Simple (yet Efficient) Function Authoring for Vectorized Engines

Laith Sakka
Meta Platforms Inc.
lsakka@meta.com

Pedro Pedreira
Meta Platforms Inc.
pedroerp@meta.com

Orri Erling
Meta Platforms Inc.
oerling@meta.com

Masha Basmanova
Meta Platforms Inc.
mbasmanova@meta.com

Kevin Wilfong
Meta Platforms Inc.
kevinwilfong@meta.com

Wei He
Meta Platforms Inc.
weihe@meta.com

Xiaoxuan Meng
Meta Platforms Inc.
xiaoxmeng@meta.com

Krishna Pai
Meta Platforms Inc.
kgpai@meta.com

Bikramjeet Vig
Meta Platforms Inc.
bikramjeet@meta.com

ABSTRACT

Vectorized execution engines process large datasets by decomposing computations into concise (tight) loops, which can be more efficiently executed by modern hardware. Providing loops that are optimal for execution usually adds burden to the software development process, as developers are required to understand details of vectorized execution, columnar data layout, data encodings, and the code compilation process itself, presenting a steep learning curve and challenges to organizations building and scaling large engineering teams. Due to their large quantity, scalar function authoring accentuates this problem. In our experience building the Velox open source execution engine, we have observed that exposing a large number of developers to the complexity inherent to vectorization resulted in a disproportionate amount of bugs and performance inefficiencies. In this paper, we describe the simple function interface (SFI) created to address this issue. SFI highly simplifies scalar function authoring by encapsulating the vectorization complexity required to generate tight loops, and presenting developers with a simpler, conciser, and more natural row-based interface - without sacrificing performance. SFI also hides columnar layout details, while providing developers the flexibility to efficiently implement advanced features such as functions with nested and recursive parameter types, type variables, variadic parameters, and generic types. Today, more than a thousand functions have been added to Velox using the SFI, implementing popular open source SQL dialects and internal domain-specific use cases at Meta, and are in active production use. While this paper presents implementation details, performance pitfalls, experimental results, and our overall experience developing the state-of-the-art Velox vectorized execution engine, we believe the concepts and trade-offs to be fundamentally equivalent and generally applicable to other vectorized engines.

PVLDB Reference Format:
Laith Sakka, Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Wei He, Xiaoxuan Meng, Krishna Pai, and Bikramjeet Vig. Simple (yet Efficient) Function Authoring for Vectorized Engines. PVLDB, 17(12): 4187 - 4199, 2024.
doi:10.14778/3685800.3685836

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685836

1 INTRODUCTION

There are two predominant strategies for the efficient execution of relational queries in modern data management systems. Execution engines can either (a) completely remove the cost of query interpretation by generating executable code at runtime (*just-in-time* compilation), or (b) amortize the interpretation overhead by processing batches of data at a time (*vectorization*). While the best strategy for a given workload is still a debated topic (including hybrid approaches) [13], vectorization is generally adopted for processing large datasets due to its simplicity, lack of compilation overhead, and generated code efficiency [18] [23] [3] [6]. In a vectorized execution engine, computation is decomposed into a sequence of simple and concise operations over a batch of data - *tight loops* - providing more predictable memory access patterns and minimizing CPU pipeline stalls caused by cache misses and branch mispredictions. Vectorization enables CPUs to fully leverage out-of-order execution and SIMD instructions [22], while providing a programming paradigm that is more naturally translatable to highly parallel hardware accelerators like GPUs.

A vectorized engine is only as efficient as the loops it executes. If loops are carefully unswitched [28] to minimize CPU stalls, and batches are large enough to amortize the cost of dispatching to the next loop, near-optimal CPU usage can be achieved. For example, in order to sum two columns, a vectorized engine usually provides one version of the loop to sum a batch of each supported data type, and a dispatch mechanism (such as a virtual function call) to locate the correct loop at runtime. Moreover, to avoid nullity checks (branches) for cases when parameters are known to be not-null, two versions of each loop may be provided, one with, and one without the nullity check. In an *encoding-aware* engine where input batches can be arbitrarily encoded [8], different versions of each loop may also be provided to handle cases when parameters have a specific encoding, such as one loop for constant-encoded columns, one for flat, and one for dictionaries. Implementing this exponential combination of loops and taking the right trade-off between runtime efficiency and binary program size is a challenging task. Beyond coding skills, it requires developers to have a deep understanding of vectorized data processing, columnar memory layout, data encodings, and compilers, presenting a steep learning curve for new developers, and major challenges to organizations trying to scale teams and speed up engine development.

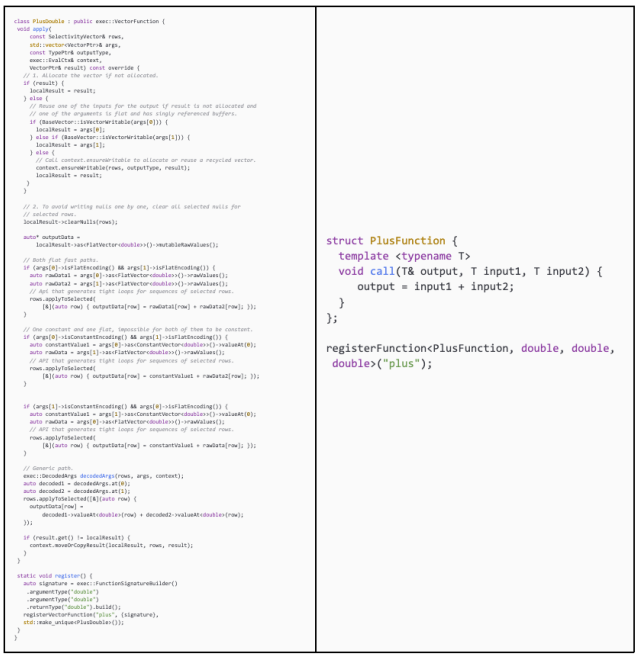
Due to the broad surface and sheer variety, scalar function authoring accentuates this problem. Popular engines like Presto [25] [27] or Spark [26] commonly expose hundreds of *built-in* scalar

functions, providing functionality such as manipulation of strings, arrays, maps, dates, timestamps, regular expressions, JSON, URLs, and many others [10], sometimes encompassing 1/3 to 1/2 of the entire engine's codebase [33]. In our experience building the Velox open source execution engine [18] [17] and adding support for prominent SQL dialects, we have observed hundreds of engineers across Meta and the open source community developing vectorized functions. While the concepts of encoding-aware execution, columnar data representation, and vectorized expression evaluation are well-understood by (a few) data system specialists, we have found them to be unintuitive and excessively complex for many developers; it resulted in a disproportionate amount of bugs caused by leaky vectorized APIs, and numerous performance inefficiencies.

To address this issue, we have developed a novel *simple function interface* (SFI) in Velox. SFI encapsulates complexity from vectorized scalar function development, hiding details of the columnar layout, and presenting developers with a simpler, conciser, and more natural row-based interface. SFI is comprehensive and allows developers to efficiently leverage advanced features such as nested and recursive parameter types, type variables, variadic parameters, and generic types, which are commonly used during scalar function authoring. By transparently applying performance optimizations such as loop unswitching, fast-path for common encodings, and efficient null handling, SFI commonly provides the same level of performance as their vectorized counterparts, while reducing the cognitive burden on developers, code verbosity and complexity, and likelihood of bugs. In fact, we have observed cases where functions migrated to use SFI present higher efficiency than their vectorized counterparts, due to missed optimization opportunities in the original vectorized implementation. To exemplify the development experience, Figure 1 contains a vectorized and an SFI-based implementation of the *plus()* function, where the left side merely illustrates the amount of code required. Both versions compile to similar executable programs and provide equivalent performance.

Today, more than a thousand functions have been written using SFI, implementing popular open source SQL dialects and internal domain-specific use cases. We have estimated that hundreds of developers from Meta and the open source community, with different levels of context in data processing, have used this interface to extend the behavior of data management systems. Furthermore, SFI is in active production use: through analysis of internal SQL batch workloads in Presto, we have found that expression evaluation alone encompasses about **15%** of CPU usage across clusters; migrating them to a state-of-the-art vectorized engine containing the optimizations (and complexity) discussed throughout this paper has improved that portion by a factor of **3.5x** on an aggregated level. We have also found that the ideas of SFI similarly apply to aggregate function development, although the details of that work will be described in a future paper. Ultimately, while this paper is presented in the context of the Velox library, we believe that the concepts of vectorization, columnar layout, and encoding-aware execution, as well as the trade-offs between code complexity, developer productivity, reliability, and efficiency, to be fundamentally equivalent and applicable to most modern vectorized execution engines.

