The VLDB Journal (2023) 32:1111–1135
https://doi.org/10.1007/s00778-023-00784-2

REGULAR PAPER

DynQ: a dynamic query engine with query-reuse capabilities embedded in a polyglot runtime

Filippo Schiavio[1] · Daniele Bonetta[2] · Walter Binder[1]

Received: 11 May 2022 / Revised: 11 May 2022 / Accepted: 18 January 2023 / Published online: 13 March 2023

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 popular programming languages. Existing LINQ frameworks rely on the type system of statically typed languages such as C♯ 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. The gap between dynamic languages and LINQ frameworks has been partially bridged by the recent work DynQ, a novel query engine designed for dynamic languages. DynQ is language-agnostic, since it is able to execute SQL queries on all languages supported by the GraalVM platform. 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. The 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. In this work, we extend DynQ addressing the problem of optimizing high-throughput workloads in the context of fluent APIs. In particular, we focus on applications which make use of data-processing libraries mostly for executing many queries on small batches of datasets, e.g., in micro-services, as well as applications which make use of data-processing libraries within recursive functions. For this purpose, we present *reusable compiled queries*, a novel approach to query execution which allows reusing the same dynamically compiled code for different queries. As we show in our evaluation, thanks to reusable compiled queries, DynQ can also speed up applications that heavily use data-processing libraries on small datasets using a typical fluent API.

Keywords Language integrated queries · Dynamic languages · Query compilation · Partial evaluation

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 [15] and emerging machine-learning frameworks such as TensorFlow [64] 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 [35] (which targets .NET languages, e.g., C♯) and the Java Stream API [48]. Microsoft's implementation of LINQ not only allows developers to query in-memory collections, but it can be extended with data-source providers [34] (e.g., LINQ-to-SQL and LINQ-to-XML) that allow developers to execute federated queries (i.e., queries that process data from multi-

The work presented in this article has been supported by Oracle (ERO project 1332) and the Swiss National Science Foundation (project 200020_188688).

✉ Filippo Schiavio
filippo.schiavio@usi.ch

Daniele Bonetta
daniele.bonetta@oracle.com

Walter Binder
walter.binder@usi.ch

[1] Università della Svizzera italiana, Lugano, Switzerland

[2] VM Research Group, Oracle Labs, Cambridge, MA, USA

ple 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 [11,20], and several optimization techniques have been proposed [31,41,42]. 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 [27]), 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 [69] and Python Pandas library [38]) 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 [60].

Due to their popularity, embedded database systems such as DuckDB [53] 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 [52]. 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 into 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 [53] 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 heterogeneous objects, such as a JavaScript map.

In this article, we present an in-depth description of the DynQ [57] LINQ engine targeting dynamic languages for the GraalVM platform [72]. 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 [70] in a novel and previously unexplored way. Indeed, the Truffle framework was designed for programming-language implementations, whereas our approach exploits Truffle as a general code-generation framework in the context of a data-processing engine.

In this work, our goal is to extend the applicability of DynQ by exposing (to developers) data-frame-like API defined by method chaining, often referred to as fluent APIs [14]. The DynQ approach to query execution has been successful on database workloads [57]; however, it still remains an open research question whether it is also suitable as a drop-in replacement for a typical data-processing library implemented in a dynamic language which exposes a fluent API (e.g., the Lodash [36] library for JavaScript). Unfortunately, our preliminary evaluation shows that the performance of DynQ is suboptimal when it comes to high-throughput workloads, i.e., workloads that process a lot of small batches of datasets. Such workloads are very common in data-intensive server-side applications such as microservices. Moreover, the performance penalties can become more evident on applications that make use of data-processing libraries within recursive functions.

To overcome this performance issue, we first introduce parametricity in DynQ with a generalization of prepared statement which extends the binding of query parameters from raw values to expressions through user-defined functions (UDFs). Then, we present *reusable compiled queries*, a novel approach to query execution which allows reusing the same compiled code for different queries. Reusable compiled queries are implemented within the fluent API exposed by DynQ, and they are completely transparent to the user.

This article makes the following contributions:

- We describe 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 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.
- We introduce *reusable compiled queries*, a novel approach to query execution relying on an efficient and flexible cache of compiled queries which allows reusing the same compiled code to execute similar queries.
- We evaluate DynQ on workloads designed for both databases and programming languages. Our evaluation shows that DynQ' performance is 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 workloads.

– We release an open-source prototype of DynQ.[1]

This article is structured as follows. In Sect. 2 we introduce relevant background information. In Sect. 3 we describe the design of DynQ, and in Sect. 4 we describe how we integrate parametricity in DynQ and how we extend its query compilation model for efficient execution of multiple similar pipelines on small datasets. In Sect. 5, we evaluate the performance of DynQ against hand-optimized queries as well as existing data-processing libraries and databases. Section 6 discusses related work, and Sect. 7 concludes this article.

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 [72] and the Truffle [70] 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♯ language with an SQL-like *query comprehension* syntax and a set of query operators [6]. 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 [58], 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 [20]. 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♯ implementation of LINQ executes queries by leveraging the pull-based model, which shares many similarities with the Volcano [17] query execution model in use by many popular relational databases, such as PostgreSQL [5]. It has been shown [32] that the main performance drawbacks of this execution model are virtual calls to the interface methods (e.g., `MoveNext()` and `Current()` in C♯, 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 [7] and data-centric query compilation [44].

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 information 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. Both pull-based and push-based models have been studied from the point of view of compilers and program transformations [29,41,42]. Interestingly, it has been shown [58] that, by leveraging compiler optimization techniques such as loop fusion, method inlining, and scalar replacement, neither model clearly outperforms the other.

A well-known disadvantage of query compilation is the overhead introduced by the compilation itself. For statically typed and compiled languages, query compilation can take place at different times, namely at application compilation time (e.g., in SBQL4J [68]) or during its execution (e.g., in Steno [41]). While the former approach has the advantage of hiding the query compilation cost during the compilation of the application, it imposes serious limitations: the queries must be expressed in the application code, meaning that a system that accepts queries as user input cannot leverage such an approach. In the context of dynamic languages (such as JavaScript or Python), this approach cannot be used in general, because the application source code is directly executed by the runtime. On the other hand, runtime

[1] DynQ is available at https://github.com/usi-dag/DynQ-VLDB.

query compilation does not suffer from these limitations, but the compilation cost can shadow the benefits obtained by the optimization passes, in particular for short-running queries. Recent research [30] addresses this issue with 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 this article.

2.3 GraalVM and the Truffle framework

DynQ is implemented targeting the GraalVM [72] 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 Java, JavaScript, Ruby, Python, and R. At its core, GraalVM relies on a state-of-the-art dynamic compiler (called Graal [71]), which brings JIT compilation to all GraalVM languages. Language runtimes for GraalVM (including DynQ) are implemented using the Truffle [70] 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 [70], i.e., language interpreters that use custom API and data structures enabling explicit and direct interaction with the underlying language VM components (including the JIT compiler). The Graal optimizing compiler has special knowledge of such API, and is capable of generating efficient machine code by means of partial evaluation [25].

In addition to JIT compilation, the Truffle framework provides mechanisms to interact with any of the dynamic languages supported by GraalVM. Thanks to these 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 dynamic loop unrolling and polymorphic inline caching [22].

3 DynQ

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

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

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 [24], 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 of

the objects in a single collection (e.g., an array) can be different from each other. 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 these 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 get invalidated, giving the query execution engine the chance to re-profile the code that is being executed, possibly leading to the generation of new machine code that now takes different runtime assumptions into account.

Thanks to its design, DynQ is able to outperform hand-optimized implementations of queries written in dynamic languages. Internally, the type system of DynQ's query engine handles two main types, 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 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, 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 Fig. 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 [70]), 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 the JIT compilation to GraalVM. Dynamic compilation is triggered by DynQ, which also takes possible runtime de-optimizations and re-compilations into account. Finally, the result of query exe-

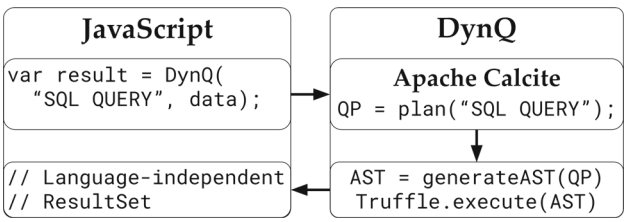

Fig. 1 High-level query life cycle in DynQ

cution, 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 uses a push-based approach and takes place by visiting the query plan generated by Calcite and converting it into Truffle nodes. The push-based query execution approach used by DynQ is inspired by the model introduced in LB2 [63]. 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 Sect. 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 [22]. 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, while 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 C for the currently visited operator O. If O is not a join (i.e., it has only one child), C will consume the rows produced by the child of O. If 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 L, and a right consumer that finds matching pairs of rows by iterating over the elements in 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 Fig. 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, DynQ returns the sorted (or aggregated) data, which is already a list of rows.

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

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

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

Fig. 2 Main interfaces in DynQ

Note that 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. Since push engines do not allow terminating the source iteration, i.e., an operator cannot control when data should not be produced anymore by its source operator, DynQ implements early exits for the *limit* query operator by throwing a special `EndOfExecution` exception.

