Language-Agnostic Integrated Queries in a Managed Polyglot Runtime

Filippo Schiavio
Università della Svizzera italiana
Lugano, Switzerland
filippo.schiavio@usi.ch

Daniele Bonetta
Oracle Labs - VM Research Group
Cambridge, MA, USA
daniele.bonetta@oracle.com

Walter Binder
Università della Svizzera italiana
Lugano, Switzerland
walter.binder@usi.ch

ABSTRACT

Language-integrated query (LINQ) frameworks offer a convenient programming abstraction for processing in-memory collections of data, allowing developers to concisely express declarative queries using general-purpose programming languages. Existing LINQ frameworks rely on the well-defined type system of statically-typed languages such as C$^\sharp$ or Java to perform query compilation and execution. As a consequence of this design, they do not support dynamic languages such as Python, R, or JavaScript. Such languages are however very popular among data scientists, who would certainly benefit from LINQ frameworks in data analytics applications.

In this work we bridge the gap between dynamic languages and LINQ frameworks. We introduce DynQ, a novel query engine designed for dynamic languages. DynQ is language-agnostic, since it is able to execute SQL queries in a polyglot language runtime. Moreover, DynQ can execute queries combining data from multiple sources, namely in-memory object collections as well as on-file data and external database systems. Our evaluation of DynQ shows performance comparable with equivalent hand-optimized code, and in line with common data-processing libraries and embedded databases, making DynQ an appealing query engine for standalone analytics applications and for data-intensive server-side workloads.

1 INTRODUCTION

In modern data processing, the boundary between *where* data is located, and *who* is responsible for processing it, has become very blurry. Data lakes [14] and emerging machine-learning frameworks such as TensorFlow [55] make it very practical for data scientists to develop complex data analyses directly "in the language" (i.e., in Python or JavaScript), rather than resorting to "external" runtime systems such as traditional RDBMSs. Such an approach is facilitated by the fact that many programming languages are equipped with built-in or third-party libraries for processing in-memory collections (e.g., arrays of objects). Well-known examples of such libraries are the Microsoft LINQ-to-Objects framework [33] (which

targets .NET languages, e.g., C$^\sharp$) and the Java Stream API [43]. Microsoft's implementation of LINQ not only allows developers to query in-memory collections, but it can be extended with data-source providers [32] (e.g., LINQ-to-SQL and LINQ-to-XML) that allow developers to execute federated queries (i.e., queries that process data from multiple sources). Many systems with similar features have been proposed (e.g., Apache Spark SQL [2]). LINQ systems have been studied from a theoretical point of view [10, 18], and several optimization techniques have been proposed [29, 37, 38]. However, the proposed solutions mostly focus on statically-typed languages, where type information is known before program execution.

Despite the many benefits it offers, LINQ support is currently missing in popular dynamic languages, i.e., languages for which the type of a variable is checked at runtime, such as Python or JavaScript. Such languages are often preferred by data scientists (e.g., in Jupyter notebooks [25]), because they are easier to use and typically come with a simple data-processing API (e.g., filter, map, reduce) and data-frame API (e.g., R dplyr library [58] and Python Pandas library [35]) that simplify quick data exploration. Besides data analytics, supporting language-integrated queries in dynamic languages would also be useful in other contexts. As an example, JavaScript and Node.JS are widely used to implement data-intensive server-side applications [51].

Due to their popularity, embedded database systems such as DuckDB [46] often provide bindings for some dynamic languages. With such an approach, the database query engine is hosted in the application process, removing the inter-process communication overhead imposed by solutions that adopt an exernal database system [45]. However, developers cannot use embedded databases to query arbitrary data that resides in the process address space (e.g., an array of JavaScript objects or a file loaded by the application). Instead, using embedded databases, it is usually required to create tables with a data schema, and then traverse the object collection and insert relevant data in such tables, a so-called ingestion phase. Some embedded databases are able to query specific data structures implemented in a dynamic language, e.g., DuckDB [46] can execute queries on both R and Pandas data frames. However, both R and Pandas data frames are implemented with a columnar data structure composed of typed arrays, and they cannot store dynamic objects, such as e.g. a JavaScript Map.

In this paper we introduce DynQ, a novel query engine targeting dynamic languages for the GraalVM platform [61]. Unlike existing LINQ systems, DynQ is capable of running queries on dynamically-typed collections such as JavaScript or R objects. Moreover, DynQ is *language-agnostic*, and can execute queries on data defined in

any of the languages supported by GraalVM. DynQ is highly optimized and benefits from just-in-time (JIT) compilation to speed up query execution. To the best of our knowledge, DynQ is the first query engine targeting multiple programming languages, which explicitly interacts with a JIT compiler. Such a tight integration with the JIT compiler is obtained by using the Truffle language implementation framework [59] in a novel and previously unexplored way. Indeed, the Truffle framework was designed for programming-language implementations, whilst with our approach we exploit Truffle as a general code-generation framework in the context of a data-processing engine.

This paper makes the following contributions:

- We introduce DynQ, a language-agnostic query engine which can execute queries on collections of objects as well as on file data (e.g., JSON files) and other data sources without requiring any data schema (neither provided nor inferred). DynQ is able to optimize itself depending on the data types encountered during query execution.
- We describe DynQ's approach to query compilation, which relies on self-optimizing abstract syntax tree (AST) interpreters and dynamic speculative optimizations.
- Our evaluation of DynQ shows performance comparable with a hand-optimized implementation of the same query and outperforms implementations based on built-in or third-party data-processing libraries in most of the evaluated workloads.

This paper is structured as follows. In Section 2 we introduce relevant background information in the context of our work. In Section 3 we describe the design of DynQ and in Section 4 we evaluate its performance against hand-optimized queries as well as existing data-processing libraries and databases. Section 5 discusses related work, and Section 6 concludes this paper.

2 BACKGROUND

In this section we give an overview of the .NET implementation of language-integrated queries (LINQ) and we discuss its execution model as well as improvements proposed in the research literature. Then, we introduce the GraalVM platform [61] and the Truffle [59] framework that we use for implementing DynQ.

2.1 Language-integrated Queries

LINQ was first introduced in Microsoft .NET 3.5 to extend the $C^\sharp$ language with an SQL-like *query comprehension* syntax and a set of query operators [5]. The following is an example of a LINQ query:

```
IEnumerable<int> xs = ...;
var evenSquares = from x in xs
                  where x % 2 == 0
                  select x * x;
```

LINQ implements a lazy evaluation strategy by converting query operators to iterators, a so-called *pull-based* model [49], i.e., each operator pulls the next row from its source operator. In the example query, the where and select clauses in the query comprehension are de-sugared into calls to the methods Where and Select defined in the IEnumerable interface.

Another important feature of LINQ is its extensibility to new data formats. LINQ can execute queries not only on in-memory object

collections, but also on any data type that extends the generic types IEnumerable or IQueryable; indeed, from a theoretical point of view, LINQ queries can be executed on any data type that exhibits the properties of a monad [18]. This great flexibility is obtained through so-called LINQ providers, i.e., data-source specific implementations of the mentioned generic types. Relevant examples of LINQ providers are LINQ-to-XML (that queries XML documents) and LINQ-to-SQL, which converts query expressions into SQL queries and sends them to an external DBMS. Despite the benefits it provides, LINQ was explicitly designed targeting statically-typed languages, and it is currently not available in dynamic languages. Our work overcomes this limitation.

2.2 Query Execution Models

The $C^\sharp$ implementation of LINQ executes queries by leveraging the pull-based model, which shares many similarities with the Volcano [16] query execution model in use by many popular relational databases. It has been shown [30] that the main performance drawbacks of this execution model are virtual calls to the interface methods (e.g., MoveNext() and Current() in $C^\sharp$, or hasNext() and next() in Java), which introduce non-negligible overhead, since they are executed for each input row of each operator in the query plan. In the context of relational databases, the most relevant optimizations for removing such overhead are vectorization [6] and data-centric query compilation [40]. Vectorized query execution, similarly to the Volcano model, uses a pull-based approach. However, the query interpretation overhead is mitigated by leveraging a columnar data representation and batched execution, i.e., instead of evaluating a single data item at a time, query operators work on a vector of items which represents multiple input rows. Data-centric query execution completely removes the interpretation overhead by generating executable code for a given query. Code generation commonly happens at runtime, using schema and type informations to generate code that is specialized for the tables used in a query. Data-centric query compilation adopts a so-called *push-based* model, i.e., each operator pushes a row to its destination operators.

A well-known disadvantage of query compilation is the overhead introduced by the compilation itself. Recent research [28] addresses this issue by introducing an adaptive query compilation model. With such a compilation model, the engine first quickly generates an executable representation of the query and executes it in an interpreter. Then, during query execution, the engine performs adaptive decisions whether to compile a query operator based on execution-time estimations. This approach is inspired by the implementation of JIT compilers in language VMs and shares similarities with the query execution model adopted by DynQ. Unlike existing SQL query compilation approaches, DynQ needs to generate machine code that is specialized to access objects located in the memory space of a running language VM. This scenario presents unique challenges that are not found in existing SQL execution runtimes, as we will discuss in the rest of the paper.

