5. Try different vector sizes
6. Give up: Exploit

Clearly, iteratively improving an execution tactic by applying these rules in order reflects its creator's biases and potentially ignores large parts of the search space. Also, when new generator rules are added, this approach needs to be maintained (extended and re-evaluated) which is recurring time-consuming process. Note that the other strategies (randomized and, the following, Monte Carlo Tree Search) do not require hard-coded decisions and, hence, are less influenced by creator's biases and are maintenance-free.

6.3 Monte Carlo Tree Search (MCTS)

When rethinking our approach of exploring large spaces, we can draw analogies to Artificial Intelligence (AI) used in complex games (e.g. chess or go). For an AI to make the next choice, it commonly first builds a tree (state/search tree) that represents all possible choices made by players, multiple steps ahead. For complex games, like chess, these trees quickly become very (exponentially) large. One of the relatively new approaches, e.g. used in AlphaGo alongside Neural Networks, is Monte Carlo Tree Search (MCTS) [44].

Generic MCTS is a randomized approach to search a tree. It starts with exploring parts of the tree and, will given enough time, eventually have fully explored the tree. MCTS has 4 phases which are repeated, until typically a time limit is reached:

1. *Selection*: A node will be selected using some policy.
2. *Simulation*: Multiple paths from the selected node to the leaves are simulated.
3. *Node Expansion*: The selected node is expanded.
4. *Back Propagation*: The information from the simulation is propagated back towards the root.

The Selection phase has significant impact on which areas of the large tree are explored. Ideally, these should be the most important areas. Suppose, we had some measure of reward, then we could visit areas with the highest reward first (exploitation). This, however, should be balanced with more risky exploration. To balance exploration and exploitation with some measure of reward is a classical MAB problem, with arms corresponding to child nodes. A problem that can be solved optimally using the Upper Confidence Bound algorithm (UCB, Equation (1)).

UCB applied to trees (UCT). For trees, or MCTS in specific, there is a variant of UCB, called UCT [28]. Until a given point in time, the following variables specific historical metrics collected so far: Let X_i be the empirical mean (of rewards) for child node i, c be some constant, t be the number of samples in the parent node and s be the number of samples of the current node:

$$uct_i = X_i + c * \sqrt{\frac{t}{s}}$$

During the Selection phase, this score is computed for each potential child node, then the child node with the highest score is chosen. This is repeated until a leaf is found. Later, during Back Propagation, we need to update the metrics (reward and number of samples) for our selected node and all nodes on the back towards the root.

Application to our Exploration Problem. The application of MCTS to our problem, efficiently exploring the design space, is relatively straightforward. Each mutation node becomes a node in the MCTS. We adapt the Simulation and Node Expansion steps: during Simulation, we execute the mutation sequences that result from the path in the tree and collect runtime statistics. Then, we expand existing node simply by applying our generator rules. In our case, MCTS brings two major advantages: (1) MCTS almost never evaluates the full search space, unless given an huge exploration budget. Instead, it focuses on promising sub-spaces, which is a crucial advantage for exploring large spaces. (2) Assuming the same pipeline is run multiple times, we can extend the existing tree in the new run (i.e. learning). Especially in a cold run, our tree does not yet contain useful information (many UCT scores are ∞ i.e. highest possible score). In the following, we aim at further improving the order in which we traverse the search space by heuristics for breaking ties between UCT scores.

Propagating Information across Branches. Suppose we already have partial knowledge, e.g. we know that whole-pipeline data-centric works well on this platform. But that knowledge is part of a different branch of the tree, one we have not traversed yet during Selection. Consequently, we would have to re-discover that data-centric execution is beneficial. To ease the burden of re-discovering good choices, we remember rewards and #samples for all mutations (many tree nodes can encode a mutation in different branches, also tree nodes can encode mutation sequences). This allows us to formulate this guessing problem as a MAB, but this time for mutations (and mutation sequences), rather than MCTS tree nodes. This steers the exploration into the direction with the highest confidence (using UCB). In practice, this turns out to work quite well, because on the first level of the tree (i.e. close to the root), we likely discover most possible decisions.

Maximum Distance. If, we are, however, at the very beginning of building the tree, we do not have such knowledge, yet. In this case, we try to steer away from already explored nodes (or clusters). Therefore, given a set of already explored siblings (children of the same parent node), we should preferably explore the most dissimilar point next. We define the similarity of two nodes x and y as $1 - d(x, y)$ according to a distance function d. Using d, we select the nodes with the maximal distance to already explored nodes, to break ties. If there are multiple such nodes, we chose randomly.

