Automatic Array Transformation to Columnar Storage at Run Time

Lukas Makor*
Johannes Kepler University
Linz, Austria
lukas.makor@jku.at

Sebastian Kloibhofer*
Johannes Kepler University
Linz, Austria
sebastian.kloibhofer@jku.at

David Leopoldseder
Oracle Labs
Austria
david.leopoldseder@oracle.com

Daniele Bonetta
Oracle Labs
Netherlands
daniele.bonetta@oracle.com

Lukas Stadler
Oracle Labs
Austria
lukas.stadler@oracle.com

Hanspeter Mössenböck
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Today's huge memories make it possible to store and process large data structures in memory instead of in a database. Hence, accesses to this data should be optimized, which is normally relegated either to the runtimes and compilers or is left to the developers, who often lack the knowledge about optimization strategies. As arrays are often part of the language, developers frequently use them as an underlying storage mechanism. Thus, optimization of arrays may be vital to improve performance of data-intensive applications. While compilers can apply numerous optimizations to speed up accesses, it would also be beneficial to adapt the actual layout of the data in memory to improve cache utilization. However, runtimes and compilers typically do not perform such memory layout optimizations. In this work, we present an approach to dynamically perform memory layout optimizations on arrays of objects to transform them into a columnar memory layout, a storage layout frequently used in analytical applications that enables faster processing of read-intensive workloads. By integration into a state-of-the-art JavaScript runtime, our approach can speed up queries for large workloads by up to 9x, where the initial transformation overhead is amortized over time.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic compilers**; **Runtime environments**; Interpreters; • **Information systems** → *Column based storage.*

KEYWORDS

Columnar Storage, Array Storage, Program optimization, Dynamic Language, Dynamic Compilation

ACM Reference Format:
Lukas Makor, Sebastian Kloibhofer, David Leopoldseder, Daniele Bonetta, Lukas Stadler, and Hanspeter Mössenböck. 2022. Automatic Array Transformation to Columnar Storage at Run Time. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3546918.3546919

*Both authors contributed equally to the paper

MPLR '22, September 14–15, 2022, Brussels, Belgium
© 2022 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium*, https://doi.org/10.1145/3546918.3546919.

1 INTRODUCTION

The amount of created, gathered and processed data is constantly increasing [46, 59, 63] and so are the demands for data analytics and computation [33, 37, 41, 50, 57]. With data visualization and computing extending more and more into dynamic languages and runtimes, the problem of optimizing those processes is no longer restricted to databases and query languages. However, while databases can resort to query planning [13, 28, 70] and storage optimization techniques [1, 2, 22], programming languages have to rely on the developers to optimize the data layout and processing.

To model large data workloads, developers use data structures available in their programming languages. One fundamental data structure available in almost every programming language are arrays. Thus, modern runtimes and compilers are tailored to optimize accesses to arrays. However, one optimization aspect that is hardly targeted by current runtimes, while playing a key role in database optimizations, is specializing the memory layout of a data structure based on its usage. Processing contiguous memory, for example, leads to a better cache utilization (fewer cache misses) than random memory access [18]. Records are typically stored as individual objects. Consequently, the objects representing the records are scattered throughout the heap. Hence, successive accesses to record data result in many cache misses and suboptimal performance.

To tackle this problem, we propose optimizing arrays of (uniform) objects on-demand via run-time transformation of the underlying memory layout to a *columnar storage format*. Columnar storage is frequently used in databases [8, 22, 64]. There, all records of a table are decomposed into their individual column values, with each column arranged linearly in memory. Records are then merely position indicators to the corresponding property values in each column. This linear representation enables more efficient data access due to improved cache utilization, for example, when iteratively accessing the same column of different records.

```
1 let totalSalaries = 0
2 for (let i = 0; i < emps.length; i++) {
3     const e = emps[i]
4     e.bonus = Math.floor(e.totalRev * 0.02)
5     e.sal = e.sal + e.bonus
6     totalSalaries += e.salary
7 }
```

Listing 1: A method that calculates new salaries in a loop

Listing 1 showcases an example of a bulk operation that may appear in a typical data-intensive application:
The bonuses and salaries of a large array of employees are updated and the resulting salaries are accumulated. The left-hand side of Fig. 1 shows the memory layout of such a data structure in a language runtime such as Google V8 [6] or GraalVM [53], where the different employee objects are scattered across the heap. For this query, a columnar layout—as depicted on the right-hand side of Fig. 1—would bring several advantages: Loading the actual object references could be omitted, as the data is directly accessed via "property columns" with linear memory layout. Object field accesses could be replaced by array accesses, which would foster data locality and cache utilization [12, 39]. On modern architectures, the improved loop structure may even enable loop vectorization [11] by using SIMD instructions [35, 36].

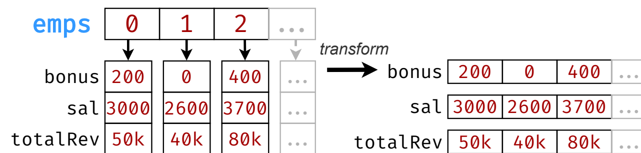


Figure 1: The original memory layout and the columnar layout of the *salary* example

We chose to implement our approach for JavaScript, because of the many different areas in which JavaScript is used, ranging from web development [27], (in-memory) databases [48], and visualizations [9] to server-side data processing using Node.js [15]. We integrated this approach into the GraalVM JavaScript runtime [51] based on the Truffle Framework [29, 65, 68], as GraalVM JavaScript's implementation simplifies the integration of columnar arrays into the language. The combination of the Truffle Framework and the high-performance GraalVM JIT Compiler [19, 44, 62] enables us to integrate custom compiler optimizations specifically targeting accesses to columnar arrays.

We focus on large arrays of objects that are used in read-intensive workloads, ideally accessed in loops (*queries*) and enable a restoration of the original representation if unsupported accesses occur. Using heuristics based on profiling information, we automatically identify suitable arrays and transform them at run time, hence the implementation imposes no additional effort for the developer.

In summary, this paper contributes the following:

(1) A novel storage strategy for arrays of objects, which enables automated transformation into a columnar layout. This includes a transition system to both selectively apply and revert the transformation depending on the array usages.
(2) An ECMAScript [23] compliant implementation of this storage strategy in the production-quality GraalVM JavaScript runtime. Our implementation achieves speed-ups of up to 9x for specific queries on large workloads.
(3) A set of novel compiler optimizations that utilize the tight integration of the Truffle framework and the GraalVM Compiler, to selectively optimize accesses to arrays based on their observed run-time state.
(4) An evaluation of our approach based on microbenchmarks and TPC-H queries [14].

The rest of this paper is structured as follows: In Section 2, we provide background information on the runtime and compiler infrastructure on which our approach is based. Section 3 details our storage transformation approach in JavaScript and describes the integration into the language implementation, while the necessary performance optimizations are part of Section 4. In Section 5, we describe some of the limitations of our current implementation. Section 6 contains an extensive evaluation of our approach. Sections 7 and 8 compare our implementation with similar approaches in literature and suggest potential extensions.

2 BACKGROUND

We implemented our approach by using the Truffle framework [65, 68] in combination with GraalVM [53] and the GraalVM Compiler [19, 44, 62]. This enables significant control over the data representation in the language itself, the run-time interpretation process as well as the partial evaluation and compilation process including the applied optimization techniques.

2.1 GraalVM

GraalVM [53] is a state-of-the-art, high-performance polyglot runtime environment. As a Java runtime, it enables the execution of Java bytecode languages such as Java, Kotlin, or Scala but also supports execution of dynamic (e.g., JavaScript and Ruby) or LLVM-based languages via Abstract Syntax Tree (AST) interpretation and compilation via partial evaluation [34, 42, 67, 69].

The GraalVM Compiler is the dynamic just-in-time (JIT) compiler of the GraalVM [19, 44, 62]. It transforms the bytecode into the graph-based intermediate representation Graal IR [19, 21] and applies aggressive optimizations to produce high-performance machine code. By using run-time profiling information, the GraalVM Compiler can furthermore apply speculative optimizations [21]. If an assumption fails, a *deoptimization* [20, 31] is triggered, the compiled code is invalidated and execution falls back to the interpreter.

In our approach, we use the GraalVM Compiler to apply optimizations to speed up accesses to transformed arrays.

2.2 Truffle Framework

Truffle [29, 65, 68] is a language implementation framework for executing guest languages on a Java virtual machine. A guest language is integrated by implementing an AST interpreter [69] using the Truffle API. Additionally, Truffle supports interoperability between different guest languages.