Query compilation example. Consider the DynQ query targeting a JavaScript array of objects shown in Fig. 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 Fig. 4. A simplified implementation of the nodes that compose the query is depicted in Fig. 5. As shown in Fig. 5, the `LessThanNode` node leverages Truffle specializations for implementing the less-than

```
var data = [{x: 1, y: 2},
            {x: 2, y: 1},
            {x: Date('2000-01-01'),
             y: Date('2000-01-02')}];

DynQ.registerTable(data, 'T');
var Q = 'SELECT COUNT(*) FROM T WHERE x < y';
var result = DynQ.execute(Q);
```

Fig. 3 Example of a DynQ query on a JavaScript array

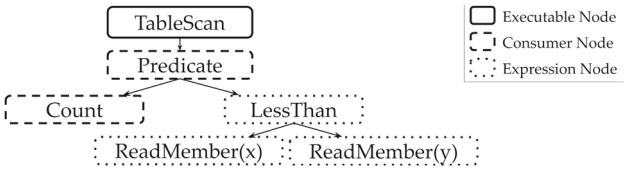

Fig. 4 AST generated by DynQ for the query in Fig. 3

operation. The `LessThanNode` implementation shown in Fig. 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 Fig. 4. If the query is 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

```
class TableScanNode implements ExecutableNode{
  ConsumerNode consumer;
  PolyglotArray input;

  public Object execute() {
    try {
      for(int i=0; i<input.numElements; i++) {
        Object row = jsArrayElement(input, i);
        consumer.consume(row);
      }
    } catch (EndOfExecution e) {}
    return consumer.getResult();
  }
}

class PredicateNode implements ConsumerNode {
  ConsumerNode consumer;
  Expression predicate;

  public void consume(Object row) {
    if(predicate.execute(row)) {
      consumer.consume(row);
    }
  }
  public Object getResult() {
    return consumer.getResult();
  }
}

class CountNode implements ConsumerNode {
  long result = 0;

  public void consume(Object row) {
    result++;
  }
  public Object getResult() {
    return result;
  }
}

class LessThanNode implements ExpressionNode {
  ExpressionNode left, right;

  public 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 l, LocalDate r) {
    return l.isBefore(r);
  }
}

class ReadMember implements ExpressionNode {
  String name;

  public Object execute(Object row) {
    return readJsMember(name, row);
  }
}
```

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

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 Graal, the 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 Sect. 5.2.

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 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 [71], 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 achieve 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.

Figure 6 shows the pseudo-code equivalent to the machine code generated by DynQ for the example query of Fig. 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

```
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;
}
```

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

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, while 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 Sect. 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 [26] and TSIMMIS [10]. 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

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

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

Fig. 7 Federated query with DynQ

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.

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 [16], 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 [33,56].

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 [18], 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 [13]. 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 Sect. 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 objects to be compared are actually JavaScript dates. 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 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, while 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.

3.5 Fluent APIs in DynQ

In this section, we focus on data-processing libraries for dynamic languages which allow developers to query heap-allocated objects using a data-frame-like API, i.e., expressing query operators as a chain of method calls. Examples of this syntax are the Spark DataFrame API [2] and LINQ queries when used with the de-sugared method-call syntax [35]. The following is an example of a pipeline built with method chaining, which is a de-sugared version of the LINQ query written with the comprehension syntax in Sect. 2.1.

```
IEnumerable<int> xs = ...;
var evenSquares = xs
    .Where(x => x % 2 == 0)
    .Select(x => x * x);
    .ToArray()
```

Such a chained method-call syntax is used by many existing data-processing libraries. Using this syntax, developers invoke an operator on enumerable objects (also called pipeline builders), passing as parameters the expressions to be evaluated by the operator. The invocation results in a new enumerable object on which another operator can be invoked, forming a chain of method calls. The method chain will result in a materialized result once the developer makes use of a terminal operator, e.g., on the de-sugared LINQ query in the example, `evenSquares.ToArray()` is called to materialize the query result into an array. From now on, we will refer to this syntax as fluent APIs. As an example of fluent APIs usage with DynQ, Fig. 8 shows how the de-sugared LINQ query in the example above can be executed with DynQ.

Note that, in contrast to SQL queries, using a fluent API developers can fragment the definition of a single query. As an example, using a fluent API one can define a function which returns an enumerable object, e.g., the representation of a table scan followed by a predicate, and then call such a function in two different contexts, appending a different terminal operator in each context. This feature greatly improve modularity; indeed, it is often offered by Object-Relational Mapping (ORM) systems [65].

Existing data-processing systems which offer fluent APIs and that are based on query compilation are implemented similarly to SQL query compilers. In particular, those systems lazily keep track of the operators composing a pipeline as well as their parameters (e.g., UDFs). Once a terminal operator is called, the sequence of operators are considered as a single, standalone query, which is then compiled as a single unit. From now on, we will refer to this approach which triggers compilation for each query execution per-query compilation.

```
var xs = [...];
var evenSquares = DynQ
  .scan(xs)
  .filter(x => x % 2 == 0)
  .map(x => x * x)
  .toArray();
```

Fig. 8 Example of fluent APIs usage with DynQ

Per-query compilation has the well-known advantage of generating code which is specialized as much as possible for a given query, which means that the generated code is, in principle, the best possible implementation of that query. DynQ offers developers the fluent APIs leveraging the described per-query compilation approach. However, as further analyzed in the next section and shown in our evaluation, while the per-query compilation approach performs very well on analytical workloads, it is suboptimal for high-throughput workloads which perform many queries on small batches of data. In the next section we will present an extension of DynQ to efficiently deal with high-throughput workloads, too.

4 Caching compiled queries

In the context of query engines based on compilation, a natural solution to the problem of improving the performance on high-throughput workloads is reducing the compilation overhead. Since DynQ is able to execute queries through interpretation before (or instead of) compiling them, compilation overhead is already mitigated. However, for high-throughput workloads, where many queries are executed on small batches of datasets, using the DynQ execution model as discussed before, the application could end up executing all those queries through interpretation.

To improve DynQ's performance on high-throughput workloads, we integrate query-reuse capabilities within the engine. In particular, we present reusable compiled queries, a novel approach to query execution inspired by the code cache implemented in managed runtimes of dynamic languages based on hot-code compilation. With hot-code compilation, the runtime first executes an application through its interpreter. Methods that are invoked often are identified as "hot" and are dynamically compiled to native code. Such an approach has the goal of reaching a stable (or steady) compiler state, i.e., eventually all hot methods which compose the running application are compiled by the JIT of the language runtime. Although the reachability of a stable compiler state is not guaranteed by the runtimes, it is typically achieved for most long-running applications. To reach a stable compiler state, the language runtime must be able to avoid recompiling the same method every time it gets called from a different code location (unless such a method gets inlined). This feature is commonly achieved by leveraging code caches, i.e., map-like data-structures which store the compiled method defined at a given code location.

Considering the context of data-processing libraries, reaching a stable compiler state means that the pipelines are

Table 1 Benefits and drawbacks of approaches to fluent API compilation: per-query, hot-code, and reusable compiled queries

Approach	Calls de-virtualization	Reachability of a stable state
per-query compilation	✓ guaranteed (all calls)	× never (by design)
hot-code compilation	× best-effort (all calls)	✓ most of the applications
reusable compiled queries	✓ guaranteed (subset of calls) × best-effort (remaining calls)	✓ most of the applications

executed in compiled code. However, it is unlikely that the performance of a single pipeline is as good as the one that could be obtained by compiling the specific pipeline using a per-query compilation approach. Indeed, the compiler might (or might not) decide to inline a certain operator as expected destination of another operator, similarly the compiler might (or might not) decide to inline a whole pipeline in a certain code location, e.g., if detected to be frequently executed.

Table 1 shows the benefits and drawbacks of the described approaches, per-query compilation, hot-code compilation, and reusable compiled queries in the context of an application that accepts queries as user input. In particular, per-query compilation can guarantee that all the virtual calls in the implementation of the query operators are de-virtualized through specialization. However, such an approach cannot reach a stable compiler state by design, as every time a query is executed, it triggers its compilation. On the other hand, hot-code compilation is commonly able to reach a stable compiler state, executing the (hot) implementation of the query operators in compiled code. However, method de-virtualization is offered only on a best-effort basis through heuristics. Although it is intuitively impossible to achieve both full de-virtualization and the reachability of a stable state, we argue that by restricting de-virtualization to a subset of calls, it is possible to design an execution model which reduces the number of compiler invocations compared with per-query compilation, but generates more specialized code than hot-code compilation. To implement reusable compiled queries, we first integrate parametricity within the query preparation, such that a single compiled query can be reused multiple times passing different parameters. Then, we leverage the pipeline builders' API to detect similar queries and so that internally we can make use of parametricity, i.e., as an automatic compiler optimization.

In this section, we first describe parametricity in its simpler form, i.e., prepared statements [47], a well-known feature offered by many database systems. Then, we introduce a parametric extension of a fluent API, a generalization of prepared statements which extends the applicability of parametricity from raw values to expressions through UDFs. Finally, we describe reusable compiled queries, a novel approach for implementing a fluent API which does not require developers to make explicit use of parametricity, without suffering from the query compilation overhead for each single query execution as done using the tradition per-query compilation approach.