Gower Distance. Our choice chains (mutation sequences) are rather complex objects, composed of categorical integers, quantitative integers and lists thereof. Therefore, we use Gower distance [16], a distance function able to handle complex objects. It is defined as the arithmetic mean of its components K:

$$d(x, y) = \frac{1}{|K|} \sum_{k \in K} d'(x, y, k) \tag{3}$$

Each component has slightly different formula (d') depending on its type. We only describe quantitative (d'_q) and categorical components d'_c, other types are defined as well but are not relevant here.

For a quantitative component k and its range r_k, we define

$$d'_q(x, y, k) := \frac{|x_k - y_k|}{r_k}$$

For a categorical component k, we define

$$d'_c(x, y, k) := \begin{cases} 1 & \text{if } x_k \neq y_k \\ 0 & \text{otherwise} \end{cases}$$

Since we only need to measure the distance between sibling nodes in the tree, we can directly apply Equation (3) for two nodes x and y. Note that to find the node(s) with maximal distance, we need to compute the pair-wise distances between all siblings. Thus, for trees with many siblings computing the distance can become costly. The trees we create are typically not very wide (nodes have roughly up to 40 siblings). When extending the mutation nodes and mutation rules over time, trees widen. In this case, we can use a random sample of sibling nodes to compute the distances.

6.4 Remembering the Past

The proposed exploration strategies have to re-explore the search space before any positive changes can be done, *for each query*. Figure 2a illustrates this for TPC-H Q1. One can increase the Risk Budget to explore a bigger part of the space, but consequently, overall query performance suffers as precious CPU cycles are wasted:

In longer-running workloads, we can exploit past knowledge.

While we cannot fully rely on the accuracy of the past (underlying data may have changed causing different performance), we should not fully disregard past knowledge.

Quick Start - Remembering Good Points. After exploring a pipeline, or query, we can remember the best choices. On the next iteration of the query, we can start checking whether these choices are still the best. While this delays the regular exploration process by a few steps, it directly feeds good points back into the exploration process. We call this Quick Start.

We implemented Quick Start by generating a fingerprint of the pipeline and mapping the fingerprint to the historic data. The historic data contains a mapping from the (design space) point to a histogram of runtimes. Both mappings can grow quite large, if they grew over a certain threshold, we use sampling to determine the surviving data points. Our fingerprints contain operator types as well operator properties (e.g. global aggregation, key join). An improved mapping could also include performance information (e.g. pipeline throughput tuples/cycle, selectivities) or system state (e.g. #threads used). Currently, we use an exact mapping between fingerprint and historical data. But, especially, when integrating performance information into the fingerprint, a best-effort match would be more desirable.

Table 4: Excalibur often significantly outperforms other systems optimized for analytics (TPC-H SF50, multi-threaded).

Name	Runtime (ms)			
	Q1	Q3	Q6	Q9
Umbra [41]	287 (1.5×)	**326** (0.9×)	91 (1.8×)	854 (1.2×)
DuckDB [1]	1325 (6.9×)	2338 (6.7×)	341 (6.6×)	15306 (21.0×)
MonetDB [24]	5488 (28.6×)	1089 (3.1×)	190 (3.7×)	1178 (1.6×)
Excalibur (heur)	**192**	349	**52**	**730**

Incremental Monte Carlo Tree Search (MCTS). MCTS has the convenient property that we can continue building the tree with following runs of the same query. Consequentially, MCTS can incrementally learn more about the design space, iteration by iteration. The challenge is to identify the same pipeline, for which we use the same fingerprinting scheme as for Quick Start.

7 EXPERIMENTAL EVALUATION

In this section we provide a compact experimental evaluation of the Excalibur VM, which we implemented in C++. For each operator, it first generates VOILA [20] code, which it then translates into LLVM IR using a particular flavor; and then into machine code on-the-fly (using LLVM's C++ API). The VM support multiple adaptive decisions (presented in Table 2) and a bouquet of exploration strategies (discussed in Section 6).

Hardware. In small scale experiments, scale factor 50 and below, we used a dual-socket Intel Xeon Gold 6126 with 24 SMT cores (12 physical cores) and 19.25 MB L3 cache each. The system is equipped with 187 GB of main memory. For large scale experiments with scale factors ≥ 100 the previous system did not have enough main memory. Therefore, for large scale experiments, we used an (older) quad-socket Xeon E5-4657L v2 with 96 SMT cores (48 "real" cores) in total, 30 MB L3 cache per chip and a total of 1 TB main memory.

Structure. First we compare Excalibur to other state-of-the-art system as well as hand-written implementations. Then, we analyze impact of the Risk Budget onto finding possible improvements and, consequently, performance. Afterwards, we compare the different exploration strategies on the TPC-H data set and investigate the adaptation with respect to various parameter values in TPC-H Q6. Then we investigate the impact of the code cache and, lastly, show the adaptation over the runtime of a query.