When executing on GraalVM, the AST is partially evaluated using profiling information gathered at run time [67]. The GraalVM

Compiler furthermore uses this information to optimize the generated machine code towards peak performance. Currently, there are a number of implementations of dynamic languages such as JavaScript [51], Ruby [54] or Python [52], but also LLVM-based languages such as C/C++ [60] are supported.

2.2.1 Truffle Object Storage Model. The Truffle object storage model (OSM) [66] is a generic, language-agnostic model that can be utilized to handle object storage for a variety of Truffle languages. As it was built with dynamic languages in mind, dynamically adding and removing properties from an object is an optimized feature of the OSM. It has built-in support for optimizations such as type specialization and polymorphic inline caches.

In the OSM, objects are equipped with immutable *shapes*, that describe their current memory layout and specify how individual properties can be accessed. Whenever an object modification would require altering the shape, the object is instead assigned a new shape and its memory layout is transformed. By caching these shapes at object accesses within the AST, the compiler can subsequently optimize property accesses by utilizing the known constant offsets. Each shape and its transition history—the property additions and removals that resulted in this shape—form a unique combination. Specifically in JavaScript, these shapes and the available optimizations are comparable to V8's *hidden classes* [3].

2.3 GraalVM JavaScript

GraalVM JavaScript [51] is a high-performance, ECMAScript [23] compliant JavaScript implementation, built using the Truffle framework. As with other Truffle languages, GraalVM JavaScript also supports language interoperability. Furthermore, a custom Node.js backend [55] enables execution of Node-based applications using the Truffle Framework.

2.3.1 JavaScript Object Layout. GraalVM JavaScript [51] builds its object representation on top of the Truffle OSM (Section 2.2.1), where objects have dedicated constant shapes that may be switched at run time. In addition to information about the properties themselves, each shape also references a type descriptor, the so-called *object type*, which can be arbitrarily defined by the language implementation. In GraalVM JavaScript, this type descriptor is used to define the *behavior* of the corresponding object, i.e., to regulate accesses to the underlying object storage. This concept is used, for example, to differentiate between regular objects, functions, promises, and JavaScript proxies, whose property accesses may trigger underlying function calls.

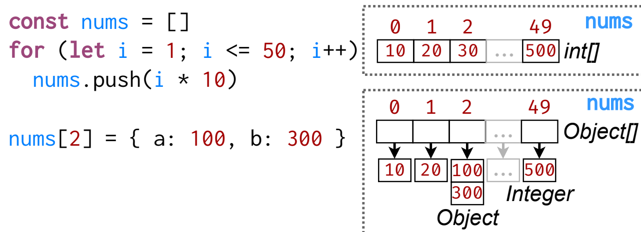


Figure 2: Exemplary array transition in GraalVM JavaScript

2.3.2 JavaScript Array Semantics. Arrays are not part of the Truffle OSM, thus GraalVM JavaScript defines its own schema to model the semantics of JavaScript arrays [23].

In JavaScript, arrays do not always constitute linear memory regions but can also have holes in between or they may not start at index 0. Moreover, they can be used to store elements of arbitrary user types, hence data may be highly polymorphic. Arrays in GraalVM JavaScript are also equipped with a *type descriptor*—the *array strategy*—, in this case representing the different array layouts. Upon modification, the array strategy therefore has to verify that the corresponding operation does not violate the current layout. Otherwise, a transition to a new strategy that allows the operation is initiated, potentially coupled with a transformation of the array storage to fit this new layout.

Fig. 2 showcases one such transformation, where a primitive integer array is transformed into an array of objects as soon as an object is inserted. This requires a transformation of all primitive integers to boxed values.

3 STORAGE TRANSFORMATION IN JAVASCRIPT

As a proof-of-concept, we previously created a run-time data analysis and storage transformation framework for the Truffle research language *SimpleLanguage* [40, 45]. However, for JavaScript—a language with a complex type system and intricate runtime semantics—a more sophisticated approach is required. Our goal is not to change the existing array implementation but rather to define a new *strategy* (cf. Section 2.3.2) that arrays can transition to at run time. Such a new strategy has to support the traditional ECMAScript semantics [23] and should add no overhead to arrays that are not in columnar format.

3.1 Array State Changes

As shown in Section 2.3.2, GraalVM JavaScript uses a complex model to represent JavaScript arrays and abide by the standard. When performing storage transformation on such arrays, we similarly have to ensure compliance and respect all allowed operations on arrays. Furthermore, transforming the array contents into a *columnar* format is an expensive process and must only happen, if we can be reasonably sure that the new structure yields benefits during further execution of the program. We achieved all that by introducing our own *array strategy* into the language. This strategy intercepts all accesses to the respective array and employs counters to automatically identify large and frequently *read* arrays. As accumulating such metrics causes overhead upon each array access, we developed a 4-state approach for how to migrate arrays to this strategy. This approach allows us to gradually introduce more tracking mechanisms as the array becomes more suitable for storage transformation.

Fig. 3 shows the different states an array has to traverse in order to be eligible for storage transformation:

(1) As soon as an element is added to an empty array, the strategy is set according to the element type. We focus on *arrays of objects*, hence we only consider arrays with the built-in strategy `JSObjectArray` (and no primitive arrays). This process is still part

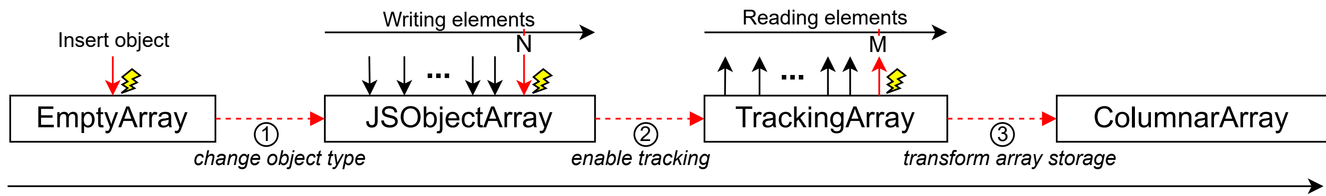


Figure 3: The different strategies that an array has to acquire to be considered a viable candidate for transformation

of the built-in GraalVM JavaScript array transitions. The `JSObjectArray` strategy also already signals that the array, *i*) only contains objects (no mixtures of primitive values and objects), *ii*) starts at 0 and *iii*) is strictly contiguous, i.e., it does not contain any holes. However, at this point the array may still contain objects of any shape. Also note that only the references to the array elements are stored in contiguous memory, while the objects themselves are still scattered all over the heap.

② As new items get added to the array, we check whether the array size exceeds a configurable threshold ($T1$). At that point, we trigger a transition to our custom `TrackingArray` array strategy. Transitioning to the `TrackingArray` strategy does not change the array layout, but only enables more sophisticated tracking: We count read accesses to the array and automatically change the strategy again whenever an invalid access occurs (e.g., removing an element from the array). Typically, the tracking and the corresponding array transitions happen during interpretation and thus do not affect partial evaluation and compilation.

③ Finally, if the read count exceeds another configurable threshold ($T2$), a transition to the `ColumnarArray` strategy is triggered and the array layout is transformed.

In the current implementation, threshold $T1$ is by default set to 50000 and threshold $T2$ to 25000. Based on our experience, these numbers strike a balance between a moderate minimum array size and still allowing for the optimizations to significantly speed up processing due to a certain number of expected accesses.

3.2 Array Transformation

The transformation of an array is split into two parts: First, we verify the actual array contents to make sure that they indeed fit into a columnar layout. Then, we allocate the resulting data structure and copy the data. This part constitutes most of the overall overhead. Fig. 4 shows the memory layout of an array before and after the transformation. The top part of the figure displays the initial layout, akin to the one presented in the example in Fig. 1. In the following sections, details about the transformation process that results in the components and memory layout depicted in the bottom part of Fig. 4 are presented.

Array Verification. As JavaScript is a dynamic language, we cannot statically check whether the array holds only values of a single type. Instead, we have to check that all objects in the array share the same *shape* (cf. Section 2.2.1).

In the same sense, we check every object property for compliance and prevent transformation if they are incompatible (e.g., ECMScript [23] *accessor properties* or non-writable properties are not supported). If the verification fails, the process is interrupted and the array is not transformed.