The remainder of this paper is organized as follows. Section 2 presents the background, discussing main concepts around modern



```
class PlusDouble : public exec::VectorFunction {
 void apply(
     const SelectivityVector& rows,
     std::vector<VectorPtr>& args,
     const TypePtr& outputType,
     exec::EvalCtx& context,
     VectorPtr& result) const override {
   // 1. Allocate the vector if not allocated.
   if (result) {
     localResult = result;
   } else {
     // Reuse one of the inputs for the output if result is not allocated and
     // one of the arguments is flat and has singly referenced buffers.
     if (BaseVector::isVectorWritable(args[0])) {
       localResult = args[0];
     } else if (BaseVector::isVectorWritable(args[1])) {
       localResult = args[1];
     } else {
       // Call context.ensureWritable to allocate or reuse a recycled vector.
       context.ensureWritable(rows, outputtype, result);
       localResult = result;
     }
   }

   // 2. To avoid writing nulls one by one, clear all selected nulls for
   // selected rows.
   localResult->clearNulls(rows);

   auto* outputData =
       localResult->asFlatVector<double>()->mutableRawValues();

   // Both flat fast paths.
   if (args[0]->isFlatEncoding() && args[1]->isFlatEncoding()) {
     auto rawData1 = args[0]->asFlatVector<double>()->rawValues();
     auto rawData2 = args[1]->asFlatVector<double>()->rawValues();
     // API that generates tight loops for sequences of selected rows.
     rows.applyToSelected(
         [&](auto row) { outputData[row] = rawData1[row] + rawData2[row]; });
   }

   // One constant and one flat, impossible for both of them to be constant.
   if (args[0]->isConstantEncoding() && args[1]->isFlatEncoding()) {
     auto constantValue1 = args[0]->asConstantVector<double>()->valueAt(0);
     auto rawData = args[1]->asFlatVector<double>()->rawValues();
     // API that generates tight loops for sequences of selected rows.
     rows.applyToSelected(
         [&](auto row) { outputData[row] = constantValue1 + rawData2[row]; });
   }


   if (args[1]->isConstantEncoding() && args[0]->isFlatEncoding()) {
     auto constantValue1 = args[1]->asConstantVector<double>()->valueAt(0);
     auto rawData = args[0]->asFlatVector<double>()->rawValues();
     // API that generates tight loops for sequences of selected rows.
     rows.applyToSelected(
         [&](auto row) { outputData[row] = constantValue1 + rawData[row]; });
   }

   // Generic path.
   exec::DecodedArgs decodedArgs(rows, args, context);
   auto decoded1 = decodedArgs.at(0);
   auto decoded2 = decodedArgs.at(1);
   rows.applyToSelected([&](auto row) {
     outputData[row] =
         decoded1->valueAt<double>(row) + decoded2->valueAt<double>(row);
   });

   if (result.get() != localResult) {
     context.moveOrCopyResult(localResult, rows, result);
   }
 }

 static void register() {
   auto signature = exec::FunctionSignatureBuilder()
     .argumentType("double")
     .argumentType("double")
     .returnType("double").build();
   registerVectorFunction("plus", {signature},
   std::make_unique<PlusDouble>());
 }
}
```

```
struct PlusFunction {
  template <typename T>
  void call(T& output, T input1, T input2) {
     output = input1 + input2;
  }
};

registerFunction<PlusFunction, double, double,
 double>("plus");
```

Figure 1: *plus()* function implementation using the vectorized (left) and SFI (right), presenting equivalent performance.

vectorized engines, columnar layout, expression evaluation, and scalar function support. Section 3 presents an overview of the SFI discussing the main interfaces, challenges, and performance considerations of scalar function authoring in vectorized engines. Section 4 lists and describes details of the features supported in SFI, discussing how it supports primitive, nested, and generic types. Section 5 discusses some limitations of SFI. Section 6 points to related work, and finally Section 7 concludes this paper.

2 BACKGROUND

2.1 Vectorization

Vectorization [5] is the predominant technique for processing large scale datasets. In a vectorized execution engine, complex operations are decomposed into smaller (tight) loops, each executing simple operations over large batches of data. As much as possible, virtual calls, dynamic dispatching and even branches (*if* and *switch* statements) - or anything that could stall the CPU pipeline - are pulled outside of the loop (*loop unswitching*). Tight loops over large batches of data can be efficiently executed by allowing CPUs to fully leverage out-of-order execution and SIMD instructions, and reducing cache misses by relying on prefetching and more predictable memory access patterns.

Loop unswitching implies that many physical versions of the same loop must be available in the program. They are either explicitly spelled out by the programmer, or more commonly expressed using techniques like templating in C++ [29]. For example, to implement a vectorized operation that can sum two batches of either integers or floats, one needs to provide a loop that expects integers and one that expects floats, and at runtime decide which one to take (*dispatch*). Generalizing this idea, vectorization is the process

of decomposing larger logical operations into tight and efficient loops, and successively executing them. Intuitively, the section between loops is typically less efficient as it has less instruction-level parallelism and predictability. As long as the time spent processing loops is sufficiently larger to amortize the time dispatching across loops, CPUs can be efficiently used.

2.2 Columnar Layout

Vectorization relies on columnar data layout. In a columnar layout, data is organized as horizontal partitions (batches or rows), where each column is vertically represented by a structure commonly referred to as *array* [2] or *vector* [34]. We refer to them as vectors throughout this paper. Each vector may contain one or more buffers, and buffers may be shared across multiple vectors. Vectors may contain a buffer (bitmask) to represent the nullity (or validity) of each value. Data types that have fixed length, such as integers, floats, decimals, timestamps, and dates are often simply stored contiguously in a sequential buffer.

Variable-length types such as strings need additional metadata. In its simpler form, a string vector is represented by a *data* buffer containing the actual string contents, accompanied by an *offset* buffer containing where each string starts - the size of a string at *i* is the difference between the offset at *i+1* and the offset at *i*. More sophisticated string representations like *StringView* [15] store a string prefix along with the size/offsets to allow for faster comparisons, in addition to allowing shorter strings (usually up to 12 bytes) to be stored inline in the metadata/offsets buffer, thus improving memory locality.

Container types like arrays and maps are commonly represented in a similar way, having one internal and recursive vector representing the container elements from every row, and a buffer containing the offsets where each array starts. Maps are stored in a similar fashion, but instead containing two vectors, one for keys, and one for values. Modern systems may also use an alternate representation that keeps two additional buffers, one for *offsets*, and one for *lengths*, called *ListView* [16]. Despite carrying redundant information in the simple case, *ListViews* allow for more flexibility while processing columnar containers, like enabling non-contiguous ranges, out of order writes, and overlapping ranges. Rows/structs are commonly represented by a set of vectors, where each vector can recursively be of any of the types described above. Rows, arrays, and maps may also carry an additional nullity bitmask to represent the case when an entire container is null, as opposed to a particular element of the container being null.

2.3 Encodings

Columnar representations share the important property of storing values of the same data type contiguously, offering ample opportunities for more compact representations. For example, columns containing many repeated values (low cardinality) can be efficiently represented by a buffer containing the distinct values (the *alphabet*) and a buffer of integers mapping each row to the index in the alphabet; also known as **dictionary encoding**. Columns with many contiguous repeated values (common in sorted datasets) can be efficiently represented by an element, and the number of times it appears (the *run-length*); also known as **run-length encoding** (RLE). As a particular case of RLE, columns that represent a single distinct value (commonly used to represent literals and partition keys) can be represented by a single value and the column size; also known as **constant encoding**.

These encodings are often referred to as *recursive* or *cascading*, as they can conceptually be applied to (or *wrapped around*) columnar buffers of any data type; including ones that are already encoded using other schemes. Conversely, *leaf encodings* can be applied to specific data types (and hence cannot be applied recursively), such as frame-of-reference, bit-packing, and *varints* for integers [12], ALP [1], Chimp [14], and Gorilla [21], for floats/doubles, FSST for strings [4], and a myriad of other encodings schemes.

Encoding techniques have traditionally been applied for saving IO while either storing columnar datasets on disk, or transferring them through the network. Modern vectorized execution engines, however, have found that the cascading property of certain encodings (like dictionaries, constant, and RLEs) can also be conducive to more efficient data processing [18]. For example, operations like scalar functions can be applied only to distinct values (not to every logical row). Furthermore, operations that decrease the cardinality of a dataset, like filtering, can be achieved by wrapping a dictionary around the data, and only including the indices of records that survived the filter. Cardinality-increasing operations like unnests and joins can be executed by wrapping an RLE containing the size of each output run (or a dictionary) to the input data. This strategy allows datasets to be efficiently processed by only modifying indirection buffers (their wrappings), but without touching their internal data, as these can be larger when representing long strings and recursively defined containers.