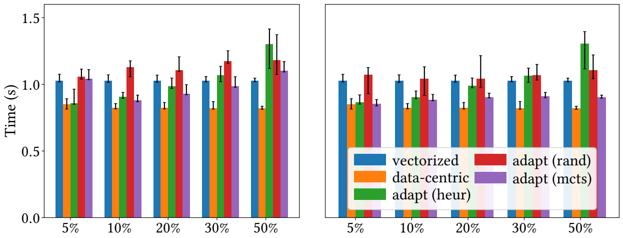
7.1 State-of-the-Art Competitors vs. Excalibur

To judge the performance of Excalibur on a relative as well as absolute level, we compare it to state-of-art systems and handwritten implementations. We selected a diverse set of queries: TPC-H Q1, Q3, Q6 and Q9. We ran these queries against the TPC-H data set with scale factor 50 and used all available hardware threads.

Systems. First, we compare to systems optimized for analytical performance. In particular, we chose the well-known state-of-the-art systems. This includes the open source systems MonetDB [24], featuring classical columnar execution, and DuckDB [1], a vectorized system. In addition, we compare to the data-centric system

Table 5: Compared to hand-written & optimized implementations, Excalibur's implementation of vectorized & data-centric execution still "leaves room for improvement".

Name	Runtime (ms)			
	Q1	Q3	Q6	Q9
Vectorized Execution				
Tectorwise [26]	248 (1.0×)	**294** (0.7×)	66 (1.3×)	793 (0.9×)
Excalibur (vec)	225	394	**49**	917
Data-Centric Execution				
Typer [26]	**137** (0.8×)	437 (0.8×)	73 (1.2×)	1193 (0.9×)
Excalibur (dc)	163	541	61	1337
Overall				
Tectorwise [26]	248 (1.3×)	**294** (0.8×)	66 (1.3×)	793 (1.1×)
Typer [26]	**137** (0.7×)	437 (1.3×)	73 (1.4×)	1193 (1.6×)
Excalibur (heur)	192	349	**52**	**730**



(a) ***Naive* Exploration: β must be high enough to discover good choices but low enough to not waste too much time.**

(b) **With *Quick Start*, we already have a good guess on the best flavor. Thus, we can lower β.**

Figure 2: Impact of Risk Budget β

Umbra [41] which uses a simple VM-based approach to dynamically switch between different JIT-compiled flavors [27]. The results are summarized in Table 4. We can see that Excalibur outperforms the three other systems on most queries, as none of the fixed execution strategies (column-at-a-time, data-centric, vectorized) dominates across all queries and Excalibur adaptively finds a good strategy and the code generated using VOILA has competitive raw performance.

Hand-written Implementations. To further delve in raw performance, we compare with the hand-optimized implementations of state-of-the-art query execution paradigms by Kersten et al. [26]: Typer, an instance of data-centric compilation [40] and Tectorwise, an implementation of vectorized execution [12]. Both, Typer and Tectorwise, perform roughly on par with the system that pioneered its respective paradigms Hyper and Vectorwise [26]. Table 5 shows the results. Most queries perform roughly on par. However, we noticed that the implementations of Excalibur are slightly slower. Most notably, the data-centric implementation of Q1, where LLVM "optimizes" our data-centric code by, instead of merging branches, replaces them with conditional move instructions (cmov).

7.2 Impact of Risk Budget

In this experiment, we measure the effect the *Risk Budget*, the budget for adaptive exploration, has on overall performance. We chose a relatively simple query, TPC-H Q1 (on SF10, single-threaded), where the best execution paradigms currently known are data-centric, or variations thereof. Consequently, our system has to switch to a completely different execution paradigm, which first has to be discovered. Here, we differentiate between non-learning exploration strategies (*naive* exploration), without knowledge of the past, and learning strategies, able to leverage past knowledge.

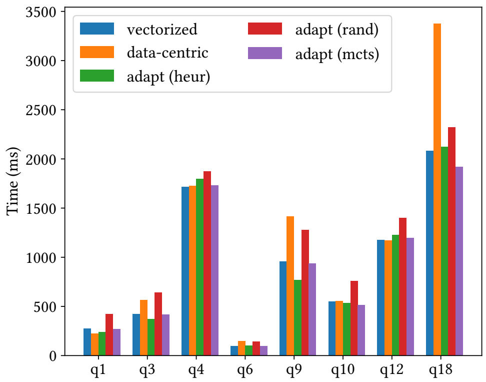


Figure 3: On medium-sized data sets, Excalibur can adapt to the best flavors (TPC-H SF50, multi-threaded).