Object Transformation. To achieve a true columnar layout, we allocate *property arrays* of the corresponding types for each property of the elements' shape and copy over the data. The bottom part of Fig. 4 depicts the resulting memory layout, with, e.g., the `bonus` values of all objects stored in one array. At the same time, the original properties in the objects are deleted to save memory. As a consequence, one problem arises: To read a property value of a particular object, typically the object reference and the property key are required. However, due to the transformation, a property access must now read the data from the given position in the corresponding property array and not from the object itself (see `propertyArrays` in Fig. 4). This means that we require additional information for property accesses, i.e., the array instance in which the property arrays are stored and the index in these arrays. Therefore, upon transformation we store those two pieces of information—the tuple representing the *array location*—as new properties `arrRef` and `arrIdx` in the object. In the example in Fig. 4, `arrRef` references `emps` and `arrIdx` contains the object's index in the array. For property accesses, this information is subsequently queried and allows access to the real data.

The *type descriptor* (see Section 2.3.1) of the objects in the array is changed to a custom *proxy* type, to reroute object accesses to the transformed storage. Additionally, we set a flag in the array to signal the layout change and to speed up status checks. In literature, objects that refer to columnar data structures are typically called *proxy objects* [47, 56], hence we also use this terminology throughout the rest of the paper. Note that these proxy objects are not to be confused with ECMAScript proxy objects [23].

3.3 Proxy-Based Object Access

To create the connection from a proxy object to the array, the two properties `arrayRef` and `arrayIdx` are installed in the proxy objects, as seen in Fig. 4. Hence, when a property of a proxy object is accessed, `arrayRef` is used to identify the transformed array and to access its property arrays. Subsequently, `arrayIdx` is used to access the property arrays at the correct position. During compilation, we apply various optimizations to improve the property access performance. These optimizations are further discussed in Section 4.

Multiple Arrays. When a proxy object is contained in multiple *transformed* arrays, it needs to track all the array locations it is part of (i.e., all array-position tuples). Therefore, we introduce an additional property in each proxy that is part of multiple arrays, that stores a *set* of all array-index tuples. Property read operations can be performed from any of these array locations, but write operations on proxies have to be performed on each columnar array that the object is part of to keep all the individual property arrays in sync.

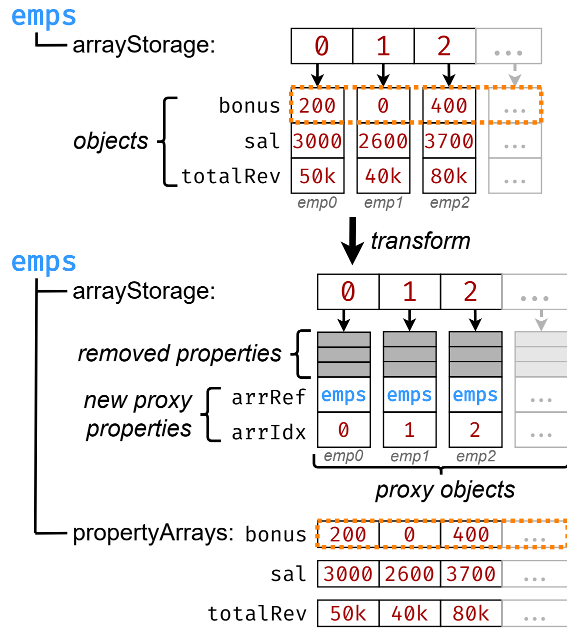


Figure 4: Storage transformation and resulting proxies

3.4 Array Restoration

Transformed arrays still have to exhibit the same semantics as common arrays. Therefore, whenever an operation occurs that is not supported by the columnar array strategy, we have to ensure that the original array is restored first. Unsupported operations include changes to an object shape—e.g., adding, removing, or changing the type of properties—and inserting new objects with a different shape into the array. To restore an array, we revert the columnar storage by recreating the original properties with the correct data from the property arrays. Subsequently, we restore all objects' type descriptors and change the array strategy (see Section 3.1).

4 PERFORMANCE OPTIMIZATION

As the integration of our approach represents a significant intrusion into the language implementation, it inherently also comes at a cost in terms of run-time performance. Fig. 5 depicts the contrast between a field access in GraalVM JavaScript without storage transformation **(a)** and reading the value from a property array via a transformed proxy object, as well as the optimizations that we apply: As shown in **(b)**, to read data from a proxy object, we first have to retrieve its *array location*—i.e., the tuple of array reference (`$arr`) and the position in the property arrays (`$i`). With this information, we can then load the fitting property array via the array reference and finally the actual value from the determined position in the property array.

By utilizing the GraalVM Compiler infrastructure, we can apply specific optimizations to the components that we introduce, to produce highly-optimized machine code and to compensate for the overhead. Some of these optimizations are depicted in Fig. 5 **(c)** to **(f)** and are described in detail in the following sections.

4.1 Access Intrinsics

By transforming the array storage and redirecting all accesses to object properties to the matching property arrays, we introduce a number of new memory accesses, which inherently slow down the performance of individual accesses.

The compiler adds a number of checks to ensure that memory accesses are valid (omitted from Fig. 5 **(b)** due to brevity). This includes boundary checks at array accesses, null checks when accessing objects, and contextually redundant type checks. By exploiting our knowledge about the created data structures, we can safely remove most of those checks during compilation by using custom compiler intrinsics that result in fast, unchecked accesses, safeguarded by the constraints of our data structure.

4.2 Marking Array Accesses

To perform optimizations at compile time, we have to detect accesses to columnar arrays and proxies in the IR. For each proxy, we also have to know both the array and the position where it was loaded from. Therefore, we mark each location where a proxy is loaded from a columnar array (`<mark>(e, emps, i)`). This is illustrated in Fig. 5 **(c)**.

4.3 Single-State-Compilation

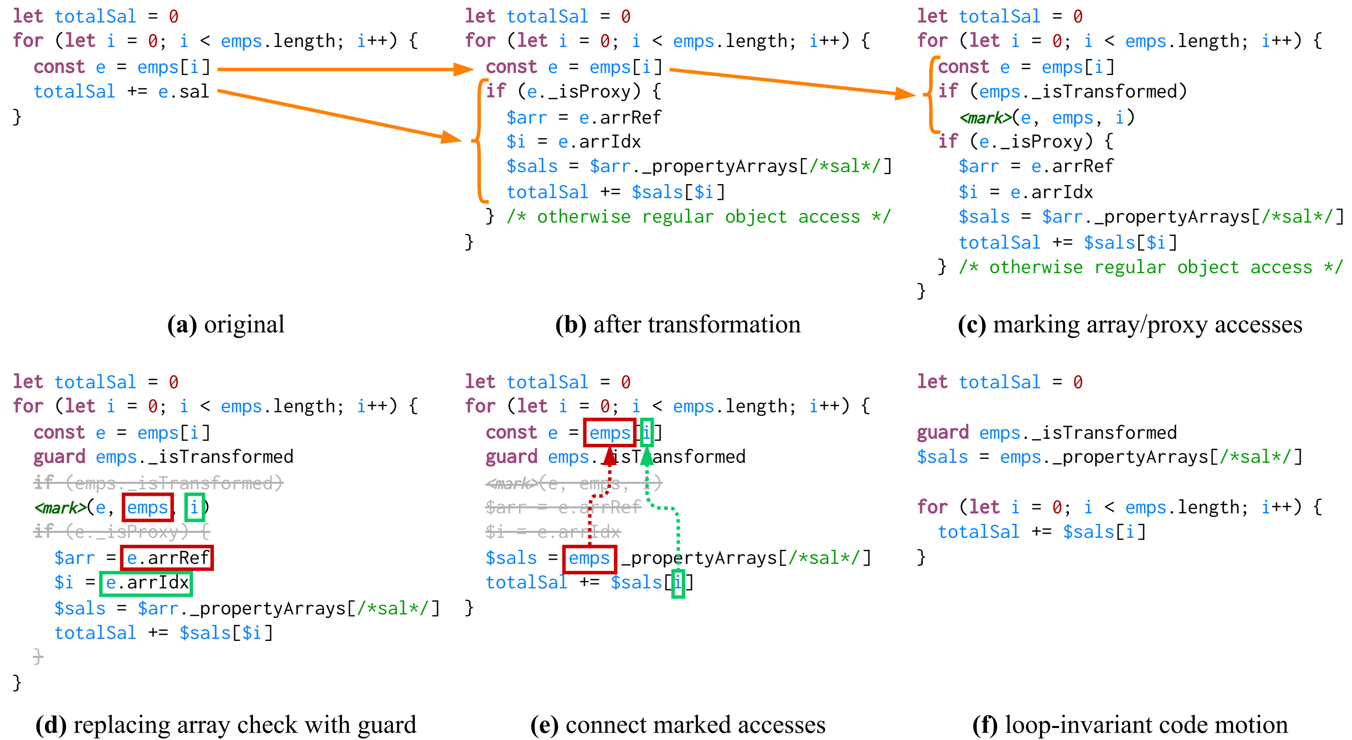
While optimizing individual accesses results in localized benefits, one goal of our approach is to enable additional compiler optimizations due to the new data layout. Upon compilation, we therefore want to ensure that only one state of an array (i.e., *unmodified* or *transformed*) is compiled. By focusing on just one of these variants, we can apply much more sophisticated optimizations on the individual accesses at the cost of possible deoptimization if the array is observed to be in a different state.