2.3 GraalVM and the Truffle Framework

DynQ is implemented targeting the GraalVM [61] platform, i.e., a polyglot language runtime compatible with the Java Virtual Machine (JVM). GraalVM is capable of executing programs developed

in a variety of popular programming languages, such as e.g. Java, JavaScript, Ruby, Python, and R. At its core, GraalVM relies on a state-of-the-art dynamic compiler (called Graal [60]), which brings JIT compilation to all GraalVM languages. Language runtimes and systems for GraalVM (including DynQ) are implemented using the Truffle [59] language implementation framework. Unlike other code generation frameworks for the JVM or the .NET platform, Truffle does not rely on bytecode generation, but rather on the concept of self-optimizing interpreters [59], i.e., language interpreters that use custom APIs and data structures enabling explicit and direct interaction with the underlying language VM's components (including the JIT compiler). The Graal optimizing compiler has special knowledge of such data structures and APIs, and is capable of generating efficient machine code by means of partial evaluation [23]. In addition to JIT compilation, the Truffle framework provides mechanisms to interact with any of the dynamic languages supported by GraalVM. Thanks to such interoperability mechanisms, DynQ can effectively inline machine code used by GraalVM language runtimes into its own query execution code. For example, DynQ can use the very same machine code used by the GraalVM JavaScript VM to read JavaScript heap-allocated objects, thereby enabling efficient access to in-memory data during SQL query execution. This approach to SQL execution allows DynQ to efficiently exploit runtime information, to benefit from optimizations that are normally used in high-performance language VMs, such as, e.g., dynamic loop unrolling and polymorphic inline caching [20].

3 DYNQ

In this section we provide a detailed description of DynQ's internals, presenting its general design (Section 3.1), dynamic query compilation (Section 3.2), and its built-in support for third-party data providers (Section 3.3). We also explain how DynQ's architecture facilitates the development of language-specific optimizations (Section 3.4). In designing DynQ we focused on the following three goals:

- *Language-independence:* DynQ should be able to execute queries on any collection of objects from any language supported by GraalVM.
- *High performance:* Query execution with DynQ from a dynamic language should be as efficient as a hand-optimized application written in the same language.
- *Extensibility and modularity:* Integrating new data sources and query operators in DynQ should impact only their respective components.

In the following subsections we describe how we designed DynQ to meet all these requirements.

3.1 DynQ Architecture

At its core, DynQ is a dynamic query engine for GraalVM that exploits advanced dynamic compilation techniques to optimize query execution. DynQ is exposed to users by means of a language-agnostic API, and is capable of executing queries on any object representation supported by GraalVM languages. Unlike the popular LINQ implementation for the .NET platform, DynQ does not extend its supported programming languages with a query-comprehension syntax, but rather relies on SQL queries expressed as plain strings.

The LINQ query-comprehension syntax allows query validation at program compilation time. However, as already discussed in the literature [22], in a dynamically-typed language, where syntactic validation and type checking take place at runtime, lacking this form of compile-time validation is not an issue. Moreover, since one of the main goals of DynQ is language independence, extending the syntax of multiple languages would not be a practical approach.

Two important differences between DynQ and existing LINQ systems are its dynamic type system and the tight integration with the underlying GraalVM platform. The flexibility of dynamic languages imposes additional performance challenges compared with query engines that process data with a known type, as the engine has to take into account that a value may be missing in an object and that runtime types may differ from the expected ones. JIT compilation is crucial in this context, as it allows DynQ to generate machine code that is specialized for the data types observed at runtime. For example, DynQ can emit offset-based machine code when accessing R data frames, or hash-lookup-based access code when reading data from JavaScript (map-like) dynamic objects. Close interaction with the platform's JIT compiler is a peculiar feature of DynQ and a key architectural difference w.r.t. other popular language-integrated approaches. Existing systems (e.g., .NET LINQ or Java 8 Streams) do not interact with the underlying language runtime; in all such systems, queries are compiled to an intermediate representation (e.g., .NET CLR or Java bytecode) like any other language construct (e.g., Java 8 Streams are converted to plain Java bytecode with virtual method calls and loops). Query compilation to such intermediate representations happens statically, before program execution. At runtime, the language VM might (or might not) generate machine code for a specific query. However, the lack of domain knowledge of the underlying JIT compiler could limit the class and scope of optimizations that the language VM can perform. For example, a language VM might (or might not) decide to inline certain methods into hot method bodies depending on runtime heuristics that have nothing to do with the structure of the actual query being executed.

DynQ, on the contrary, takes a radically different approach as it explicitly *interacts* with the underlying VM's JIT compiler to drive query compilation. In this way, DynQ can effectively propagate its runtime knowledge of any given query to machine code generation, resulting in high performance. As an example, DynQ can effectively *force* the inlining of the predicates of a given query expression into table-scan operators, ensuring efficient data access. Moreover, the tight integration with the language VM's JIT compiler unlocks a class of optimization that are not achievable with existing LINQ-like systems, namely, *dynamic* speculative optimizations: not only can DynQ apply an optimization (e.g., inlining) when it sees potential performance gains, but it can also *de-optimize* the generated machine code when certain runtime assumptions are invalidated, giving a chance to its query execution engine to re-profile the data that is being processed, possibly leading to the generation of new machine code that now takes into account different runtime assumptions.

Thanks to its design, DynQ can outperform hand-optimized implementations of queries written in dynamic languages. Internally, the type system of DynQ's query engine handles two main types,
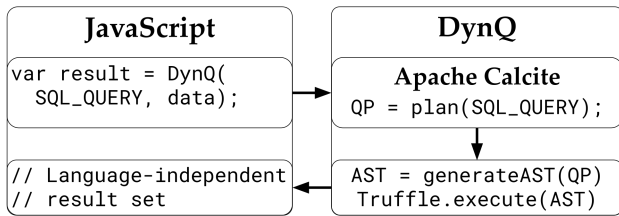
JavaScript	DynQ
``var result = DynQ(
SQL_QUERY, data);```	**Apache Calcite** ```QP = plan(SQL_QUERY);```
``// Language-independent
// result set```	```AST = generateAST(QP)
Truffle.execute(AST)```

Figure 1: High-level query life cycle in DynQ.

namely primitive types and structured types. Primitive types include all Java primitive types as well as String and Date. Structured types include arrays and nested data structures, i.e., objects with properties of any of the mentioned types; multiple nesting levels are supported as well. As expressions, DynQ supports logical and arithmetic operators, the SQL LIKE function on strings, and the EXTRACT function on dates. Moreover, DynQ seamlessly supports user-defined functions (UDFs), as it can directly inline code from any GraalVM language into its SQL execution code. In this way, UDFs from any of the GraalVM languages can be called during SQL evaluation with minimal runtime overhead. In particular, it is not required that a UDF is written in the same language as the application that is using DynQ, e.g., it is possible to use DynQ from JavaScript, executing a query with a UDF written in R. As GraalVM is compatible with Java, DynQ can leverage existing Java-based components to perform SQL query parsing and initial query planning. To this end, our implementation currently leverages the state-of-the-art SQL query parser and planner Apache Calcite [4]. While using Calcite as an SQL front-end has the notable advantage that DynQ's implementation can focus on runtime query optimization *after* query planning, it is important to note that DynQ's design is not bound to Calcite's API, and other SQL parsers and planners could be used as well.

A high-level overview of the life-cycle of a query executed with DynQ from a dynamic language (JavaScript in the example) is depicted in Figure 1. As the figure shows, as soon as a developer has defined a dataset in the form of an object collection (e.g., an array) it is possible to execute an SQL query on the in-memory data. DynQ is invoked from the host dynamic language, passing (as parameters) a string representation of the query and a reference to the input data. DynQ leverages Calcite for parsing and validating the SQL query; if successful, the validated query is converted into an optimized query plan. Then, DynQ traverses the query plan, generating an equivalent executable representation (i.e., Truffle nodes [59]), which is our form of a physical plan. By generating Truffle nodes, the code generation phase of DynQ is very efficient, as Truffle nodes are ready to be executed by GraalVM. Query execution thus begins by executing the Truffle nodes generated by DynQ. As soon as the DynQ runtime detects that the AST (or parts of it) are frequently executed, it delegates (to GraalVM) the JIT compilation to machine code. As discussed, dynamic compilation is triggered by DynQ, which also takes into account possible runtime de-optimizations and re-compilations. Finally, the result of query execution, i.e., a language-independent data structure accessible by any GraalVM language, is returned to the application.