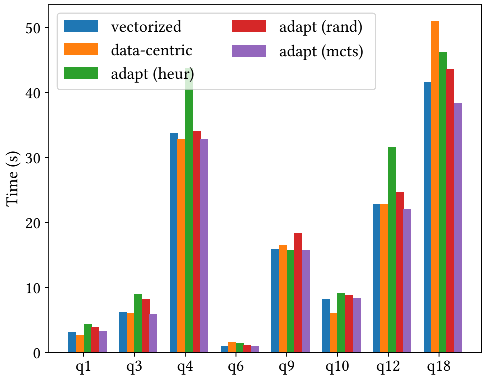


Figure 4: On the older hardware platform used for the larger scale, the difference between data-centric and vectorized execution blurs. The extra execution time does allow *mcts* to consistently beat *heur* (TPC-H SF300, multi-threaded).

Naive Exploration. Figure 2a visualizes impact of varying Risk Budgets on overall query performance for non-learning exploration strategies. We can see that there is no clear optimal budget and it depends on the exploration strategy: We need a minimum budget to be able to discover a reasonably good solution, but using too much is counter-productive. For large search spaces, it is hard to adapt to the better flavor in time, especially with a low budget.

Learning Exploration. Using Quick Start, we remember good points and, in the next run, explore them early. Figure 2b shows that using Quick Start allows lowering the Risk Budget needed to find good points. For example, using the MCTS strategy, it could be lowered to 5% whereas without learning even with a Risk Budget of 50% we were not likely to discover good points.

7.3 Various Scale Factors & Multi-Threading

Analytical queries tend to behave differently with (a) varying data sizes as well as (b) with/without parallelism (i.e. multi-threading). Larger tables significantly impact query performance, e.g. hash tables grow bigger leading to increased memory access cost. Similarly, parallelism also causes different performance characteristics.

When running using one core that core can consume most of the system's memory bandwidth. On the other hand, when using all available cores, memory bandwidth has to be shared between them, which leads to higher memory access cost on each core (more cycles spent on memory accesses/waiting for memory locally). Since, these factors impact performance, it is reasonable to assume they can also impact the best flavor of a query. Therefore, we experiment on different scale factors of the TPC-H data set.

Medium-Scale Multi-Threaded. We start with a multi-threaded experiment on scale factor 50. The resulting runtimes are visualized in Figure 3. We see that there is significant performance diversity between the data-centric and vectorized flavors, most notably in Q9 and Q18, but less extreme also in Q1 and Q3. It is visible that Excalibur can adapt to the best flavor, depending on the exploration strategy used. Notably adaptive strategies can beat static flavors (e.g. on Q9 heuristic beats vectorized and data-centric leading to roughly 2× improvement). Usually the heuristic strategy (*heur*) behaves best thanks to the relatively small space explored (certain hard-coded points), but is closely followed by the Monte Carlo Tree Search-based strategy (*mcts*). Less elaborate strategies (i.e. the randomized approach *rand*) tend to perform worse than MCTS, this is due to (a) to the learning nature, trees can be extended over multiple runs, and (b) better exploration behaviour, as good candidate sub-trees more likely to be re-visited. In these multi-threaded experiments, the absolute budget is relatively low (high throughput, queries run quickly), thus exploration strategies do not have much time to discover good points.

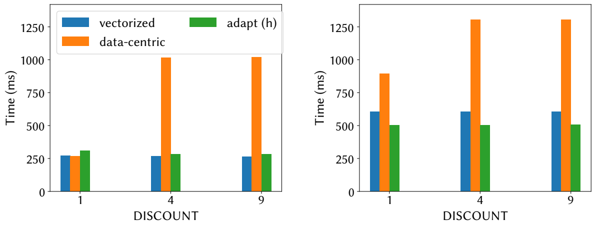
Large-Scale Multi-Threaded. Large data sets, however, give Excalibur more time for exploration, thanks to the higher query runtime. Thus, we ran the same queries on a roughly 6× bigger data set. Figure 4 shows the resulting runtimes. The underlying hardware has more main memory and cores, but the older CPU means queries have relatively higher runtime and performance of flavors behaves differently. But also here Excalibur adapts to the best flavor. The heuristic exploration is now consistently outperformed by the MCTS strategy: thanks to significantly higher query runtime, the absolute risk budget is higher (proportional to query runtime) and allows exploring more points in the design space.

7.4 Adaptation to Varying Query Parameters

Especially in real-world queries, cardinalities are hard to predict and frequently *wrong by orders of magnitude* [36]. One possible solution is to use real-life observed metrics to optimize the query at runtime (i.e. adaptivity) by e.g. re-ordering filters. This is (1) challenging for JIT-compiling systems, as it would require expensive re-compilation and (2) affects the best flavor. Therefore, we experiment how our JIT-compiling system Excalibur adapts to changing selectivities. In particular, we evaluate TPC-H Q6 with different parameters:

```sql
SELECT SUM(l_extendedprice*l_discount) AS revenue FROM lineitem
WHERE  l_shipdate >= DATE '[DATE]' AND l_quantity < [QUANTITY]
  AND  l_shipdate < DATE '[DATE]' + INTERVAL '1' YEAR
  AND  l_discount BETWEEN [DISCOUNT] - 0.01 AND  [DISCOUNT] + 0.01
```

Different parameter choices consequently lead to different selectivities in each of the WHERE clauses. For simplicity, we chose the DATE to start 01-01 (January 1st) in a specific year and from hereon only specify the YEAR. Figure 5 shows our results. We observe that full vectorized execution often is the best tactic, beating full data-centric execution. However, adaptive execution using the heuristic search strategy is able to identify this: it generally is on par with the best static tactic, and in Figure 5b even beats it.