Fig. 5 **(d)** shows an example of such an optimization: The check `if (emps._isTransformed)` is replaced with a *guard* on the same condition, which would switch execution back from compiled code to interpretation if the guard fails. If the guard succeeds, we can perform additional optimizations on checks that are transitively true on transformed arrays. In the example, we can eliminate the check `if (e._isProxy)`, because we know that a transformed array contains only proxy objects.

4.4 Array Location Access Optimization

As stated above, after transformation, access to a property value must be rerouted to load the value from the respective property array. Without optimization, this process negatively impacts performance, as we effectively trade loading the array element and accessing the property for loading the array location tuple, loading the property array, and eventually loading the value.

While we cannot get rid of this overhead during interpretation, we can make use of the GraalVM Compiler to improve the situation: At compile time, the Truffle AST is partially evaluated, a process that optimizes the code but also inlines most of the nested method calls to enable subsequent optimizations [67]. In many cases, the result of this is that whole loops—such as depicted in the initial example (Listing 1)—end up in the same compilation unit, hence in a single Graal IR graph. For our approach this means that very often both loading of the array element (`const e = emps[i]`) and subsequently



(a) original **(b)** after transformation **(c)** marking array/proxy accesses

(d) replacing array check with guard **(e)** connect marked accesses **(f)** loop-invariant code motion

Figure 5: (Compile-time) Optimizations on property accesses (high-level representation)

accessing a property of the corresponding proxy object (`e.sal`) appear in *the same* IR graph.

We can detect such accesses at compile time using the *markings* described in Section 4.2. In a custom compiler phase, we detect array element accesses (`<mark>(e, emps, i)`) followed by proxy accesses (`e.arrRef`, `e.arrIdx`) as depicted in Fig. 5 **(d)**. For such patterns, we can skip loading the `arrRef` and the `arrIdx` by directly utilizing the marked values at the element access, i.e., using `emps` and `i` instead of `e.arrRef` and `e.arrIdx`.

Going back to the memory layout of a transformed array in Fig. 4, the marked proxy, array and index match the `arrRef` and `arrIdx` values stored in the proxy. Hence, rerouting those values results in the same program semantics but skips the additional property accesses, as depicted in Fig. 5 **(e)**. After this optimization, loading property values is effectively independent of loading the proxy object, as accessing the property arrays is decoupled from accessing the proxy object. Naturally, we have to ensure that no memory violation (e.g., a method call that may result in an array restoration) occurs between the marked access and the `arrRef` and `arrIdx` usages before applying this optimization.

4.5 Preserving Cache Locality

Objects may be part of multiple arrays. In this case, the transformed proxy objects must hold a *set* of array references and array indices to indicate the positions of the properties in the different property arrays. This is required to propagate property writes to all affected columnar arrays (cf. Section 5.3).

As shown in Section 4.4, we have to access the `arrRef` and `arrIdx` properties to access the real data of a proxy. However, from the perspective of the proxy, we cannot say which array location is the "right" one, i.e., the array and index this proxy was originally loaded from (`const e = emps[i]`). As each array contains exactly the same data for this proxy in the property arrays, we can, in fact, pick an arbitrary array location. While this is semantically correct, picking the "wrong" one may have a severe impact on the performance: If a proxy is loaded in a loop, choosing different array locations may result in accesses to different property arrays per iteration, essentially voiding all the benefits of the columnar layout.

Fortunately, the compiler optimization described in Section 4.4 already takes care of this problem: By always replacing the marked proxy property accesses with the marked arrays and indices, we ensure that we always access the same property arrays in each loop iteration. Thus, we can preserve cache locality in compiled code.

4.6 Follow-up Optimizations

The optimizations described above not only reduce the overhead of property accesses but also enable optimization of the loop itself. As shown in Fig. 5 **(f)**, the guard as well as loading the property array are now *loop-invariant* operations, hence existing compiler optimizations such as *loop-invariant code motion* [4] can move those instructions before the loop. Therefore, only the property array access remains within the loop.

5 CURRENT LIMITATIONS

Like all runtime-only optimizations based on heuristics, our technique might not always result in better performance. In this section, we summarize the main aspects that may limit its effectiveness in JavaScript applications.

5.1 Array Writing after Transformation

Inserting an object into a columnar array requires deconstruction of the object into the individual properties and N individual array writes for N properties. Additionally, the object has to be transformed into a proxy. Therefore, write operations to columnar arrays can cause significant slowdowns.

While write-intensive workloads are not a target of our approach, we nevertheless have to handle insertions in columnar arrays. As our focus was on optimizing read-intensive workloads, we decided to perform a restoration of the original array in case of a write access to the array. While this causes some overhead due to the restoration process, we can limit the overhead of the actual write operation, as the object no longer has to be deconstructed. Also, consecutive writes no longer suffer from this performance penalty.

5.2 Modification of Transformed Objects

While we allow modifications to transformed objects, we have to ensure that those modifications abide by the shared shape's constraints. Therefore, if a modification threatens to violate this contract, we force a restoration of the whole array. One problematic aspect of this is storing primitive numbers: In Truffle, those types are mapped to Java primitives, i.e., separate types for various integer ranges and floating-point numbers. Due to JavaScript's numeric type encompassing both integer and floating-point numbers, a restoration may also happen if we assign to a numeric property, previously assumed to be integer, a floating-point value.

5.3 Objects in Multiple Arrays

When a proxy object is part of multiple columnar arrays, all arrays have to contain the object's property data in their property arrays. Therefore, if the object is modified, the changes have to be mirrored in all arrays that contain the object. We specifically check whether an object is part of only a single array and introduce optimizations for this case. However, if a proxy is contained in N columnar arrays, modifying the object requires N writes to the different property arrays, or—in the worst case—a restoration of all N arrays if the modification is not supported by the columnar layout.

5.4 Contiguous Array Requirement

As mentioned before, we rely on an existing array strategy to signal that an array is eligible for storage transformation. However, this dependency also means that we currently do not support storage transformation of non-contiguous arrays (*arrays with holes*) or non-zero-based arrays.

5.5 GC Problems Due to Proxies

While the link of proxy objects to their corresponding arrays is necessary to access the corresponding data, it also may cause increased memory consumption: If an array is no longer referenced from the user code, array references within a proxy may keep it alive with respect to garbage collection. Fundamentally, a transformed array can only ever be collected when none of its proxy objects are referenced from outside the array.

6 EVALUATION

Our approach is primarily designed to speed up frequently executed queries on large arrays. Over time, the optimizations should amortize the initial tracking and transformation overhead. Unfortunately, typical JavaScript benchmarks rarely contain such use cases. Rather, most data-intensive benchmarks use database languages such as SQL. Hence, we evaluated the performance of our approach both with custom microbenchmarks to analyze specific scenarios and with a custom JavaScript port of the TPC-H Decision Support Benchmark [14] to measure the impact on business-oriented queries. Additionally, we evaluated the impact of the transformation process itself. To ensure compliance with the standard, we tested our implementations by utilizing the V8 and ECMAScript test suites [24]. Combined, these suites consist of around 51000 test cases of which around 2000 attempted a storage transformation, with around 500 succeeding. The array verification prevented transformation of the other arrays. All benchmarks were executed on a dual-socket Intel(R) Xeon(R) CPU E5-2690 @ 8x2.90GHz with 32 logical cores and 192G available RAM. Our approach is based on GraalVM EE version 22.2.

6.1 Query Microbenchmarks

We designed a number of JavaScript microbenchmarks [1] to test the capabilities of our approach in various scenarios. Most microbenchmarks implement a specific query that operates on a large (array) dataset with a predefined *workload size*, notably *filter operations* (collecting all objects of an array that satisfy a certain filter criterion), *aggregation operations* (aggregating certain properties of the objects in the array) and *property writing* operations (updating certain properties of all objects in the array).

