BOSS - An Architecture for Database Kernel Composition

Hubert Mohr-Daurat
Imperial College London
h.mohr-daurat19@imperial.ac.uk

Xuan Sun
Imperial College London
xuan.sun@imperial.ac.uk

Holger Pirk
Imperial College London
hlgr@ic.ac.uk

ABSTRACT

Composable Database System Research has yielded components such as Apache Arrow for Storage, Meta's Velox for processing and Apache Calcite for query planning. What is lacking, however, is a design for a general, efficient and easy-to-use architecture to connect them. We propose such an architecture. Our proposal is based on the ideas of partial query evaluation and a carefully designed, unified exchange format for query plans and data. We implement the architecture in a system called BOSS[1] that combines the Apache Arrow, the GPU-accelerated compute kernel ArrayFire and the CPU-oriented Velox kernel into a fully-featured relational Data Management System (DMS). We demonstrate that the architecture is general enough to incorporate practically any DMS component, easy-to-use and virtually overhead-free. Based on the architecture, BOSS achieves significant performance improvement over the CPU-only Velox kernel and even outperforms the highly-optimized GPU-only DMS HeavyDB for some queries.

PVLDB Reference Format:
Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. BOSS - An Architecture for Database Kernel Composition. PVLDB, 17(4): 877 - 890, 2023.
doi:10.14778/3636218.3636239

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

1 INTRODUCTION

The use cases of Data Management Systems (DMSs) have evolved from simply storing and retrieving data to managing all aspects of the data lifecycle as well as hardware resources. To this end, DMSs have to account for many kinds of heterogeneity. First, there is increasing workload heterogeneity: classic scenarios like Online Transactional and Analytical Processing are joined by new applications such as data cleaning, data integration, model training or inference. In addition, applications require various heterogeneous data models such as relations, graphs, documents, key-value-pairs and even trained models. Last but not least, systems must manage heterogeneous hardware devices such as CPUs, GPUs, Smart-Storage Devices, Trusted Enclaves, FPGAs, TPUs and other Application-Specific Integrated Circuits (ASICs).

To address the challenge of heterogeneity, systems currently have two options. The first option is to extend the DMS kernel to support heterogeneous devices, workloads and data models. While this approach usually yields good performance, it requires substantial engineering effort. This effort leads to a fractured ecosystem with many systems supporting one or two "aspects of heterogeneity" (GPUs [7, 10, 31], data cleaning [9], JSON-fields [14], ...) but, to the best of our knowledge, none supporting more. Alternatively, systems can wrap special-purpose libraries in UDFs to support heterogeneity. While this is substantially less effort, it comes at the cost of inferior performance [24, 41].

We propose a third option that combines the best of both: a novel DMS design based on the idea of partial evaluation. Under this paradigm, a query is no longer processed by a single kernel but goes through a sequence of stages, each of which progresses toward the final result. Between stages, the query is passed in a simple, unified format containing all information required to produce the result (data and code). New functionality (such as a kernel or library supporting a specific hardware device) can be integrated as a stage in the evaluation process. As the entire query (including inputs) is passed between kernels, each kernel can opportunistically evaluate as many (or few) operators as deemed beneficial. As long as the last kernel in the pipeline supports the full evaluation of the query, this paradigm guarantees that the query is fully evaluated while giving each kernel complete freedom to implement special data model semantics, support new workloads, exploit hardware-specific features or apply sophisticated optimizations. This insight is reflected in a recent trend towards composable DMSs [39].

There are, however, several challenges when realizing such a design. First, the kernels (and underlying libraries) follow fundamentally different designs, leading to an "impedance" mismatch when combining them: some kernels are "pull-driven" while others are "push-driven"; some provide dynamically-typed declarative APIs while others are imperative and statically typed; some are stateful while others maintain no internal state; virtually all of these kernels maintain metadata that is exploited during query evaluation. Efficiently combining these kernels is non-trivial and requires significant amounts of boilerplate/glue code that is not only expensive and tedious to write and maintain but also error-prone.

We propose addressing these challenges through a number of technical contributions:

- We introduce the idea of partial database query evaluation as a paradigm for database system composition
- We introduce a unified physical in-memory data representation that is generic enough to support the exchange of data and code between DMS kernels yet simple to use and free from runtime overhead. Where state-of-the-art systems need to combine different interchange formats for data, metadata and code, the proposed unified format solves all these problems using a powerful, elegant and highly efficient representation.

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636239

[1] project available under MIT license at http://boss.lsds.uk

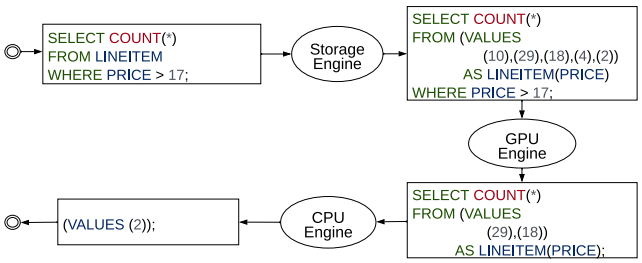

Figure 1: A Pipeline using Partial Query Evaluation

• We introduce a DMS-specific compile-time programming framework (type system, problem-specific language and memory management model) which extends well-known programming language techniques to solve the problem of integrating database kernels with minimal boilerplate code and virtually no performance overhead.
• We demonstrate the practical feasibility of these techniques by presenting the first data management system that composes multiple (unmodified) DMS kernels/libraries into a full-featured data analytics system: the Apache Arrow storage library [1], the CPU-oriented Velox database kernel [38], and the GPU-accelerated tensor compute kernel ArrayFire [32]. The system effectively demonstrates that DMS-components like Arrow, Velox and ArrayFire can be composed to complement their feature sets without performance penalty (compared to a monolithic architecture).

We describe the general concept of *partial query evaluation* in Section 2 and establish objectives and design principles in Section 3. Next, we introduce the representation of data atoms (i.e., the type system), complex expressions that are used to represent both complex data structures (such as columns, relations and hashtables) as well as operators (such as join, aggregation and select) and the memory management model in Section 4. We discuss the programming framework in Section 5 and its use to construct a system in Section 6. We study the system's performance in Section 7, discuss related work in Section 8 and conclude in Section 9.

2 PARTIAL QUERY EVALUATION

The goal of our work is to develop a DMS-architecture that allows the composition of separately designed components with minimal engineering effort. We propose for that a simple architecture: query processing is fragmented into stages, each implemented (or integrated from off-the-shelf systems) as "Engines" with more straightforward, smaller kernels rather than a monolithic one. For example, the implementation we will present in Section 6 integrates three engines: the storage engine is solely responsible for resolving the table and column names and loading the data; the GPU engine accelerates selected relational operators; the CPU engine integrates a CPU-based DMS supporting all the relational operators.

To develop this DMS while minimizing the complexity of integrating engines and orchestration logic, we propose the concept of *partial query evaluation*: the queries are passed through a linear pipeline of engines. Each engine accepts an input query and returns a result query. The returned query is equivalent to the input query (i.e., a correct interpretation of the input query) but "closer" to the final result[1]. After the last stage, no code shall be left for evaluation in the output query.

The example in Figure 1 illustrates these steps: the input query (on the top left) is passed to the storage engine; `FROM LINEITEM` is replaced by the content of that table (on the top right). Next, the GPU engine evaluates the `WHERE PRICE > 17` predicate and filters the data but leaves the `SELECT COUNT(*)` unmodified (resulting in the query on the bottom right). Finally, the CPU engine evaluates the aggregation and produces the output (on the bottom left).

Naturally, SQL is an ill-suited representation for this approach. Manipulating such a format incurs implementation and execution overhead as well as scalability issues when exchanging data between engines. Instead, we design a data & code representation as well as a programming framework that suits the requirements.

3 DESIGN PRINCIPLES

Partial Query Evaluation requires designing the DMS-architecture such that engines are implemented and integrated with negligible runtime overhead and minimal boilerplate code. This overarching objective can be achieved by following four guiding principles that we outline in the following.

Remain Unopinionated. To ensure that different kernels with different designs can be integrated easily, a composable DMS must not impose design decisions on kernels. In software engineering, this principle is commonly referred to as "unopinionated" design. In the context of DMSs, the principle applies to aspects as diverse as logical data model, data representation, memory model, execution model or concurrency control. A framework must, for example, support the integration of Volcano-style query processors [19], Bulk-processors [4], X100-style [56] processors and just-in-time compiled execution engines [35].

Minimize Boilerplate. Without an appropriate development framework, integrating external database kernels requires substantial boilerplate code. Such code is a productivity hazard for developers and a major source of bugs and performance overhead. Integrating a GPU-coprocessing library, for example, requires code to convert data objects, manage data transfers, schedule kernel execution, control concurrent execution and transfer results back. The amount of boilerplate code can easily exceed that of actual data processing code. The codebase of the Ocelot extension to support GPU-accelerated data processing in MonetDB [23], for example, contains roughly 5,000 lines of OpenCL kernel code for kernel operations and more than 23,000 lines of boilerplate C-code to coordinate execution.