(a) YEAR=1999, QUANTITY=1 **(b)** YEAR=1992, QUANTITY=1

Figure 5: Adaptive execution beats static on Q6 with varying parameters. Values for DISCOUNT have been multiplied by 100 (i.e. 0.01 becomes 1). (TPC-H SF300, multi-threaded)

Table 6: Runtime of smaller scale factors is significantly affected by compilation latency, which can be eased using a code cache or using parallelism. Impact of code cache on query runtime for TPC-H SF0.1 without adaptive execution.

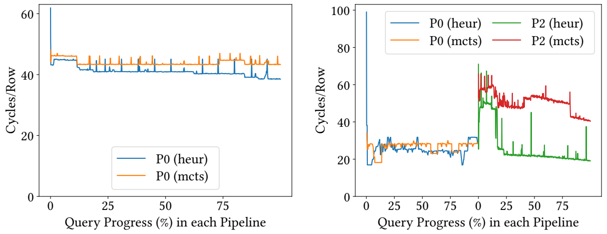
Cache Size	Runtime (s)					
(#fragments)	1 Thread			8 Threads		
	Q1	Q9	Q18	Q1	Q9	Q18
0	29.1	54.6	59.0	5.1 (6×)	10.6 (5×)	11.2 (5×)
8	13.9 (2×)	29.6 (2×)	28.8 (2×)	2.9 (10×)	6.4 (9×)	7.7 (8×)
16	11.1 (3×)	25.9 (2×)	25.5 (2×)	2.6 (11×)	6.7 (8×)	6.0 (10×)
32	4.5 (6×)	19.3 (3×)	19.1 (3×)	1.8 (16×)	5.3 (10×)	4.8 (12×)
64	1.1 (27×)	6.0 (9×)	6.0 (10×)	0.4 (69×)	2.1 (26×)	2.3 (25×)
128	1.1 (26×)	1.9 (28×)	2.0 (30×)	0.4 (68×)	0.8 (68×)	0.9 (68×)
1024	1.1 (26×)	2.0 (28×)	2.0 (30×)	0.4 (72×)	0.8 (68×)	0.8 (74×)
16384	1.1 (26×)	2.0 (28×)	2.0 (30×)	0.4 (68×)	0.8 (68×)	0.8 (73×)

7.5 Code Cache

For short-running queries, compilation latency tends to be a major bottleneck. In our model, many fragments can be cached, thus reducing compilation latency. In the following we investigate the impact of the code cache's size on the query runtime. This size refers to the number of fragments stored. 0 refers to the code cache being disabled. Table 6 shows the results.

General Observations. It can be seen that with increasing size of the code cache, query runtime improves. For simple queries, like Q1, with 32 fragments cached, we can improve the runtime by 6×. The plateau is reached at about 64 cached fragments, where the runtime is roughly 26× faster than without the code cache. More complex queries, like Q18, contain more code fragments and, therefore, require larger code caches. In case of Q18, a code cache size of around 128 fragments captures all code fragments at that point runtime is roughly 30× faster than without the code cache.

Multi-Threaded Compilation. If we compare single-threaded execution (mostly compilation time on SF0.1) against multi-threaded, we see that compilation time improves significantly (5 – 6×). Unfortunately, compilation does not seem to scale linearly. When compiling code fragments concurrently, we do not introduce any additional locking (besides checking the code cache, reserving an



(a) Q1 **(b) Q12**

Figure 6: Over time different flavors are tried and adapted. Pi denotes pipeline i. (TPC-H SF50, single-threaded).

entry in the cache and the execution pipeline waiting until compilation is complete). Thus, we suspect the additional overhead must come from LLVM.

High Compilation Time without Code Cache. It can be seen that without code cache (size 0), the initial compilation time is extremely high as all vectorized primitives (code fragments) need to be generated. This has multiple reasons: Machine Code generation in Excalibur is not optimized: We generate relatively straightforward LLVM IR from VOILA and rely on compiler optimizations (like auto-vectorization/SIMDzation, and others commonly included in -O3) afterwards. This can of course be improved. For example, we could directly emit SIMDized code, i.e. skipping compiler auto-vectorization. We consider optimizing query compilation latency a very relevant, yet rather orthogonal challenge for our research into the possibility and benefits of an adaptive fine-grained runtime exploration of the execution strategy search space. Thus, we expect low-latency JIT query compilation techniques pioneered in other code-generating systems, most notably Hyper [40] and Umbra [41] (both regrettably not in open source), to be beneficial for Excalibur; e.g. better register allocation [29], directly emitting assembly [27], avoiding certain combinations of optimizations [40]. Improvements in compilation latency will be even more impactful in Excalibur than Hyper and Umbra, as we generate multiple alternative code fragments during execution. This currently also leads to additional setup and tear-down costs (`LLVMContext` and `TargetMachine`).