To enable these optimizations, an **encoding-aware** execution engine needs to be implemented in a way such that (a) each vectorized operation is able to receive and produce a batch of arbitrarily-encoded data, like flat, constant, dictionary, and RLEs, and (b) that they are designed in a way such that the input data encoding can be leveraged for more efficient execution. While this enables a wide range of optimization opportunities, it highly increases the engine complexity, and the likelihood of bugs due to the exponential combination of possible code paths.

2.4 Expression Evaluation

Expression evaluation in vectorized execution engines works by decomposing larger expressions into several vectorized operations (or *sub-expressions*), one at a time, each of which consuming a batch of parameters and producing a batch of results that are consumed by the next sub-expression. For example, the expression "*a + func(b)*" is executed by first applying *func()* to an input batch of *b*, producing partial results, then applying the *plus()* function to *a* and the partial result from the previous sub-expression.

Expressions are usually represented by expression trees. Generally, nodes in this tree represent either (a) input columns, (b) scalar function calls, or (c) special expression forms that require special semantics, like conjuncts (AND/OR), *try* expressions, conditionals, lambdas, and casts. Literals are usually eliminated by constant folding subtrees before execution. Furthermore, each node in the tree often carries semantic metadata that can be used to optimize execution, such as determinism (*does it always return the same result*

for the same inputs?), and null propagation behavior (*does it always produce null if any of the inputs are null?*). For example, deterministic functions over dictionaries and constants can be applied over the alphabet only (or the constant value), and sub-expression execution can be completely skipped if only nulls are found when OR'ing the nullability bitmask for each input parameter.

The execution process consists of a recursive descent of the expression tree, passing down a mask identifying the active rows. Additional rows can be masked out as more sub-expressions return nulls (depending on their null propagation behavior). Therefore, at each sub-expression level, one needs to check and only execute the expression over the active rows from that batch. Allocation of buffers for partial results may present overhead if executed too often; as much as possible, vectorized expression evaluation engines try to recycle existing buffers using local memory arenas, or reuse buffers from input parameters that are not needed after that point. However, additional logic must be put in place to ensure that buffers that are being reused are indeed writable (not shared across other vectors), and materialized (in many cases vectors can be *lazy* and only loaded upon first use).

Lastly, vectorized execution of conditionals poses additional complexity to expression evaluation. The state-of-the-art [18] dictates that first the expression that decides which branch each row will take needs to be evaluated (the *condition*), then executing the rows that match the condition (the *then* branch), then rows that do not (the *else* branch). This logic can also be generalized to support multiple branches (*switch* statements). Since different branches will independently write to the same output buffer, sub-expressions need to be able to generate output values which are out of order (e.g, first writing even rows, then odd rows).

2.5 Scalar Functions

The logic applied by sub-expressions is predominantly composed of *scalar functions*. Scalar functions are functions that provide a 1:1 mapping between input and output; they take a set of parameters from one row, and produce one output result, as opposed to aggregate functions which may take many input rows and produce a single output. They implement most of the domain-specific logic made available to engine users, ranging from arithmetic operations, to string, json, date/time, regex, array and map manipulation, among many others. Execution engines usually provide APIs that allow users to add their own application-specific business logic, which can either be compiled as part of the same process (built-in/unfenced), or as an external UDF (fenced), depending on isolation and performance requirements.

Besides the function implementation, developers need to provide a function name, and a *function signature* describing the acceptable input parameter types and the output type a function produces; this process is referred to as *function registration*. The metadata produced by function registration is often stored in a *function registry*, which is used at type resolution time to ensure that user-supplied expressions are valid, that expected types match for each sub-expression, and to locate the corresponding vectorized code (tight loop) to dispatch to for each batch. Oftentimes, functions may be able to receive a large (sometimes unbound) number of types as parameters, so function signatures in modern execution engines must support *type variables* in addition to concrete types, e.g. *array_min(array<T>) -> T*, such that the type returned by the function is dependent on its parameter types.

2.6 Velox Engine

Velox is an open source unified execution engine created by Meta [17]. Velox is implemented as a C++ library that can be integrated with existing data management systems, providing efficiency wins due to its state-of-the-art vectorized engine. Velox also provides reusability benefits, as features and optimizations can be implemented once and be (re-)used across multiple engines. As an execution engine component, Velox does not provide a language or global query optimization capabilities; rather, it takes a fully optimized query plan as input and executes it using the local host resources. As such, Velox implements the execution layer as described in the composable data management stack [19]. In addition to traditional SQL analytic query processing, Velox is also currently integrated into stream processing, low-latency interactive use cases, feature engineering, data preprocessing for ML, database and datawarehouse ingestion, timeseries processing, log messaging, with a total of about a dozen data systems at Meta [7], providing efficiency wins, more consistency across engines, and engineering efficiency through reusability.

At its core, Velox is a vectorized engine that fully embraces encoding-aware execution and makes extensive use of dictionaries, constants, lazily loaded buffers, SIMD execution, and a variety of other vectorization techniques such as the ones described above for efficient expression evaluation [20]. Velox provides the core vectorized operators and frameworks, and exposes extensibility APIs to allow developers to fully customize its behavior and implement a specific semantic. The scalar function API, the main focus of this paper, is the most commonly used extension point. Velox provides implementations of Presto and Spark functions using the SFI and vectorized scalar functions interface as part of the library, allowing developers to achieve compatibility with PrestoSQL [11] and SparkSQL [26].

3 OVERVIEW

This section presents an overview of scalar function authoring in vectorized engines, highlighting some of their most common efficiency optimizations, the complexity they bring, and other development pitfalls. It also presents a new simple function interface (SFI) built with the purpose of encapsulating this complexity, discussing how SQL and C++ types are mapped, and how SFI automatically generates vectorized code using C++ templates and metaprogramming.

3.1 Scalar Function Authoring

In order to add a scalar vectorized function to an engine like Velox, developers have to implement a vector function that receives vectors as input parameters, and produces results as output vectors. Listing 1 illustrates the usual components of a scalar function API in a vectorized engine.

- *apply():* a virtual function that will be called to dispatch each batch of columnar data. Note that while this is a virtual

function call that may stall the CPU pipeline, it is only called once per batch.
- *rows:* a selectivity vector that represents the current active rows.
- *args:* input parameter vectors, which could be encoded in arbitrary ways.
- *outputType:* the return type of the function.
- *context:* additional expression evaluation and query plan context, metadata, and access to other query-wide structures.
- *output:* the output vector. The output vector may be already allocated or not, and functions are free to produce any type of encoding.
- *register():* in addition to the function implementation, authors also have to register it by associating the function with a name and a signature. For example, the function above has the name *plus*, and a signature *(double, double) -> double*.

```
class PlusDouble: public VectorFunction {
  void apply(
      const SelectivityVector& rows,
      std::vector<VectorPtr>& args,
      const TypePtr& outputType,
      EvalCtx& context,
      VectorPtr& output) {}
};
```

Listing 1: A vectorized API for scalar function.

This relatively simple API enables the expression evaluation engine to generate efficient vectorized code. However, in order to fully leverage its potential, developers are required to have deep context on the concepts discussed below.

Encodings. In an encoding-aware engine, operators, expressions, and other processing kernels are allowed to produce data using any type of encodings, and consequently, may also receive input vectors which are encoded in arbitrary ways. For example, to implement the *plus()* function described above with maximum efficiency, one needs to consider the input encodings: if one of the inputs is constant-encoded, a special loop can be provided to hold one value in a local variable (register) and iterate through the other vector. Other specialized loops may be provided for cases where one (or both) of the input vectors are dictionary encoded, in addition to a base case where both parameters are flat. As described in Section 2, the tighter the loops, the higher the efficiency. CPU pipelines will less likely be stalled, the CPU will issue cache misses earlier, predict branches more accurately, and compilers will generally have a better chance of autoSIMD'izing loops. However, the developer authoring the function is responsible for providing the unswitched loops, increasing the development burden and likelihood of bugs that may only get triggered in very specific situations.

Columnar data layout. Vectors of nested types have complex columnar layouts, as they can be arbitrarily combined. For example, a vector representing a column of the type *map(array(integer), row(integer, double))* will be composed of six vectors: one map, one array, one row, two integers and one double. Each vector may contain their own null buffer and be encoded in arbitrary ways, such as dictionaries and constants.