Zero Copies, Minimal Transformations. Creating copies of data is a costly operation for high-performance DMSs. A well-designed monolithic system, therefore, does not create copies of data unless absolutely required. To be performance-competitive, a composed DMS must not unnecessarily copy data either.

Interestingly, many kernels have the same internal data representation, making zero-copy data transfer possible: MonetDB, Velox, ArrayFire and Arrow, e.g., share almost the same data representation (the only difference being minor optimizations in string representations). Upon close inspection, this is not surprising, as

[1]note that it is, in principle, acceptable for an engine to return its input query unmodified

all of these are designed to maximize CPU efficiency, and modern CPUs are optimized for a specific data representation: that of the ANSI C language. Most kernels store datasets in C-arrays augmented with additional metadata (such as histograms or sortedness flags). However, all kernels we studied support the extraction of the C-arrays and the construction of the kernel-specific representation from C-arrays without copying data. If exploited effectively, this enables zero-copy data transfers of data between kernels.

Language over Library. This principle is, to a large extent, a consequence of the first and the second but deserves special mention due to its effect on the design. A composition framework that is lightweight and unopinionated must resort to the least common denominator of the kernels it aims to support. That least common denominator is the host programming language (usually C or C++, though Rust and Swift are viable alternatives). This largely precludes the use of libraries to coordinate execution, generate code or transform data. By avoiding libraries, the framework allows a state-of-the-art compiler to perform optimizations such as function inlining, loop unrolling or vectorization even across kernel boundaries. Note, however, that this only applies to the framework and does not prevent kernels from using libraries internally.

In the remainder of this paper, we describe a framework that follows these guidelines and, through careful design and sophisticated metaprogramming, achieves high-performance processing in a surprisingly small codebase (fewer than 5K lines of modern C++ code).

4 CROSS-KERNEL COMMUNICATION

The first challenge to address when connecting multiple kernels without incurring prohibitive copy-overhead is the definition of an appropriate exchange format for data (base tables, indices, histograms, etc.) as well as code (i.e., query plans, selection predicates, UDFs, etc.). While projects like Apache Arrow [1] define an efficient bulk-oriented data exchange format, this format is ill-suited to represent query plans. On the other hand, projects like Substrait [50] define a format for query plans but do not address the data exchange problem. Combining both to orchestrate the operation of multiple database kernels requires substantial boilerplate code and often comes at significant runtime overhead to convert the exchange format to the internal representation. Efficiently connecting multiple kernels requires a unified, lightweight, in-memory representation of data and code that can be interpreted with minimal boilerplate. Such a representation must address several non-trivial challenges. First, it must support the bulk-oriented data format that is shared by virtually all high-performance in-memory DMSs: (statically typed) C-arrays. Next, it must support the (dynamically-typed) arguments that come in from frontends like SQL, Python or R. It must also support placeholders (we refer to them as "symbols") representing table or column references in an execution plan. The representation must, further, be extensible to allow kernels to represent internal datatypes (such as tensors or GPU memory objects) without the need for physical data transformation. Finally, it must provide a concise API that avoids runtime overhead by being amenable to compile-time optimizations like inlining and common subexpression elimination.

```
template<typename... ExtensionTypes>
typedef variant<bool, int64, double, string,
                ExtensionTypes...> CWDTypedAtom;
```

Figure 2: A (slightly simplified) implementation of CWDT using C++ Templates.

While designing such a representation may seem daunting, focusing on data-processing systems provides several opportunities to constrain the problem and, thereby, make it tractable. First, the type system of such systems is "closed" at runtime, i.e., no new types can be defined once the system has been compiled. Next, the type system can be divided into a "core", i.e., types that every kernel must support and internal "extensions", i.e., types that cannot be passed between kernels. Lastly, data is usually "typed" in bulk, i.e., the type of the first value in a column (or partition) is the same as all other values in that column. Type-interpretation can, therefore, be performed once per-column rather per-value (this concept is referred to as "evidence-typing" in programming language research [26]).

Motivated by this analysis, we propose a unified representation for data and code that addresses the challenges by exploiting these opportunities. It combines several techniques, inspired by established programming language techniques but adapted to the needs of DMSs. We discuss them bottom-up in the rest of this section.

4.1 Extensible Closed-World Dynamic Typing

A unified exchange format's most fundamental challenge is the efficient and easy-to-use representation of dynamically-typed values (we use the term Atoms). Typically, this is efficiently implemented using "tagged union", such as C++ variants, but requires the set of types to be statically defined across the kernels. However, a kernel integration might require the declaration of further dynamic types, e.g., one supporting TensorFlow might require a tf::TensorBuffer type. As illustrated in Figure 2, our approach extends standard C++ variant by instantiating it with a set of "core" types (bool, int64, etc.) while allowing the compile-time extension with further types. A core-only type system, e.g., can be declared by instantiating the template without extension types (`CWDTypedAtom<>`) while the type system that also supports TensorFlow would be declared as `CWDTypedAtom<tf::TensorBuffer>`. This approach supports the combination of multiple different type-systems in the same executable. While data can only be exchanged *between* kernels in the standard type system, kernels can use a custom type system internally with *transformation-overhead-free* communication between kernels[2]. As the type system is fully defined at compile time (a.k.a., closed at runtime), the compiler can generate type-specific operators statically (we will illustrate this in Section 5).

4.2 Memory-Managed Spans for Data Exchange

High-performance data processing kernels store and process data in collections rather than individual atoms. While many kernels store and maintain metadata (histograms, min/max values, etc.), the core data structure of virtually all of them is plain C-arrays. While

[2]the ordering of the core-types preceding the extension types ensures that core types have identical tags in any derived type system

```
template<typename ElementType>
struct Span {
  ElementType* data;
  size_t size;
  void* payload;
  void (*destructor)(); // function pointer
};
```

Figure 3: Spans capture bulk-allocated data

this lack of diversity might seem surprising initially, it is a logical consequence of the hardware support for that format. A universal data exchange format should, therefore, be based on C-arrays.

However, plain C-arrays have no support for memory management, requiring kernels to implement their own mechanism, often based on reference counting or mark-and-sweep garbage collection. As every kernel has its own memory management scheme, unifying them without substantial overhead is non-trivial. However, focusing on (analytical) data processing systems suggests an approach: as allocated memory objects tend to be large (megabytes to gigabytes), small, constant allocation/deallocation overheads are negligible. We propose a simple memory management mechanism based on function pointers and opaque payloads.

Figure 3 illustrates the implementation of the idea in the form of a *BOSS Span*: inspired by C++20 spans, BOSS Spans are type-generic, thin wrappers around C-arrays their memory deallocation managed in the form of a function pointer and an untyped payload. Upon destruction, the function is called and can, e.g., delete an underlying C-array, unmap a memory-mapped file or not do anything if the span does not own its memory (the default). Note that the payload and the data can point to the same address but do not have to. If, e.g., a span is created from an input that stems from an external library and has a header preceding the data in the same allocated piece of memory, `data` would point to the first data item while `payload` would point to the address of the beginning of the allocated object, i.e., the header.

Thus implemented, spans are unopinionated with respect to ownership. Most importantly, they have no notion of "shared ownership". While the function pointer can be used to decrease a reference count, there is no built-in mechanism for that purpose. Reference counting can, however, be implemented outside the span type (in fact, some of the kernel wrappers we present in Section 6 do just that). However, we found that true shared ownership of memory objects is surprisingly rare in DMS kernels: persistent tables are owned by the storage manager, while intermediate results merely have their ownership transferred from one function to another. To illustrate this, consider the example of the `process` and `zeroWhere` functions in Figure 4 that implement a simplified select query (one that zeroes out values that do not qualify): while the `process` function creates (and thereby owns) the `input` span initially (in lines 10 & 11), it transfers ownership to `zeroWhere` in line 12 through a `move`. After executing the operation (lines 2 to 4), `zeroWhere` returns ownership in line 5 (through another `move`). `Process` (re-)accepts ownership through assignment to `input` in line 12. While this process might sound error-prone, it can be enforced by the C++ compiler: we propose to implement Spans as move-only/not-copyable types. If a move of a Span is omitted, the compilation will fail. If a value is used

```
1  auto zeroWhere(auto&& input, auto predicate){
2    for(auto& inputValue : input) {
3      if(predicate(inputValue))
4        inputValue = 0;
5    return move(input);
6  }
7
8  void process() {
9    auto inputVector = vector{1, 17, 9, 2};
10   auto input = Span{data = inputVector.data(),
11                     size = inputVector.size()};
12   input = zeroWhere(move(input),
13                     [](auto v) { return v < 8; });
14   print(input);
15 }
```

Figure 4: Move-only spans simplify the memory model

after a move (e.g., if the result of `zeroWhere` were not assigned to `input`), the compiler warns (or fails) with a "use after move" error.