7.6 Adaptation over Query Runtime

Over the runtime of a query, Excalibur tries to find better execution tactics. To highlight this adaptive behaviour, we visualize the execution traces as measured by Excalibur without Quick Start. If a pipeline has less than 10 samples, we omit it in the plot. Note that the x-axis shows query progress rather than the absolute time, consequently all pipelines have the same length (from 0% to 100%).

Q1. The results for Q1 are visualized in Figure 6a. Both exploration strategies, heuristic and MCTS, start at around 60 cycles/row, and from there on quickly choose better flavors. Unsurprisingly, we can see that with the hard-coded heuristic (*heur*) strategy more quickly find a faster flavor. The reasons are that (a) the heuristic is heavily biased and (b) only explores a rather small space. The other strategy (*mcts*) explores a significantly larger space and is, in this time frame, unable to find a flavor faster than 45 cycles/row.

Q12. The results for Q12 are visualized in Figure 6b. Again, the heuristic tends to outperform. In the first pipeline $P0$, it finds a faster flavor (30 cycles/row vs. 100 cycles/row initially). But it is closely tracked by *mcts*. In the second pipeline in the plot ($P2$), the heuristic outperforms by a wider margin (20 cycles/row vs. 70 cycles/row initially). Also here, *mcts* tends to find better flavors (40 cycles/row) but is, due to the large search space, unlikely to find the winning flavor in the time frame.

8 RELATED WORK

Historically, database engines had to make a choice between interpreted execution and compilation i.e. they either ship and have to maintain one, or both. Maintaining both engines will eventually lead to significantly higher maintenance effort (and consequently higher costs). Instead, Excalibur effectively operates as a virtual machine that supports JIT-compilation. It features a very simple (and low maintenance) interpreter with code generation being used to bootstrap most expressions/statements. This work implements many ideas sketched earlier [18] where we proposed a virtual machine that would (a) bridge the gap between interpretation and compilation, (b) allow adaptive compilation and (c) adaptive offloading to heterogeneous hardware. The heterogeneous hardware aspect, however, still remains as future work.

Kohn et al. [29] showed how Hyper was extended with an LLVM interpreter. This way, Hyper can efficiently handle short queries using interpretation (skipping IR to assembly compilation), while long running queries can benefit from optimized execution in assembly. In Hyper a whole pipeline is either interpreted or compiled. Excalibur operates on finer granularity (code fragments). Compared to Hyper, code-fragment-granularity has two major benefits: (a) code fragments can be cached, effectively minimizing compilation time, and (b) code fragments can easily be replaced (e.g. with optimized versions, essentially micro-adaptivity [42]). After the additions by Kohn et al. Hyper effectively ships a compiler translating physical plans to LLVM IR, and an interpreter running their LLVM IR-alike language which needs to handle every expression in their language. Excalibur, instead, ships a very simple interpreter where customized code fragments can be plugged in. Consequently, Excalibur only needs to explicitly handle a few auxiliary statements/expressions while most relevant expressions can be JIT-ed and cached.

LLVM compilation is also used in Umbra [27] which as a low-latency option provides an X86 back-end that directly generates assembly (reducing JIT latency). However, generating assembly is not portable to other platforms, which is problematic with the rise of non-X86 high performance processors [21] (ARM, RISC-V plus accelerators). Due to its architectural proximity to Hyper, it also inherits Hyper's limitations, namely compiling whole pipelines, and a fixed choice for a single (data-centric) code generation flavor.

Other systems compile to languages that provide a JIT-compiling virtual machine. A notable example is Spark which can (a) interpret operations on Resilient Distributed Data Sets (RDDs) [46] or (b) generate Java code [4]. Both rely on the Java Virtual Machine providing efficient interpretation and compilation. Evidently, performance of these architectural choices on database workloads was sub-par, culminating in the development of a new vectorized engine in C++ with "2-4x average speedups [...] on SQL workloads" [8].

Another interesting approach has been taken by the Collection VM (CVM) project [39]. CVM constitutes a compilation framework based around an Intermediate Representation with back-ends that compile to different targets. The IR is centered around processing collections (bags, sets, structs thereof, and sets of structs ...). CVM produces complex intermediates and, to guarantee top-notch performance, will have to remove them with a computationally hard process (i.e. Deforestation [45]), a problem VOILA [20] does not struggle with (intermediates are vectors of atomics, or tuples of such vectors). Moreover, CVM did not intend to cover different query execution paradigms, and it is not clear, if it can. However, CVM offers automatic parallelization on one or multiple machines.