4.1 Explicit parametricity

Prepared statements have been designed to efficiently execute the same query multiple times with differently bound variables. DynQ supports prepared statements which are implemented as instances of `ExecutableNode` that accept parameters. In particular, when a query is prepared, DynQ generates an equivalent AST as discussed in the previous sections. During the AST generation, when DynQ encounters a query variable, i.e., the question mark symbol, it creates an expression node which acts as a placeholder for a value to be bound at query execution time. During the execution of a prepared statement, DynQ binds the placeholders to their values which are retrieved from the local scope of the currently executing query, i.e., its stack frame. Thanks to this approach, once the AST generated from a prepared statement is compiled by the JIT compiler, all subsequent invocations of the prepared statement will execute the same compiled code. Note that, similarly to the case of executing a query without bound parameters, prepared statements may be subject to re-compilation, too. However, since prepared statements are designed to be executed multiple times, re-compilation could take place among different executions. In particular, re-compilation could take place if the types of the prepared statements' parameters change among the different invocations of the same prepared statement, as DynQ would need to generate new machine code specialized for different types.

Figure 9 shows an example of a prepared statement with DynQ. The prepared statement in the example is very similar to the example query in Fig. 3, with the only difference that the expression `x < y` in Fig. 3 is now `x < ?`. Figure 10 shows the AST generated from the prepared statement in Fig. 9. As expected, the AST is very similar to the one shown in Fig. 4, the only difference is the node `PlaceHolder$0`, i.e., the placeholder for the prepared statement variable, which replaces the node `ReadMember(y)`. Note that also the compiled code for the AST depicted in Fig. 10 is very similar to the one shown in Fig. 6, with the only difference that the generated function now takes a parameter to be compared with the property `x` of each row.

```
var data = [{x: 1, y: 2},
            {x: 2, y: 1},
            {x: Date('2000-01-01'),
             y: Date('2000-01-02')}];

DynQ.registerTable(data, 'T');
var Q = 'SELECT COUNT(*) FROM T WHERE x < ?';
var prepared = DynQ.prepare(Q);
var result1 = prepared(3);
var result2 = prepared(Date('2000-01-02'));
```

Fig. 9 Example of a DynQ prepared statement on a JavaScript array

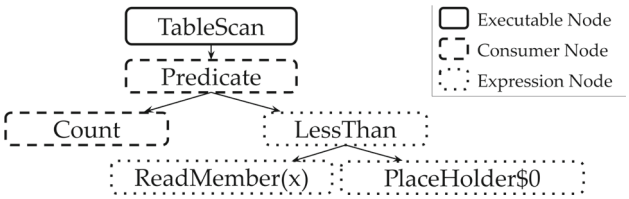

Fig. 10 AST generated by DynQ for the prepared statement in Fig. 9

As we will show in Sect. 5.3, using prepared statements, DynQ shows performance in line with an equivalent hand-written function which takes the variables of the prepared statement as arguments.

In the implementation of DynQ, we generalize the notion of prepared statement in the context of a fluent API. To this end, we introduced a special marker in our fluent API: `DynQ.par`, as well as a special operator: `prepare`. The parameter marker `DynQ.par` acts as the question mark symbol in prepared statement. However, in contrast to prepared statements, parameters are not limited to placeholder replacements for raw values, since placeholder nodes in the fluent API can represent any UDF. When a UDF is provided as argument to an operator appended into a pipeline-builder, e.g., `.map(x => x*x)`, as with per-query compilation DynQ forces the inlining of the UDF into the query code, de-virtualizing the call to such UDF. On the other hand, when the marker `DynQ.par` is used to indicate that a parameter will be provided at query execution time, DynQ introduces a virtual call into the generated code pointing to the placeholder location, where DynQ will place the reference to the UDF provided at query execution. Consider again the DynQ fluent API example in Fig. 8, which selects the squares of the even numbers in a given array. Figure 11 shows an example of parametricity defining a similar query which is parametric for the predicate expression. Such a parametric fluent API can be later invoked by passing (as parameter) an arbitrary UDF as predicate expression. Indeed, as the figure shows, the same compiled query can be used to evaluate the squares of even numbers as well as the squares of the numbers which pass any given predicate, e.g., the odd numbers.

As we will show in Sect. 5.3.3, both prepared statements and parametric fluent API are efficient solutions for query

```
var xs = [...];
var squaresOf = DynQ
  .prepare()
  .scan(xs)
  .filter(DynQ.par)
  .map(x => x * x)
  .toArray();

var evenSquares = squareOf(x => x % 2 == 0);
var oddSquares = squareOf(x => x % 2 == 1);
```

Fig. 11 Example of parametric fluent API usage with DynQ

reuse, since the query compilation happens only once and its overhead is mitigated by multiple executions of the same compiled code with different parameters. However, unfortunately, even if parametric fluent API offer a great performance benefit, its usage has many limitations in comparison with a traditional (i.e., nonparametric) fluent API. This is motivated by the fact that both prepared queries and parametric fluent API offer parametricity in an explicit manner. First, all queries must be expressed in the application code, meaning that a system that accepts queries as user input cannot leverage such an approach. Moreover, reusing queries requires developers to carefully refactor each code location in the application which makes use of a fluent API. As an example, in order to leverage the benefit of parametric fluent API, a developer needs to be aware of all the possible code locations within an application which are suitable for being expressed with parametric fluent API, which may not be the case for large applications. Moreover, the process of switching from a common data-processing library with a fluent API to a parametric fluent API version as offered by DynQ may require rewriting a large part of the application. Finally, parametric fluent API cannot be used for cross-library optimizations. In particular, suppose an application makes use of multiple libraries that internally use the same pipeline. To execute that pipeline on the same compiled code, a developer should create a separate module with the definition of the pipeline and refactor those libraries' code such that they all make use of the introduced pipeline in the shared module. In the next sections, we will describe a novel approach for implementing a fluent API which does not require developers to make explicit use of parametricity, without suffering from the query compilation overhead for each single query execution as done using the tradition per-query compilation approach.

4.2 Reusable compiled queries

In this section, we introduce *reusable compiled queries*, a novel approach to query compilation which gets the best of the two abovementioned approaches, per-query and hot-code compilation. The main design goal of reusable

compiled queries is to leverage the DynQ query-compiler to de-virtualize a strict subset of the virtual calls in the implementation of the query operators and to share the same compiled code across multiple similar pipelines. Therefore, also reusable compiled queries are suitable for executing similar pipelines multiple times on small datasets, as well as for executing recursive functions that use the same pipeline with different parameters. However, instead of requiring developers to make explicit use of parametricity, reusable compiled queries internally detect the usage of similar pipelines and leverage parametricity to reuse previously compiled code transparently with respect to the user prospective, i.e., as an automatic compiler optimization.

The subsets of calls that are ensured to be de-virtualized with reusable compiled queries are all the calls to DynQ's Truffle nodes of type `ConsumerNode`, i.e., `consume(row)` and `getResult()`. We avoid forcing inlining of calls to nodes of type `Expression`, leaving the inlining decisions of expression nodes to the underlying JIT compiler (i.e., Graal), as done with all methods during the execution of an application on a VM with hot-code compilation. Consider again the even-squares example query in Fig. 8, the sequence of operators which composes such a query starts with a source table scan operator, followed by a predicate, a projection, and finally a sink `toArray` operator which materializes the rows into an array. The representation of this query partially specialized by de-virtualizing the calls to nodes of type `ConsumerNode` is equivalent to the following parametric fluent API.

```
/var partiallySpecialized = DynQ/
    .prepare()
    .scan(DynQ/.par)
    .filter(DynQ/.par)
    .map(DynQ/.par)
    .toArray();
```

Let's now consider a similar query to the one in the example, i.e., a query which returns the cubes (instead of squares) of the odd numbers (instead of even) in a given array. Since the sequence of operators which composes this query is exactly the same as the even-square query in the example, the partially specialized function shown above can be used for executing both queries, even if their predicate and projection expressions are different. In particular, the odd-cubes pipeline can be executed on the existing compiled code for the function `partiallySpecialized`, passing as parameters `xs`, `x => x % 2 == 1` and `x => x * x * x`.

Reusable compiled queries are an optional feature of our fluent API, which developers can enable globally as well as for a single query. We implemented the caching strategy behind reusable compiled queries within the pipeline builders, leveraging the parametric fluent API described in Sect. 4.1. In particular, the reusable compiled queries are stored with a tree shape in a memory location which is shared among the whole application. The pipeline tree is composed of two kinds of nodes, intermediate nodes and leaves nodes. Each leaf contains a prepared query generated with a parametric fluent API, while each intermediate node represents a query operator. Thus, a path from the root to a leaf represents a sequence of operators, i.e., a pipeline, and such a leaf contains a reference to the compiled representation of that pipeline, i.e., a DynQ executable node. Since there must be a single root in a tree and there can be multiple scan implementation, the root of pipelines tree is an empty operator; all scan operators are children of the empty root.

Reusable compiled queries are transparently created by DynQ through the pipeline builder instances created when developers make use of a fluent syntax. Figure 12 shows the internal (simplified) Java implementation of the pipeline builders. As the figure shows, each pipeline-builder object contains two fields, a reference to a (shared) node in the pipeline tree, and an array of actual parameters. Each node in the pipeline tree contains three fields, a reference to a (partially) prepared query using a parametric fluent API, a map which stores the children nodes by operator, and a reference to an executable node generated by the (fully) prepared query, which is non-null only for leaf nodes. Note that the empty root of the pipeline tree is created with an empty map and a reference to a prepared query with parametric fluent API without any operator, i.e., `root.query = DynQ.prepare()`.