Null handling. Developers are also responsible for respecting the input selectivity vector and only evaluating active rows. Other than wasting compute, evaluating functions over inactive rows may produce inaccurate results; for example, rows may have already been evaluated by other branches in conditional execution (if/switch). As an optimization, loops are also usually unswitched based on whether all rows are active (to avoid the check for every row), or the base case when there exist both active and inactive rows. Additionally, developers need to be aware of the function's *null behavior*; if *default*, then any nulls in the input must result in null output, so output nulls are often automatically set by the expression evaluation engine. Conversely, if *non-default*, then developers are required to explicitly set the null buffer of the output result.

Output. Scalar function APIs need to support cases where a function call needs to write to a vector that is already allocated, to support vectorized conditional evaluation of IF and SWITCH statements. Additionally, since expression evaluation APIs are commonly used in multiple parts of a relational evaluation engine, it needs to be flexible to allow operator developers to fully control vector allocations and reuse. This means that when writing scalar functions, developers need to be aware that the output vector may already be pre-allocated, and only allocate a new vector in case one was not provided. Furthermore, it is possible that the vector provided to the function may not be writable (not singly-referenced) or have a different encoding type than the one the developer intended to return. For example, when evaluating a conditional, the sub-expression that evaluates the first branch (the *then* branch) may produce a dictionary - or even a constant. The second sub-expression (the *else* branch) will receive the dictionary (or the constant) as the output buffer, with a different set of active rows.

Exceptions. Scalar function developers also need to be aware that while it is generally adequate to stop evaluation once an exception/error is found, at times exceptions may need to be swallowed/ignored, e.g. if the sub-expression being evaluated is within a *try()* expression, or during conjunct evaluation. Moreover, as a rule-of-thumb, throwing exceptions during vectorized evaluation should be discouraged as exception handling is generally inefficient in languages like C++. However, this may be hard to enforce as scalar functions often leverage third-party libraries that can throw exceptions themselves, e.g. regex and json parsing libraries.

Considering the aspects discussed above, writing a highly efficient vectorized function requires not only deep context on vectorized processing, but also a large amount of code. To reduce the code verbosity of scalar functions, internal abstractions are usually provided to wrap some of the common logic. For example, (a) selectivity vectors may have helper APIs to help developers apply logic (lambdas) only to active rows, (b) abstractions can be provided to encapsulate error handling logic (either capture or throw/cancel), (c) helper functions that ensure output vectors are allocated and writable (via copy-on-write if not singly-referenced), and (d) decoding abstractions may be provided to expose a logically consistent API over arbitrarily-encoded buffers, at the cost of some performance - loops that are not as *tight*. While these abstractions help reduce the amount of boilerplate code, developers still need to understand exactly how and when to use them.

3.2 A Simpler Function API

As expected, considering the complexity added by vectorized APIs, we have found that developers can more easily reason about row-based APIs, or the ones that take one row in, and produce a single scalar output value. Therefore, to simplify the authoring experience in Velox given the large developer base, we have created a **simple function interface** (SFI) that allows developers to express their function logic as a single row-based function. As an example data point, 8 bugs were found in a single function (*map_from_entries()*) [30] implemented as a vectorized function in Velox before SFI was created, but no bugs were found in the similar and arguably more complex function *multimap_from_entries()* added later on using SFI [31].

Considering SFI uses C++ metaprogramming and template expansion techniques, it can maintain *the same level of performance as their vectorized counterpart implementations*. In fact, we have observed that in some cases a vectorized function migrated to SFI, besides having a more concise and easier to understand codebase, also provided better performance because the SFI framework automatically enables optimizations that may have been overlooked by the original author. SFI has the following properties:

(1) **Easier to write.** SFI hides the complexity inherent to vectorization and abstracts details of the columnar layout, lowering the cognitive burden on developers and flattening the learning curve.
(2) **Easier to read and review.** SFI removes much of the boilerplate code, leaving the function implementation to focus on the expected functionality, thus making it more concise and easier to reason about.
(3) **More reliable.** Considering SFI encapsulates complexity and reduces duplication, the resulting code is easier to test and less error-prone.
(4) **Efficient.** SFI leverages C++ template expansion and metaprograming techniques, relying on modern compilers to fully inline, auto-SIMDize, and generate code that is as efficient as their vectorized equivalent implementation.
(5) **Expressible.** SFI allows developers to use advanced features commonly needed in scalar function development, such as nested parameter types, type variables, variadic parameters, and generic types.

The code sample shown in the right-hand side of Figure 1 illustrates the API. In SFI, a struct with a *call()* function needs to be provided by the author. The *call()* function is responsible for performing the operation over a single row, and writing a single scalar output value through the reference received as the first argument. The *call()* function may (a) return *void*, in which case it signals the expression evaluation engine that it will never return a null value, (b) return a *boolean* dictating whether the return value is null or not, or (c) return a *status* object, allowing it to also return errors without having to explicitly throw and catch exceptions. All these signatures are transparently supported using C++ metaprogramming.

registerFunction() is responsible for registering the scalar function into the function registry, but is also responsible for instantiating the C++ templates (generating the vector function). It receives the struct that implements the *call()* function, followed by the output and input types as template arguments (in this case it returns *double* and receives two *double* arguments). Within the registration function call, a vector function is automatically instantiated through a **SimpleFunctionAdapter** [32] using template expansion. The pseudocode shown in Listing 2 is used to expand the simple function into a vector function implementing the *apply()* method discussed before.

```
template<typename Func,  typename TReturn,
         typename... TArgs>
class SimpleFunctionAdapter : public VectorFunction {
  void apply(...) override {
    for (...) {
      Func().call(out, input1, input2, ...)
    }
  }
};
```

Listing 2: *SimpleFunctionAdapter* pseudo-code.

Naturally, modern compilers aggressively inline the *call()* function (no runtime dispatch), generating executable code that is similar to its vectorized counterpart definition. Section 4 describes in detail how SFI automatically applies many of the vectorized optimizations described in Section 3.1.

3.3 Type System

When a simple function is registered, its output and input types need to be specified as template arguments using C++ types. While there is a trivial 1:1 mapping between C++ and SQL types for primitive types such as integers with different precision, doubles, floats, and booleans, other container and string types do not have direct counterparts in C++ since they have special vectorized representations (as discussed in Section 2.2). For these cases, custom C++ objects are provided, such as *Varchar* and *Varbinary*, in addition to templated variants to represent nested types, like *Array<integer>*, *Map<integer, boolean>*. To prevent an extra copy while processing these types (for example, to avoid copying vectorized strings into a C++ *std::string* for each row, or an array into an *std::vector*), custom C++ proxy types are designed to allow for efficient writes and reads of the underlying columnar vectors, while presenting a familiar std-like interface for developers. Proxy types are further discussed in Section 4.2.

Table 1 shows how SQL types are mapped to C++ types, and what are the read (argument type) and write (output type) associated with each. For primitives there is a trivial 1:1 mapping; *i.e,* in the *plus()* function above, *double&* is used to represent the output and *double* is also used for the input. However, SQL types like *Varchar* and *Varbinary* need proxy types that can be used as input parameters (*StringView*) and output (*StringWriter*). The convenience macros *arg_type<X>* and *out_type<X>* are provided to map a simple type into its input and output proxy counterparts.

3.4 Simple Function Adapter

The simple function adapter is the component responsible for converting the simple row-based function provided by the developer into an efficient vectorized loop. The adapter receives the simple function struct as template parameter, along with its input and output types. Internally, the adapter makes use of *vector reader* and *vector writer* APIs, which are abstractions that hide columnar layout details and are able to create read and write proxy types for

Table 1: SQL, C++, and proxy type mappings.

SQL Type	C++ Types	Input proxy	Output Proxy
TinyInt, ..., BigInt	int8_t, ..., int64_t	int8_t, ..., int64_t	int8_t&, ..., int64_t&
Real, Double	float, double	float, double	float&, double&
Boolean	bool	bool	bool&
Varchar	Varchar	StringView	StringWriter
Array<V>	Array<V>	ArrayView<V>	ArrayWriter<V>
Map<K, V>	Map<K, V>	MapView<K, V>	MapWriter<K, V>
Row<T_1, ..., T_n>	Row<T_1, ..., T_n>	RowView<T_1, ..., T_n>	RowWriter<T_1, ..., T_n>
T	Generic<X>	GenericView	GenericWriter
T, ...	Variadic<T>	VariadicView<T>	N/A

a particular record in the vector. Therefore, *VectorReader<T>* and *VectorWriter<T>* create *arg_type<T>* and *out_type<T>*, respectively.

On a high-level, the simple function adapter executes the following steps:

(1) For each input argument, a vector reader is created. For the *plus()* function described above, two readers are created, one for each of the double inputs.
(2) If the result vector is already allocated, it ensures it is writable (singly-referenced and flat encoded); if not, a new result vector is allocated. Then a vector writer is created for the output type.
(3) For each active row, vector readers and writers are used to get input values (or proxies) and output writers for that row, which are passed to the *call()* function.
(4) The nullity value returned from the *call()* function is written to the nulls buffer in the output vector.