We executed all microbenchmarks with different workload sizes (i.e., different array lengths) and varying numbers of iterations (i.e., how often the array was processed in that way). We ensure appropriate warmup for 180 seconds for each benchmark. Fig. 6 shows the results of this evaluation. As the numbers show, our approach is not suitable for smaller workloads or for workloads with only few iterations. This is due to the initial transformation overhead, which in some cases requires significantly more time than the actual operations in the loop (see Section 6.4).

As workload sizes and iterations increase, however, we see that the performance of some benchmarks improves significantly, with `aggregate` and `writeProperty` reaching speedups of over $7x$ and $5x$, respectively. The `combinedOps` benchmark benefits the most with a speedup of around 9x on the largest workload size. While other benchmarks such as `filter` and the `salaries` example from before are boosted as well, others—particularly those that use complex properties for filtering (`filterByDate`, `filterByString`)—are not or negatively impacted.

[1] The source code is available at: https://github.com/lmakor-jku/data-intensive-js-benchmarks

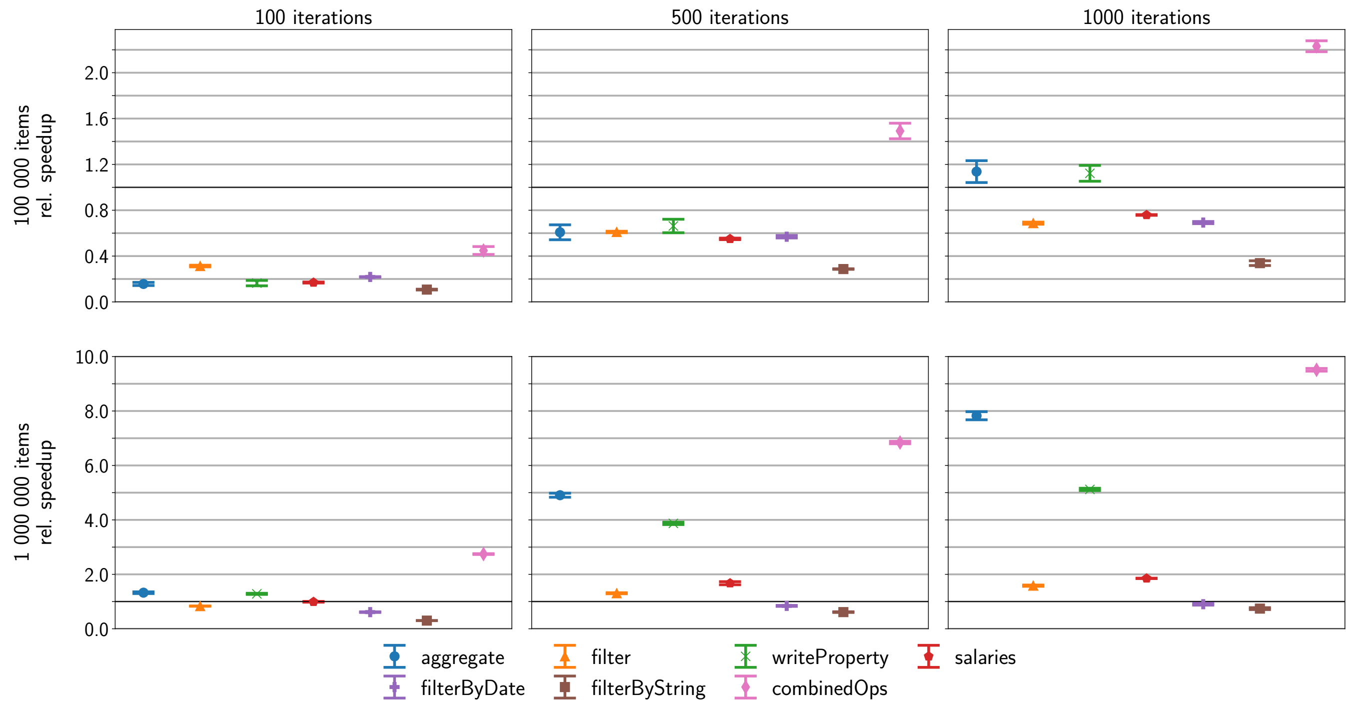


Figure 6: Microbenchmark throughput of our approach relative to baseline without storage transformation *(higher is better)*

The evaluation furthermore emphasizes that the ideal targets for our approach are large arrays: As depicted in the charts, even with 1000 iterations the overhead on the smaller workload is too impactful and prevents performance improvements on most benchmarks. With large workloads, however, even with fewer numbers of iterations we can already see speedups, as the columnar layout causes performance benefits in each loop iteration; only the filter-based benchmarks show regressions. By analyzing the emitted IR at various stages, we observed that filtering does not reach the same levels of improvement as the other operations, as the writing process (i.e., filling the resulting array) is more costly than reading and checking the actual property values. Similarly, date and string properties require additional property accesses on the resulting objects (e.g., reading the actual bytes of a string, accessing the timestamp value), hence our current approach for storage transformation cannot significantly improve such operations.

To summarize, we see that transforming an array into a columnar layout can indeed improve performance on bulk-processing operations over time. On the other hand, our approach suffers from the transformation overhead and is less suitable if the targeted loops exhibit more complex operational patterns.

6.2 Lodash Microbenchmarks

Lodash [16] is a real-world JavaScript library providing higher-order utility functions for processing arrays and objects. At the time of writing, the Lodash *npm* package [2] has more than 40 million weekly downloads. Hence, it seems reasonable to evaluate our approach using Lodash as an example for a library used in production to show the applicability of storage transformation. We used Lodash version 4.17.21.

In the following, we present the results for 4 different workloads, based on the already presented microbenchmarks, but this time implemented using Lodash instead of vanilla JavaScript. In each benchmark, we create an array with 1 million elements and we repeat the Lodash function 500 times on the whole array. For a benchmark execution, 10 warm-up iterations followed by 10 measured iterations were performed. In the following sections, only the call of the respective Lodash function is listed.

Filter. To implement the filter microbenchmark, we used Lodash's `filter` function. The `filter` function allows passing an array and a predicate, i.e., a function that returns a boolean value for a given input. It returns a new array containing all elements for which the predicate returns true.

```
_.filter(arr, function(o) {
    return o.a == 1
})
```

Aggregate. To implement the aggregate microbenchmark, we used Lodash's `reduce` function. The `reduce` function takes the array to be processed, an accumulation function, and an initial value. The accumulation function takes the accumulator and the current array element and returns a new accumulator that is subsequently used in the next iteration. For the first iteration, the given initial value is used as the accumulator. The result of the function is the

[2] https://www.npmjs.com/package/lodash

last computed accumulator. Hence, we use the `reduce` function to accumulate the properties `e` of all array elements.

```
_.reduce(arr, function(sum, o) {
    return sum + o.e
}, 0)
```

WriteProperty. To implement the writeProperty microbenchmark, we used Lodash's `forEach` function. `ForEach` takes the array to be processed and a function, which is called for each array element.

```
_.forEach(arr, function(o) {
    o.salary = o.salary + o.age * 10
})
```

Salaries. To implement the salary calculation microbenchmark, we used Lodash's `reduce` function, as it allows us to accumulate the total of the salaries and to update the `bonus` and `salary` properties.

```
_.reduce(arr, function(totalSalary, o) {
    o.bonus = Math.floor(o.totalRevenue * 0.02)
    o.salary = o.salary + o.bonus
    return totalSalary + o.salary
}, 0.0)
```

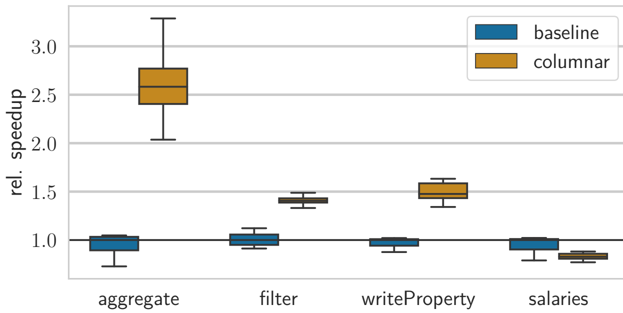


Figure 7: Lodash query time of our approach relative to baseline without storage transformation *(higher is better)*

The results of these Lodash-based microbenchmarks are depicted in Fig. 7. The numbers show that for most workloads the transformed version is faster than the baseline without our optimization. In the *salaries* benchmark, the compiler cannot remove all checks related to the `reduce` operation, hence the accesses do not become loop invariant. This results in moderate performance numbers. While the aggregate microbenchmark shows the biggest speedup of more than 2.5x on average, also the filter and write workloads are sped up by around 1.5x.