When a developer makes use of a fluent API with reusable compiled queries, DynQ internally creates a new pipeline builder composed of an empty array as actual parameters and a reference to the (shared) root of the pipeline tree. Then, every time an operator Op with parameters $(p_1, ..., p_n)$ is appended on a pipeline builder P_0 through method call, the method $P_0.appendOperator$ is called, and a new pipeline-builder instance P_1 is created (line 44 in Fig. 12). The actual parameters of P_1 are defined as $P_0.actualParameters ++ [p_1, ..., p_n]$, where $++$ denotes the array concatenation (line 45 in Fig. 12). The reference to a the pipelines-tree node in P_1 (i.e., $P_1.sharedNode$) will be evaluated as follows. If the node $P_0.sharedNode$ has already a child node for the operator Op (say $node'$), then we define $P_1.sharedNode = node'$ (line 18 in Fig. 12). Otherwise, a new (shared) node $node''$ is created for the operator O as a child of $P_0.sharedNode$ in the pipeline tree, and $P_1.sharedNode$ will be assigned to $node''$ (lines 20-32 in Fig. 12). Finally, if the operator Op is a terminal operator, then $P_1.sharedNode$ is a leaf in the pipeline tree. If such a leaf was freshly created, DynQ creates the `ExecutableNode` through the parametric fluent API instance in the tree node and cache it in the tree node itself. Otherwise DynQ reuses the already generated (i.e., cached) `ExecutableNode`. Note that the generated AST does not contain any expression node but parameters, since all actual parameters provided by the developer are internally stored in the pipeline-builder instances and replaced

```
class ParametricFluent {
  // Implementation omitted for brevity
  ParametricFluentAPI append(
    Operator op, Object[] params) { ... }

  ExecutableNode toExecutable() { ... }
}

class PipelineNode {
  ParametricFluent query;
  Map<Operator, PipelineNode> children;

  // note: non-null only for leaf nodes
  ExecutableNode executable;

  PipelineNode getOrCreate(Operator op) {
    if(children.contains(op)) {
      return children.get(op);
    }
    int n = op.parametersCount;
    Object[] params = new Object[n];
    for(int i=0; i<n; i++) {
      params[i] = DynQ.par;
    }
    PipelineNode next = new PipelineNode();
    next.query = this.query.append(op,params);
    if(op.isTerminal) {
      next.executable =
        next.query.toExecutable();
    }
    children.put(op, next);
    return next;
  }
}

class PipelineBuilder {
  Object[] actualParameters;
  PipelineNode sharedNode;

  Object appendOperator(
      Operator op, Object[] params) {

    PipelineBuilder next =
      new PipelineBuilder();
    next.actualParameters = arrayConcat(
      this.actualParameters, params);
    next.sharedNode = this.sharedNode
      .getOrCreate(op);
    if(next.executable != null) {
      // terminal operator:
      // invoke the executable node
      // and return the result
      return next.executable.execute(
        next.actualParameters);
    } else {
      // intermediate operator:
      // return the new enumerable object
      return next;
    }
  }

}
```

Fig. 12 Java-like pseudocode of PipelineBuilder objects for reusable compiled queries

with `DynQ.par` within the generated executable node, making that node reusable for any other query composed of the same sequence of operator. Once the `ExecutableNode` has been retrieved (either cached or freshly created) it is automatically invoked by passing as parameters the array $P_1.actualParameters$ (line 54 in Fig. 12). It is important to note that reusable compiled queries do not prevent additional speculative compiler optimizations, e.g., the compiler could decide to speculatively inline a UDF in a query as in hot-code compilation.

Note that reusable compiled queries share a similar design goal with the code cache implemented in language runtimes, i.e., avoiding recompiling the same code location multiple times. There are however important differences between reusable compiled queries and general-purpose code caches. In particular, a code-cache can store a compiled method, it cannot automatically partially specialize such a method for subsequent similar reuse, as in the case of reusable compiled queries. Moreover, the lookup in the code-cache is more expensive than the one in the pipeline tree, as each lookup is performed on a single global map on the code-cache, while in the case of the pipeline tree, each lookup is local at the level of a specific operator. In particular, the cost of a single lookup in the pipeline tree is constant, i.e., a lookup only checks whether a specific attribute of an object is `null`. Reusable compiled queries can be seen as an optimized data-processing-specific code cache for fluent API which is able to detect similar queries and reuse their compiled representation.

Thanks to the normal (i.e., nonparametric) fluent syntax offered by reusable compiled queries, a developer can switch from using a typical data-processing library implemented in a dynamic language to DynQ with minimal effort. In particular, there is no need to take care of rewriting the pipelines with explicit parametricity leveraging the parametric fluent API. Moreover, as we will show in Sect. 5.3.3, thanks to reusable compiled queries, DynQ outperforms Lodash [36], a popular data-processing library for JavaScript, making DynQ an attractive drop-in replacement for existing data-processing libraries.

We note that reusable compiled queries may reduce peak performance of long-running queries when compared to per-query compilation. This is expected, because per-query compilation guarantees that all calls in the query-operator implementations are de-virtualized at compilation time, which is not the case for reusable compiled queries. However, it is important to note that, when reusable compiled queries are enabled, the virtual calls that are not de-virtualized by DynQ are still candidates to be de-virtualized by the underlying VM on a best-effort basis, which in many cases is effectively applied. As an example, if a hot code location makes use of the DynQ's fluent API and the calls to DynQ are inlined within that code location, the parameters passed

to the fluent API (e.g., the UDF for a `filter` operation) can be de-virtualized by the VM once the generated code for DynQ's fluent API is inlined and optimized in the context of the caller.

5 Evaluation

In this section, we evaluate the performance of DynQ. First, we describe our evaluation plan (Sect. 5.1), explaining the setup and the motivation for each experiment. Then, we evaluate DynQ with two dynamic languages, R (Sect. 5.2) and JavaScript (Sect. 5.3). We evaluate DynQ on existing, established workloads designed for both databases and programming languages. Moreover, we also evaluate DynQ in a realistic scenario by recasting an existing server-side data-processing application to make use of DynQ.

As database workloads, we evaluate DynQ using the TPC-H benchmark [66] 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 2, have been presented in the context of *stream-fusion engine* [58], and they belong to the following categories:

- Queries consisting of selection and aggregation (without group by), leading to a single row (i.e., queries 1, 2, 3).
- Queries consisting of selection, projection, which return a list of rows (i.e., query 4), with also a limit operator (i.e., query 6) and with both sort and limit (i.e., query 5).
- A query consisting of selection and join, followed by an aggregation operator (i.e., query 7).

From now on, 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 18-core Intel i9-10980XE (@3.0 GHz) with 256 GB of RAM. The operating system is a 64-bit Ubuntu 20.04 and the language runtime is GraalVM Community Edition 21.3.0, i.e., the latest LTS release at the time of writing. 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 we do not measure the time spent for query parsing and planning done by Calcite since it is not an optimized component of our system and, on some queries, planning is currently rather slow on Calcite, a performance issue which can be solved with more engineering effort. However, it is important to note that measured time takes into account the generation of our physical plan representation (i.e., Truffle nodes), and also their JIT compilation, which happens during query execution. 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.

5.1 Evaluation plan

On R, we evaluate DynQ against the `data.table` API, DuckDB [53], and MonetDB [23]. In this setting, we import the TPC-H tables into R data frames. Since TPC-H is based on a strict (relational) schema, and the data is imported into R data frames, which is a typed data structure, the evaluation on R does not highlight the DynQ peculiarity of efficiently accessing data with unknown schema. Indeed, for all the experiments in this setting, DynQ uses the schema information from the data frames. However, DynQ currently uses the schema only for the data-access operations, all the query operators nodes as well as the other expression nodes share the same implementations as in the case of unknown schema, as described in Sect. 3.4. The main goal of this evaluation is to show that on relational database workloads the flexibility of DynQ in accessing data formats which are not directly

Table 2 Micro-benchmark queries from *stream-fusion engine* [58]

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'`

managed by the query engine does not impair performance, in contrast to other data-processing systems.

On JavaScript, we evaluate DynQ in very different settings. First, we evaluate DynQ against AfterBurner [14] using AfterBurner's memory layout, i.e., a columnar layout composed of typed JavaScript arrays. In this setting we use TPC-H and the microbenchmark queries as workloads. Since AfterBurner is a relational database, also this setting uses a strict schema, and similarly to the evaluation on R the goal of this evaluation is to show that query execution performance with DynQ on relational data is in line with a query engine which reads data using its own memory layout.

Then, we evaluate DynQ on datasets stored as JavaScript object arrays. For all the experiments in this setting, no schema information is provided to DynQ. Here, we first evaluate DynQ against Lodash [36] and handwritten implementations using the microbenchmarks. Then, we evaluate DynQ on existing code bases (the npm module *cities* [12]), leveraging prepared statements (Sect. 5.3.2) to implement a web-service backend module to search locations based on user input. Those experiments highlight the ability of DynQ to efficiently process dynamic objects with unknown schema. Finally, we evaluate reusable compiled queries (Sect. 5.3.3) on a JavaScript implementation of two relevant benchmarks that use a fluent API for data processing that we recasted from the Renaissance [50] benchmark suite, which was originally implemented in Java. One of these two latter experiments also show the ability of DynQ to handle efficient query execution on polymorphic types, since the engine needs to deal with mixed types of input arrays.