4.3 Semi-statically Typed Expressions

While Spans allow representing contiguous in-memory data, this abstraction does not allow to represent complex data structures (e.g., columns, tables, trees, hashtables), query plans/programs or any dynamic behaviour (e.g., on-demand data loading from file). We propose to implement all these abstractions in a single, unified representation, on top of closed-world-typed atoms and spans, with "symbolic expressions (s-expressions)". Classic s-expressions, as introduced with LISP [34], are nested (linked) lists of dynamically typed atoms (including symbols). In LISP, s-expressions are represented in the iconic notation using parenthesis: `(list 7 2 3 6)` represents a data structure while `(lambda (x) (+ x 9))` represents a function, i.e., code. In general, the high interpretation overhead makes s-expressions manifestly unsuited as a data model for a high-performance DMS. Instead, we propose several novel extensions: most importantly, we propose to statically type some expression arguments at compile-time while allowing some in (closed-world) dynamically typed form. Further, we propose to use the span concept we introduced earlier to store consecutive, identically-typed arguments of expressions. While Figure 5 illustrates the implementation in C++, let us discuss the extensions in more detail.

Dynamically Typed Arguments. In their simplest form, we propose expressions that are an (almost) faithful reimplementation of classic s-expressions: they differ from classic s-expressions only in that the first element (a.k.a. the head) must be a Symbol (see line 5 in Figure 5). While this has no practical impact on expressivity (one could map expressions with a non-symbolic head to one where the head is a symbol with an empty name), we found it a practical restriction that enables several optimizations we discuss later. The API is simple: an expression representing a column of Extensible Closed-World Dynamic Typing-typed (CWDT-typed) values would, e.g., be created as `Expression("Column", 5, 9.2, "seventeen", false)`.

Statically Typed Arguments. While CWDT-typed expressions are very flexible, they come at substantial overhead: on the one hand, a system needs to represent the runtime type using a type tag

```
1  typedef CWDTypedAtom<Expression> DynamicArgument;
2  typedef variant<Span<int>,
3                  Span<float>, /*...*/> SpanArgument;
4  template <typename... StaticArguments> class Expression {
5    Symbol head;
6    vector<DynamicArgument> dynamicArguments;
7    tuple<StaticArguments...> staticArguments;
8    vector<SpanArgument> spans;
9  public: // API
10   // slow but generic argument access as DynamicArgument
11   DynamicArgument getArgument(int);
12   // fast and specialized access using specific type
13   template<int i> auto getArgument();
14   SpanArgument getSpanArgument(int);
15 };
```

Figure 5: Semi-statically typed expressions

(usually an integer); On the other hand, operating on dynamically typed values requires dynamic dispatching of values to operators (a.k.a., visitation), which translates into a (virtual) function call. Most importantly, however, most data processing kernels operate on static types. Translating between dynamic and static types requires significant amounts of boilerplate code.

To address these problems, we propose to support static typing of expression arguments (see line 4 in Figure 5). Declaring an expression as `Expression<int, float, string, bool>("Column", 5, 9.2, "seventeen", false)`, e.g., creates an expression with statically typed arguments. Statically typed expressions save memory, CPU cycles and boilerplate code. They are also the basis for the Semi-Static Dispatching technique we introduce in the next section. First, however, let us discuss the last and arguably, most consequential category of arguments: Spans.

Span Arguments. As discussed earlier, Spans are statically typed collections of atoms. In addition to statically and dynamically typed arguments, we propose to implement expressions that can take arbitrarily many Spans of potentially different types as arguments (see line 8 in Figure 5). The elements of the Spans are "logically" exposed arguments of the expression through the same API as is used to access other arguments (line 11 in Figure 5). Physically, however, the Spans remain plain C-arrays that can be accessed directly using a "physical" API (line 14).

Argument APIs. To conclude the description of our proposed expression arguments, let us describe the APIs used to access them. There are two APIs with distinct use cases: the "slow and unified" API (line 11 in Figure 5) is designed for non-performance-critical cases like rule-based query optimization or printing results to the console while the "fast and specific" API is designed for performance-critical cases like the tight loop of a processing operator (line 13 & 14 show the API to access static arguments and spans).

The "Slow Path": Dynamically-Typed Wrappers. The slow path provides access to all arguments of expressions through a "least-common-denominator" API: CWDT-typed arguments. As static arguments and span elements can be converted to CWDT arguments but not the other way around, CWDT types are the least common denominator (line 11 in Figure 5).

```
1  // decompose() API
2  class Expression {
3    tuple<Symbol, Statics, Dynamics, Spans> decompose() &&;
4  };
5
6  // Decomposition and recomposition example
7  Expression removeDynamicArguments(Expression&& input) {
8   auto [head, statics, dynamics, spans]
9                              = move(input).decompose();
10  return Expression(head, statics, {}, spans);
11 }
```

Figure 6: Destructive Decomposition API and Example

The "Fast Path": The fast API provides (read-only) access to each of the specific argument categories using designated functions: the static arguments are accessed using a templated function that takes the argument position as a compile-time constant, the dynamic arguments by their runtime position and the spans as a collection of (read-only) references to spans. While all elements of a single span have the same static type (they wrap C-arrays, after all), different spans in the collection can have different element types (useful to support weakly typed SQL engines like SQLite). In addition to being faster by providing access to an entire span object at once (which avoids converting every single element to a dynamic argument), the fast path is also type-safe for statically typed arguments.

The proposed abstractions provide convenient *read-only* access to data and code in line with state-of-the-art dataflow systems. However, due to the simplicity of the memory model (i.e., the lack of reference counting), even trivial operators like projections in a column-store would have to perform expensive copies when performing only structural changes to the expressions (i.e., when the data itself is not changed). To address this challenge, we propose a novel design pattern for dataflow systems that requires neither copies nor reference counting: *Destructive Decomposition*.

4.4 Destructive Decomposition

Like Spans, we propose implementing Expressions as non-copyable (move-only in C++-nomenclature) types. In fact, they must be move-only types because one of their components (the Span arguments) is move-only. However, while they cannot be copied, they can be decomposed into components which can be reassembled to implement operations like projections. Figure 6, line 3, illustrates the API: a `decompose()` function that returns the expression components (head, statics, dynamic arguments and spans) as a tuple. The `&&`-suffix to the function indicates that the function can only be applied to an rvalue-reference, i.e., an object that is about to be destroyed. It is the very semantics that is required for destructive decomposition.

Lines 7 to 11 in Figure 6 illustrate how destructive decomposition can be used to structurally modify an expression (in this example, to remove the dynamic arguments): according to the established ownership model, the function owns the expression object; it can safely destroy the object and extract its arguments. To do so, the function must mark `input` for destruction (by calling `move`) and call `decompose()`. Just like for Spans, the use of `move` is required and enforced by the compiler. Using the input expression after it has been

moved results in a warning or compilation error (depending on compiler flags). The elements of the returned tuple are assigned directly to variables using the `auto []`-syntax. They are then reassembled into a new Expression object. Using destructive decomposition this way avoids expensive copies and (opinionated) reference counting.

5 KERNEL-WRAPPER CODE SYNTHESIS

Based on the data abstractions defined in the last section, let us now define compile-time programming abstractions that allow kernel integrators to generate (practically) zero-overhead kernel-wrapper code with minimal boilerplate. In particular, we propose compile-time abstractions that practically eliminate the need for physical data transformation by providing a compile-time translation between a kernel's in-memory data representation and the expected Expression API. To ease their interaction with the host system language, these abstractions are purposely leveraging existing programming language techniques, extended to solve the problem of composing off-the-shelf systems into a full DMS system with minimal boilerplate and integration complexity.

5.1 Semi-static dispatching

Since DMSs have to support arbitrary workloads, operators need to be generic to support a variety of types. To provide type generality, most high-performance systems (such as MonetDB and DuckDB) rely on compile-time code generation to expand templates into type-specific operators. To minimize runtime overhead, the dispatching of dynamically-typed columns or tables to statically-typed operators is performed per-column or per-microbatch. Without adequate language support, however, this requires significant boilerplate code for type dispatching and template expansion. The MonetDB team, e.g., used to use Mx, a purpose-built template expansion system, for that purpose [4] (now, they use C-macros).

Data-intensive programming languages like Julia solve the problem of mapping dynamic types to operators by implementing a technique called dynamic dispatching [27]. Dynamic dispatching allows the overloading of functions with static types, which are matched against the runtime types of function calls. In Julia, this is supported by just-in-time code generation to maintain competitive performance. Systems languages like C++, Rust and Swift do not support dynamic dispatching[3]. However, with the restricted CWD-Typing defined in Section 4 allowing the set of types to be statically defined within each kernel, we found that a limited form of dynamic dispatching is sufficient to address the needs of composable DMS development. We call this technique *Semi-static Dispatching*.