6.3 TPC-H

As *TPC-H* benchmarks are typically based on SQL, we used custom JavaScript ports to make them executable on the GraalVM JavaScript Node.js engine, while making sure that all queries are defined in native JavaScript without use of external libraries. We decided on 5 queries that mostly use array operations without subselects or more complex SQL operations, namely 1, 6, 12, 14, 19. We used the standardized TPC-H dataset generator with scale factor 1 to

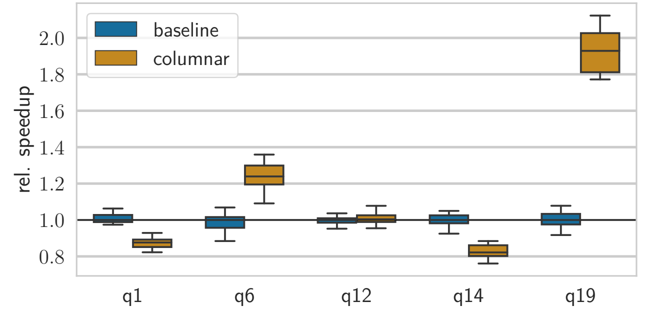


Figure 8: TPC-H query time on our approach relative to baseline without storage transformation *(higher is better)*

generate approximately 1GB of data and used a custom wrapper to load the resulting data and to execute the query. We took 20 measurements for each query and within each measurement applied the query 500 times on the data set.

Fig. 8 depicts the relative speedups on each of those queries. Particularly the performance of query 6 and query 19 improved, with the former achieving speedups of around 30% and the latter of more than 80%. The performance of query 12 is similar to the baseline, while query 1 and 14 are negatively impacted, with slowdowns of around 10% and 20%, respectively. We explain these differing results with the nature of the queries: Queries 6 and 19 mostly feature iteration over the individual tables and eventual grouping, while queries 1 and 14 use grouping and sorting operations or a complex aggregation in combination with date functions, respectively.

6.4 Transformation Impact

Table 1: Transformation overhead

Iterations (#)	Total (ms)	Query (ms)	Transf. (ms)	Transf. (%)
Aggregate				
1	1647	512	1135	68.9
100	1922	955	967	50.3
1000	2487	1444	1043	41.9
Filter				
1	1927	886	1041	54.0
100	4827	3804	1023	21.2
1000	25 145	24 042	1103	4.4
WriteProperty				
1	2385	1342	1043	43.7
100	3632	2638	994	27.4
1000	10 088	8995	1093	10.8
Salaries				
1	1838	655	1183	64.3
100	2536	1321	1215	47.9
1000	3538	2430	1108	31.3

While the transformation overhead is an issue, it is necessary for our approach and—as the previous figures showed—can be amortized by the speedups achieved through compiler optimizations. To better estimate the overall impact, however, we also evaluated this transformation overhead. Our setup for these benchmarks is as follows:

We create a randomized array of 1 million objects (with equal shapes to make them viable for transformation) and loop over this array while performing a particular operation (aggregation, filtering, etc.) on each element. Depending on the benchmark configuration, we execute the same operation on the whole array repeatedly (1, 100, 1000 repetitions). We measure the total time (i.e., overall time for all repetitions of a query) as well as the time required to perform the transformation.

Table 1 contains the results of this evaluation. If the benchmark is only executed with one iteration, the overhead in the queries takes up from 45% to around 70% of the total time. But the results show that this overhead does not scale with the number of iterations, such that with 1000 iterations it only takes up from 4% to 40% of the total time. As the transformation overhead is independent of the number of iterations, its share of the overall time decreases the more iterations are performed.

Overall, the results show that our approach is mostly beneficial for workloads that are bulk-processed a number of times, such that the performance improvements in repeated iterations can amortize the overhead over time.

7 RELATED WORK

7.1 Columnar Arrays

Mattis et al. [47] implemented a custom library that provides columnar data structures. They use *proxy objects* to represent the objects in an array and reroute property accesses to property arrays. The JIT compiler of the PyPy runtime subsequently applies optimizations to speed up accesses.

While they offer an API for programmers to enable a columnar storage layout, we automatically transform suitable arrays of objects at run time. While their API enables more explicit selection of the arrays that should be columnar, it also shifts the responsibility to the programmer. With our approach, the developer does not have to make estimates about the operations that will be performed on an array or about the intricacies of the data that it may hold.

In contrast to our approach, they lazily create proxies upon read access, to potentially remove them via escape analysis during compilation. However, this also prevents them from preserving *object identity* in proxies. As we always transform the contained objects into proxies, we cannot remove their allocation or accesses during compilation, but preserve referential identity. Nevertheless, as discussed in Section 4.6, we *can* move accesses out of loops.

In a similar work, Pivarski et al. [56] augment an analysis framework used in the field of high energy physics to efficiently use columnar arrays. Their datastore already supports a columnar layout, but as their queries are typically implemented in C++ using complex nested loops, the framework forces a materialization of the corresponding objects beforehand. They perform code transformation at AST level to replace object accesses with accesses to the columnar storage to improve performance and to prevent allocations. In contrast to both Mattis et al. [47] and our approach, they perform this transformation statically, without the use of a JIT compiler. Additionally, as their data is already in a columnar layout, they do not require additional run-time transformations to optimize accesses or queries.

7.2 Memory Layout Optimizations

Researchers have further shown a variety of other techniques to improve the memory layout of objects and data structures. These include *object splitting* that separates fields based on access frequency [26], *allocation pools* to partition heap objects based on profiling information and temporal relationships to improve data locality [10, 43], as well as implementations and theoretical models of *struct of array* layouts (columnar arrays) in areas such as high performance computing (HPC) [25, 32]. Due to our integration into GraalVM, we do not have a similar level of control over the memory management. In contrast to some of these works, our approach should apply optimizations without interference by the developer. Hence, it neither requires explicit specification of the desired memory layouts nor does it leave the choice of a fitting memory layout for particular data up to the developer.

7.3 Self-Managed Collections

Nagel et al. [49] implemented a C# API for *self-managed collections* (SMCs), which place their content in off-heap memory regions to circumvent garbage collection and to improve the data layout. In such collections, individual data blocks contain all objects of the same type. Each slot within a data block may contain an object and also tracks the current slot state, i.e., whether it contains an object and whether the contained object is used or can be reclaimed. Accesses are regulated via a global indirection table, where object references point to an entry in the indirection table, which in turn points to the slot containing the actual object data.

The off-heap memory model allows them to utilize fast, *unsafe* operations on low-level pointers to facilitate speedups. Modifications to the JIT compiler allow optimization of the indirect accesses. Additionally, SMCs are thread-safe and use *compaction* to reduce memory consumption.

Their approach requires the use of dedicated collection data structures while we try to optimize plain arrays. Their approach is designed to improve performance of LINQ queries, hence a specific collection API makes sense. In JavaScript (and other dynamic languages), however, such query APIs are far less widespread and—more importantly—not natively part of the language, thus we aim at optimizing data structures for more abstract query patterns, i.e., loop structures.

As JavaScript is increasingly used for data visualization and analysis, libraries and frameworks with JavaScript bindings such as *Apache Arrow* [5] and *RAPIDS* [58] are gaining popularity. They offer a columnar storage format for dynamic languages but provide no means of automatically transforming arrays into this format.

7.4 Storage Strategies for Collections

Storage strategies are also available in other runtimes and languages, such as described by Bolz, Diekmann, and Tratt [7]. They integrated this concept for Python collections into the PyPy runtime. Based on

the assumption that even in dynamic languages homogeneous collections of elements appear frequently, this allows them to adapt the storage strategies on-demand, depending on the inserted elements. Thus, they can reduce boxing overhead on primitive collections, bypass type checks, and speed up certain operations based on the type information.

Their storage strategies appear to be somewhat similar to the array storage strategies already available in GraalVM JavaScript, with specific focus on efficiently modeling primitive arrays. We extended these strategies for arrays of objects, for a more complex but also more restricted transformation with significant gains for certain workloads. Whereas they target multiple collection types within the language, our approach is currently restricted to arrays.