5.2 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 [66] loaded into an R data frame. Since, like DynQ, DuckDB [53] 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 0.3.0), executed on GnuR [51] (version 3.6.3). Since the measured execution time with DynQ does not take into account query planning time, we slightly modified the DuckDB R plugin so that queries can be planned and executed in two different steps, so that the measured execution time on DuckDB does not take into account query planning as well. 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 10 executions.

Due to the different query planners and implementation choices in DynQ and DuckDB (DuckDB is vectorized and interpreted while 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, the query plans are 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 2, 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 Fig. 14. As the figure shows, DynQ is slower than the `data.table` API only on MQ7 and outperforms it on all other queries by speedup factors ranging from 1.16x (MQ4) to 27.8x (MQ5). The speedup on MQ6 against `data.table` (i.e., 826x) is because DynQ chains query operators and stops the computation once it finds the first 1000 elements that satisfy the predicate (i.e., the limit operator). On MQ6, DynQ performs comparably with DuckDB, with speedup factors of about 1.18x against DuckDB(df) and 0.63x against DuckDB(preload), with a query execution time of about 1ms, showing the effectiveness of our exception-based approach for implementing early exits for the LIMIT operator. 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 4.38x (MQ7) to 48.14x (MQ1). Those speedups against DuckDB(df) are motivated by the fact that the micro-benchmark queries are simple and mostly dominated by table scans. DuckDB(df) requires table-scan operations to convert data on-the-fly from data frames into the DuckDB physical data representation, which introduces high overhead. On the other hand, DynQ can execute queries on R data frames in-situ, i.e., without any conversion. Indeed, 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. In particular, DynQ is slower than

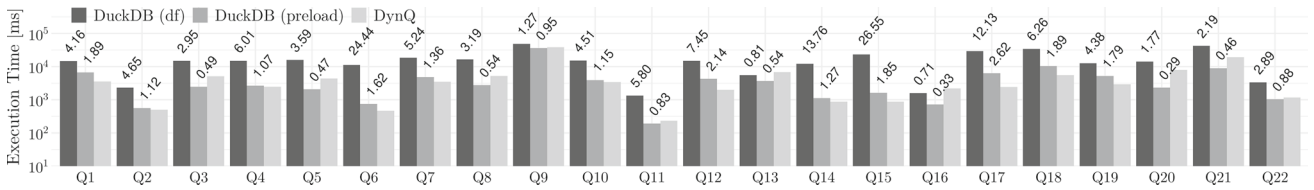

Fig. 13 R TPC-H benchmark (SF-10)

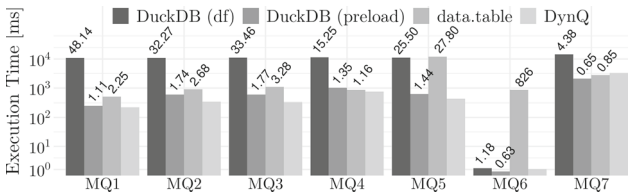

Fig. 14 R micro-benchmark (SF-10)

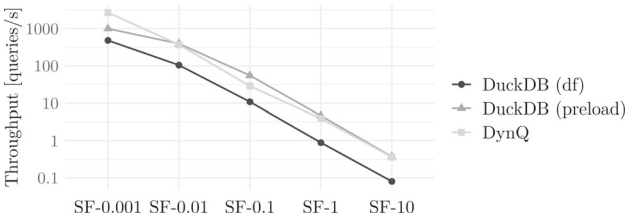

Fig. 15 Geometric mean of queries/s (TPC-H)

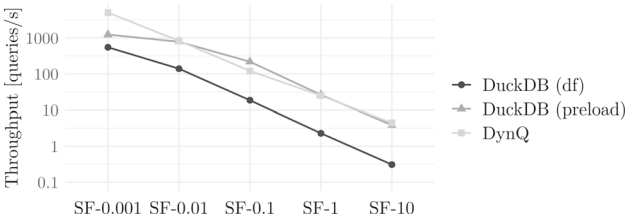

Fig. 16 Geometric mean of queries/s (micro-benchmark)

DuckDB(preload) only on queries MQ6 (factor 0.63x) and MQ7 (factor 0.65x), and outperforms DuckDB(preload) on all other queries, with speedup factors ranging from 1.11x (MQ1) to 1.77x (MQ3).

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 Fig. 13.

As the figure shows, DynQ is slower than DuckDB(df) only on Q13 (factor 0.81x) and Q16 (factor 0.71x), in all other queries DynQ outperforms DuckDB(df), with speedup factors ranging from 1.27x (Q9) to 26.55x (Q15). In comparison with DuckDB(preload), DynQ is faster on 12 queries (i.e., Q1, Q2, Q4, Q6, Q7, Q10, Q12, Q14, Q15, Q17, Q18, Q19).

Latency Benchmarks. As discussed in Sect. 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, this approach is commonly faster than compiling queries, since the compilation overhead may not be paid off.

For this evaluation, we consider an experiment similar to the one performed in the context of Umbra [45]. Such an experiment [28] 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 Fig. 16 for the micro-benchmark and in Fig. 15 for TPC-H. As the figures show, for both the micro-benchmark and TPC-H, DynQ outperforms DuckDB(df) on all evaluated scale factors. In particular, on the micro-benchmark DynQ outperform DuckDB(df) of factors 9.17x (SF 0.001), 5.94x (SF 0.01), 6.51x (SF 0.1), 11.02x (SF 1) and 14.29x (SF 10). On TPC-H, DynQ outperform DuckDB(df) of factors 5.59x (SF 0.001), 3.56x (SF 0.01), 2.65x (SF 0.1), 4.31x (SF 1) and 4.4x (SF 10).

In comparison with DuckDB(preload), the evaluation shows interesting trends. On the smallest scale factors (SF 0.001 and 0.01), DynQ fully executes all queries in the interpreter and it never triggers compilation. On scale factor 0.001, the DynQ throughput differs from DuckDB(preload) by a factor of 4.08x on the micro-benchmark, and of 2.7x on TPC-H. On scale factor 0.01, the DynQ throughput is in line with DuckDB(preload), in particular, DynQ shows a throughput factor improvement of 1.06x on the micro-benchmark, and of 0.95x on TPC-H. On scale factors 0.1, DynQ starts compiling parts of the queries; however, since the datasets are still small, most of the query execution is still in the interpreter. On such scale factor, the DynQ throughput is smaller than the one of DuckDB(preload), by factors 0.58x on the

micro-benchmark, and of 0.56x on TPC-H. Then, on scale factor 1, in DynQ query compilation is paid off on the micro-benchmark, reaching a throughput in line with the one of DuckDB(preload), i.e., 0.96x factor. This is not the case for TPC-H, where the throughput of DynQ is factor 0.84x 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 by a factor of 1.15x, and becomes comparable with DuckDB(preload) on the TPC-H queries, by a factor of 0.98x.

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.

Comparison with Native DBMS. In this section, we evaluate DynQ against MonetDB [23], a modern, interpreter-based, RDBMS 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 11.43.9 (Jan2022-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 10 executions.

The benchmark results are depicted in Fig. 17 for TPC-H and in Fig. 18 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, with the only exception of Q12, where MonetDB and DynQ show very similar execution times. All remaining queries are rather simple and mostly dominated by table scans. For those queries, DynQ is faster than MonetDB; in particular DynQ outperforms MonetDB by a speedup factor of 3.3x on Q1 and 1.3x on Q6. Concerning the remaining micro-benchmark queries, on MQ6 DynQ shows a speedup

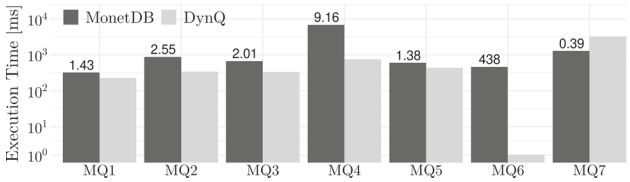

Fig. 18 Micro-benchmark against MonetDB (SF-10)

of 438x; 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 9.16x, 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, MQ5 DynQ outperforms MonetDB by speedup factors of 1.43x, 2.55x, 2.01x, and 1.38x.

5.3 JavaScript benchmarks

Here, we evaluate DynQ using the JavaScript programming language. For this evaluation, we first compare DynQ against AfterBurner [14], which is 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 Fig. 3. In this setting, we evaluate DynQ on the micro-benchmark against handwritten implementations in JavaScript and implementations that rely on Lodash [36], which is arguably the most efficient and popular data-processing library for JavaScript. Finally, we evaluate DynQ on existing code bases, comparing the performance of a JavaScript library against equivalent implementations using DynQ.