To illustrate the advantage of semi-static dispatching, let us introduce the example of implementing a sum operator in the DMS. Because the operator handles generic types as arguments, but the code operates on statically-typed data for performance, the implementation in a typical DMS requires: (a) type-checking for the arguments passed to the operator; (b) conversion to the static type system; (c) a memory management model for direct access to the data without a copy; and (d) conversion back to the dynamic type representation. The DMS also requires (e) an operator registry that allows

```
1 operators["Sum"_] = []<NumericType ArgumentsType>
2 (ComplexExpression<ArgumentsType>&& input) {
3   auto result = input.getArgument<0>();
4   for(auto& span : input.getSpans())
5     for(auto& element : Span<ArgumentsType>(span))
6       result += element;
7   if(isnan(result)) return "Table"_("Sum"_("NULL"_));
8   return "Table"_("Sum"_(result));
9 };
```

Figure 7: Combining all the techniques to implement a DMS-operator with minimum boilerplate

dynamically typed operators. Each of these problems requires a large amount of boilerplate code in the operator implementation.

The code in Figure 7 shows how little boilerplate is required with the semi-static dispatching approach: The operator is defined as a generic lambda and inserted into the operator registry (e) in line 1: the `[]`-syntax declares a lambda while the angle brackets declare a type parameter `ArgumentsType` that is expanded to the different types of arguments the operator accepts (float, int, ...). The `NumericType` qualifier restricts the set of possible template arguments using C++20 concepts. The implementation of the lambda is relatively straightforward: it takes a single `ComplexExpression` as input in line 2 using move paradigm (c) and initializes the result from the first argument of the input expression in line 3 which is statically typed, avoiding type-checking (a). It, then, iterates over all input spans (line 4) and casts them (b) to the same type as the first argument in line 5. This cast is safe as the framework ensures that the spans have the same type as the first argument at runtime. After that, the operator merely needs to iterate over the span elements (also in line 5), sum them up in line 6 and return the result (d) in line 7.

As demonstrated in this example, all of the problems we enumerated are solved by encapsulating all the techniques we presented previously into one compile-time API. This, supported by the host language, allows significantly reducing boilerplate and code complexity without compromising the high-performance requirements for a DMS kernel. As a qualitative comparison, the Greater operator implemented in ArrayFire takes less than 23 lines of code, whereas implementing the equivalent functionalities but without semi-static dispatch takes more than 80 lines of code[4].

The execution framework takes advantage of the type parameter `ArgumentsType` (line 1) being known and statically expanded at compile-time. Thus, at runtime, the expression's dynamic arguments are efficiently typed-checked against static types in the dispatch code. If one of the argument types does not match, no evaluation is performed, and the expression is returned as-is (i.e., will be evaluated by subsequent stages). If the arguments match, they are replaced with static arguments (moved with zero copies) and the function is called for evaluation.

5.2 An Embedded Expression DSL

As covered in Section 4, we propose a simple yet effective data model. Creating expressions in that model, however, is quite

[3]Note that dynamic dispatching is often confused with virtual function calls, leading to false claims of dynamic dispatch support in systems languages

[4]We provide the two versions in the Evaluation folder of the code repository at https://github.com/lsds/MultiKernelBOSS/tree/main/Evaluation.

```
1  ;; input query:
2    (Select Lineitem (Where Mode != "Truck"))
3  ;; after Storage engine substitutes symbols and Strings:
4    (Select (Table (Key 1 2 3 4 5) (Mode 0 0 1 2 0))
5            (Where Mode != 0))
6  ;; after ArrayFire partially-evaluates the query:
7    (Gather (Table (Key 1 2 3 4 5) (Mode 0 0 1 2 0)) 2 3)
8  ;; after Velox evaluates the query:
9    (Table (Key 3 4) (Mode 1 2))
10 ;; after Storage engine re-write the Strings back:
11   (Table (Key 3 4) (Mode "Mail" "Air"))
```

Figure 8: Query evaluation in a staged pipeline

boilerplate-heavy. To reduce the boilerplate, we extended the C++ host language with a domain-specific language by overloading the underscore (_) operator. That overload provides two functions: when applied to a string literal, it encodes a symbol; when applied and invoked, it constructs an expression. This allows the concise construction of expressions directly in the host language. To illustrate this, consider lines 7 and 8 in Figure 7: the result of the computation must be returned as a value in a column named `Sum` that is wrapped in a `Table` expression (line 8). However, the computation may also produce an invalid (specifically, a NaN) value. This is represented in SQL as a `NULL`. To handle this case, the operator returns a `NULL`-symbol in the column if `isnan(result)` returns true.

6 PARTIALLY-EVALUATING ENGINES

To demonstrate the feasibility of multi-kernel DMSs and assess the utility of the abstractions we introduced in Sections 4 and 5, we implemented them in a system named BOSS, short for **B**ulk-**O**riented **S**ymbol-**S**tore [6]. BOSS supports CPU/GPU co-processing of relational queries by pipelining a storage & loading kernel, a GPU-accelerated relational kernel and a CPU-based relational kernel that supports a subset of what is required for relational data analytics. Let us start by introducing the processing pipeline at a high level before presenting the different kernels.

6.1 Engine Pipelining

BOSS combines three engines: the Apache Arrow persistence engine [1], the CPU-oriented relational processing engine Velox [38], and the ArrayFire parallel computing kernel [32] to add GPU co-processing support. The example in Figure 8 illustrates the role of each engine in the execution pipeline. The initial query (line 2) is, first, evaluated by the Storage engine (lines 4 & 5) to substitute symbols referring to tables and columns with expressions containing the data of those columns. Next, the ArrayFire engine opportunistically evaluates the parts of the plan that benefit from GPU acceleration (line 7). Finally, the Velox engine evaluates all unevaluated operators and returns an evaluated result (line 9). There is no need for explicit operator assignment to kernels to decide which operators are evaluated by the ArrayFire or the Velox engine: the only orchestration logic is the order of the engines in the pipeline.

Let us, now, discuss the technical details of each of these engines and how they achieve zero-overhead interaction.

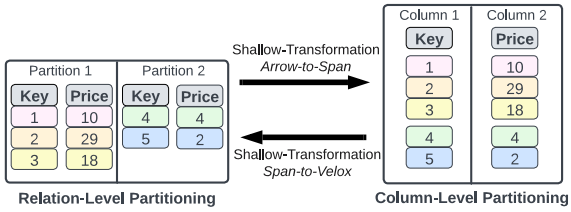

Figure 9: Conversion between relation-level (Arrow & Velox) and column-level (BOSS Span & ArrayFire) partitioning

6.2 Storage & Loading

The storage engine implemented in BOSS has two functions: loading relational data from files into memory and substituting the table identifier symbols with the stored relational expressions.

Loading Relational Data. We implement the storage engine on top of Apache Arrow [1], which provides in-memory columnar storage in the form of *Arrow Arrays*. Arrow Arrays are wrappers around C-arrays. They can, therefore, be converted to BOSS Spans with zero-copy by passing the C-array pointer to the Span's constructor. Because the Arrow Array owns the C-array, to ensure that the C-array is released only when the Span is deallocated, the Arrow Array's shared pointer is passed as the payload for the Span's destructor.

To partition column data into smaller arrays that fit in the CPU's last-level cache, both Apache Arrow and BOSS support partitioning but with a different layout. As shown on the left of Figure 9, Apache Arrow uses a *relation-level partitioning layout*, i.e., relations are sets of partitions which are sets of column arrays. We adopted in BOSS the *column-level partitioning layout*, i.e., columns are composed of multiple partitions, shown on the right of Figure 9. This layout, supported by the Expression API by storing multiple Spans in the expressions (see Section 4.3) is simpler for manipulating data per column compared with Apache Arrow's layout and has less structural redundancy (e.g., column metadata is stored only once).

Data in one layout is converted into the other representation with minimal overhead using *shallow-transformation*, i.e., the target data structure is created, but the data is transferred without copies.

Symbol Substitution. As illustrated in Figure 8 lines 1 to 5, table identifier symbols are substituted by the storage engine with stored relational expressions, such as replacing the `Lineitem` symbol in line 2 with `(Table (Key 1 2 3 4 5) (Mode 0 0 1 2 0))` in line 4. To efficiently perform this substitution, table identifier symbols are stored in a dictionary of table identifiers to relational expressions. During query execution, the storage engine traverses query plan expressions and replaces table identifier symbols with stored relational expressions. As the storage engine owns persistent relational expressions (i.e., the `Table` expressions), they are shallow-copied into the plan: the storage engine instantiates new spans referencing the input C-arrays without passing a destructor function or payload. This effectively makes the spans non-owning. As the storage engine outlives the execution of the query, handing out non-owning references to spans is safe.

Compressed String Dictionaries. When a database contains string columns with low cardinality (i.e., few unique strings), storing

duplicates of the strings in memory would be wasteful. In addition, executing equality predicates on a string is inefficient when only an integer comparison suffices. DMSs, therefore, typically implement a compressed string dictionary, i.e., strings are stored with an integer column whose values are indexed into a list of unique strings. The following illustrates how the partial query evaluation paradigm allows the implementation of this feature in the storage engine without changes in the other engines.