3.2 Query Compilation in DynQ

Query compilation in DynQ is implemented using a push-based approach, and takes place by visiting the query plan generated by Calcite and converting it into Truffle nodes. It is important to stress that Truffle nodes are an executable representation of the DynQ query that the GraalVM JIT compiler can efficiently compile to machine code. The push-based query execution approach used by DynQ is inspired by the model introduced in LB2 [54]. In this model, each operator produces a result row that is consumed by an executable callback function. Rather than relying on statically generated callback functions, however, DynQ propagates result rows to Truffle nodes. In this way, those nodes can specialize themselves on the actual data types observed at runtime. Internally, DynQ relies on two classes of Truffle nodes, namely (1) Expressions and (2) Query-operators.

Expression nodes represent the supported SQL expressions and UDF functions introduced in Section 3.1. Since DynQ is a schema-less query engine, each expression node used in a query has initial unknown input (and output) type. During query execution, Truffle nodes rewrite themselves to *specialized* versions capable of handling the actual types observed during query execution. This specialization mechanism is natively supported by the Truffle framework, and allows DynQ to handle type polymorphism in a way analogous to language runtimes, resorting to runtime optimization techniques such as polymorphic inline caches [20]. In this way, an expression can be specialized during query execution to handle multiple data types.

Query-operator nodes are responsible for executing SQL operators, eventually producing a concrete result value. DynQ relies on two categories of query-operator nodes, namely *consumer* nodes and *executable* nodes. Intuitively, each query operator (excluding table scans) has its own consumer node, whilst only table-scan and join operators implement an executable node. The main executable node of a query, i.e., the one containing the root operator, takes care of producing the result set for that query.

DynQ generates a query's root executable node by visiting the plan generated by Calcite. In particular, DynQ generates a consumer node $\mathscr{C}$ for the currently visited operator $\mathscr{O}$. If $\mathscr{O}$ is not a join (i.e., it has only one child), $\mathscr{C}$ will consume the rows produced by the child of $\mathscr{O}$. If $\mathscr{O}$ is a table scan, DynQ generates an executable node which iterates over a data structure (which acts as a table), invoking the generated chain of consumer nodes for each row. The implementation of the consumer nodes generated by visiting a join operator depends on the join type. DynQ supports nested-loop joins and hash-joins (possibly with non-equi conditions). In case of nested-loop joins, DynQ creates a left consumer which inserts all rows into a list $\mathscr{L}$, and a right consumer that finds matching pairs of rows by iterating over the elements in $\mathscr{L}$ for each row. In case of hash-joins, the left consumer inserts the rows in a hash-map, which is used by the right consumer to find matching pairs. The corresponding Java interfaces ExpressionNode, ConsumerNode, and ExecutableNode are shown in Figure 2. When the query root operator is not a materializer (e.g., for queries composed of projections and predicates), DynQ adds a custom consumer which fills a list of rows, since DynQ always outputs an array data structure. On the other hand, when the root operator is a sort or an aggregation,

```
interface ExecutableNode {
 Object execute();
}

interface ConsumerNode {
 void consume(Object row) throws EndOfExecution;
 Object getResult();
}

interface ExpressionNode {
 Object execute(Object row);
}
```

Figure 2: Main interfaces in DynQ.

```
var T = [{x: 1, y: 2},
         {x: 2, y: 1},
         {x: Date('2000-01-01'), y: Date('2000-01-02')}];
var result = DynQ('SELECT COUNT(*) FROM T WHERE x < y',T);
```

**Figure 3: Example of a DynQ query on a JavaScript array.**

Figure 4: AST generated by DynQ for the query in Figure 3.

DynQ returns the sorted (or aggregated) data, which is already a list of rows. Moreover, since push engines do not allow terminating the source iteration, i.e., an operator cannot control when data should not be produced any more by its source operator, DynQ implements early exits for the *limit* query operator by throwing a special `EndOfExecution` exception, which is part of the signature of the method `ConsumerNode.consume(row)`. Stateful operators do not need any specific executable node, since they are implemented using the `ConsumerNode` methods `consume(row)` and `getResult()`. As an example, if a query has a group-by operator (which is not the root operator in the query plan), its implementation of `consume(row)` updates the internal state (a hash-map) and the implementation of `getResult()`, which is invoked by its source operator once all input tuples have been consumed, sends all tuples from the aggregated hash-map to its destination (a `ConsumerNode`), calling the `consume(row)` method for each aggregated row, and finally returns the value obtained by calling the `getResult()` method on its destination consumer.

Query compilation example. Consider the DynQ query targeting a JavaScript array of objects shown in Figure 3. The query execution plan for the example query is composed of a table-scan operator, a predicate operator, and an aggregation operator that counts the number of rows that satisfy the predicate. The AST of Truffle nodes generated by DynQ for the example query is depicted in Figure 4. The implementation of the nodes that compose the query is depicted in Figure 5. As shown in Figure 5, the `LessThanNode` node leverages Truffle specializations for implementing the less-than operation. The `LessThanNode` implementation shown in Figure 5 presents only the specializations for Int and Date types, because those types are the ones used in the example. The actual implementation contains specializations for all types supported in DynQ as well as their possible combinations (e.g., Int/Double and Double/Int). In particular, DynQ specializations with mixed types respect the implicit type conversion (i.e., type cast) semantics commonly integrated in a query planner, but in DynQ the detection of such casts must take place during data processing (instead of during query planning), since at query planning time types are not known in DynQ. Consider the method `execute(Object row)` defined in the class `LessThanNode`. This method first executes the left and right children expression nodes (i.e., property reads in the example query). Then, the method call to `executeSpecialized` (internally) performs a type check for the two arguments (i.e., `fst` and `snd`). If both values have type `int`, the specialization `execute(int, int)` is executed; if they are both dates, the method `execute(LocalDate,`
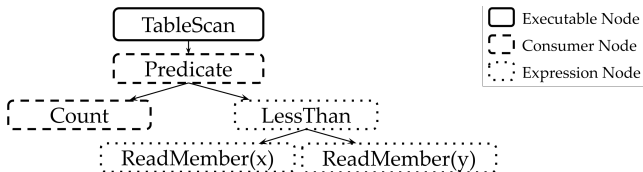
`LocalDate`) is executed; otherwise, the current tuple is discarded. Note that, although our current implementation is permissive, i.e., it does not stop the query execution throwing an exception in case a malformed row is encountered, implementing different error handling strategies would be trivial.

Consider again the AST generated by DynQ for the example query depicted in Figure 4. When the query would be executed on an R data frame, DynQ would generate *the same tree*, but the `TableScan` executable node and the `ReadMember` expression nodes would specialize in different ways, depending on the runtime types. The flexible design of DynQ allows reusing the very same query-operator nodes for executing queries on different data structures, like a JavaScript array of objects or an R data frame. Thanks to this design, we achieve all the three goals listed in the beginning of this section. In particular, the extensibility and modularity of our design allow adding new data sources (e.g., a data structure in a dynamic language or an external source like a JSON file) by integrating only the expression nodes which take care of accessing data from such a data source, without requiring any modification to the query-operator nodes.

Dynamic machine code generation. By implementing DynQ on top of Truffle, DynQ has fine-grained control over GraalVM's JIT compiler. Dynamic compilation is triggered based on the runtime profiling information collected during query execution, and the Graal JIT compiler applies (to DynQ queries) all optimizations that are commonly used in dynamic language runtimes. Examples of optimizations applied by Graal include aggressive inlining, loop unrolling, and partial escape analysis. JIT compilation is performed by GraalVM using a configurable number of parallel compiler threads. This leads to short compilation times, as we will further discuss in Section 4.1.1.

In contrast to many engines based on query compilation, DynQ does not need to generate machine code before executing a query. Query execution in DynQ begins as soon as the Truffle nodes have been instantiated. First, the execution starts by *interpreting* those nodes; during this phase the runtime collects type information for the nodes that leverage Truffle specializations (e.g., `LessThanNode` in the previous example). Then, once the runtime detects that some nodes are frequently executed (e.g., the main loop in `TableScanNode`), it initiates machine-code generation. Once the runtime has collected type information for those rows which have

```
class TableScanNode implements ExecutableNode {
  ConsumerNode consumer;
  PolyglotArray input;
  Object execute() {
    try {
      for(int i = 0; i < input.numElements, i++) {
        consumer.consume(readJsArrayElement(input, i));
      }
    } catch (EndOfExecution e) {}
    return consumer.getResult();
  }
}

class PredicateConsumerNode implements ConsumerNode {
  ConsumerNode consumer;
  Expression predicate;
  void consume(Object row) {
    if(predicate.execute(row)) { consumer.consume(row); }
  }
  Object getResult() { return consumer.getResult(); }
}

class CountConsumerNode implements ConsumerNode {
  long result = 0;
  void consume(Object row) { result++; }
  Object getResult() { return result; }
}

class LessThanNode implements ExpressionNode {
  ExpressionNode left, right;
  boolean execute(Object row) {
    Object fst = left.execute(row);
    Object snd = right.execute(row);
    return executeSpecialized(fst, snd);
  }
  @Specialization
  boolean execute(int left, int right) {
    return left < right;
  }
  @Specialization
  boolean execute(LocalDate left, LocalDate right) {
    return left.isBefore(right);
  }
}

class ReadMemberNode implements ExpressionNode {
  String name;
  Object execute(Object row) {
    return readJsMember(name, row);
  }
}
```