In reality, the simple function adapter contains an extensive set of optimization to improve **performance** of the generated loops, in addition to features that allow developers to express a plurality of usage scenarios (such as nested types, variable types, variadic parameter list, generic types, and more), increasing their **productivity**. These features are discussed in detail in the next section.

4 IMPLEMENTATION DETAILS

This section describes implementation details of the SFI. It dives into challenges faced to efficiently support different SQL logical types, flexible function signatures, and performance optimizations that are transparently enabled for developers. We start by describing functions that operate over primitive types. Next, strategies for efficiently handling nested types are presented (due to their ubiquity in modern workloads), followed by discussions about nullity handling, and methods to efficiently support variadic and generic types.

4.1 Primitive Types

As shown in Table 1, when operating over primitive types, the input and output types passed to the *call()* function match the native representations used to store data in vectors. For example, a vector of doubles is stored in an array of doubles. The *call()* function receives direct references to the output array items for outputs, and copies for input. It is important to avoid any transformation overhead to the data, especially for functions which have a very concise body (like simple arithmetic functions).

In order to provide maximum efficiency, it is important to consider the following aspects. First, VectorReaders are wrappers that abstract the internal data encoding. As such, reading data of a generic vector may not be a direct memory access, since it involves checking the encoding type of the vector. Ideally, if the input vectors have flat encodings, it is faster to switch the encoding check out of the loop (*loop unswitching*). Second, setting the output nullity value involves writing to a bit vector, for each row. In reality, it might also involve checking if the bit vector is allocated or not; for example, in Velox the non-existence of a null buffer means all rows are non-null. We have empirically found that the additional checks and logic required by the items above usually prevent modern compilers to automatically apply SIMD to the generated code.

Fast-paths. Successively applying *call()* for each row function is the hottest path in scalar function execution. Therefore, to provide loops which are as tight as possible, optimized custom cases (*fast-paths*) can be provided for common scenarios. For example, it is common for arithmetic functions to have two inputs which are both flat and null-free, having all input rows active/selected. For these cases, a fast-path execution loop can be provided. Given the conciseness and simplicity of these unswitched loops, compilers can usually inline and automatically apply SIMD to the generated code. Through experiments, we have found fast-paths for the *plus()* function, for example, to provide orders of magnitudes speed up over the basic adapter loop described above.

Furthermore, there are many other conditions under which fast-path implementations are beneficial. For instance, it is common to find cases where one input is flat and one is constant (e.g, *plus(a, 5)*)), or cases where all inputs are flat, but not all rows are active; or cases where the function might return nulls. The adapter is written in such a way as to automatically create many of these fast-paths (hot tight loops) under different conditions, while automatically handling the following optimizations:

Buffer reuse. When the output type of a function matches one of the input types, and the input buffer is not needed after function invocation, the input buffer can be used for results instead of allocating a new buffer. For example, in the expression *plus(a, b)*, if *a* is stored in a flat vector which will not be needed after the invocation of the plus function, then *a*'s buffer may be used to store the computation result.

Bulk null setting. Writing each bit to the output null buffer can be an expensive operation, particularly for cases when the function body is very concise. Since in most cases functions return not-null values, SFI optimizes for this case and bulk sets nulls to not-null by default.

Null setting avoidance. The adapter can also statically infer if a function never generates null. In the simple function interface, if the *call()* function return type is *void*, it means the output is never null; if it is bool, then the function returns true for not-null and false for null. When the function is known to be non-null generating, the code path that sets the nullity can be completely removed from the innermost hot loop (note that previous optimizations already set all nulls to not-null). This is crucial to allow compilers to auto-SIMD.

Constant inputs pre-processing. SFI provides an API for developers to pre-process constant inputs of functions, hence avoiding repeated computation. If provided, an *initialize()* function is called before the loop starts, allowing developer to pre-process constant

inputs, For example, a regex function with a constant pattern could use the *initialize()* function to compile the pattern expression only once per thread of execution.

Encoding-based fast-path. The generic way of reading a vector of arbitrary encoding is to use a general-purpose decoding API to simplify data access (also called a decoded vector). Even though decoded vectors provide a consistent API and make it easier to handle arbitrarily encoded input data, they pose overhead; each time an input value is accessed, the encoding of the vector needs to be checked, potentially triggering an indirect access (for dictionaries, for example). For operations where the function body is concise (like *plus()* or *minus()*, for example), this overhead is non-negligible.

To optimize for these cases, encoding-based fast paths are added, creating inner loops that are tight, free of encoding checks, and auto-SIMD'izable. A downside of this optimization is that the core loop is replicated multiple times, increasing the generated program size and compilation times. In general, the number of times the loop is replicated is n^3, where n is the number of arguments, and 3 is the number of encodings. To control the program size increase, we only apply this optimization when all input arguments are primitives, and the number of input arguments is <= 3.

To counterbalance the effect of fast-path optimizations on the program size, we have created a *pseudo-specialization* mode that does not affect the program size, but reduces the overhead of decoding to a single multiplication per argument. This mode is enabled when all the primitive arguments are either flat or constant, reducing the number of loops required in the program. When the input vector is constant, the value can always be read from index 0 of the values buffer (the only index available for a constant-encoded vector), When the vector is flat, the value can be read from the row index. This can be achieved by assigning a factor to either 0 or 1 and reducing the decoding operation per row into a multiplication with that factor, adding a single instruction to the hot path. Note that such a multiplication does not prevent auto-SIMD. The code in Listing 3 illustrates the idea.

```
if (allArgsConstantOrFlat()) {
  auto factor0 = args[0].isConstant()? 0 : 1;
  auto factor1 = args[1].isConstant()? 0 : 1;
  ...
  for (i in selectedRows) {
    SimpleFunc().call(
      output[i],
      rawData0[factor0*i],
      rawData1[factor1*i]);
  }
}
```

Listing 3: Illustration of pseudo-specialization for a function with two arguments.

To illustrate the effectiveness of these optimizations, we evaluated the expression *clamp(0.05*(20+one_hot(c0, 1)), -10, 10)*, which is an example of a common pattern in ML preprocessing use cases. The pseudo-specialization makes the program **2x** faster, while the complete specialization makes the program around **4x** times faster.

4.1.1 Boolean Types. Boolean columns are usually represented by a bit vector, though a boolean variable is used to represent values in the input and output. Therefore, boolean values need to be packed/unpacked for each row, limiting potential optimizations of pure boolean to boolean computations, like AND, OR, or XOR or other bitwise operations that could be more efficiently performed using the underlying bit vector. While it is possible to introduce separate APIs that expose the bit vectors for more efficient simple implementations, they have not been implemented yet in Velox because bitwise scalar functions do not exhibit high usage in our internal workloads.

4.1.2 Varchar and Varbinary. Strings are commonly represented in vectorized engines using the *StringView* format described in Section 2.2. A StringView vector usually holds shareable ownership over the buffer containing the actual string data, allowing multiple vectors to share the same string contents without copying the data. In SFI, StringViews are passed to simple functions as inputs, while StringWriters are used for string output. A StringWriter object directly writes data to the underlying output vectors, without storing data in intermediate representations (such as *std::string*).

StringWriter tracks the buffer capacity, exponentially growing the size of the output string. StringWriters automatically write data to the output buffer, creating the StringView metadata pointing to the written string contents once the function invocation is complete. The code in Listing 7 illustrates an example of a function that concatenates input strings. Inputs are represented as StringView and the output is a StringWriter that directly manipulates the underlying vector. Following are crucial optimizations for string processing, considering their ubiquity in scalar function authoring.

ASCII fast-path. Functions with string inputs can usually be further optimized when their inputs are known to be ASCII-only. For example, *length()* can be trivially calculated for ASCII strings by simply returning its physical size in bytes, while for generic strings potentially containing non-ASCII inputs, a linear computation is required. To allow for this type of optimizations, SFI enables developers to define a *callAscii()* function that is automatically called when all the string input arguments are known to be ASCII-only. Through microbenchmarks, we have observed that a *substr( )* ASCII-only fast-path performs **2.8x** faster than a general implementation.

ASCII behavior. Expression evaluation engines need to run ASCII detection code to decide whether a string buffer is ASCII-only or not. To avoid this overhead, SFI allows developers to specify the ASCII behavior of a function, informing the engine about whether a function is guaranteed to produce ASCII-only output for ASCII-only input - which is the case in all examples we have encountered.