During loading, low-cardinality string columns and dictionary-compressed. During the query evaluation, the Storage engine evaluates string predicates on the dictionary and re-writes the query plan into one that operates directly on the (integer) keys (lines 4 & 5 in Figure 8). This allows "downstream" engines to efficiently process compressed strings without having to implement the functionality. To turn keys back to strings in the result, the storage engine wraps the query in a `StringLookup` operator (turning line 9 into 11).

6.3 Relational Processing on CPU

We implement a general-purpose CPU execution engine for the Multi-kernel DMS to handle relational processing based on Meta's Velox kernel [38]. Integrating Velox into BOSS, requires transferring data from BOSS to Velox and back and translating BOSS query plans to Velox. For that purpose, we implemented a thin wrapper around the Velox kernel we call the *Velox Engine*.

Data Transfer. Velox is a vectorized execution engine that processes data in batches of tuples in decomposed format. The fundamental materialization unit for decomposed data in Velox is a `FlatVector`, which represents a column or column-partition in a single memory block. Consequently, data As can be converted between Velox' `FlatVector` and BOSS Spans without copy. Composed from multiple `FlatVectors`, `RowVectors` are used to represent a group of columns that are being passed between operators.

A Velox `FlatVector` can assume ownership of a memory object, evaluate operations on it and return ownership without requiring copying. BOSS' Velox Engine adopts this Owned mode, by providing `spanToVelox` and `veloxtoSpan` functions to transform Span to a `FlatVector` and vice-versa.

Illustrated in Figure 9, *Velox-to-Span* implements a *shallow transformation* between BOSS Span and Velox' `RowVectors`. A vector of `RowVector` represents multiple Spans generated based on the *relation-level partitioning layout*. For example, the first batch of column `Key` and column `Price` is transformed to `RowVector 1`, while the second batch is transformed to `RowVector 2`.

Expression Translation. The Velox Engine uses Velox' `PlanBuilder` API to create a query plan tree. It builds a query plan tree bottom up, starting with the source node (`ValueNode`), followed by computation nodes such as `FilterNode`, `ProjectionNode` and `AggregationNode`. `ValueNodes` are created from `RowVectors` (created as explained above). The inputs of the remaining functions are obtained by traversing the BOSS expression. We implemented a recursive `bossExprToVelox` function to iterate over BOSS expressions and perform the transformation.

To illustrate the process, let us use `FilterNode` as an example: Figure 10 shows a code snippet illustrating the process of transforming `"Where"_("Greater"_(Price, "17"_))` to inputs of Velox'

```
1  void bossExprToVelox(Expression&& e, QueryBuilder& b) {
2    e.visit(
3      [&](ComplexExpression&& e) {
4        auto [head, statics, dynamics, spans] =
5          move(e).decompose();
6        for (auto& argument: dynamics)
7          bossExprToVelox(move(argument), b);
8        if (head == "Greater") {
9          auto& args = b.getFilterArguments();
10         b.addFilter(format("{} > {}", args[0], args[1]));
11       }
12     },
13     [&](auto valueOrSymbol) {
14       b.addFilterArgument(getAtomicExpr(valueOrSymbol));
15     });
16 }
```

Figure 10: Translating BOSS expression to Velox using its `PlanBuilder`.

`filter` function. The dispatching of dynamically typed BOSS expressions to statically Velox-typed functions is encoded using the `e.visit` function in line 2. In the first level of recursion, the `Where`-expression is decomposed into its head (the `Where`-symbol) (line 4), and its sole argument (`"Greater"_(Price, "17"_)`). The `Greater`-expression is recursed-upon (line 7). In the second recursive step, the expression is still visited as a `ComplexExpression`, its head is `Greater`, and the two arguments will be traversed as atomic expressions in the next two recursive steps. In the third level of recursion `Price` is visited as a symbol (line 13) while 17 is visited as a constant value (also line 13). The atomic values are saved as filter arguments. After returning to the second level of recursion, the `FilterNode` plan is created (lines 9 to 12) from the saved arguments.

Opportunities. When integrating Velox kernel into BOSS, we found that the Velox engine does not yet exploit all optimization opportunities. The `join` operator provided by Velox, e.g., does not exploit available indices, such as primary/foreign keys. Furthermore, the current version of Velox does not support GPU acceleration. A secondary engine can improve performance by exploiting these opportunities without modifying the Velox kernel itself. Let us in the following illustrate how BOSS offloads compute-intensive operators onto a GPU by using the ArrayFire Tensor compute kernel.

6.4 Acceleration on GPU

Due to limited capacity of GPU memory, transfer cost and massively parallel architecture, not all operators can be GPU accelerated [20]. Consequently, we implemented a GPU-accelerated engine that determines which parts of a query to execute on the GPU. Taking advantage of partial query evaluation, the queries are traversed during execution to opportunistically evaluate some operators, leaving the rest of the query unevaluated (for a CPU-kernel to evaluate).

To demonstrate the use of off-the-shelf kernels, we build on the ArrayFire Tensor Processing Kernel [32] to implement the GPU-accelerated engine. ArrayFire provides a unified API to manipulate tensors, called *af::arrays*, with backends for CUDA, OpenCL and classic CPUs. Our engine implements relational operations on top.

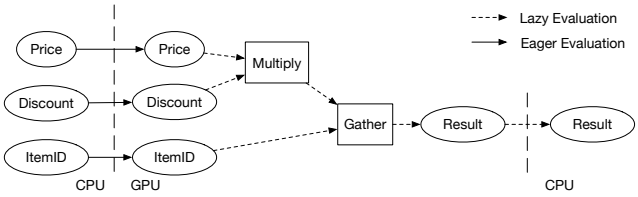

Figure 11: ArrayFire computation is lazy, copies are eager

Relational Operators Implementation. We limit our efforts to cases that can efficiently be implemented using only ArrayFire functions.

Projection. In its simple form, this operator is a no-op, i.e., only renaming columns using destructive decomposition. However, some projections evaluate arithmetic expressions. These are straightforward to implement using ArrayFire's arithmetic operations.

Selection. This operator is implemented using ArrayFire by evaluating the predicate(s) to a bit array using ArrayFire's boolean operations on af::arrays and transforming the bit array into a position list (i.e., an array of indices) using ArrayFire's `af::where` operator. The kernel opportunistically evaluates as many predicates as possible given the GPU memory constraints. As illustrated in Figure 8 (line 7), the ArrayFire Engine leaves some selections unevaluated to minimize cross-device data transfer. Those that are evaluated are transformed to a `Gather` operator, taking an (unmodified) relation and the position list as arguments to be evaluated by the Velox Engine.

Join. The typical implementation of a join on the GPU is a partitioned hash join [31, 47] which is non-trivial to implement with ArrayFire. However, ArrayFire supports the case of indexed primary/foreign key joins, which are, arguably, the most common joins. The join is implemented using ArrayFire's `af::lookup` function to resolve the probe side of the joined relation. If the input data is partitioned into multiple spans as discussed in 6.2, they are combined into a single contiguous array in the GPU memory. This operation is performed at no extra cost when the data is transferred to the GPU.

Lazy GPU Transfer & Evaluation. Unlike BOSS Spans, af::arrays are no mere wrappers for data. Instead, most ArrayFire functions return unevaluated "proxies" that contain a DAG of operators to be executed lazily when data is accessed (on the CPU). However, when af::arrays are initialized with data, data is eagerly transferred to the GPU. This creates a hybrid execution graph (illustrated in Figure 11), in which some edges are evaluated eagerly and some lazily.

This hybrid evaluation plan can become a performance liability when the required dataset cannot be determined before executing the plan. If, e.g., the GPU memory is insufficient, a transfer can fail, making other transfers unnecessary. In Figure 11, e.g., the transfer of the discount column is required if, and only if, the price column is transferred as well. To avoid unnecessary data transfers in BOSS, we transform the eager evaluation into a lazy one through a well-known design pattern: we encapsulate the eager computation in a functional closure and only evaluate it if necessary. The execution graph (lazy as well as eager edges) can, thereby, be composed using virtual function calls. For that purpose, we exploit the extensible type system introduced in Section 4.1. We extend it with an additional atomic type: a closure that holds and returns an ArrayFire array (implemented as `function<af::array()>`). When traversing the BOSS plan expression, subexpressions that are amenable to GPU acceleration are evaluated to such closures (which are passed between operators as BOSS atoms). When returning the result to the next kernel in the pipeline (i.e., Velox), the ArrayFire Engine turns the closure into a core-BOSS atom by evaluating it.

7 EVALUATION

To assess the benefits of GPU-accelerating relational operators with our approach, we evaluate the performance of BOSS [6] with and without GPU acceleration and compare it with CPU-based DMSs and a GPU-based DMS. Then, we verify more precisely the effects of partially-evaluating queries for datasets not fitting into GPU memory and the benefits of the zero-copy data transfer design in BOSS.