Figure 5: Simplified Truffle-node implementation in DynQ for the nodes used in the example query in Figure 3.

been executed in the interpreter, it speculatively generates machine code assuming that the subsequent rows will have the same types. If such speculative assumptions get invalidated (e.g., because a subsequent row has an unexpected type), the compiled code gets invalidated and the execution falls back to interpreted mode. Then, the runtime can update the collected type information and later re-compile the nodes to machine code accordingly. It is important to note that, even if triggering recompilation has a cost, specializations stabilize quickly [60], typically incurring only minor overhead. By leveraging a state-of-the-art dynamic compiler like Graal, DynQ can selectively compile single components of the query's physical plan. In particular, each table-scan executable node can be selectively compiled to self-contained machine code. Thanks to this approach, a query does not need to be fully compiled to machine code to benefit from high performance, since e.g. executing a join operator could lead to the evaluation of one child node in the interpreter (if it has few elements) and another child in compiled machine code.

```
executeMethodAfterJITCompilation() {
  result = 0;
  for(int i = 0; i < input.numElements, i++) {
    row = // read i-th array element
    fst = // read property "x" of row
    snd = // read property "y" of row
    // Type checking for predicate
    if(/* fst and snd are integers */) {
      if(fst < snd) { result++; }
    }
    else if(/* fst and snd are dates */){
      if(fst.isBefore(snd)) { result++; }
    }
  }
  return result;
}
```

Figure 6: Pseudo-code equivalent to the machine code generated by DynQ, executing the example query in Figure 3.

Figure 6 shows the pseudo-code equivalent to the machine code generated by DynQ for the example query of Figure 3, once both types in the example are encountered (i.e., both x and y properties have either type Int or Date). As the figure shows, all the calls to the interface methods are aggressively inlined by the compiler. The operations listed at the beginning of the while loop that interact with the host dynamic language (i.e., reading the current array element and its properties x and y) are inlined by the compiler as well. Moreover, the predicate node is compiled into two if statements that check whether in the current row the fields x and y have one of the expected types. If this is not the case, in general, the generated code would be invalidated as described above, whilst in this specific example, since there is no other specialization in the less-than node, the current row is discarded and the generated code does not need to be invalidated.

3.3 DynQ Providers

As introduced in Section 2.1, LINQ queries are not limited to object collections, instead they can be executed on any data format for which a so-called LINQ provider (i.e., a data-source specific implementation of the enumerable and queryable interfaces) is available. Such flexibility is an appealing feature for developers, since it allows executing federated queries within the same programming model, leaving the complexity of orchestrating different data sources to the system. In the context of DBMS, orchestration of federated queries is a widely studied topic, pioneered by systems like Garlic [24] and TSIMMIS [9]. As an example of custom providers in DynQ, consider a scenario where a developer needs to analyze a web-server log file in JSON format, counting the number of accesses for each user who registered to the website after a specific date, with user registration data however stored in a database. Figure 7 shows how such a log analysis can be executed with DynQ. As the figure shows, developers do not have to deal with opening/closing any file or database connection; they only need to provide a file name and configurations for accessing the database (e.g., the URL, credentials, and database name) to DynQ, which takes care of everything else. Moreover, the Calcite query planner detects the operators that can be pushed to external data sources. When executing the example query, DynQ sends (to the database) the SQL query with the predicate on the date field and retrieves only user names of the rows matching the predicate. Hence, the operation can

be executed more efficiently (i.e., exploiting database optimizations) and communication overhead is reduced.

```
var path = 'file://.../log.json';
DynQ.registerJSON('logs', path);
var config = // DB url, credentials, ...
DynQ.registerJDBC('users', config);

var result = DynQ.executeQuery(`
  SELECT users.name, COUNT(*) as count
  FROM users, logs
  WHERE logs.user.id = users.id
  AND users.registration_date > DATE ...
  GROUP BY users.name`);
```

Figure 7: Federated query with DynQ.

Implementing a DynQ provider requires defining a specific table-scan operator, which takes care of iterating over the rows in the input data source, and a data-accessor operator, which takes care of accessing the fields of each row. Our JSON provider builds on Jackson [15], an efficient JSON parser for Java, for accessing fields in JSON objects. This approach can be further extended with more complex parsers that integrate predicate execution during data-scan operations, which is an approach already explored in the literature [31, 48].

Besides the query parser and planner, Apache Calcite has another appealing feature for DynQ, namely its flexibility in integrating new data sources by defining specific adapters. A Calcite adapter takes care of representing a data source as tables within a schema, i.e., a representation that can be processed by the query planner. Similarly to LINQ providers, from a query execution point of view, a Calcite adapter takes care of converting the data from a specific source to a Calcite enumerable that can be integrated into the query engine, allowing the execution of federated queries.

3.4 Language-Specific Type Conversions

Although GraalVM allows efficient interactions among different languages [17], it may introduce overhead related to data conversion operations. As an example, dates are represented as `LocalDate` instances once shared among different languages, but the internal representation in a specific language may be different, e.g., in JavaScript dates are represented as long values, as the number of milliseconds from the epoch day January 1, 1970-01-01, UTC [12]. As an example, consider the following simple query:

`SELECT COUNT(*) FROM T WHERE X < DATE '2000-01-01'`

Suppose DynQ executes such a query (without language-specific type conversions) on JavaScript objects, in a first step (before query execution) it would create a `LocalDate` instance for the constant date (2000-01-01), then during predicate evaluation, for each row:

- It would check that the current row contains the field X and that it is actually a date instance (this step cannot be avoided in the context of dynamic languages).
- It would convert the JavaScript date into a `LocalDate` instance.
- Finally, it would compare the converted `LocalDate` instance with the constant one (2000-01-01).

On the other hand, evaluating the predicate in JavaScript would require only the first step above (i.e., checking that the field exists and has type date), if so the date comparison is executed using the JavaScript internal representation of dates, that is, a single comparison of two primitive longs, which is of course much more efficient than the steps above.

The reason for those data conversions is that different languages may internally represent the same data type differently, but exposing those types to other languages requires a common representation. To overcome these inefficiencies related to the type conversions introduced by language interoperability, DynQ provides an extension mechanism that can be used to implement language-specific type conversions. As discussed in Section 3.2, DynQ relies on two main categories of nodes, namely expression nodes and query operator nodes. Language-specific type conversions can be implemented by extending expression nodes with new specializations for types of a certain language.

Considering for example JavaScript dates, language-specific type conversions can be implemented to extend comparison nodes by taking care of checking if the object subject to a comparison is actually a JavaScript date. If so, the comparison can be executed more efficiently by delegating it to the JavaScript engine, an operation that could be inlined by the Graal compiler into the DynQ's query operator nodes. Note that language-specific type conversions do not break the high modularity of DynQ, since only expression nodes are extended with such optimizations, whilst adding new query operators, data sources, or features of the query engine (e.g., parallel query execution) would impact only query operator nodes. Moreover, language-specific type conversions are an optional extension, i.e., DynQ can execute queries on objects of a language for which no language-specific type conversions are implemented. In this case, depending on the data type of the processed objects, DynQ may have to execute data-conversion operations.
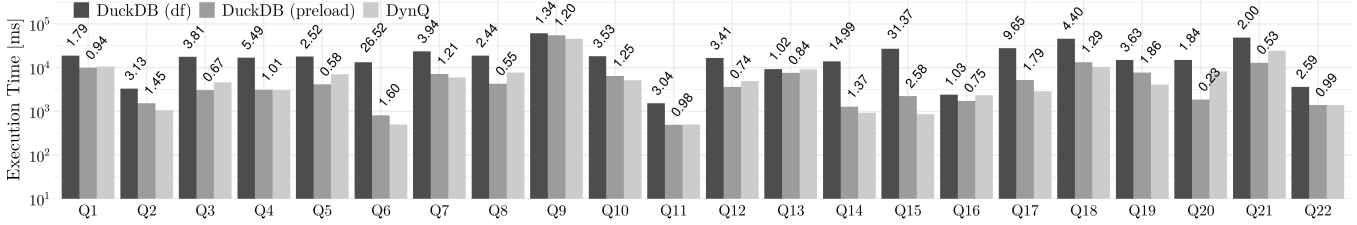
4 EVALUATION

In this section we evaluate the performance of DynQ. We evaluate DynQ with two dynamic languages, R (Section 4.1) and JavaScript (Section 4.2). In both settings, we evaluate DynQ using the TPC-H benchmark queries and a micro-benchmark composed of a set of queries based on the dataset of the TPC-H benchmark. Those queries, listed in Table 1, have been presented in the context of a *stream-fusion engine* [49]. We refer to the *i*-th query in TPC-H as Q*i*, and to the *j*-th query in the micro-benchmark as MQ*j*.