7.5 Query Optimization

In literature, there are many works on *query optimization* in programming languages:

Zhang et al. [71] optimize JavaScript loop structures via both offline and online analysis: First, they generate a number of query plans (i.e., variants of a loop pattern) and measure their performances. At run time, they then try the different plans on actual input data chunks and identify the best plan based on the execution time. Over time, their system can dynamically adapt this plan based on new performance observations. For loops, plan changes manifest themselves in loop body rewrites to reorder conditions and utilize SIMD operations. They integrated this system into the Truffle framework, where each query plan at run time is represented by a different AST, which is then compiled to machine code by the GraalVM Compiler.

Babelfish by Grulich, Zeuch, and Markl [30] goes in a similar direction. They also provide their own Truffle extension to enable optimization of polyglot user-defined functions (UDFs). They unify the representation of both the queries and their embedded UDFs and use custom Graal IR to access the data set and mimic query operations. Subsequently, the query is compiled to native machine code. By leveraging Truffle's partial evaluator, they can specialize operations and inline the nested queries to prevent the call overhead and enable follow-up optimizations such as scalar replacement and loop optimizations.

Schiavio, Bonetta and Binder [61] use Truffle and GraalVM for a polyglot query API. They consume SQL strings, optimize them with an external query planner and subsequently generate new AST nodes. Furthermore, their queries allow accesses to both in-language collections as well as to external data structures, by defining specific *providers*.

In contrast to our work, the above-mentioned approaches focus on optimizing individual queries without specifically transforming or modifying the underlying data structure. While we argue that optimization of individual queries may also benefit a data structure that is already in columnar layout, the goal of our work is to showcase the transformation of an array at run time and subsequently optimize accesses to the array. The latter enables query optimization in the compiler but to a lesser degree than the approaches listed above. Hence, it may be part of a future extension to specifically investigate query optimizations with columnar storages in mind.

7.6 Run-Time Data Structure Selection

There is also work on other dynamically adapting data structures, e.g., describing a high-level API to encode potential storage transformations [17] or automated indexing and transformation within a dedicated API based on observed query behavior [38]. In contrast to our work, however, those approaches typically require changes to the underlying program to explicitly *choose* the given APIs or explicitly define expected data structure transitions.

8 FUTURE WORK

Future extensions to our approach aim at *i*) investigating transformation potential of reference type properties (e.g., dates, strings), *ii*) preventing compilation-deoptimization cycles and *iii*) polyglot support for other Truffle languages.

8.1 Meta-Circular Storage Transformation

Arrays of objects with reference type properties often do not benefit from our approach. The emerging problem is best illustrated via an example: Assuming that a *transformable* array contains objects with `Date` type properties. `Date` objects typically store the actual timestamp in primitive properties, hence, using the date for a comparison requires accessing this property. If this access occurs for a columnar array in a loop, our optimizations can only replace the loading of the object holding the date property with an access to a property array (holding all the date references), but the access to the actual `Date` object containing the timestamp remains.

In the example, we could transform the property array containing all date objects, which would allow us to simplify the loop body to one access to a new property array containing the timestamp. Hence, we could skip loading the original proxy object and loading the date object.

8.2 Compile-Time Query Duplication

Due to the restrictions mentioned in Section 4.3, some methods may not reach a "stable" state: If a method is alternatingly called with a transformed and a not-(yet-)transformed array, we repeatedly have to deoptimize the already compiled method and start compilation anew. To solve that problem, we can duplicate loops that contain both, the fast path handling the columnar array and the slow path handling the original array representation, introducing a branching instruction prior to the duplicated loops and optimizing them respectively. Naturally, one aspect of such a compiler optimization is the increase in code size, so additional heuristics will have to be used in order to decide, what and when to duplicate.

8.3 Polyglot Support

While we implemented our prototype for JavaScript, we expect that it could also speed up certain workloads in other Truffle languages such as Python or Ruby. Unfortunately, the array representations of the individual language implementations do not share commonalities via the core framework similar to the Truffle OSM. Hence, porting our approach would require some engineering effort and may further require an alternative to our object-to-proxy transformation (unless supported by the language implementation).

9 CONCLUSION

In this work, we developed an approach for automated array storage transformation in JavaScript. The transformation creates *columnar arrays* from arrays of objects and adapts accesses to this new data structure. As a consequence, we can speed up accesses to these arrays, hence our approach is especially suited for processing of arrays within loops. We integrated our approach into the GraalVM JavaScript runtime, based on the Truffle Framework, that uses the GraalVM infrastructure to enable run-time optimization by using profiling information and speculative optimizations. We gain performance benefits by applying dedicated optimizations at compile time, such as removing redundant checks, compiling methods with respect to the array state, and subsequently moving loop invariant accesses out of hot loops. If an array is modified in a way that conflicts with our representation, we enable a restoration of the original array.

An evaluation of our approach on a set of microbenchmarks as well as the TPC-H benchmark suite shows that we can achieve significant speedups on bulk operations on large arrays, while suffering from the transformation overhead on smaller or more complex operations. We argue that future work such as compile-time duplication of loops could improve upon our approach and yield better results, even on smaller, less repetitive workloads.

ACKNOWLEDGMENTS

This research project was partially funded by Oracle Labs. We thank all members of the Virtual Machine Research Group at Oracle Labs. Oracle, Java, GraalVM, and HotSpot are trademarks or registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. We also thank all researchers at the Johannes Kepler University Linz's Institute for System Software for their support of and valuable feedback on our work.

REFERENCES

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 671–682. https://doi.org/10.1145/1142473.1142548

[2] Daniel Abadi, Daniel Myers, David DeWitt, and Samuel Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, Istanbul, Turkey, 466–475. https://doi.org/10.1109/ICDE.2007.367892

[3] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. *SIGPLAN Not.* 49, 6 (June 2014), 496–507. https://doi.org/10.1145/2666356.2594332

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley Pub. Co, Reading, Mass.

[5] Apache Software Foundation. 2022. Apache Arrow. The Apache Software Foundation.

[6] Lars Bak. 2022. V8 JavaScript Engine. https://v8.dev/. (accessed 2022-06-29).

[7] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, Indianapolis Indiana USA, 167–182. https://doi.org/10.1145/2509136.2509531

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

[9] Mike Bostock. 2022. D3.Js - Data-Driven Documents. https://d3js.org/. (accessed 2022-06-29).

[10] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-Conscious Data Placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*. Association for Computing Machinery, New York, NY, USA, 139–149. https://doi.org/10.1145/291069.291036

[11] D. Callahan, J. Dongarra, and D. Levine. 1988. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing (Supercomputing '88)*. IEEE Computer Society Press, Washington, DC, USA, 98–105.

[12] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. 1994. Compiler Optimizations for Improving Data Locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. Association for Computing Machinery, New York, NY, USA, 252–262. https://doi.org/10.1145/195473.195557

[13] Richard L. Cole and Goetz Graefe. 1994. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD '94)*. ACM, New York, NY, USA, 150–160. https://doi.org/10.1145/191839.191872

[14] Transaction Processing Performance Counci. 2021. *TPC Benchmark H - Standard Specification.* Technical Report 3.0.0. Transaction Processing Performance Counci (TPC), San Francisco, CA, USA. 138 pages.

[15] Ryan Dahl. 2022. Node.Js. https://github.com/nodejs/node. (accessed 2022-06-29).

[16] John-David Dalton. 2022. Lodash. https://github.com/lodash/lodash. (accessed 2022-06-29).

[17] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B. Sartor, and Wolfgang De Meuter. 2015. Just-in-Time Data Structures. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 61–75. https://doi.org/10.1145/2814228.2814231

[18] Ulrich Drepper. 2007. What Every Programmer Should Know about Memory. *Red Hat, Inc* 11 (2007), 2007.

[19] Gilles Duboscq, Lukas Stadler, Thomas Wuerthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An Extensible Declarative Intermediate Representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. Shenzhen, China, 9.

[20] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, Cracow, Poland, 187–193. https://doi.org/10.1145/2647508.2647521

[21] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages (VMIL '13)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[22] Amit Dwivedi, C. Lamba, and Shweta Shukla. 2012. Performance Analysis of Column Oriented Database Vs Row Oriented Database. *International Journal of Computer Applications* 50 (July 2012), 31–34. https://doi.org/10.5120/7841-1050

[23] ECMA International. 2020. *ECMA-262, 12th Edition, June 2021.* Technical Report 12. ECMA (European Association for Standardizing Information and Communication Systems), San Francisco, CA, USA. 879 pages.