Zero-copy optimization. In many cases, string functions can be efficiently implemented by performing a shallow copy of the StringView objects (the pointers and string sizes) and sharing the buffer containing string contents. This optimization can be used in functions such as *substr()*, *trim()*, *split()* and similar. To enable it, SFI allows developers to specify whether the output string vector should share the string buffers of the output with any of the input parameters. The zero-copy optimization provides another **2x** speedup over the ASCII-only fast-path version of the *substr()* function discussed above.

4.2 Nested types (Maps, Arrays, and Structs)

Similarly to strings, maps, arrays, and structs/rows have a more complex columnar representation. For example, a map in Velox consists of two nested vectors (one storing *keys* and one storing *values*) and two buffers (one for the *lengths* and one for the *offsets* of each row). In addition, not only the map's children can be arbitrarily encoded, but the map itself might be wrapped in a dictionary or other encoding. This complexity makes function authoring over complex types non-trivial and error-prone.

SFI improves authoring by providing efficient proxy types to read and write columnar vectors representing nested types [24], encapsulating complexity and without materializing elements into temporary containers like *std::vector* and *std::unordered_map*. The following subsections discuss strategies to support nested types while maintaining performance and simplicity.

4.2.1 Nested Input Types. The simplest approach to support nested types as inputs to scalar functions is, for each row, to copy elements from the columnar buffers into standard containers, like *std::vector*, *std::unordered_map*, *std::tuple*, and *std::optional* (to represent nullable values). Despite the simplicity and providing APIs which most developers are familiar with, this approach has two key inefficiencies:

(1) *Unnecessary copy.* All data must be first read from the original columnar buffer, decoded, written into the temporary container, then potentially read again.
(2) *Eager materialization.* Input vector elements are decoded and copied into the temporary container before calling the function - even if they are not needed. For example, the *length()* function only needs to access the size of each array; the *subscript()* function only needs to access one element; *array_first()* only needs to access the first element.

In order to avoid this overhead, proxy type abstractions are provided for inputs. To maximize efficiency, the view types should be trivial to construct and lazy (do not materialize the underlying data unless they are explicitly accessed by the function author). Velox provides the *ArrayView*, *MapView*, and *RowView* proxy types, each providing std-like APIs to be developer-friendly.

ArrayViews store the length of the array, its offset within the elements vector, and a pointer to the elements vector. Only when an element is accessed in the ArrayView, an *OptionalView* is created, containing the decoded index of the accessed element and a pointer to the vector containing the value. Similarly, only when the developer calls *has_value()* on the *OptionalView* accessor, the nullity of the value is checked, and only when the developer calls *value()*, the value is accessed. This strategy reduces the memory footprint as less data needs to be read from memory into caches, and minimizes cache misses. While creating such temporary objects may sound like potential overhead, compilers are usually capable of inlining all required functions and generating efficient tight loops. The goal is keeping the authoring simple, but achieving the same level of performance of a complex vectorized function implementation.

Performance. While developing view types, we implemented a baseline version based on std containers for performance comparison. Figure 2 shows the results. The average speedup for arrays was around **2x**; we have also found the speed up for maps to be higher (**+10x**), because materializing the intermediate representation involves hashing all elements while constructing hashmaps. The overhead of materialization for deeply nested types was also high, as illustrated by *row_arrays_sum()*.

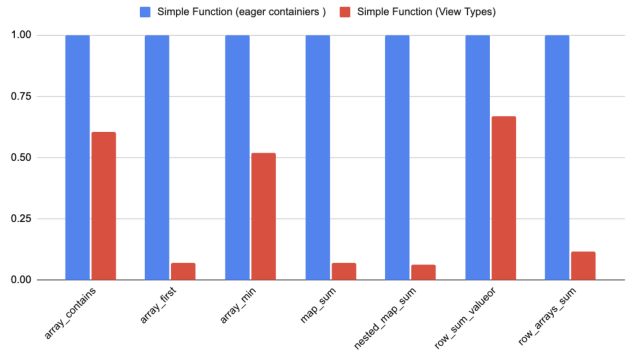


Figure 2: Performance of view types compared to eager materialized for different functions.

4.2.2 Nested Output Types. Similar challenges exist with functions that return nested types. The trivial way to support nested types is to write the single row results into a temporary std container, then copy the results into the columnar buffer at the end of the function call. This approach has the overhead of *double-writing*; data is written twice, once in the temporary container, and then again when copied to the columnar buffer. To avoid this overhead, efficient modern vectorized engines must provide writer proxy types, which directly manipulate the underlying output columnar buffers. Map writers also avoid unnecessary sorting and hashing of map keys. Velox provides *ArrayWriter*, *MapWriter* and *RowWriter* as output proxy types. For better user experience, writer types are designed to have an API close to standard C++ containers.

To illustrate the API and behavior, consider the example of a function that constructs and creates an array $[0, n-1)$ for each input n, shown in Listing 4. outerArray is an *ArrayWriter* object, and the *push_back()* method directly writes data to the underlying vector.

Incremental resize. In a vector function implementation, the output buffer size can usually be computed by a first pass on the data, followed by a single large allocation. While this is not possible in row-based APIs like SFI, writer objects keep track of the output buffer capacity and re-allocate it (increasing exponentially) as needed, resulting in resize overhead amortized across batches. APIs to calculate the exact output buffer size upfront can be added if the resize overhead is not amortized, leveraging the lazy materialization capability of read proxy types.

Moving elements. Moving elements between input and output containers is a common pattern, used in functions such as *array_concat()*, *flatten()*, *array_normalize()* and many others. To make these operations more convenient and performant, a specialized API may be provided by the engine. For example, in Velox we provide the *add_items()* and *copy_from()* APIs to allow developers to efficiently move elements across containers.

```
struct MakeArray {
 void call(ArrayWriter<int64_t>& writer, int64_t size) {
   for (int i = 0; i < n; i++) {
     arrayWriter.push_back(n);
   }
 }
};

struct MakeArrayOfMaps {
 void call(ArrayWriter<Map<int64_t,int64_t>>& writer) {
   auto& mapWriter1 = writer.add_items();
   mapWriter1.emplace(1, 2);
   mapWriter1.emplace(2, 4);
   // Not allowed to add to mapWriter1 after this point.
   mapWriter2.emplace(-1, -2);
 }
};
```

Listing 4: MakeArray and MakeArrayOfMaps functions written in SFI.

Further, we automatically improve the performance of copying large amounts of elements by providing fast-paths for flat and null-free buffers and implementing some of the optimizations discussed in Section 4.1. Through microbenchmarks, we have observed these optimizations to provide a 20-30% performance improvement while executing functions like *array_concat()*, which combines several arrays into a single one. In addition, when elements are strings, SFI can automatically detect and capture the underneath string content buffers in the output, preventing deep string copies. We found this optimization to provide another 30-40% speed in the *array_concat()* function with *Array<Varchar>* inputs.

In-order elements writing. Because writer types modify the underlying vector directly, it is not always possible to allow out-of-order element writes, or provide std-like APIs, particularly for cases of deeply nested types. For example, for a function returning an *Array<Map<...>>*, it is not possible to add three maps then write to them concurrently, as writing to a previous map would require subsequent elements to be rearranged. Furthermore, the standard *push_back()* API cannot be respected as it requires pre-creating an intermediate map first, invalidating the in-place mutation requirement for efficiency.

The function *MakeArrayOfMaps* in Listing 4 exemplifies the provided API. Calling *add_item()* on the ArrayWriter returns a MapWriter and, since the elements of the maps are primitives, the *emplace()* API can be used to add elements.

Figure 3 shows the performance effect of using writer types in contrast to using temporary std containers. Mutating the underlying vector directly enhances the performance significantly, by almost **4x** for arrays, and more for maps due to the hashing cost of the intermediate container used.

4.3 Generic Types

In many cases, due to the recursive nature of nested types, scalar functions may need to be able to operate over an infinite number of type sets. For example, the *cardinality()* function has the signature *array(T) -> integer*, where *T* can represent any type, including nested types defined recursively. Similarly, functions such as *equal(T, T) -> bool*, may accept any two inputs of the same type as long as they are *comparable*. Functions with generic inputs may also need to return generic output types. A trivial example is the *subscript()* function, which has the following signature: *array(T), int -> T*.

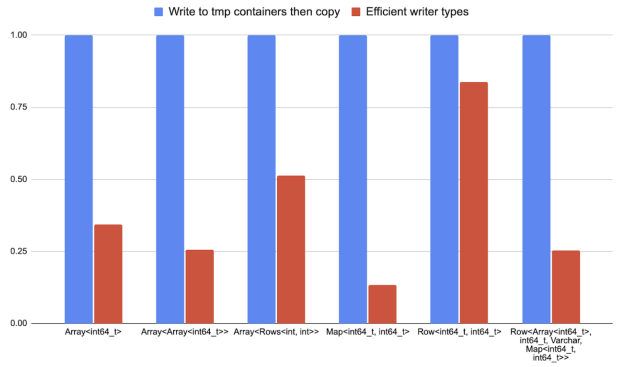


Figure 3: Performance of writer types compared to an implementation that stores data in temporary containers.