We run all our experiments on an 8-core Intel Xeon E5-2680 (2.7 GHz) with 64 GB of RAM. The operating system is a 64-bit Linux Ubuntu 18.04 and the language runtime is GraalVM Community Edition 20.1.0. Unless otherwise specified, for all experiments the reported execution times include the query preparation time, i.e., the Truffle nodes generation obtained by traversing the query plan generated by Calcite and the actual query execution time. Note that query execution time takes into account also the JIT compilation of the Truffle nodes. Unless otherwise indicated, all the figures presented in this section are bar plots that show the query execution time for each implementation. The numbers on top of the bars represent the speedup (factors) achieved by DynQ. Speedup factors below 1 indicate that DynQ is slower.

Table 1: Micro-benchmark queries from *stream-fusion engine* [49].

MQ1	SELECT COUNT(*) FROM lineitem WHERE l_shipdate >= DATE '1995-12-01'
MQ2	SELECT SUM(l_discount * l_extendedprice) FROM lineitem WHERE l_shipdate >= DATE '1995-12-01'
MQ3	SELECT SUM(l_discount * l_extendedprice) FROM lineitem WHERE l_shipdate >= DATE '1995-12-01' AND l_shipdate < DATE '1997-01-01'
MQ4	SELECT l_discount * l_extendedprice FROM lineitem WHERE l_shipdate >= DATE '1995-12-01'
MQ5	SELECT l_extendedprice FROM lineitem WHERE l_shipdate >= DATE '1995-12-01' ORDER BY l_orderkey LIMIT 1000
MQ6	SELECT l_discount * l_extendedprice FROM lineitem WHERE l_shipdate >= DATE '1995-12-01' LIMIT 1000
MQ7	SELECT SUM(o_totalprice) FROM lineitem, orders WHERE o_orderkey = l_orderkey AND o_orderdate >= DATE '1995-12-01' AND l_shipdate >= DATE '1995-12-01'

Figure 8: R TPC-H benchmark (SF-10).
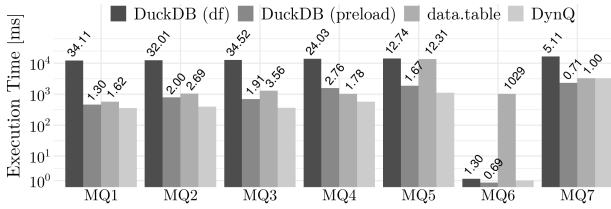
Figure 9: R micro-benchmark (SF-10).

4.1 R Benchmarks

In this section we evaluate DynQ with the R programming language. Here, we use the dataset from the TPC-H benchmark generated with the original dbgen tool [56] loaded into an R data frame. Since, like DynQ, DuckDB [46] allows executing SQL queries directly on R data frames, we evaluate DynQ on the TPC-H benchmark queries and the micro-benchmark queries against DuckDB, on a dataset of scale factor 10; the dataset size is 10GB in a text format. In particular, we use DuckDB (version 0.2.0), executed on GnuR [44] (version 3.6.3). DuckDB provides two ways for executing queries on R data frames, i.e., directly on the data-frame data structure, and in a managed table, which is much more efficient but requires an ingestion phase. We refer to the former setting as DuckDB(df), and to the latter one as DuckDB(preload). Note that, by comparing DynQ against DuckDB, the fair comparison is with DuckDB(df), since the data is accessed directly on R data frames, as in DynQ. Moreover, in evaluating DuckDB(preload), we do not measure the time spent in the ingestion phase. In this evaluation, we measure the median of 20 executions.

Due to the different query planners and implementation choices in DynQ and DuckDB (DuckDB is vectorized and interpreted whilst DynQ is tuple-at-a-time and JIT compiled), the goal of this performance evaluation is not to compare two very different systems, but rather to demonstrate that DynQ achieves performance competitive with an established, state-of-the-art data-processing system. We consider the micro-benchmark queries important in our evaluation, since, due to their simplicity, we expect the query plans to be the same in DynQ and in other systems. Moreover, since the micro-benchmark queries are rather simple, they stress data-access operations, showing that the extensibility of DynQ in accessing data in different formats does not impair query execution performance, which we consider a great achievement.

Micro-benchmarks. Due to the simplicity of the queries in the micro-benchmarks listed in Table 1, we manually implement them using the data.table API, which is arguably the most efficient library for processing R data frames. The benchmark results are depicted in Figure 9. As the figure shows, DynQ is comparable with the data.table API on MQ7 and outperforms it on all other queries by speedup factors ranging from 1.62x (MQ1) to 12.31x (MQ5). The impressive speedup on MQ6 against data.table (i.e., 1029x) is because DynQ chains query operators and stops the computation once it finds the first 1000 elements that satisfy the predicate. On MQ6, DynQ performs comparably with DuckDB, with speedup factors of about 1.3x against DuckDB(df) and 0.69x against DuckDB(preload), with a query execution time of about 1ms, showing the effectiveness of our exception-based approach for implementing early exists for the LIMIT operator. Moreover, as we will discuss in Section 4.1.1, we consider such a low query execution time a great achievement for DynQ, since the existing query engines based on compilation commonly suffer from a latency overhead due to query compilation. DynQ outperforms DuckDB(df) in all other queries as well, with speedup factors ranging from 5.11x (MQ7) to 34.52x (MQ1). DynQ performance is closer to DuckDB(preload), which significantly outperforms DuckDB(df), showing that the great flexibility of DynQ in accessing data in different formats does not impair performance.

TPC-H Benchmark. Here, we evaluate DynQ using the TPC-H benchmark. Like in our previous experiment, we compare DynQ

against DuckDB executing queries directly on the data frame, i.e., DuckDB(df) and with data loaded into a managed memory space, i.e., DuckDB(preload). The benchmark results are depicted in Figure 8. As the figure shows, DynQ outperforms DuckDB(df) in all queries, with speedup factors ranging from 1.02x (Q13) to 31.37x (Q15). In comparison with DuckDB(preload), DynQ is faster on 11 queries (i.e., Q2, Q4, Q6, Q7, Q9, Q10, Q14, Q15, Q17, Q18, Q19).

4.1.1 Latency Benchmarks. As discussed in Section 3.2, even if DynQ is an engine based on query compilation, it is able to start executing a query before compiling it, by executing the Truffle nodes which represent the query in the interpreter. This feature is crucial for obtaining high throughput when executing queries on small datasets. Here, we evaluate the throughput of DynQ against DuckDB. Since DuckDB is based on interpretation and vectorization, it does not spend any time on code generation and query compilation. On small datasets, such an approach is commonly faster than compiling queries, since the compilation overhead may not be paid off.

For this evaluation, we considered an experiment similar to the one performed in the context of Umbra [41]. Such an experiment [26] evaluates the throughput (by calculating the geometric mean of queries per second for all TPC-H queries) over different scale factors. In our experiment we evaluate the throughput over scale factors 0.001, 0.01, 0.1, 1, and 10, first on the micro-benchmark queries and then on the TPC-H queries. The benchmark results are depicted in Figure 11 for the micro-benchmark and in Figure 10 for TPC-H. As the figures show, for both the micro-benchmark and TPC-H, DynQ outperforms DuckDB(df) on all evaluated scale factors, ranging from a factor of 1.76x (SF 0.001) to 13.31x (SF 10) on the micro-benchmark, and from a factor of 1.67x (SF 0.001) to 3.66x (SF10) on TPC-H.

In comparison with DuckDB(preload), the evaluation shows interesting trends. On the smallest scale factor (SF 0.001), DynQ fully executes all queries in the interpreter; in this case, the throughput is comparable with DuckDB(preload). In particular, the DynQ throughput differs by the one of DuckDB(preload) by a factor of 0.84x on the micro-benchmark, and of 1.1x on TPC-H. On small scale factors 0.01 and 0.1, DynQ starts compiling parts of the queries; however, since the datasets are still small, the compilation is not well paid off. Indeed, the DynQ throughput is smaller than the one of DuckDB(preload), by factors 0.43x and 0.33x on the micro-benchmark, and of 0.57x and 0.58x on TPC-H. Then, on scale factor 1, in DynQ query compilation is paid off on the micro-benchmark, reaching a throughput comparable with DuckDB(preload), i.e., 1.04x factor. This is not the case for TPC-H, where the throughput of DynQ is factor 0.76x compared with DuckDB(preload). The reason is that the TPC-H queries are much more complex than the micro-benchmark queries, leading to longer query compilation times. Finally, on scale factor 10, DynQ outperforms DuckDB(preload) on the micro-benchmark queries by a factor of 1.71x, and becomes comparable with DuckDB(preload) on the TPC-H queries, by a factor of 0.99x. Our evaluation on the query latency shows that JIT compilation in DynQ is not a source of performance concerns, differently from most existing query engines based on compilation.