[24] ECMA International. 2022. Tc39/Test262. Ecma TC39.

[25] Juliana Franco, Martin Hagelin, Tobias Wrigstad, Sophia Drossopoulou, and Susan Eisenbach. 2017. You Can Have It All: Abstraction and Good Cache Performance. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 148–167. https://doi.org/10.1145/3133850.3133861

[26] Michael Franz and Thomas Kistler. 1998. *Splitting Data Objects to Increase Cache Utilization.* Technical Report.

[27] Google. 2022. Angular. https://angular.io/. (accessed 2022-06-29).

[28] Goetz Graefe. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.* 25, 2 (June 1993), 73–169. https://doi.org/10.1145/152610.152611

[29] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. Association for Computing Machinery, New York, NY, USA, 78–90. https://doi.org/10.1145/2816707.2816714

[30] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient Execution of Polyglot Queries. *Proc. VLDB Endow.* 15, 2 (Oct. 2021), 196–210. https://doi.org/10.14778/3489496.3489501

[31] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[32] Holger Homann and Francois Laenen. 2018. SoAx: A Generic C++ Structure of Arrays for Handling Particles in HPC Codes. *Computer Physics Communications* 224 (March 2018), 325–332. https://doi.org/10.1016/j.cpc.2017.11.015

[33] Liang Hong, Mengqi Luo, Ruixue Wang, Peixin Lu, Wei Lu, and Long Lu. 2018. Big Data in Health Care: Applications and Challenges. *Data and Information Management* 2, 3 (Dec. 2018), 175–197. https://doi.org/10.2478/dim-2018-0014

[34] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. 2014. A Domain-Specific Language for Building Self-Optimizing AST Interpreters. In *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE 2014)*. Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/2658761.2658776

[35] Intel. 2010. A Guide to Vectorization with Intel® C++ Compilers.

[36] intel. 2022. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, Chapter 5 - Instruction Set Summary.*

[37] Xiaolong Jin, Benjamin W. Wah, Xueqi Cheng, and Yuanzhuo Wang. 2015. Significance and Challenges of Big Data Research. *Big Data Research* 2, 2 (June 2015), 59–64. https://doi.org/10.1016/j.bdr.2015.01.006

[38] Oliver Kennedy and Lukasz Ziarek. 2015. Just-In-Time Data Structures. In *CIDR*. www.cidrdb.org, Monterey, CA, USA, 11.

[39] Tanvir Ahmed Khan, Ian Neal, Gilles Pokam, Barzan Mozafari, and Baris Kasikci. 2021. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Virtual, 163–181.

[40] Sebastian Kloibhofer. 2021. Run-Time Data Analysis to Drive Compiler Optimizations. In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2021)*. Association for Computing Machinery, New York, NY, USA, 9–12. https://doi.org/10.1145/3484271.3484974

[41] Alexandros Labrinidis and H. V. Jagadish. 2012. Challenges and Opportunities with Big Data. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 2032–2033. https://doi.org/10.14778/2367502.2367572

[42] Florian Latifi, David Leopoldseder, Christian Wimmer, and Hanspeter Mössenböck. 2021. CompGen: Generation of Fast JIT Compilers in a Multi-Language VM. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Dynamic Languages (DLS 2021)*. Association for Computing Machinery, New York, NY, USA, 35–47. https://doi.org/10.1145/3486602.3486930

[43] Chris Lattner and Vikram Adve. 2005. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. *SIGPLAN Not.* 40, 6 (June 2005), 129–142. https://doi.org/10.1145/1064978.1065027

[44] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-Based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. ACM Press, Vienna, Austria, 126–137. https://doi.org/10.1145/3168811

[45] Lukas Makor. 2021. Run-Time Data Analysis in Dynamic Runtimes. In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2021)*. Association for Computing Machinery, New York, NY, USA, 6–8. https://doi.org/10.1145/3484271.3484973

[46] MarketsandMarkets. 2022. Big Data Market Size, Share and Global Market Forecast to 2026 | MarketsandMarkets. https://www.marketsandmarkets.com/Market-Reports/big-data-market-1068.html. (accessed 2022-04-27).

[47] Toni Mattis, Johannes Henning, Patrick Rein, Robert Hirschfeld, and Malte Appeltauer. 2015. Columnar Objects: Improving the Performance of Analytical Applications. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. ACM, Pittsburgh PA USA, 197–210. https://doi.org/10.1145/2814228.2814230

[48] Joe Minichino. 2022. LokiJS. https://github.com/techfort/LokiJS. (accessed 2022-06-29).

[49] Fabian Nagel, Gavin Bierman, Aleksandar Dragojevic, and Stratis Viglas. 17. Self-Managed Collections: Off-heap Memory Management for Scalable Query-Dominated Collections. , 71 pages. https://doi.org/10.5441/002/edbt.2017.07

[50] Simone Ferlin Oliveira, Karl Fürlinger, and Dieter Kranzlmüller. 2012. Trends in Computation, Communication and Storage and the Consequences for Data-intensive Science. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE, Liverpool, UK, 572–579. https://doi.org/10.1109/HPCC.2012.83

[51] Oracle. 2021. Graal.Js. https://github.com/graalvm/graaljs. (accessed 2020-09-09).

[52] Oracle. 2021. GraalPython. https://github.com/graalvm/graalpython. (accessed 2020-09-09).

[53] Oracle. 2021. GraalVM. https://www.graalvm.org/. (accessed 2020-07-23).

[54] Oracle. 2021. TruffleRuby. https://github.com/oracle/truffleruby. (accessed 2020-09-09).

[55] Oracle. 2022. Node.Js Runtime. https://www.graalvm.org/22.0/reference-manual/js/NodeJS/. (accessed 2022-04-25).

[56] Jim Pivarski, Peter Elmer, Brian Bockelman, and Zhe Zhang. 2017. Fast Access to Columnar, Hierarchically Nested Data via Code Transformation. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, Boston, MA, USA, 253–262. https://doi.org/10.1109/BigData.2017.8257933

[57] Samira Pouyanfar, Yimin Yang, Shu-Ching Chen, Mei-Ling Shyu, and S. S. Iyengar. 2018. Multimedia Big Data Analytics: A Survey. *ACM Comput. Surv.* 51, 1 (Jan. 2018), 10:1–10:34. https://doi.org/10.1145/3150226

[58] RAPIDS Development Team. 2018. *RAPIDS: Collection of Libraries for End to End GPU Data Science.*

[59] David Reinsel, John Gantz, and John Rydning. 2018. The Digitization of the World from Edge to Core. *International Data Corporation, Framingham* 16 (2018), 28.

[60] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. 2016. Sulong - Execution of LLVM-based Languages on the JVM: Position Paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems - ICOOOLPS '16*. ACM Press, Rome, Italy, 1–4. https://doi.org/10.1145/3012408.3012416

[61] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2021. Language-Agnostic Integrated Queries in a Managed Polyglot Runtime. *Proc. VLDB Endow.* 14, 8 (April 2021), 1414–1426. https://doi.org/10.14778/3457390.3457405

[62] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, Orlando, FL, USA, 165–174. https://doi.org/10.1145/2581122.2544157

[63] Statista. 2021. Total Data Volume Worldwide 2010-2025. https://www.statista.com/statistics/871513/worldwide-data-created/. (accessed 2022-04-27).

[64] Mike Stonebraker, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2005. C-Store: A Column-Oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*. VLDB Endowment, Trondheim, Norway, 553–564.

[65] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12)*. ACM, Tucson, Arizona, USA, 13–14. https://doi.org/10.1145/2384716.2384723

[66] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages, and Tools - PPPJ '14*. ACM Press, Cracow, Poland, 133–144. https://doi.org/10.1145/2647508.2647517

[67] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[68] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, Indianapolis, Indiana, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[69] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

[70] Rui Zhang, Saumya Debray, and Richard T. Snodgrass. 2012. Micro-Specialization: Dynamic Code Specialization of Database Management Systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. Association for Computing Machinery, New York, NY, USA, 63–73. https://doi.org/10.1145/2259016.2259025

[71] Wangda Zhang, Junyoung Kim, Kenneth A. Ross, Eric Sedlar, and Lukas Stadler. 2021. Adaptive Code Generation for Data-Intensive Analytics. *Proc. VLDB Endow.* 14, 6 (Feb. 2021), 929–942. https://doi.org/10.14778/3447689.3447697