5.3.1 Evaluation on AfterBurner

For evaluating DynQ against AfterBurner [14], we implemented 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

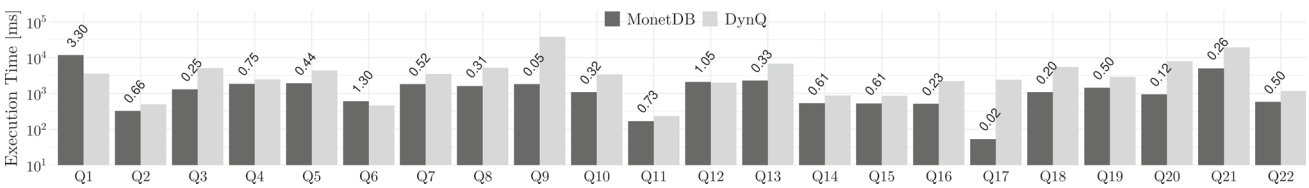

Fig. 17 TPC-H benchmark against MonetDB (SF-10)

setting we evaluate AfterBurner both on GraalVM and on V8 [67] (Node.JS version 14.17.6). 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 2, we manually implemented them using the AfterBurner API, which is a fluent API inspired by Squel.js [59]. The benchmark results are depicted in Fig. 20. 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 145x on V8, and 826x on GraalVM. DynQ outperforms AfterBurner running on GraalVM for all other queries, too, ranging from a speedup factor of 2.56x (MQ4) to 12.33x (MQ5). When executing AfterBurner on V8, AfterBurner is faster than DynQ on MQ1, MQ2, MQ4 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 fluent 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 Fig. 19. As the figure shows, DynQ outperforms AfterBurner executed on GraalVM on all queries, with speedup factors ranging from 3.21x (Q20) to 25.05x (Q11). When executing AfterBurner on V8, DynQ is slower on queries Q1 (0.82x), Q14 (0.88x), Q18 (0.65x) and Q20 (0.41x). On all remaining queries, DynQ outperforms AfterBurner on V8 with speedup factors ranging from 1.26x (Q6) to 6.03x (Q17), since AfterBurner materializes more intermediate results than DynQ.

5.3.2 Evaluation on object arrays

Here, we evaluate DynQ using JavaScript object arrays as datasets. First, we evaluate DynQ on the micro-benchmark against equivalent handwritten implementations. Then, we evaluate DynQ on an existing code base, by comparing the original implementation of an npm [46] module with an equivalent one based on DynQ. Here, we measure query execution time at peak performance.

Micro-benchmarks. Similarly to the evaluation on R, we manually implemented the micro-benchmark queries in the JavaScript language. In this setting, we evaluate the micro-benchmark queries against handwritten 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 Fig. 21. In this setting, we measure the median of 20 executions. As the figures show, DynQ outperforms implementations based on Lodash for all queries. In particular, DynQ outperforms Lodash with speedup factors ranging from 1.92x (MQ4) to 7.84x (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 performance are comparable with the hand written implementations in most of the queries. In particular, DynQ is slower than the hand written implementations only on MQ2 (0.91x), and faster on MQ6 (2.46x) and MQ7 (2.04x). There are multiple reasons why DynQ is able to outperform the handwritten queries. First, the JavaScript semantics may enforce additional operations which are not required in data processing; as an example, JavaScript's Map performs hashing by converting each value into a string representation. Moreover, during the execution of handwritten 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 handwritten implementations of the micro-benchmark queries count 160 lines of code (LOC), the Lodash imple-

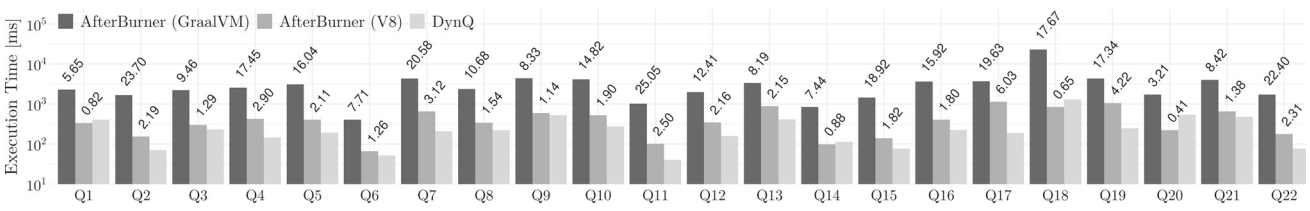

Fig. 19 JS TPC-H benchmark on AfterBurner (SF-1)

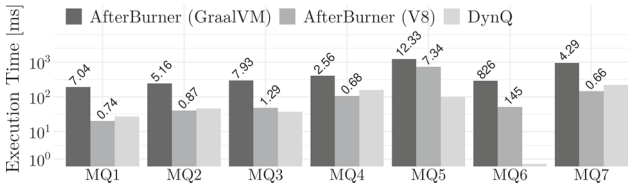

Fig. 20 JS micro-benchmark on AfterBurner (SF-1)

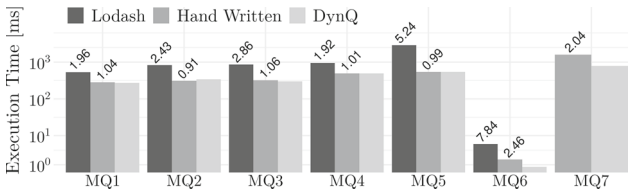

Fig. 21 JS micro-benchmark (SF-1)

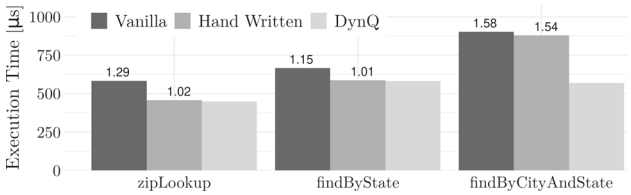

Fig. 22 JS benchmark on *cities* module

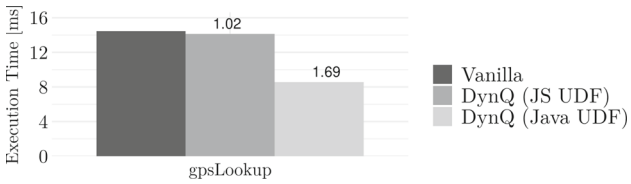

Fig. 23 JS benchmark on *cities* module with UDF

mentations count 58 LOC, and the DynQ implementations count 40 LOC.

Benchmarks on existing code bases. 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* [12], which exposes a dataset of locations and offers an API for selecting and filtering elements. In this setting, we measure the median of 1000 executions (after a warmup of 5000 executions).

The npm module *cities* stores data in a single table (i.e., in a JavaScript array). The API offered by *cities* are listed below.

- `findByState`: finds the first location which matches a given state name.
- `findByCityAndState`: finds all the locations which match a given city name and state name.
- `zipLookup`: finds the first location which matches a given zip code.
- `gpsLookup`: finds the closest location of a given point (by latitude and longitude).

This module implements the first three API using Lodash, while the fourth API is manually implemented with hand-optimized code, which relies on the npm module *haversine* [21] for evaluating the distance between two points. Due to the simplicity of the API of the *cities* module, we also implemented an hand-optimized version of the first three API. 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, while 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.

Since all the four API in the *cities* module are implemented as functions which take as parameters simple raw values (e.g., `findByState` accepts the name of a state) we implemented those API in DynQ leveraging prepared statement introduced in Sect. 4.1. As an example, the following is the DynQ implementation of the `findByState` API.

```
Var locations = { ... }
DynQ.registerTable(locations, 'locations');
Q = 'SELECT * FROM locations WHERE state=?';
findByState = DynQ.prepare(Q);
```

The benchmark results are depicted in Fig. 22 for the first three API, and in Fig. 23 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), Fig. 23 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 API. 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 Sect. 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 1.69x. 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.

5.3.3 Reusable compiled queries

Here, we evaluate reusable compiled queries, our novel approach to query compilation discussed in Sect. 4.2. As a benchmark, we re-implemented two workloads, i.e., Scrabble and Mnemonics, which are part of the Renaissance [50], a well-known Java benchmark suite. The original Java implementation of these benchmarks evaluate the Java 8 Stream API, a data-processing API offered by the Java class library. In this setting, we implemented the benchmarks on JavaScript using Lodash and DynQ with different implementations of the fluent API: the per-query compilation approach (Sect. 3.5; DynQ-per-query in the figures), with explicit parametricity (Sect. 4.1; DynQ-par in the figures) and with reusable compiled queries (Sect. 4.2; DynQ-rcq in the figures).

As expected, implementations based on DynQ with explicit parametricity outperform all other implementations on both benchmarks, so we use this configuration of DynQ as a baseline. In particular, the bar plots show, above each bar, the speedup of DynQ with explicit parametricity against the implementation referred by that bar. However, as discussed in Sect. 4.2, explicit parametricity introduces limitations in term of modularity. On the other hand, reusable compiled queries do not suffer from this limitation, as they offer a classic fluent API. In particular, modifying an application to switch from a common data-processing library (e.g., Lodash) to DynQ with reusable compiled queries requires only minimal effort. The first goal of this evaluation is to show that both explicit parametricity and reusable compiled queries outperform Lodash and DynQ using per-query compilation approach. The second goal is showing that the enhanced modularity of reusable compiled queries introduces a very low overhead w.r.t. explicit parametricity.