4.1.2 Comparison with Native DBMS. In this section we evaluate DynQ against MonetDB[21], i.e., a modern, interpreter-based,
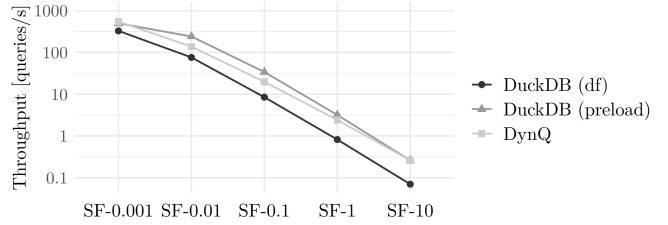
Figure 10: Throughput, geometric mean of queries/s over the TPC-H queries.
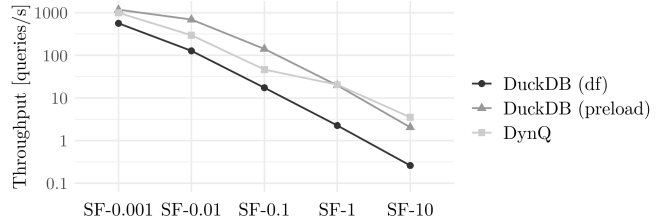
Figure 11: Throughput, geometric mean of queries/s over the micro-benchmark queries.

native DBMS featuring high-performance vectorized execution. Although we do not consider MonetDB a direct competitor to DynQ, this evaluation should be considered an indication of how DynQ performs in comparison with a native RDBMS. For this evaluation, we use MonetDB Database Server Toolkit v11.39.7 (Oct2020-SP1), executing the queries with mclient. We measure the end-to-end query execution time, taking into account the cost of inter-process communication for sending result sets from the server to the client process. For fairness, we configure MonetDB for executing in a single-thread, since we have not yet implemented parallel query execution in DynQ. For this experiment, we evaluate DynQ on R data frames, using a scale factor of 10 for both the micro-benchmark and TPC-H; we present the median of 20 executions.

The benchmark results are depicted in Figure 12 for TPC-H and in Figure 13 for the micro-benchmark. As the figures show, MonetDB outperforms DynQ in all queries containing the join operator, i.e., in MQ7 and in all TPC-H queries but Q1 and Q6. Moreover, MonetDB outperforms DynQ in MQ5. All remaining queries are rather simple and mostly dominated by table scans. For those queries, DynQ is faster than MonetDB; in particular, in both Q1 and Q6, DynQ outperforms MonetDB by a speedup factor of about 1.6x. Concerning the remaining micro-benchmark queries, on MQ6 DynQ shows an impressive speedup of about 700x; this is because MonetDB (like the data.table R package) does not stop the query execution once the first 1000 elements (i.e., the limit operator) have been found. On MQ4, DynQ outperforms MonetDB by a speedup factor of 15.04x, because MQ4 returns a large result set that MonetDB needs to serialize and transfer to the client process, whereas DynQ (being an embedded query engine) does not incur such an overhead. Finally, on MQ1, MQ2, and MQ3, DynQ outperforms MonetDB by speedup factors of 1.37x, 3.34x, and 2.1x.
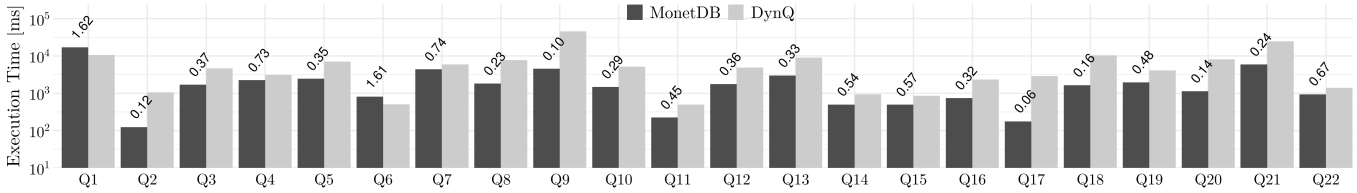
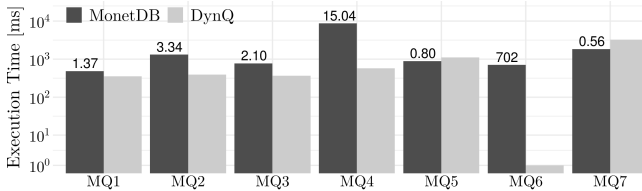**Figure 12: TPC-H benchmark against MonetDB (SF-10).**

Figure 13: Micro-benchmark against MonetDB (SF-10).

4.2 JavaScript Benchmarks

Here, we evaluate DynQ with the JavaScript programming language. For this evaluation, we first evaluate DynQ against AfterBurner [13], an in-memory database entirely written in JavaScript, on both the micro-benchmark and on TPC-H. Then, we evaluate DynQ querying data loaded into a JavaScript array of objects, like in the example of Figure 3. In this setting, we evaluate DynQ on the micro-benchmark against hand-written implementations in JavaScript and implementations that rely on Lodash [34], which is arguably the most efficient and popular stream API for JavaScript. Finally, we evaluate DynQ on existing code bases, comparing the performance of a JavaScript library against equivalent implementations using DynQ.

4.2.1 Evaluation on AfterBurner. For evaluating DynQ against AfterBurner [13], we created a specific DynQ provider for the memory layout implemented in AfterBurner, i.e., a columnar layout composed of JavaScript typed arrays. The implementation of such a specific data-source provider required only about 1000 lines of code, which shows the great extendibility of DynQ. In this setting we evaluate AfterBurner both on GraalVM and on V8 [57] (Node.JS version 12.15.0). All our experiments on AfterBurner are executed using only scale factor 1; we cannot evaluate AfterBurner on bigger datasets due to a limitation in the Node.js file parser used in AfterBurner, which cannot parse files exceeding 2GB. In this setting, we measure the median of 20 executions.

Micro-benchmarks. Due to the simplicity of the queries in the micro-benchmark listed in Table 1, we manually implemented them using the AfterBurner API, which is a streaming-like API based on a method-chaining notation, inspired by Squel.js [50]. The benchmark results are depicted in Figure 15. As the figure shows, even if AfterBurner is based on query compilation, it does not optimize the early exit for the limit operator. Thus, for MQ6, DynQ outperforms AfterBurner by a speedup factor of 56.32x on V8, and 985x on GraalVM. DynQ outperforms AfterBurner running on GraalVM for all other queries, too, ranging from a speedup factor of 4.89x (MQ1) to 21.89x (MQ5). When executing AfterBurner on V8, AfterBurner is faster than DynQ on MQ1, MQ2, and MQ7; the reason is that V8's

compiler is faster than GraalVM on these queries, so the benefit of compilation is almost immediate.

TPC-H Benchmark. We evaluate DynQ against AfterBurner on TPC-H using the original AfterBurner benchmark [1]. Since AfterBurner uses a streaming-like API, there is no query parsing and planning phase, and the query plan is made explicit by the API usage. For fairness, we manually fine-tuned the queries in our evaluation such that Calcite generates the same query plans used by AfterBurner. The benchmark results are depicted in Figure 14. As the figure shows, DynQ outperforms AfterBurner executed on GraalVM on all queries, with speedup factors ranging from 7.16x (Q20) to 36.6x (Q19). When executing AfterBurner on V8, DynQ shows comparable performance on Q1, and is slower only on Q20 by a factor of 0.77x. On all remaining queries, DynQ outperforms AfterBurner on V8 with speedup factors ranging from 1.12x (Q14) to 4.87x (Q17), which is motivated by the fact that AfterBurner materializes more intermediate results than DynQ.

4.2.2 Evaluation on Object Arrays. Here, we evaluate DynQ using JavaScript object arrays as datasets. First, we evaluate DynQ with the micro-benchmark against equivalent hand-written implementations. Then, we evaluate DynQ on an existing code base, by comparing the original implementation of a utility Node.JS npm [42] module with an equivalent one based on DynQ. In this setting we evaluate all experiments measuring only query execution time at peak performance.

Micro-benchmarks. Similarly to the evaluation on R, we manually implemented the micro-benchmark queries in JavaScript. In this setting, we evaluate the micro-benchmark queries against hand-written implementations and implementations that use Lodash. Since Lodash does not offer an API for the join operator, we do not evaluate MQ7 using Lodash. The scale factor used for our JavaScript evaluation is 1 (whereas we used a scale factor of 10 for the R evaluation). This is motivated by the fact that querying R data frames is more efficient than JavaScript object arrays, since R data frames are internally implemented using a columnar data format composed of typed arrays, whereas JavaScript arrays are a more flexible data structure that can be composed of heterogeneous objects.