To support these features in a generic manner, type resolution needs to happen in two layers. First, function signatures need to provide support for type variables to allow scalar functions that are valid for an infinite number of parameter types to be registered. Beyond allowing arbitrary parameters, type variables enable developers to place restrictions on the relationship between types. For example, *equal(T, T) -> bool* accepts any type as *T*, but places a restriction that the first and second parameters need to be of the same type; i.e, *equal(integer, float)* fails type resolution. Similarly, *array(T) -> T* adds a restriction that the output type is the same as the elements of the input array.

Secondly, once type resolution resolves type variables into concrete types, the vectorized engine needs to find the function implementation to dispatch to at execution time. For common functions, this means locating the tight unswitched loop instantiated for that particular set of parameters, for instance, *equal(double, double)*. The remainder of this section discusses strategies to efficiently support generic types while still providing a simple and intuitive row-based API for function authors.

4.3.1 Generic Inputs. In Velox, support for generic types is enabled using the *Any* and *Generic<T>* types. While Any allows developers to express generic types without restrictions, *Generic<Tx>* allows developers to express relationships with other parameter types. For instance, *equal(Generic<T1>, Generic<T1>)* expresses that the function expects two inputs of the same type. Using this API, a *cardinality()* function that supports generic arrays and maps as parameters can be trivially defined as shown in Listing 5

```
struct Cardinality {
 void call(int64_t& out, ArrayView<Any>& arrayView) {
   out = arrayView.size();
 }

 void call(int64_t& out, MapView<Any, Any>& mapView) {
   out = mapView.size();
 }
}
```

Listing 5: Cardinality function written in SFI.

GenericViews are small and efficient objects that only lazily instantiate its internal elements, similar to the complex type view objects discussed in Section 4.2.1. In most cases, they are fully inlined in the body of the caller function by the compiler without

adding any overhead. In the example above, the GenericView object is not even constructed since ArrayViews elements are lazily instantiated, and the function never accesses its elements - only its size. Therefore, through microbenchmarks we have observed that the runtime of the simple *cardinality()* function above is the same as the runtime of the vectorized cardinality function, while the number lines of code required to express the function is more than an order of magnitude lower, highly reducing its authoring complexity.

To support a larger extent of functions, Generic objects should also be *comparable*, *orderable*, *hash-able*, and *cast-able*. Comparable property is needed to implement *equal(T, T)* functions in a general manner; hash-able property types are needed for functions such as *array_duplicates()*, *array_intersection()*, *array_frequency()*, and many others that require a small temporary hash table; orderable property is needed to implement functions like *array_sort()*; castable property is needed to implement functions like *to_json()* and *to_string()* which may need to cast nested types recursively, in addition to being used to efficiently implement the *copy_from()* operation is discussed in subsection 4.3.2.

Primitives fast-path. The generic comparison, hashing, ordering, and casting APIs provide a simple and convenient interface that recursively works for any types, but may add overhead due to a dynamic dispatch (or a large switch statement) per row. Although this overhead can be amortized for nested types across the cost of processing each container element, this cost is particularly visible for primitive types, when the cost of the operation itself is very low. For example, since comparing two primitive types can be done in single instruction, a dynamic dispatch per row is prohibitively expensive.

To avoid this overhead, developers are able to register primitive fast-path implementations of functions that use generic types. The idea is simple: a generic base implementation is provided, which is slower (for primitives) but that works for any type, in addition to specialized functions (loops) for primitive types, since these are more performance-sensitive. For example, for a complete implementation of the equal function, a general implementation based on a generic (but comparable) type can be provided as a catch-all that works for any nested type, in addition to fast-path implementation for every primitive type, such as integers and floats or different precisions. In our experiments, a primitive type fast-path implementation for the *eq()* function provided a 2x speedup by eliminating the type check per row inherent to the generic type support.

Conflicting function resolution. To enable the behavior described above, a vectorized engine needs to allow multiple versions of the same function to be registered, and define the resolution order between them. For example, with generic and variadic type support (discussed in Section 4.4), all functions below are valid and able to handle a single input argument of type integer:

```
void call(bool& out, int32_t input);
void call(bool& out, VariadicView<int32_t> input);
void call(bool& out, GenericView input);
void call(bool& out, VariadicView<GenericView>input);
```

For performance, during function resolution the engine needs to ensure that the least generic version of the function is selected - the tighter loop. For example, *int32_t* is less generic than *Variadic<int32_t>*, which is less generic than *Any*, which is less generic *Variadic<Any>*.

In addition to Generic and Any, SFI provides *Comparable<Tx>* and *Orderable<Tx>* which are similar to Generic except that they limit the types that can be represented by variables to those satisfying the properties.

4.3.2 Generic Output. It is common for functions that operate over generic types to also return generic types, e.g, *array_flatten()*, *subscript()*, *array_trim()*, and similar functions. In Velox, generic outputs are represented by *GenericWriter* objects. By analyzing existing scalar function implementation, we have observed that the most commonly used APIs in GenericWriter are *copy_from(GenericView)*, to copy an element from the input, or simply assigning an input *GenericView* to a *GenericWriter*. The code in Figure 6 shows how the function *array_flatten()* can be written using SFI.

```
struct ArrayFlatten {
 void call(out_type<Array<Generic<T1>>>& out,
      arg_type<Array<Array<Generic<T1>>>>& arrays) {
    for (auto& array : arrays) {
      if (array.has_value()) {
       // The loop below can be replaced with out.add_items(array.value());
       // In that case there is no need for primitive fast-path since
       // add_items is already optimized for that.
       for (auto& item:array){
         if (item.has_value()){
           auto& genericWriter = out.add_item();
           genericWriter.copy_from(item);
         } else{
           out.add_null();
         }
       }
      }
    }
  }
};
```

Listing 6: ArrayFlatten function written in SFI.

A key inefficiency in the function above is the type checking required for each element of the inner-most loop. As previously discussed, this problem can be solved for primitive types by registering a fast-path implementation for cases when T1 is a primitive type. Alternatively, one can call the *add_items()* API on the outer array to copy all of its elements as mentioned in the comment in code in Listing 6. While *add_items()* on an array with generic elements requires a dynamic dispatch on the element type, it gets amortized by the cost of moving the elements since it can be done only once for all the elements, similarly to *copy_from()* when performed on complex types as explained earlier. Using *add_items()* has additional performance advantages also since it implements other optimizations described in section 4.2.2, such as avoiding deep copying strings, and providing fast-paths for flat encoding.

4.4 Variadic Inputs

Finally, it is also common for SQL functions to allow users to specify an arbitrary number of parameters of the same type, like *concat(str1, str2,..., strN)*. These variadic arguments are represented in the *call()* function using a VariadicView which that have APIs similar to the ArrayView type presented in subsection 4.2.1. For example, the concat function described above can be trivially expressed as illustrated in Listing 7.

4.5 Null Behavior

In addition to the *call()* function used through out the paper and explained earilier, SFI provides two other ways variants; *callNullable( )* can be used if the function may return anything other than null in the presence of null input. This function receives pointers to the input types specific in Table 1, where a *nullptr* represents a null input. If a function has even stricter null handling semantics and returns null if any input is null, including nested container elements (*e.g*, if any element in an input array is null), *callNullFree()* can be used. In that case, nested types are recursively represented using a corresponding view type from Table 1 that returns child values directly, i.e, not wrapped in the *OptionalAccessor* interface described in Section 4.2.1.

```
struct Concat {
 void call(StringWriter& out, StringView& first,
   StringView& second, VariadicView<Varchar>& rest) {
     out.append(first);
     out.append(second);
     for (auto& input: rest){
       out.append(input);
     }
  }
};
```

Listing 7: Concat function written in SFI.

5 LIMITATIONS

While SFI covers a variety of features related to scalar function authoring, it has limitations. First, functions that can be implemented zero-copy as vectorized cannot be expressed using SFI. For example, *map_keys()* can be implemented zero-copy by moving the *keys* vector from the map; *is_null()* can be implemented by simply returning the nullity buffer. Second, in SFI authors do not have access to the encodings of the input vectors, nor control over the encoding of the result vector. Hence, it is not possible to leverage a specific encodings to optimize functions, e.g, *array_sort()* does not need to re-order elements and copy them during sorting; instead, it can wrap the input using a dictionary and sort the indices. Lastly, today SFI does not support lambda functions. However, we estimate that in practice these limitations are only encountered by a limited number of functions (<5%).