Scrabble is a simulation of the well-known board game, which evaluates the list of words that results in higher score among a given list of 124,455 words. The benchmark results are depicted in Fig. 24. On Scrabble, the execution time is mostly dominated by the UDFs in charge of filtering words and evaluating their score. For this reason, in comparison to the slowest implementation, i.e., Lodash, DynQ with explicit parametricity obtains a moderate speedup of 1.25x. In comparison with DynQ with per-query compilation approach, explicit parametricity obtains a speedup of 1.23x, which is motivated by the fact that per-query compilation approach requires DynQ to start executing the query in interpreted mode even if the same query is executed multiple times, since the compiled code is not shared among multiple runs. Finally, in comparison with DynQ with reusable compiled queries, explicit parametricity obtains a minimal speedup of 1.04x, showing that the better modularity of reusable compiled queries does not impair performance.

Mnemonics uses Java streams to compute mnemonic phone codes [37]. Since Mnemonics uses only simple query

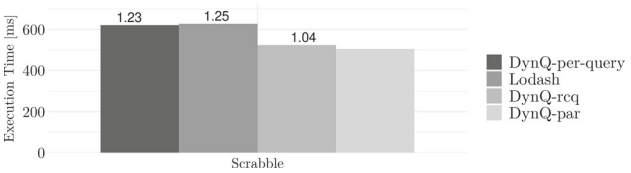

Fig. 24 Scrabble benchmark on JavaScript with reusable compiled queries

operators, we implemented this benchmark also with the JavaScript simple (but performance-oriented) implementation of the query operators (JS-Push in the figure). Since Mnemonics is implemented as a recursive function which invokes two queries on each recursive step, we consider it a relevant application for evaluating reusable compiled queries. Once the recursion is close to its halting case, those queries are executed on very small arrays, meaning that high throughput of a single call is required to achieve high performance in the whole computation. As discussed in Sect. 4, DynQ approach to query execution without leveraging parametricity or reusable compiled queries, i.e., DynQ-per-query, is not suitable for those kinds of application, as creating a fresh AST for each query execution leads to executing the whole workload through interpretation most of the time.

The benchmark results are depicted in Fig. 25. As expected, the slowest implementation is DynQ-per-query, which is outperformed by explicit parametricity by a factor of 6.92x. The speedup of DynQ with explicit parametricity and reusable compiled queries against Lodash (2.69x for DynQ-par) is because the Mnemonics benchmark uses the `flatMap` operator and Lodash implements that operator by materializing intermediate results. Our JavaScript implementations of the query operators are similar to the ones in Lodash, but the `flatMap` operator is implemented without materializing intermediate results, as in DynQ, which explains the performance improvement of our JavaScript implementations w.r.t. Lodash. Although the JavaScript implementation of the query operators is conceptually very similar to those in DynQ, leveraging explicit parametricity leads DynQ to a speedup of 1.57x against the JavaScript implementation. Also reusable compiled queries outperform the JavaScript implementation, since, as discussed in Sect. 4.2, reusable compiled queries can guarantee that the sequence of operators composing a pipeline is fully devirtualized. Finally, we note that explicit parametricity leads to a speedup of 1.15x in comparison with reusable compiled queries, a higher speedup w.r.t. the one observed on Scrabble. This is expected, since Mnemonics is a function which, for the benchmark input, executes 338 recursive calls for a single run of the benchmark, so the overhead of reusable compiled queries w.r.t. explicit parametricity is amplified in comparison to Scrabble, since each recursive call involves a query execution.

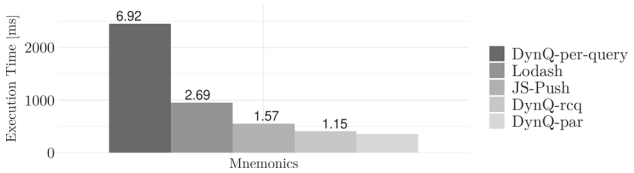

Fig. 25 Mnemonics benchmark on JavaScript with reusable compiled queries

Our evaluation on shared compiled queries shows that, besides high performance query execution, DynQ is now able to speed up high-throughput workloads, making DynQ an appealing drop-in replacement for data-processing libraries which offer a fluent API independently from the workload characteristics.

6 Related work

Query compilation in relational databases dates back to System-R [9] and it has been studied in many research work [44,54]. Recently, query compilation is increasingly gain interest both in the research community and in industrial systems. In the context of stream libraries and fluent API, Steno [41] exploits query compilation in LINQ for the C♯ language. Nagel et al. [42] further improve LINQ query compilation in C♯ by using more efficient join algorithms and by generating native C code which is able to access C♯ collections that reside on the heap of the managed runtime. OptiQL [62] is a stream library for the Scala language which leverages the Delite [61] framework for generating optimized code. Strymonas [29] is a stream library for Java, Scala, and OCaml. Strymonas leverages the LMS [55] 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 API (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 [24] is a JavaScript implementation of LINQ, which has been extended [43] with a provider for querying the MongoDB [40] database. JSINQ compiles queries to JavaScript source code. Due to this design, JSINQ cannot outperform a handwritten implementation of a query, in contrast to DynQ.

Afterburner [14] 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. Another drawback of an ingestion phase is that, if the dataset is already stored in a collection (e.g., in an array), copying the data into a managed memory space may significantly increase the memory footprint. 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, which are common in object-oriented languages. On the other hand, DynQ supports primitive types as well as arrays and nested data structures. Finally, our approach targets any language supported by GraalVM, while 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 [53] 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.

Caching the generated machine code of a compiled query for later reuse has been recently used in a PostgreSQL query compiler [49]. However, such approach is similar to DynQ prepared statement, i.e., it allows reusing compiled queries by modifying only bind variables. On the other hand, with reusable compiled queries DynQ is able to reuse compiled queries also by modifying any expression, e.g., a predicate or a projection. Permutable compiled queries [39] also addresses the problem of avoiding recompilation. However, such an approach has been designed for integrating adaptivity in compiled queries, and does not allow reusing the previously compiled code for executing subsequent queries, as done in DynQ with reusable compiled queries.

The Truffle framework has been successfully adopted for optimizing existing libraries. FAD.js [8] is a runtime library for Node.js which optimizes JSON data access by parsing data lazily and incrementally when the data is actually con-

sumed by the application. FAD.js focuses on optimizing data access, while 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.

Speculative optimizations based on Truffle have been proposed [56] 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), while 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 [56] the query compilation is obtained by combining Spark code generation and ASTs, leveraging Truffle nodes for speculative optimizations and using Spark original code as fallback, while in DynQ the whole query is compiled into an AST. Moreover, the speculative optimizations discussed in [56] are complementary to our approach, and such optimizations can be integrated into our DynQ providers for textual data sources.

Recently, a query engine which leverages Truffle for optimizing polyglot UDF execution has been proposed in Babelfish [19]. Babelfish and DynQ share some implementation choices, i.e., leveraging Truffle nodes as representation of the operators in a physical query plan. However, Babelfish's goal is efficiently integrating a polyglot UDF execution within a database system with static type information (i.e., the schema). On the other hand, DynQ's goal is integrating a query engine within dynamic language in LINQ style. Besides efficient UDF execution, DynQ also deals with query execution on unknown types both on the datasets and the expressions, e.g., executing queries with UDFs on arrays of JavaScript objects. This flexibility is crucial for embedding LINQ-style queries and a fluent API within a dynamic language.

7 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. We extended DynQ with reusable compiled queries, a novel approach to query compilation which is suitable for applications that perform data processing on many small datasets.

Our evaluation shows that the performance of query evaluation with DynQ is comparable with, and sometimes better than, hand-optimized implementations, outperforming existing data-processing systems and embedded databases in most of the benchmarks. Moreover, thanks to reusable compiled queries, DynQ is also able to outperform data-processing libraries also on high-throughput workloads that perform data processing on many small datasets. 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, since DynQ is a language-agnostic data-processing framework, and its flexible on executing queries on different data representation, DynQ could be exploited for implementing query execution in the context of other existing data processing frameworks written in any language supported by GraalVM. As an example, DynQ could be used for implementing query execution on external files with possibly malformed data (e.g., JSON files) in a similar way as done in our previous work on Spark SQL [56], i.e., by leveraging speculative optimizations.

Finally, besides the high impact that DynQ can provide being the first LINQ system for dynamically typed languages, we believe that DynQ is a great tool for further research. As an example, DynQ could be extended for implementing runtime predicate reordering in the area of adaptive data-processing. Indeed, we think that releasing DynQ is an important contribution for the database and programming-language research communities. For this reason, we released DynQ as an open-source project, available at https://github.com/usi-dag/DynQ-VLDB.

Funding Open access funding provided by Università della Svizzera italiana.