The benchmark results are depicted in Figure 16. In this setting, we measure the median of 20 executions. As the figures show, DynQ outperforms both implementations for all queries. In particular, DynQ outperforms Lodash with speedup factors ranging from 1.58x (MQ2) to 48.62x (MQ6). The high speedup on MQ6 is motivated by the fact that, similarly to the `data.table` API in R, also Lodash does not chain the filter with the limit operation, unlike DynQ. Moreover, DynQ also outperforms all the hand-written implementations, with speedup factors ranging from 1.15x (MQ6)
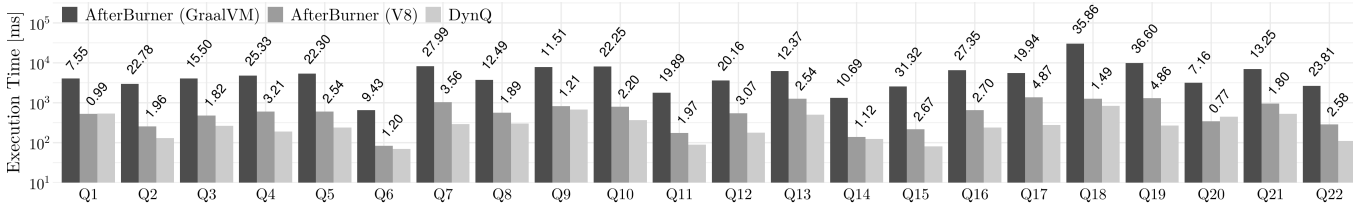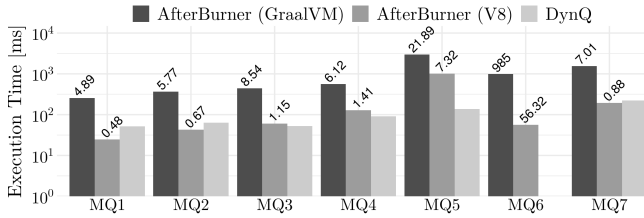
Figure 14: JS TPC-H benchmark on AfterBurner (SF-1).
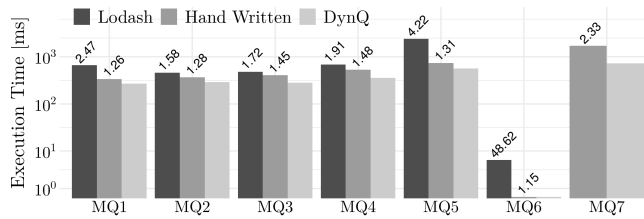

Figure 15: JS micro-benchmark on AfterBurner (SF-1).
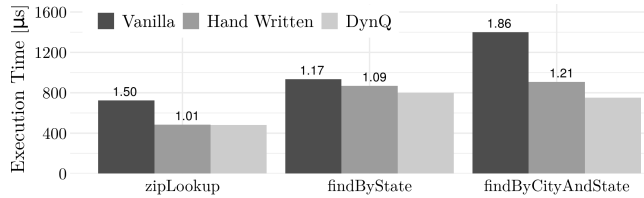

Figure 16: JS micro-benchmark (SF-1).

Figure 17: JS benchmark on *cities* module.

Figure 18: JS benchmark on *cities* module with UDF.

filtering elements. In this setting, we measure the median of 100 executions (after a warmup of 500 executions).

The npm module *cities* stores data in a single table (i.e., in a JavaScript array). The API offered by *cities* consists of `findByState`, `zipLookup`, `findByCityAndState`, and `gpsLookup`. This module implements the first three APIs using Lodash, whilst the fourth API is manually implemented with hand-optimized code, which relies on the npm module *haversine* [19] for evaluating the distance between two points. Due to the simplicity of the APIs of the *cities* module, we also implemented an hand-optimized version of the first three APIs. We have not reimplemented the `gpsLookup` API, since the original version is already hand-optimized and it does not use any third-party data-processing library. For evaluating DynQ on the `gpsLookup` API, we use two versions; one version (DynQ (JS UDF)) uses the JavaScript module `haversine` as UDF for calculating the distance between two points, whilst the other version (DynQ (Java UDF)) uses a Java UDF instead of the JavaScript one. We manually implemented the Java UDF by carefully replicating the JavaScript version, such that the executed algorithm is exactly the same.

The benchmark results are depicted in Figure 17 for the first three APIs, and in Figure 18 for the `gpsLookup` API. Since for the latter experiments we use DynQ in two different ways (i.e., implementing the UDF in JavaScript and Java), Figure 18 shows (above the bars of those two implementations) their respective speedups against the original implementation. As the figures show, DynQ outperforms both Lodash and the hand-optimized implementations in all APIs. Moreover, the evaluation of the `gpsLookup` API shows that evaluating an UDF with DynQ does not introduce any overhead when the UDF is implemented in the host dynamic language (i.e., JavaScript). This is expected, since, as discussed in Section 3, GraalVM can inline the machine code generated from the JavaScript UDF within the query execution code. Moreover, when the UDF is implemented in Java, performance improves, i.e., we measure a speedup factor of 1.64x. This is expected, since executing the JavaScript UDF requires more type checks than executing the UDF in Java. Our evaluation on existing codebases shows that, besides data analytics, DynQ is also a promising library for server-side Node.JS applications that perform in-memory data processing.

to 2.33x (MQ7). There are multiple reasons why DynQ is able to outperform the hand-written queries. First, the JavaScript semantics may enforce additional operations which are not required in data processing; as an example, JavaScript's Map object performs hashing by converting each value into a string representation. Moreover, during the execution of hand-written queries the JavaScript engine needs to perform more runtime checks than DynQ. Besides performance, the implementations using DynQ are the most concise ones. In particular, the hand-written implementations of all the queries that compose the micro-benchmark count 160 lines of code (LOC), Lodash implementations count 58 LOC, and DynQ implementations count 40 LOC.

Benchmarks on Existing Codebases. Here, we evaluate DynQ on an existing code base, by comparing the performance of an existing JavaScript library against an equivalent implementation that uses DynQ. In particular, we selected the npm module *cities* [11], which exposes a dataset of locations and offers an API for selecting and
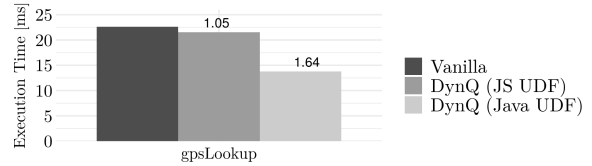
5 RELATED WORK

Query compilation in relational databases dates back to System-R [8] and has recently gained more interest both in the research community and in industrial systems. In the context of stream libraries, Steno [37] exploits query compilation in LINQ for the C$^\sharp$ language. Nagel et al. [38] further improve LINQ query compilation in C$^\sharp$ by using more efficient join algorithms and by generating native C code which is able to access C$^\sharp$ collections that reside on the heap of the managed runtime. OptiQL [53] is a stream library for the Scala language which leverages the Delite [52] framework for generating optimized code. Strymonas [27] is a stream library for Java, Scala, and OCaml. Strymonas leverages the LMS [47] framework for ahead-of-time query compilation for Java and Scala, and MetaOCaml for the OCaml language. Those libraries are designed for statically-typed languages and exploit type information for generating specialized programs during the code generation.

Most dynamic languages such as JavaScript or Python offer standard data-processing APIs (e.g., filter, map, reduce), as well as more advanced streaming libraries. However, little research has focused on optimizing integrated queries in dynamic languages. Among them, JSINQ [22] is a JavaScript implementation of LINQ, which has been extended [39] with a provider for querying the MongoDB [36] database. JSINQ compiles queries to JavaScript source code. Due to this design, JSINQ cannot outperform a hand-written implementation of a query, in contrast with DynQ.

Afterburner [13] is an in-memory, relational database embedded in JavaScript. Afterburner leverages optimized JavaScript data structures (i.e., typed arrays) and generates ASM.js [3] code, i.e., an optimized subset of JavaScript with only primitive types. Although this design offers very fast query evaluation, it comes with many limitations compared to our approach. First, it cannot execute queries on arbitrary JavaScript objects, i.e., the data needs to be inserted into a database-managed space before query execution, which introduces overhead, increases the memory footprint, and requires users to provide a data-schema, whereas our approach allows objects to be queried in-situ without any user-provided schema. Moreover, Afterburner is designed for relational data of few primitive types (i.e., numbers, dates, and strings), with no support for querying arrays and nested data structures, unlike DynQ. Finally, our approach targets any language supported by GraalVM, whilst Afterburner is specifically designed for JavaScript. In particular, the approach proposed in Afterburner cannot be easily replicated in other dynamic languages, since most of them do not offer efficient data structures like typed arrays and an efficient subset of the language to operate on primitive datatypes, like ASM.js.