6 RELATED WORK

DuckDB [23] is a vectorized DBMS that leverages the concepts presented about columnar layout and encoding-aware execution, providing a vectorized scalar function API similar to the one described in this paper. DuckDB also provides template-based executors that extend single row operations into optimized vectorized loops (kernels), implementing some of the ideas discussed. For example, *UnaryExecutor* and *BinaryExecutor* are helper methods that can be used to implement binary and unary functions over primitive types, and *ArrayGenericBinaryExecutor* can be used for functions with two input arrays that have primitive inputs. While they could potentially be consolidated into a single API that automatically dispatches to one of the executors, they do not provide a unified and seamless authoring experience today. DuckDB also does not have support for arbitrarily nested types, variadic parameters, generic types, and other advanced authoring features.

Photon [3] is a proprietary vectorized engine developed by Databricks, implementing similar ideas about specializing kernels for different input types and properties, such as null-only and no-nulls. Although some of these optimizations are mentioned in [3], only limited information about the engine internals is available. DataFusion [9] is an open source query engine written in Rust, which operates on Apache Arrow [2] in-memory format. DataFusion provides a columnar function authoring interface (vectorized) but does not provide a simple row-based authoring interface.

Beyond vectorization, codegen (JIT) based engines such as Spark [26] use a row-based representation to express data during expression evaluation, computing expressions one row at a time in a data-centric manner. For these cases, a row-based API is already a natural fit, and many of the challenges discussed in this paper that come from the need to handle encodings and columnar representations do not apply. Presto [25], besides being a vectorized engine, implements a similar JVM codegen-based approach for expression evaluation.

7 CONCLUSION

While vectorization is a commonly used data processing technique in modern execution engines, writing efficient vectorized code is difficult. It requires a deep understanding of vectorized processing, columnar data layout, and encodings, presenting a steep learning curve for developers and challenges to organizations building large scale engineering teams. Particularly when developing scalar functions, due to their large variety, having a large number of developers exposed to such complexity resulted in a disproportionate amount of bugs and performance inefficiencies.

In this paper we discussed our experience building a simple function interface (SFI) in the Velox open source execution engine. SFI simplifies scalar function authoring by encapsulating the complexity required to generate efficient tight loops, presenting developers with a simpler, conciser, and more natural row-based interface - without sacrificing vectorized performance. Today, more than a thousand functions have been added to Velox using the SFI, implementing popular open source SQL dialects and internal domain-specific use cases at Meta. These scalar functions are currently powering many production workloads within Meta and across the industry.

Outside of scalar function, we have built a similar simple function API for aggregate functions, which is left to be described in detail in a future paper. Although the goals of complexity encapsulation and API simplicity are similar, the motivation is less compelling since there are fewer aggregate and window functions (dozens), if compared to scalar functions (thousands). Ultimately, we believe SFI provides the best trade-off between efficiency and engineering efficiency, and that while the concepts are described in the context of Velox, they are applicable to any modern vectorized engine.

ACKNOWLEDGMENTS

The work presented in this paper was only possible due to the numerous contributions from the Velox community. The authors would like to extend a special thank you to Jack Langman, Zhenyuan Zhao, and Jake Jung for their work on an initial version of SFI.

REFERENCES

[1] Azim Afroozeh, Leonardo X. Kuffo, and Peter Boncz. 2023. ALP: Adaptive Lossless floating-Point Compression. *Proc. ACM Manag. Data* 1, 4, Article 230 (dec 2023), 26 pages.

[2] Apache Arrow. [n.d.]. A cross-language development platform for in-memory analytics. https://arrow.apache.org/. Accessed: 2024-03-21.

[3] 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 *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2326–2339.

[4] Peter Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: fast random access string compression. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2649–2661.

[5] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research (CIDR)*.

[6] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew Mccormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Ebenstein, Nikita Mikhaylin, Hung-ching Lee, Xiaoyan Zhao, Tony Xu, Luis Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Selcuk Aya, Vera Lychagina, and Brett Elliott. 2019. Procella: unifying serving and analytical data at YouTube. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2022–2034.

[7] Biswapesh Chattopadhyay, Pedro Pedreira, Sameer Agarwal, Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. In *Conference on Innovative Data Systems Research - CIDR*.

[8] Zhiyuan Chen, Johannes Gehrke, and Flip Korn. 2001. Query optimization in compressed database systems. *SIGMOD Rec.* 30, 2 (may 2001), 271–282.

[9] Apache DataFusion. [n.d.]. Apache Arrow DataFusion Documentation. https://arrow.apache.org/datafusion/. Accessed: 2024-03-21.

[10] The Presto Foundation. [n.d.]. Presto Documentation: Functions and Operators. https://prestodb.io/docs/current/functions.html. Accessed: 2024-03-21.

[11] The Presto Foundation. [n.d.]. Presto: Free, Open-Source SQL Query Engine for any Data. https://prestodb.io/. Accessed: 2024-03-21.

[12] J. Goldstein, R. Ramakrishnan, and U. Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*. 370–379.

[13] 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. *Proc. VLDB Endow.* 11, 13 (sep 2018), 2209–2222.

[14] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: efficient lossless floating point compression for time series databases. *Proc. VLDB Endow.* 15, 11 (jul 2022), 3058–3070.

[15] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org.

[16] Pedro Pedreira. [n.d.]. Aligning Velox and Apache Arrow: Towards composable data management. https://engineering.fb.com/2024/02/20/developer-tools/velox-apache-arrow-15-composable-data-management/. Accessed: 2024-03-21.

[17] Pedro Pedreira, Masha Basmanova, and Orri Erling. [n.d.]. Introducing Velox: An Open Source Unified Execution Engine. https://engineering.fb.com/2023/03/09/open-source/velox-open-source-execution-engine/. Accessed: 2024-03-21.

[18] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3372–3384.

[19] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow.* 16, 10 (jun 2023), 2679–2685.

[20] Pedro Pedreira, Deepak Majeti, and Orri Erling. 2024. Composable Data Management: An Execution Overview. *Proc. VLDB Endow.* 14, 1 (aug 2024).

[21] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: a fast, scalable, in-memory time series database. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1816–1827.

[22] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1493–1508.

[23] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984.

[24] Laith Sakka. [n.d.]. Velox Blog: Simple Functions: Efficient Complex Types. https://velox-lib.io/blog/simple-functions-2/. Accessed: 2024-07-17.

[25] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813.

[26] Apache Spark. [n.d.]. Apache Spark - Unified Engine for large-scale data analytics. https://spark.apache.org/. Accessed: 2024-03-21.

[27] Yutian Sun, Tim Meehan, Rebecca Schlussel, Wenlei Xie, Masha Basmanova, Orri Erling, Andrii Rosa, Shixuan Fan, Rongrong Zhong, Arun Thirupathi, Nikhil Collooru, Ke Wang, Sameer Agarwal, Arjun Gupta, Dionysios Logothetis, Kostas Xirogiannopoulos, Amit Dutta, Varun Gajjala, Rohit Jain, Ajay Palakuzhy, Prithvi Pandian, Sergey Pershin, Abhisek Saikia, Pranjal Shankhdhar, Neerad Somanchi, Swapnil Tailor, Jialiang Tan, Sreeni Viswanadha, Zac Wen, Biswapesh Chattopadhyay, Bin Fan, Deepak Majeti, and Aditi Pandit. 2023. Presto: A Decade of SQL Analytics at Meta. *Proc. ACM Manag. Data* 1, 2, Article 189 (jun 2023), 25 pages.

[28] Linda Torczon and Keith Cooper. 2007. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[29] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. 2017. *C++ Templates: The Complete Guide (2nd Edition)* (2nd ed.). Addison-Wesley Professional.

[30] Velox. [n.d.]. Bugs fixed in mapFromEntries function. https://github.com/facebookincubator/velox/issues?q=label:map_from_entries_bugs. Accessed: 2024-03-21.

[31] Velox. [n.d.]. MultiMapFromEntries function source code. https://github.com/facebookincubator/velox/blob/main/velox/functions/prestosql/MultimapFromEntries.h. Accessed: 2024-03-21.

[32] Velox. [n.d.]. SimpleFunctionAdapter source code. https://github.com/facebookincubator/velox/blob/main/velox/expression/SimpleFunctionAdapter.h. Accessed: 2024-03-21.

[33] Velox. [n.d.]. Velox: A C++ vectorized database acceleration library. https://github.com/facebookincubator/velox. Accessed: 2024-03-21.

[34] Velox. [n.d.]. Velox Documentation. https://facebookincubator.github.io/velox/. Accessed: 2024-03-21.