References

1. AfterBurner Team. AfterBurner TPC-H Benchmark, 2020. https://github.com/afterburnerdb/afterburner/blob/master/src/tpch/benchmark_tpch.js
2. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: SIGMOD, pp. 1383–1394 (2015)
3. ASM.js Team. asm.js (2020). http://asmjs.org
4. Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M.J., Lemire, D.: 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, pp. 221–230 (2018)
5. Behandelt PostgreSQL. Postgresql, 1996. www.PostgreSQL.org/about
6. Bierman, G., Meijer, E., Torgersen, M.: Lost in translation: formalizing proposed extensions to c#. In: OOPSLA 2007, pp. 479–498 (2007)
7. Boncz, P.A., Zukowski, M., Nes, N.: Monetdb/x100: hyper-pipelining query execution. In: CIDR 2005, pp. 225–237 (2005)
8. Bonetta, D., Brantner, M.: FAD.js: fast JSON data access using JIT-based speculative optimizations. Proc. VLDB Endow. **10**, 1778–1789 (2017)
9. Chamberlin, D.D., Astrahan, M.M., King, W.F., Lorie, R.A., Mehl, J.W., Price, T.G., Schkolnick, M., Griffiths Selinger, P., Slutz, D.R., Wade, B.W., Yost, R.A.: Support for repetitive transactions and ad hoc queries in system r. ACM Trans. Database Syst. **6**, 7–94 (1981)
10. Chawathe, S.S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J.: The tsimmis project: integration of heterogeneous information sources. In: IPSJ (1994)
11. Cheney, J., Lindley, S., Wadler, P.: A practical theory of language-integrated query. SIGPLAN Not., pp. 403–416 (2013)
12. cities Team. cities-npm (2020). https://www.npmjs.com/package/cities/
13. ECMAScript Team. ECMAScript Language Specification-ECMA-262 Edition 5.1 (2020). https://www.ecma-international.org/ecma-262/5.1/#sec-15.9.1.1
14. El Gebaly, K., Lin, J.: In-browser interactive sql analytics with afterburner. In: SIGMOD 2017, pp. 1623–1626 (2017)
15. Fang, H.: Managing data lakes in big data era: what's a data lake and why has it became popular in data management ecosystem. In: CYBER, pp. 820–824 (2015)
16. Friesen, J.: Processing JSON with Jackson, pp. 323–403 (2019)
17. Graefe, G., McKenna, W.J.: The volcano optimizer generator: extensibility and efficient search. In: Proceedings of IEEE 9th international conference on data engineering, pp. 209–218 (1993)
18. Grimmer, M., Seaton, C., Schatz, R., Würthinger, T., Mössenböck, H.: High-performance cross-language interoperability in a multi-language runtime. In: SIGPLAN Not., pp. 78–90 (2015)
19. Grulich, P.M., Zeuch, S., Markl, V.: Babelfish: efficient execution of polyglot queries. Proc. VLDB Endow. **15**(2), 196–210 (2021)
20. Grust, T., Rittinger, J., Schreiber, T.: Avalanche-safe linq compilation. Proc. VLDB Endow. **3**, 162–172 (2010)
21. haversine Team. cities-haversine, 2020. https://www.npmjs.com/package/haversine/
22. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In: Proceedings of the European Conference on Object-Oriented Programming, ECOOP, pp. 21–38 (1991)
23. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K.S., Kersten, M.L.: MonetDB: two decades of research in column-oriented database architectures. IEEE Data Eng. Bull. **35**(1), 40–45 (2012)
24. Jäger, K.: Jsinq: a javascript implementation of linq to objects (2009)
25. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. **28**, 480–503 (1996)
26. Josifovski, V., Schwarz, P., Haas, L., Lin, E.: Garlic: a new flavor of federated query processing for db2, pp. 524–532 (2002)
27. Jupyter Team. Project Jupyter, 2020. https://jupyter.org/
28. Kersten, T., Leis, V., Neumann, T.: Tidy tuples and flying start: fast compilation and fast execution of relational queries in Umbra. VLDB J. **30**, 883–905 (2021)
29. Kiselyov, O., Biboudis, A., Palladinos, N., Smaragdakis, Y.: Stream fusion, to completeness. In: SIGPLAN Not., pp. 285–299 (2017)
30. Kohn, A., Leis, V., Neumann, T.: Adaptive execution of compiled queries. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 197–208 (2018)
31. Kowalski, T.M., Adamus, R.: Optimisation of language-integrated queries by query unnesting. Comput. Lang. Syst. Struct. **47**, 131–150 (2017)
32. Krikellas, K., Viglas, S., Cintra, M.: Generating code for holistic query evaluation. In: ICDE 2010, pp. 613–624 (2010)
33. Li, Y., Katsipoulakis, N.R., Chandramouli, B., Goldstein, J., Kossmann, D.: Mison: a fast JSON parser for data analytics. Proc. VLDB Endow. **10**, 1118–1129 (2017)
34. LINQ Team. Language Integrated Query (LINQ) provider for C# - Finance & Operations |Dynamics 365, 2020. https://docs.microsoft.com/en-us/dynamics365/fin-ops-core/dev-itpro/dev-tools/linq-provider-c
35. LINQ Team. LINQ to Objects (C#), 2020. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/linq-to-objects
36. Lodash Team. Lodash, 2020. https://lodash.com/
37. Martin Odersky. State of Scala, 2011. http://days2011.scala-lang.org/sites/days2011/files/01
38. Wes Mckinney. Data structures for statistical computing in python. *Proceedings of the 9th Python in Science Conference*, 2010
39. Menon, P., Ngom, A., Ma, L., Mowry, T.C., Pavlo, A.: Permutable compiled queries: dynamically adapting compiled queries without recompiling. Proc. VLDB Endow. **14**(2), 101–113 (2020)
40. MongoDB Team. The most popular database for modern apps |MongoDB, 2020. https://www.mongodb.com/
41. Murray, D.G., Isard, M., Yu, Y.: Steno: automatic optimization of declarative queries. In: SIGPLAN Not., pp. 121–131 (2011)
42. Nagel, F., Bierman, G., Viglas, S.D.: Code generation for efficient query processing in managed runtimes. Proc. VLDB Endow. **7**, 1095–1106 (2014)
43. Nakabasami, K., Amagasa, T., Kitagawa, H.: Querying mongodb with linq in a server-side javascript environment. In: 16th International Conference on Network-Based Information Systems, pp. 344–349 (2013)
44. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. **4**, 539–550 (2011)
45. Neumann, T., Freitag, M.J.: Umbra: a disk-based system with in-memory performance. In: CIDR (2020)
46. NPM Team. npm |build amazing things, 2020. https://www.npmjs.com/
47. Oracle. Using Prepared Statements, 2021. https://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html
48. Oracle, Team Java. Stream (Java Platform SE 8), 2020. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
49. Pantilimonov, M., Buchatskiy, R., Zhuykov, R., Sharygin, E., Melnik, D.: Machine code caching in postgresql query jit-compiler. In: 2019 Ivannikov Memorial Workshop (IVMEM), pp. 18–25 (2019)
50. Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuma, P., Studener, M., Bulej, L., Zheng, Y., Villazon, A., Simon, D., Wuerthinger, T., Binder, W.: Renaissance: benchmarking suite for parallel applications on the JVM (2019)

51. R Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna (2020)
52. Raasveldt, M., Mühleisen, H.: Don't hold my data hostage: a case for client protocol redesign. Proc. VLDB Endow. **10**, 1022–1033 (2017)
53. Raasveldt, M., Mühleisen, H.: Duckdb: an embeddable analytical database. In: Proceedings of the 2019 International Conference on Management of Data, pp. 1981–1984 (2019)
54. Rao, J., Pirahesh, H., Mohan, C., Lohman, G.: Compiled query execution engine using JVM. In: ICDE '06, p. 23, USA. IEEE Computer Society (2006)
55. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In: GPCE, pp. 127–136 (2010)
56. Schiavio, F., Bonetta, D., Binder, W.: Dynamic speculative optimizations for sql compilation in apache spark. Proc. VLDB Endow. **13**, 754–767 (2020)
57. Schiavio, F., Bonetta, D., Binder, W.: Language-agnostic integrated queries in a managed polyglot runtime. Proc. VLDB Endow. **14**(8), 1414–1426 (2021)
58. Shaikhha, A., Dashti, M., Koch, C.: Push vs. pull-based loop fusion in query engines. J. Funct. Program. **28**, 10 (2018)
59. Squel.js Team. Squel.js, 2020. https://hiddentao.github.io/squel/
60. StackOverflow Team. Stack Overflow Developer Survey 2019, 2020. https://insights.stackoverflow.com/survey/2019/
61. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: a compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst. **13**, 1–12 (2014)
62. Sujeeth, A.K., Rompf, T., Brown, K.J., Lee, H., Chafi, H., Popic, V., Wu, M., Prokopec, A., Jovanovic, V., Odersky, M., Olukotun, K.: Composition and reuse with compiled domain-specific languages. In: ECOOP 2013, pp. 52–78 (2013)
63. Tahboub, R.Y., Essertel, G.M., Rompf, T.: How to architect a query compiler, revisited. In: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, pp. 307–322 (2018)
64. TensorFlow Team. TensorFlow, 2020. https://www.tensorflow.org/
65. Torres, A., Galante, R., Pimenta, M.S., Martins, A.J.B.: Twenty years of object-relational mapping: a survey on patterns, solutions, and their implications on application design. Inf. Softw. Technol. **82**, 1–18 (2017)
66. TPC. TPC-H - Homepage, 2019. http://www.tpc.org/tpch/
67. V8 Team. V8 Engine, 2020. https://v8.dev/
68. Wcisło, E., Habela, P., Subieta, K.: A java-integrated object oriented query language. In: International Conference on Informatics Engineering and Information Science, pp. 589–603 (2011)
69. Wickham, H., François, R.: dplyr: a grammar of data manipulation (2014)
70. Wimmer, C., Würthinger, T.: Truffle: a self-optimizing runtime system. In: SPLASH, pp. 13–14 (2012)
71. Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D., Grimmer, M.: Practical partial evaluation for high-performance dynamic language runtimes. SIGPLAN Not., pp. 662–676 (2017)
72. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Onward!, pp. 187–204 (2013)