MLIR [34] is a flexible Intermediate Representation (IR) which allows combining different languages or programming paradigms and allows optimization across languages. A new IR style can relatively easily added. This is a orthogonal project regarding to VOILA [20], e.g. VOILA could be integrated into MLIR [9]. LingoDB [25] is a recent query executor prototype built on MLIR, which significantly reduces software engineering effort, and allows it to blend query optimization and code optimization techniques. It compiles pipelines using a single flavor, though, relying on static rules.

AWS Redshift is an analytical database service, built on what was originally an on-premise MPP engine (ParAccel) but subsequently much evolved [5]. It uses JIT query compilation via textual C++ code generation, and applies extensive code caching to mitigate compilation latency, including a cloud-based compilation and caching service that Redshift instances contact for help if a fragment is not in the local cache. We think that the more fine-grained code fragments that Excalibur generates, could thrive in such an architecture and lead to even higher cache hit ratios.

Adaptive Query Execution. Besides the adaptive execution provided by Virtual Machines, data processing systems tend to also explicitly implement adaptive execution. Hyper [29] and Umbra [27] allow adaptivity on the pipeline-level i.e. switch the whole pipeline against a better one. This comes down to a choice between a limited number of back-ends for interpretation and compilation. Compared to Excalibur, they cannot (a) make fine-grained expression-level decisions, (b) do not make high-level decisions adaptively, (c) minimize compilation effort when switching pipelines (unaffected fragments will hit Excalibur's code cache) and (d) require a fully re-engineered back-end for each target flavor.

Micro-adaptivity in Vectorwise only affects the current expression tree [42] and typically only affects single code fragments. Excalibur, however, can leverage adaptivity on multiple levels and additionally gain from JIT-compilation.

MonetDB [24] executes queries column-at-a-time. This allows dynamically choosing processing kernels based on properties assigned to input columns, e.g. a join can be a merge join (kernel) when both inputs are sorted on the join keys (properties). This, property-based, approach ignores the impact of runtime effects, e.g. a hash join could be faster than merge join when the hash table is very small, thanks to data parallelism. But it also operates on a too large granularity: we can only adjust execution after the full column is processed i.e. we cannot execute part of the column and find a reasonably well-performing kernel at runtime.

With Permutable Compiled Queries (PCQ), Menon et al. [38] propose techniques allowing adaptivity without recompilation. When changes to the plan are made, Excalibur might recompile a small part of the pipeline (often code fragments are in the code cache). When, e.g., only operators are re-ordered all code fragments already reside in the cache and no actual compilation takes place (happens e.g. in Section 7.4). PCQ introduces a layer of indirection at specific points of the query. By using a Virtual Machine-based approach, Excalibur, naturally operates with a layer of indirection on the language-level. Consequently, Excalibur is not limited to a handful of plan permutations, like PCQ, but allows to continuously re-optimize the plan at runtime with limited compilation effort.

In SparkSQL [4], queries often contain shuffle operations that redistribute data across nodes. However, such a shuffle requires all results to be materialized first. Adaptive Query Execution in Spark [14] allows re-optimizing the following stages based on information gain until the currently materialized shuffle.

9 CONCLUSION

With Excalibur, we presented a system that is aimed to exploit the increasing performance diversity resulting from today's ever more heterogeneous hardware. It achieves adaptive execution of diverse flavors using a single code-base, which avoids a software explosion and the resulting technical debt. The key idea is to execute VOILA query plans using an micro-adaptive Virtual Machine (VM). VOILA is a high-level language to describe relational operator algorithms and can generate code using many different flavors, among which the well-known vectorized and data-centric execution strategies. The Excalibur VM explores the design space of query execution and dynamically exploits good points, *on-the-fly*. Exploring the search space is cast into a Multi-Armed Bandit (MAB) problem.

In our experiments, Excalibur can outperform open-source systems by up to 28×, the state-of-the-art system Umbra by up to 1.8×, and its static execution strategies by up to 2×, while its adaptive execution is never much slower than any of them. When the design space is large, the gains depend on the exploration strategy. In experiments we see that on short-running queries, heuristic exploration finds the best plans, while in long-running queries, Monte Carlo Tree Search exploration finds even better plans.

Discussion. This research shows that the idea of a query code generator that can not only generate many execution flavors (the VOILA DSL), but also quickly and adaptively find a good flavor without relying on brittle physical costing (the Excalibur VM) can be realized in practice. This is an encouraging result. We do note that Excalibur is a prototype with limited functionality and still suffers from high compilation latencies. Also, its overall performance gains are modest, such that at this point one can ask whether the complexity of VOILA and Excalibur software infrastructure (which also constitutes technical debt) already merits industrial adoption. However, we believe hardware heterogeneity is likely to increase and the benefits of approaches like Excalibur with that; and there is plenty of future work to further improve our approach.