7.1 Experimental setup

Systems. We integrated Velox v0.0.1a0 (git rev. fb33fbfec5895) [38] for the implementation of the CPU-based kernel and ArrayFire v3.8.3 [32], compiled with CUDA v12.0, for the GPU-accelerated kernel. To evaluate the performance of co-processing in BOSS in comparison with CPU-based DMS, we compare BOSS with MonetDB Jun2023_SP2_release [56] and DuckDB v0.8.1 [40]. To evaluate the performance with a GPU-based DMS, we compare BOSS with HeavyDB v6.4.0 [22]. While HeavyDB is a GPU-only system and we, therefore, do not expect to outperform it in terms of runtime, *there exists, to the best of our knowledge no system that is designed for general-purpose co-processing.*

Because Velox does not implement the primary/foreign keys optimization for the joins that we implemented in the GPU-accelerated kernel, to assess the benefits of this optimization independently from the GPU acceleration, we also compare Velox and BOSS with an alternative version of BOSS fully implemented on the CPU (using ArrayFire's CPU backend)

Hardware. All experiments are performed on a server with two Intel Xeon Silver 4114 2.20 GHz CPUs, each with 10 physical cores, a 14 MB LLC cache and 196 GB of memory. The GPU is a GeForce GTX Titan Xp with 12 GB of memory with NVIDIA driver v525.105.17. We use Ubuntu 18.04 with Linux kernel 4.15.0-209 and compile all code with Clang version 14 using the compiler flags `-O3 -mavx2`.

Workload. In all the experiments, we use the TPC-H benchmark [13]. We store numerical values with double-precision floating-point type for BOSS (due to better performance for GPU-acceleration). We vary the scale factor (SF) from 1 to 100 (i.e., 1 to 100 GB). Following established practice [28], we evaluate the five queries that capture the benchmark's choke points [5]. None of these queries performs integer arithmetic besides the aggregations, which ensures fair comparison since the ArrayFire engine does not check for arithmetic overflow yet (and does not evaluate aggregations). Due to the lack of support for non-dictionary compressed string columns in the storage engine yet, Q9 is modified to filter the `PART` table on `P_RETAILPRICE` rather than `P_NAME` and Q18 to group by `C_CUSTKEY` rather than `C_NAME`. However, the cardinalities are preserved. Q1, Q3 and Q6 are the original TPC-H queries.

Query Plans. To ensure a fair comparison of BOSS's hand-written query plans with other DMSs, we apply fixed heuristics that we plan to integrate into an automated query optimizer in the future.

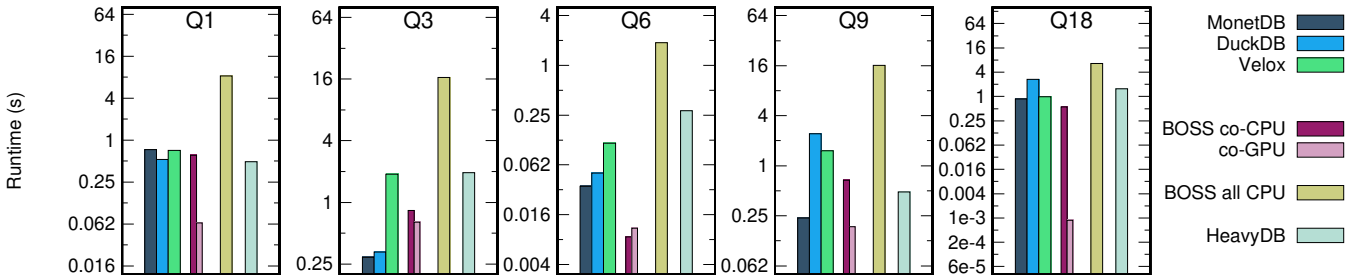

Figure 12: Runtime for TPC-H on five representative queries with scale factor 10 (i.e., 10GB)

for all the queries to decide the join order: we iteratively pick the two smallest among all the pairs of relations that can be joined next and always use the smaller of the two on the build side. To ensure that the query plan maximizes the operations that the GPU-accelerated kernel evaluates, we apply two more rules when using this kernel. First, we choose, as the next joined relations, the ones having indexed PK/FK keys over smaller relations. Second, we push the selections on joined tables down the query plan and order the predicates based on their independent selectivity (estimated by sampling the tables). For example, for Q9, `ORDERS` and `LINEITEM` are joined first because they have indexed PK/FK even though they are not the smallest relations, and the selection on `P_RETAILPRICE` is applied only after joining the `PART` table, so the indexed FK/PK on `PARTSUPP` and `PART` can be exploited.

Query Concurrency. For the interpretability of our experiments, we run the queries in isolation. If the queries were run concurrently, co-processing could be overlapped, benefiting the throughput but not the latency. For transparency, we report CPU and GPU runtimes broken down in separate bars for SF 10. To assess latency, one must add CPU and GPU runtimes; to assess throughput, one must take the maximum of the two. Because we consider throughput more important for analytics DMSs, we present BOSS runtime as the maximum between CPU and GPU when the breakdown is not indicated.

7.2 Performance with TPC-H Benchmark

To start the evaluation, we study the typical case for GPU acceleration: a medium-sized dataset by executing the TPC-H queries with SF 10. The results are shown in Figure 12. Overall, the figure shows a significant performance benefit from co-processing but with variations depending on the query characteristics. We will now analyze the results for each query in order of increasing query complexity.

Heavy scan with multi-predicate selection (Q6). Velox is outperformed by MonetDB and DuckDB by a factor of 3x and 2x, respectively. This result is explained by the implementation of the filter operator provided by Velox, which only applies to a single predicate column, while Q6 has multiple predicates. Velox uses dictionary vectors to avoid the materialization of intermediate position lists for successive filter operators, but the nesting of index indirections affects the performance. BOSS significantly improves Velox' runtime (by a factor of 14x), benefiting from GPU-accelerating the selection leaving Velox to compute grouping and aggregation. Combining the efficiency of the CPU-based operators and the GPU-accelerated operators allows BOSS to outperform the two CPU-based baselines by a factor of 3x (MonetDB) and 5x (DuckDB). The GPU-only baseline HeavyDB is even outperformed by BOSS by a factor of 25x. HeavyDB performance is also affected by the runtime overhead of the system, which we will study later in this section.

Large computation and aggregation (Q1). For this query, which is dominated by arithmetic computation, MonetDB, DuckDB and Velox perform similarly. BOSS improves Velox runtime by executing the arithmetic operations on the GPU by a factor of 1.3x. Only a minor improvement is gained because the high cardinality of the filter impedes the benefits of running the calculations on the GPU: more than 99% of the column values participating in the arithmetic operations are transferred back to the CPU. HeavyDB aggregates on the GPU and transfers only a small fraction of data back to the CPU, slightly outperforming the CPU/GPU BOSS.

Joins on filtered columns (Q3). With this query, dominated by join calculation, Velox is outperformed by MonetDB and DuckDB by a factor of respectively 6x and 7x due to Velox not taking advantage of the indexed primary/foreign key present for the join relations of this query. The Velox team considers taking advantage of an index structure outside their project's scope. However, BOSS, which implements indexed primary/foreign key joins, outperforms Velox by a factor of 3x. HeavyDB performs similarly to Velox, and is thus outperformed by BOSS by a similar factor.

Large unfiltered joins (Q9). Q9 shows similarities with Q3. The execution is dominated by join calculation but at higher cardinality. In that case, Velox takes better advantage of multi-threading and outperforms DuckDB by a factor of 1.8x. MonetDB still outperforms Velox by a factor of 6x. Similar to Q3, GPU-accelerating the join operations allows BOSS to outperform Velox by a factor of 2x. However, MonetDB outperforms BOSS by a factor of 3x due to the inferior performance of the sorting and the aggregation evaluated by Velox on the CPU. HeavyDB outperforms BOSS by a factor of 1.5x due to the more efficient GPU-accelerated implementation of sorting and grouping but is still not as efficient as MonetDB's.

High-cardinality grouping and aggregation (Q18). Q18 also executes large unfiltered join like Q9, but aggregation dominates the query runtime due to the high cardinality of the grouping. MonetDB and DuckDB outperform Velox due to better performance of the grouping operator (which accounts for 70% of the total execution time in Velox). By benefiting from GPU acceleration only for one of the two joins, BOSS outperforms all the baselines by the factors 2x (MonetDB), 3x (Velox), 4x (HeavyDB) and 6x (DuckDB).