DuckDB [46] is an embedded database with bindings for multiple dynamic languages, i.e., Python and R. Differently from many other embedded databases, DuckDB is able to execute queries directly on data structures managed by a dynamic language, in particular Python and R data frames. However, DuckDB cannot execute queries on arbitrary objects (e.g., on an array of heterogeneous objects), in contrast to DynQ. Hence, DuckDB does not need to face the challenge of dealing with unexpected types during query execution. Moreover, our evaluation shows that when DuckDB is configured for executing queries directly on the R data frame, DynQ outperforms DuckDB on all the evaluated queries.

The Truffle framework has been successfully adopted for optimizing existing libraries. FAD.js [7] is a runtime library for Node.js which optimizes JSON data access by parsing data lazily and incrementally when the data is actually consumed by the application. FAD.js focuses on optimizing data access, whilst DynQ focuses on data processing. Moreover, the approach described in FAD.js is complementary to our approach and can be synergistic with DynQ, i.e., we could integrate FAD.js in the DynQ JSON provider.

Recently, speculative optimizations based on Truffle have been proposed [48] in the context of Spark SQL for optimizing query execution on textual data formats. However, the described approach targets only the leaves of a query plan (i.e., table scans with pushed-down projections and predicates), whilst DynQ is a standalone query engine that can execute a whole query plan. Another important difference w.r.t. the mentioned work and DynQ is that in [48] the query compilation is obtained by combining Spark code generation and Truffle ASTs, leveraging Truffle nodes for speculative optimizations and using Spark original code as fallback, whilst in DynQ the whole query is compiled into a Truffle AST. Moreover, the speculative optimizations discussed in [48] are complementary to our approach, and such optimizations can be integrated into our DynQ providers for textual data sources.

6 CONCLUSION

In this paper we introduced DynQ, a new query engine for dynamic languages. DynQ is based on a novel approach to SQL compilation, namely compilation into self-specializing executable ASTs. Our approach to SQL compilation relies on the Truffle framework and on GraalVM to dynamically compile query operators during query execution. Truffle was designed as a programming-language implementation framework; however, in DynQ we exploit it in an innovative and previously unexplored way, i.e., as a code-generation framework integrated in a query engine. DynQ has been evaluated with two programming languages, namely R and JavaScript, against existing data-processing libraries and hand-optimized queries. Our evaluation shows that the performance of query evaluation with DynQ is comparable with hand-optimized implementations, outperforming existing data-processing systems and embedded databases in most of the benchmarks. To the best of our knowledge, DynQ is the first system which integrates a query engine within a polyglot VM directly interacting with its JIT compiler, and allowing execution of federated queries on object collections as well as on file data and external database systems for multiple dynamic languages.

Besides the features that DynQ offers to end users, we believe that DynQ would also be a useful framework in other data processing domains. Indeed, being a language-agnostic data-processing framework, DynQ could be exploited for implementing query execution in the context of other existing data processing frameworks.

REFERENCES

[1] AfterBurner Team. 2020. *AfterBurner TPC-H Benchmark*. https://github.com/afterburnerdb/afterburner/blob/master/src/tpch/benchmark_tpch.js

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

[3] ASM.js Team. 2020. *asm.js*. https://http://asmjs.org

[4] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 221–230.

[5] Gavin Bierman, Erik Meijer, and Mads Torgersen. 2007. Lost in translation: Formalizing proposed extensions to C#. In *OOPSLA 2007*. 479–498.

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

[7] Daniele Bonetta and Matthias Brantner. 2017. FAD.js: fast JSON data access using JIT-based speculative optimizations. *Proceedings of the VLDB Endowment* (2017), 1778–1789.

[8] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnick, P. Griffiths Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost. 1981. Support for Repetitive Transactions and Ad Hoc Queries in System R. *ACM Trans. Database Syst.* (1981), 70–94.

[9] Sudarshan S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. 1994. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *IPSJ*.

[10] James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-Integrated Query. *SIGPLAN Not.* (2013), 403–416.

[11] cities Team. 2020. *cities - npm*. https://www.npmjs.com/package/cities/

[12] ECMAScript Team. 2020. *ECMAScript Language Specification - ECMA-262 Edition 5.1*. https://www.ecma-international.org/ecma-262/5.1/#sec-15.9.1.1

[13] Kareem El Gebaly and Jimmy Lin. 2017. In-Browser Interactive SQL Analytics with Afterburner. In *SIGMOD 2017*. 1623–1626.

[14] H. Fang. 2015. Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem. In *CYBER*. 820–824.

[15] Jeff Friesen. 2019. *Processing JSON with Jackson*. 323–403.

[16] G. Graefe and W. J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218.

[17] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-Performance Cross-Language Interoperability in a Multi-Language Runtime. *SIGPLAN Not.* (2015), 78–90.

[18] Torsten Grust, Jan Rittinger, and Tom Schreiber. 2010. Avalanche-Safe LINQ Compilation. *Proceedings of the VLDB Endowment* 3 (Sept. 2010), 162–172.

[19] haversine Team. 2020. *cities - haversine*. https://www.npmjs.com/package/haversine/

[20] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 21–38.

[21] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.

[22] Kai Jäger. 2009. JSINQ–A JavaScript implementation of LINQ to Objects. (2009).

[23] Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* (1996), 480–503.

[24] Vanja Josifovski, Peter Schwarz, Laura Haas, and Eileen Lin. 2002. Garlic: a new flavor of federated query processing for DB2. 524–532.

[25] Jupyter Team. 2020. *Project Jupyter*. https://jupyter.org/

[26] Timo Kersten, Viktor Leis, and Thomas Neumann. [n.d.]. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. ([n. d.]).

[27] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. *SIGPLAN Not.* (2017), 285–299.

[28] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 197–208.

[29] Tomasz Marek Kowalski and Radosław Adamus. 2017. Optimisation of language-integrated queries by query unnesting. *Computer Languages, Systems & Structures* 47 (2017), 131–150.

[30] Konstantinos Krikellas, Stratis Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE 2010*. 613–624.

[31] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proceedings of the VLDB Endowment* (2017), 1118–1129.

[32] LINQ Team. 2020. *Language Integrated Query (LINQ) provider for C# - Finance & Operations | Dynamics 365*. https://docs.microsoft.com/en-us/dynamics365/fin-ops-core/dev-itpro/dev-tools/linq-provider-c

[33] LINQ Team. 2020. *LINQ to Objects (C#)*. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/linq-to-objects

[34] Lodash Team. 2020. *Lodash*. https://lodash.com/

[35] Wes Mckinney. 2010. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference* (2010).

[36] MongoDB Team. 2020. *The most popular database for modern apps | MongoDB*. https://www.mongodb.com/

[37] Derek Gordon Murray, Michael Isard, and Yuan Yu. 2011. Steno: Automatic Optimization of Declarative Queries. *SIGPLAN Not.* (2011), 121–131.

[38] Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. 2014. Code Generation for Efficient Query Processing in Managed Runtimes. *Proceedings of the VLDB Endowment* (2014), 1095–1106.

[39] K. Nakabasami, T. Amagasa, and H. Kitagawa. 2013. Querying MongoDB with LINQ in a Server-Side JavaScript Environment. In *16th International Conference on Network-Based Information Systems*. 344–349.

[40] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* (2011), 539–550.

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

[42] NPM Team. 2020. *npm | build amazing things*. https://www.npmjs.com/

[43] Oracle, Team Java. 2020. *Stream (Java Platform SE 8)*. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

[44] R Core Team. 2020. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/

[45] Mark Raasveldt and Hannes Mühleisen. 2017. Don't Hold My Data Hostage: A Case for Client Protocol Redesign. *Proceedings of the VLDB Endowment* (2017), 1022–1033.

[46] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[47] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE*. 127–136.

[48] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2020. Dynamic Speculative Optimizations for SQL Compilation in Apache Spark. *Proceedings of the VLDB Endowment* (2020), 754–767.

[49] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push vs. Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming* 28 (10 2018).

[50] Squel.js Team. 2020. *Squel.js*. https://hiddentao.github.io/squel/

[51] StackOverflow Team. 2020. *Stack Overflow Developer Survey 2019*. https://insights.stackoverflow.com/survey/2019/

[52] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. *ACM Trans. Embed. Comput. Syst.* (2014).

[53] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. 2013. Composition and Reuse with Compiled Domain-Specific Languages. In *ECOOP 2013*. 52–78.

[54] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 307–322.

[55] TensorFlow Team. 2020. *TensorFlow*. https://www.tensorflow.org/

[56] TPC. 2019. *TPC-H - Homepage*. http://www.tpc.org/tpch

[57] V8 Team. 2020. *V8 Engine*. https://v8.dev/

[58] Hadley Wickham and Romain François. 2014. *dplyr: A Grammar of Data Manipulation*.

[59] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *SPLASH*. 13–14.

[60] 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. *SIGPLAN Not.* (2017), 662–676.

[61] 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 *Onward!* 187–204.