Future Work. At runtime, Excalibur explores the design space and adapts physical execution, if beneficial. However, a query might have more than one optimal flavor which might change (i.e. behaviour is non-stationary). Currently, Excalibur would only detect a deterioration in performance and choose the next best flavor. Furthermore, Excalibur could certainly be extended to dynamically offload certain tasks to accelerator devices, such as GPUs or FPGAs.

REFERENCES

[1] [n.d.]. https://duckdb.org/.

[2] 2022. https://web.archive.org/web/20220626185617/https://stackoverflow.com/questions/56852812/simd-instructions-lowering-cpu-frequency. Accessed: 2022-02-16.

[3] Gene M. Amdahl. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities *(AFIPS '67 (Spring))*. 483–485.

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.

[5] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD*. 2205–2217.

[6] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2 (2002), 235–256.

[7] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *PVLDB* 8, 4 (2014), 353–364.

[8] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2326–2339.

[9] Paul Blockhaus. 2022. *A Framework for Adaptive Reprogramming Using a JIT-Compiled Domain Specific Language for Query Execution*. Master's thesis. Otto-von-Guericke University Magdeburg.

[10] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426.

[11] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. 61–76.

[12] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*. 225–237.

[13] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2020. Getting swole: Generating access-aware code with predicate pullups. In *ICDE*. 1273–1284.

[14] Wenchen Fan, Herman van Hövell, and MaryAnn Xue. 2020. Adaptive Query Execution: Speeding Up Spark SQL at Runtime. https://web.archive.org/web/20200611173835/https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html.

[15] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. 1994. Fast, Randomized Join-Order Selection - Why Use Transformations?. In *VLDB'94*. 85–95.

[16] John C Gower. 1971. A general coefficient of similarity and some of its properties. *Biometrics* (1971), 857–871.

[17] Tim Gubner. 2014. *Achieving many-core scalability in Vectorwise*. Master's thesis. Technical University of Ilmenau.

[18] Tim Gubner. 2018. Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware. In *ICDE*. 1684–1688.

[19] Tim Gubner and Peter Boncz. 2017. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. In *ADMS*.

[20] Tim Gubner and Peter Boncz. 2021. Charting the Design Space of Query Execution using VOILA. In *PVLDB*, Vol. 14. 1067–1079.

[21] Tim Gubner and Peter Boncz. 2021. Highlighting the Performance Diversity of Analytical Queries using VOILA. In *ADMS*.

[22] Tim Gubner, Diego Tomé, Harald Lang, and Peter Boncz. 2019. Fluid Co-processing: GPU Bloom-filters for CPU Joins. In *DaMoN*. 9:1–9:10.

[23] Laura M. Haas, Wendy Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene Shekita. 1990. Starburst Mid-Flight: As the Dust Clears. *IEEE Trans. on Knowl. and Data Eng.* (1990), 143–160.

[24] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering* 35, 1 (2012), 40–45.

[25] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401.

[26] 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. *PVLDB* (2018), 2209–2222.

[27] 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.

[28] Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*. 282–293.

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

[30] Vlad Krasnov. 2017. On the dangers of Intel's frequency scaling. https://web.archive.org/web/20220810211446/https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[31] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.

[32] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit algorithms*. Cambridge University Press.

[33] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*.

[34] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A compiler infrastructure for the end of Moore's law. *arXiv preprint arXiv:2002.11054* (2020).

[35] 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 *SIGMOD*. 743–754.

[36] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015), 204–215.

[37] Guy M. Lohman. 1988. Grammar-like Functional Rules for Representing Query Optimization Alternatives. *SIGMOD Rec.* (1988), 18–27.

[38] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C Mowry, and Andrew Pavlo. 2020. Permutable compiled queries: dynamically adapting compiled queries without recompiling. *PVLDB* 14, 2 (2020), 101–113.

[39] Ingo Müller, Renato Marroquín, Dimitrios Koutsoukos, Mike Wawrzoniak, Sabir Akhadov, and Gustavo Alonso. 2020. The Collection Virtual Machine: An Abstraction for Multi-Frontend Multi-Backend Data Analysis *(DaMoN '20)*. Article 7.

[40] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9 (2011), 539–550.

[41] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.

[42] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro adaptivity in vectorwise. In *SIGMOD*. 1231–1242.

[43] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *PVLDB* 6, 11 (2013), 1080–1091.

[44] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529 (2016), 484–489.

[45] Philip Wadler. 1988. Deforestation: Transforming Programs to Eliminate Trees. In *ESOP*. 231–248.

[46] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.