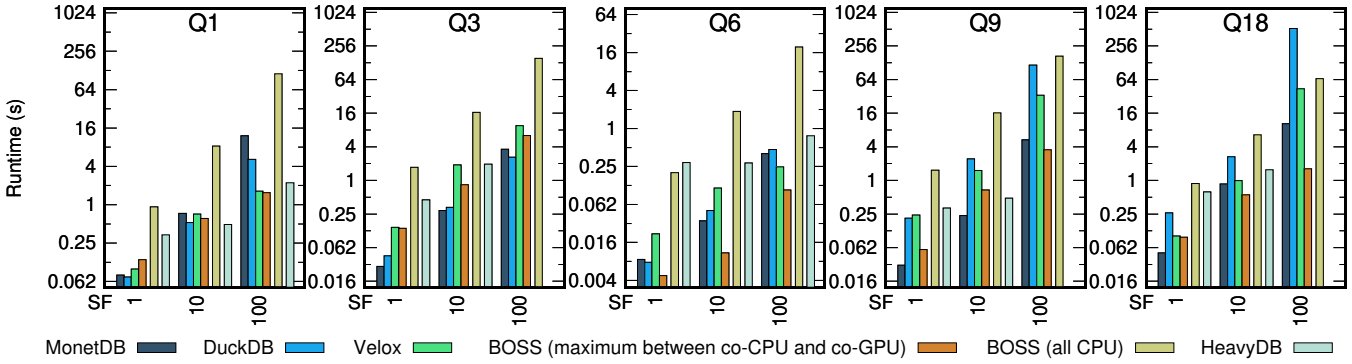

Figure 13: Runtime for TPC-H on five representative queries with SFs 1, 10 and 100

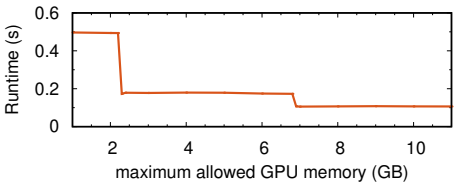

Figure 14: Q6 runtime (SF 100) for various GPU memory size

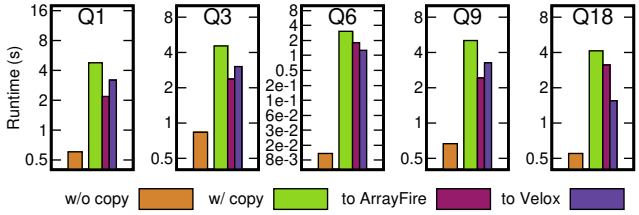

Figure 15: TPC-H Query runtime (SF 10) with and without data copy between engines

These results show that minimal integration efforts to GPU-accelerate relational operators allow BOSS to significantly improve the performance of the Velox CPU-only implementation and outperform, for some queries, other highly efficient CPU-based DMSs.

7.3 Scalability

To assess the scalability of our co-processing approach, we run the same TPC-H queries at SFs from 1 to 100.

At SF 1, (Figure 13), all CPU-based DMS and BOSS, perform (relatively to their scale) similar to SF 10. It shows that the runtime overhead of these systems is not significant, even on such a small dataset. However, HeavyDB has such overhead for evaluating Q6: the results for SF 1 and 10 are identical, which explains why heavyDB is outperformed by BOSS on this query.

At SF 100, data required for the GPU processing does not entirely fit into the GPU memory for any of the five queries. The missing bars for HeavyDB in Figure 13 are due to HeavyDB failing to process Q3, Q9 and Q18 because the implementation does not provide a fallback method when the GPU memory is insufficient. Unlike HeavyDB, BOSS' GPU-accelerated kernel can partially evaluate the query and leave it to the CPU kernel to evaluate the remaining operations, as explained in Section 6.4. The results show that, while fewer operations are GPU-accelerated, the implementation always benefits from GPU acceleration: BOSS outperforms the Velox implementation for all queries but to a lesser extent than with smaller datasets for the queries that operate with high cardinality (Q1, Q3 and Q6). For Q9 and Q18, BOSS outperforms Velox by factors of 8x and 32x, taking advantage of indexes and GPU-accelerated joins.

7.4 Effect of scarce GPU memory

To assess the effects of the partial evaluation strategy designed for the GPU-accelerated kernel, we measure how much the data not fitting into the GPU memory affects the runtime execution. For this experiment, we evaluate TPC-H Q6 with SF 100 and vary the maximum size of data allowed to be transferred to the GPU from 0GB (i.e., no GPU acceleration) to 11GB (i.e., maximum GPU transfer allowing to GPU-accelerate two of the three selections). As expected, the results in Figure 14 show a significant performance improvement at 2.3GB when the memory is sufficient to evaluate the first (low-cardinality) selection and a smaller gain at 6.9GB (when the kernel can evaluate the second, higher-cardinality, selection). The partial evaluation allows to maximize GPU acceleration's benefits for any available GPU memory resources.

7.5 Ablation Study

To assess the benefits of the design choices for our framework, we measure the runtime overhead with two modified versions of BOSS: (1) With data copy between the kernels to assess the benefit of zero-copy data transfer; (2) With the dynamic dispatch of the data elements one by one to assess the benefits of fast-path in the expression API. For both, we evaluate TPC-H queries at SF 10.

For the first experiment, the results in Figure 15 show a high cost for data movement: between four and 250 times the execution runtime depending on the query. The transfer cost from the GPU-accelerated kernel to the CPU kernel is generally lower (but still significant) than the transfer from the storage kernel since the relations passed by the storage kernel are not yet filtered before evaluating the relational operators. However, Q1, Q3 and Q9 have

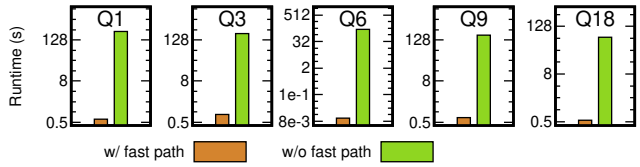

Figure 16: TPC-H Query runtime (SF 10) with and without fast-path for the dynamic dispatch of the column data

Figure 17: Normalized execution time breakdown

a high GPU-to-CPU transfer cost due to the high cardinality before the aggregation on the CPU. This result confirms the significant performance benefit from avoiding data transfer between the kernels, which is made possible by the proposed data exchange format used to integrate DMSs into a single data processing pipeline.

For the second experiment, the results in Figure 16 show an even higher cost when the fast-path is disabled: the query execution increases by a factor of 100x to 6000x depending on the query. This demonstrates that the fast-path approach avoids the execution overhead of dynamic dispatching.

7.6 BOSS Framework Overhead

To assess the execution overhead for traversing and modifying expressions (for the query execution and the shallow-transformation of the data layouts), we measure the execution time of the BOSS framework. For this experiment, we run the TPC-H queries with SFs 1 to 100 and profile the execution with Intel VTune v2020 [25] to isolate the execution time of the functions calls from the BOSS framework. The results in Figure 17 confirm that BOSS framework's overhead is negligible compared to the total execution time.

8 RELATED WORK

Composable DMSs. While DMS researchers have studied extensible designs for a long time [11, 19], the focus on the composition of separately designed components is a relatively recent trend. Meta's Velox [38] demonstrates the utility of a single execution engine for multiple systems. It achieves high-performance data processing by integrating features from different DMS into a unified execution engine. Delta Lake [29] provides a unified storage format to bring ACID transactions to big data workloads on cloud object stores. GoogleSQL [18] serves as a standard for various Google products, facilitating sharing of parser, optimizer, and operators. All of these focus on improving the reusability of the components. We, however, focus on a framework to compose kernels.

GPU Acceleration for DMS. GPU acceleration has been widely used in data analytics. Numerous contributions have been made to utilize GPU resources to speed up classic algorithms, such as sorting [33, 44, 46, 49] and nearest neighbor search [16, 17, 52]. DBMS acceleration using GPU has also received attention. Two main options exist: (1) offloading operators, (2) building a standalone GPU DMS.

While Offloading Operators, the CPU coordinates the execution process, and the GPU serves as co-processor. Join algorithms on GPUs have been extensively [37, 42, 43, 48, 51], while Yuan et al. [55] study non-join operators. Although some researchers studied offloading entire subplans [7, 55], frequently transferring data across devices results in poor performance. However, to the best of our knowledge, our proposal is the first to study runtime-opportunistic offloading of operators in the context of a full DMS.

GPU DMSs avoid the PCI-bottleneck but require the GPU to process all operators. Commercial systems such as HeavyDB [22] and BlazingDB [2] adopt GPUs as their primary execution engine. Ocelot [23] is a hardware-oblivious extension for databases that include a set of GPU acceleration operators. To improve the efficiency, some systems implement fine-grained pipeline optimizations [12, 15, 30, 53] while others [36] pipelines entire operators through OpenCL 2.0 pipes [21]. For CPU/GPU co-processing systems, data placement and transfer are the primary challenges [3, 8, 45]. Yuan et al. [55] propose to use Universal Virtual Addressing to extend the GPU memory, while CoGaDB [7] and Yogatama et al. [54] cache data on the GPU. All these techniques are applicable but orthogonal to our work.

9 CONCLUSION

Composable DMS design promises a new generation of DMSs that are more extensible, easier to develop, cheaper, safer and more efficient while taking advantage of the latest techniques and technologies. However, current DMS architectures obstruct this goal by requiring substantial boilerplate code and expensive copies between components. To address this problem, we propose a fundamentally new architecture developed around the idea of partial query evaluation and a unified exchange format for data and executable query plans. We demonstrate the feasibility and utility of our approach by implementing BOSS [6], a system that integrates the Apache Arrow storage library, the GPU-accelerated ArrayFire compute kernel and the CPU-oriented Velox kernel. We demonstrated that BOSS is virtually overhead-free and achieves significant performance improvement over the CPU-only Velox kernel and even outperforms the highly-optimized GPU-only DMS HeavyDB for some queries.

We argue that Partial Query Evaluation is the right framework for designing and implementing composable DMSs and will enable the integration of many data-oriented techniques and technologies. This includes the integration of new hardware, such as smart storage devices or application-specific circuits, as well as cloud services, such as serverless computation and smart cloud object stores. It will also accelerate the integration of new data management techniques, such as learned indices and new data models. Finally, this new paradigm holds the potential to simplify existing DMS architectures: kernels could, e.g., apply a mix of processing and optimization, effectively implement an adaptive query optimizer. We will explore such opportunities in future work.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

REFERENCES

[1] Apache Arrow. 2023. Retrieved 2023-02-24 from https://arrow.apache.org

[2] BlazingSQL. 2023. BlazingDB. Retrieved April 14, 2023 from https://github.com/BlazingDB/blazingsql

[3] Nils Boeschen and Carsten Binnig. 2022. GaccO - A GPU-Accelerated OLTP DBMS. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1003–1016. https://doi.org/10.1145/3514221.3517876

[4] Peter Boncz and M. L Kersten. 2002. *Monet: A next-Generation DBMS Kernel for Query-Intensive Applications*. Ph.D. Dissertation. Universiteit van Amsterdam.

[5] 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*. Springer, 61–76.

[6] BOSS. 2023. Retrieved 2023-12-12 from http://boss.lsds.uk

[7] Sebastian Breß. 2014. The Design and Implementation of CoGaDB: A Column-Oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14 (2014), 199–209.

[8] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-Accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1891–1906. https://doi.org/10.1145/2882903.2882936

[9] José Cambronero, John K Feser, Micah J Smith, and Samuel Madden. 2017. Query Optimization for Dynamic Imputation. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1310–1321.

[10] Jiashen Cao, Rathijit Sen, Matteo Interlandi, Joy Arulraj, and Hyesoon Kim. 2023. Revisiting Query Performance in GPU Database Systems. arXiv:2302.00734 [cs.DB]

[11] Michael J Carey, David J DeWitt, Goetz Graefe, David M Haight, Joel E Richardson, Daniel T Schuh, Eugene J Shekita, and Scott L Vandenberg. 1988. The EXODUS Extensible DBMS Project: An Overview. (1988).

[12] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. HetExchange: Encapsulating Heterogeneous CPU-GPU Parallelism in JIT Compiled Engines. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 544–556. https://doi.org/10.14778/3303753.3303760

[13] The Transaction Processing Council. 2013. TPC-H Benchmark (Revision 2.16.0). Retrieved 2023-02-24 from http://www.tpc.org/tpch/

[14] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *Proceedings of the 2021 International Conference on Management of Data*. 445–458.

[15] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. 2018. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1603–1618. https://doi.org/10.1145/3183713.3183734

[16] Vincent Garcia, Eric Debreuve, and Michel Barlaud. 2008. Fast k Nearest Neighbor Search Using GPU. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*. 1–6. https://doi.org/10.1109/CVPRW.2008.4563100

[17] Vincent Garcia, Éric Debreuve, Frank Nielsen, and Michel Barlaud. 2010. K-Nearest Neighbor Search: Fast GPU-based Implementations and Application to High-Dimensional Feature Matching. In *2010 IEEE International Conference on Image Processing*. 3757–3760. https://doi.org/10.1109/ICIP.2010.5654017

[18] Google. 2023. Google SQL. Retrieved 2023-04-13 from https://cloud.google.com/spanner/docs/reference/standard-sql/overview

[19] G. Graefe. Feb./1994. Volcano-an Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering* 6, 1 (Feb./1994), 120–135. https://doi.org/10.1109/69.273032

[20] Chris Gregg and Kim Hazelwood. 2011. Where Is the Data? Why You Cannot Debate CPU vs. GPU Performance without the Answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 134–144.

[21] K. O. W. Group. 2023. The OpenCL Specification. Retrieved April 15, 2023 from https://registry.khronos.org/OpenCL/specs/opencl-2.0.pdf

[22] HEAVY.AI. 2023. HeavyDB. Retrieved April 14, 2023 from https://www.heavy.ai/product/heavydb

[23] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for in-Memory Column-Stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. https://doi.org/10.14778/2536360.2536370

[24] Denis Hirn and Torsten Grust. 2021. One WITH RECURSIVE Is Worth Many GOTOs. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, 723–735. https://doi.org/10.1145/3448016.3457272

[25] Intel. 2023. VTune Profiler. Retrieved 2023-02-24 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[26] Will Jones, Tony Field, and Tristan Allwood. 2012. Deconstraining DSLs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. ACM, Copenhagen Denmark, 299–310. https://doi.org/10.1145/2364527.2364571

[27] Julia. 2023. Retrieved 2023-02-24 from https://julialang.org

[28] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proceedings of the VLDB Endowment* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[29] Delta Lake. 2023. Retrieved 2023-04-19 from https://delta.io/

[30] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1647–1658. https://doi.org/10.14778/3007328.3007331

[31] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2020. Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 1633–1649. https://doi.org/10.1145/3318464.3389705

[32] James Malcolm, Pavan Yalamanchili, Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, and John Melonakos. 2012. ArrayFire: A GPU Acceleration Platform. In *SPIE Defense, Security, and Sensing*, Eric J. Kelmelis (Ed.). Baltimore, Maryland, USA, 84030A. https://doi.org/10.1117/12.921122

[33] Tobias Maltenberger, Ivan Ilic, Ilin Tolovski, and Tilmann Rabl. 2022. Evaluating Multi-GPU Sorting with Modern Interconnects. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1795–1809. https://doi.org/10.1145/3514221.3517842

[34] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

[35] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[36] Johns Paul, Jiong He, and Bingsheng He. 2016. GPL: A GPU-Based Pipelined Query Processing Engine. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1935–1950. https://doi.org/10.1145/2882903.2915224

[37] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1413–1425. https://doi.org/10.1145/3448016.3457254

[38] 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. https://doi.org/10.14778/3554821.3554829

[39] 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. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2679–2685.

[40] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science Towards Embedded Analytics. (2020).

[41] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proceedings of the VLDB Endowment* 11, 4 (Dec. 2017), 432–444. https://doi.org/10.1145/3186728.3164140

[42] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient Join Algorithms for Large Database Tables in a Multi-GPU Environment. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 708–720. https://doi.org/10.14778/3436905.3436927

[43] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM '17)*. Association for Computing Machinery, New York, NY, USA, Article 17. https://doi.org/10.1145/3085504.3085521

[44] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. 2020. Bonsai: High-performance Adaptive Merge Tree Sorting. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 282–294. https://doi.org/10.1109/ISCA45697.2020.00033

[45] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. A Study of the Fundamental Performance Characteristics of GPUs and CPUs for Database Analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595

[46] Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient Top-K Query Processing on Massively Parallel Hardware *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1557–1570. https://doi.org/10.1145/3183713.3183735

[47] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, Macao, Macao, 698–709. https://doi.org/10.1109/ICDE.2019.00068

[48] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 698–709. https://doi.org/10.1109/ICDE.2019.00068

[49] Elias Stehle and Hans-Arno Jacobsen. 2017. A Memory Bandwidth-Efficient Hybrid Radix Sort on GPUs. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 417–432. https://doi.org/10.1145/3035918.3064043

[50] Substrait. 2023. Retrieved 2023-02-24 from https://substrait.io

[51] Yuchao Tao, Xi He, Ashwin Machanavajjhala, and Sudeepa Roy. 2020. Computing Local Sensitivities of Counting Queries with Joins. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 479–494. https://doi.org/10.1145/3318464.3389762

[52] Patrick Wieschollek, Oliver Wang, Alexander Sorkine-Hornung, and Hendrik Lensch. 2016. Efficient Large-Scale Approximate Nearest Neighbor Search on the Gpu. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2027–2035.

[53] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. 2012. Kernel Weaver: Automatically Fusing Database Primitives for Efficient GPU Computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 107–118. https://doi.org/10.1109/MICRO.2012.19

[54] Bobbi W. Yogatama, Weiwei Gong, and Xiangyao Yu. 2022. Orchestrating Data Placement and Query Execution in Heterogeneous CPU-GPU DBMS. *Proc. VLDB Endow.* 15, 11 (July 2022), 2491–2503. https://doi.org/10.14778/3551793.3551809

[55] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. 2013. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.* 6, 10 (Aug. 2013), 817–828. https://doi.org/10.14778/2536206.2536210

[56] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100-A DBMS in the CPU Cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.