ScootR: Scaling R Dataframes on Dataflow Systems

Andreas Kunft
Technische Universität Berlin
andreas.kunft@tu-berlin.de

Lukas Stadler
Oracle Labs
lukas.stadler@oracle.com

Daniele Bonetta
Oracle Labs
daniele.bonetta@oracle.com

Cosmin Basca
Oracle Labs
cosmin.basca@oracle.com

Jens Meiners
Technische Universität Berlin
jens.meiners@tu-berlin.de

Sebastian Breß
DFKI GmbH
sebastian.bress@dfki.de

Tilmann Rabl
Technische Universität Berlin
rabl@tu-berlin.de

Juan Fumero
The University of Manchester
juan.fumero@manchester.ac.uk

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

ABSTRACT

To cope with today's large scale of data, parallel dataflow engines such as Hadoop, and more recently Spark and Flink, have been proposed. They offer scalability and performance, but require data scientists to develop analysis pipelines in unfamiliar programming languages and abstractions. To overcome this hurdle, dataflow engines have introduced some forms of multi-language integrations, e.g., for Python and R. However, this results in data exchange between the dataflow engine and the integrated language runtime, which requires inter-process communication and causes high runtime overheads.

In this paper, we present ScootR, a novel approach to execute R in dataflow systems. ScootR tightly integrates the dataflow and R language runtime by using the Truffle framework and the Graal compiler. As a result, ScootR executes R scripts directly in the Flink data processing engine, without serialization and inter-process communication. Our experimental study reveals that ScootR outperforms state-of-the-art systems by up to an order of magnitude.

CCS CONCEPTS

• **Information systems → Query languages for non-relational engines**; *Record and buffer management*; Data management systems;

KEYWORDS

Dataflow Engines, Language Integration, Data Exchange

ACM Reference Format:
Andreas Kunft, Lukas Stadler, Daniele Bonetta, Cosmin Basca, Jens Meiners, Sebastian Breß, Tilmann Rabl, Juan Fumero, and Volker Markl. 2018. ScootR: Scaling R Dataframes on Dataflow Systems. In *SoCC '18: ACM Symposium on Cloud Computing (SoCC '18), October 11–13, 2018, Carlsbad, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3267809.3267813

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267813

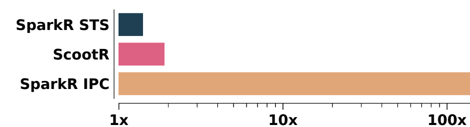


Figure 1: R function call overhead compared to the native execution on the dataflow system. Source-to-source translation (STS) and inter-process communication (IPC) are compared to native Spark. ScootR is compared to native Flink.

1 INTRODUCTION

Extracting value from data is a very important, but complex task. Typically, data analysts rely on complex execution pipelines composed of several stages, (e.g., data cleansing, transformation, and preparation) that need to be executed before the actual analysis or machine learning algorithm can be applied [34]. Often, these pipelines are repeatedly refined to obtain the best suited subset of data for prediction and analysis. Therefore, programming languages with rich support for data manipulation and statistics (provided as library functions), such as *R* and *Python*, have become increasingly popular [2]. More recently, such languages also started receiving increased attention in other domains such as enterprise software ecosystems [23]. While these languages are convenient for non-expert programmers, they are typically designed for a single-machine and in-memory usage. Thus, they run out of memory if data exceeds the available capacity and cannot scale-out without significant implementation efforts. In contrast, parallel dataflow systems, such as Apache Flink [3] and Apache Spark [33], are able to handle large amounts of data. However, data scientists are often unfamiliar with the systems' native language and programming abstraction, which is crucial to achieve good performance [4]. To overcome this barrier, dataflow engines provide additional programming interfaces in *guest languages*, such as R and Python, which build on familiar abstractions, e.g., *dataframes*. Current state-of-the-art solutions integrate guest languages in two fundamental ways. They either use inter-process communication (IPC) or source-to-source translation (STS).

Inter-process communication. In this approach, the guest language runtime runs in a separate process. Input and output data

has to be exchanged via IPC between the process running the dataflow engine and the process running the guest language. IPC supports any valid code in the guest language but can incur major performance overhead in the form of *data exchange* between the processes and *serialization* to a format readable by both languages.

Source-to-source translation. In this approach, guest language code is translated to host language code, e.g., to the dataflows's native API. While STS translation achieves near native performance, as the translation happens before program execution, it is limited to a restricted set of functions and library calls. Support for a rich set of language features would require a full-fledged compiler. The impact on the execution time for both methods is demonstrated in Figure 1, by comparing SparkR [27], which supports STS translation and IPC. In this case, the execution of a simple user-defined function (UDF) via IPC is more than 100× slower compared to STS translation[1]. Thus, current approaches either yield sub-optimal performance or restrict the set of usable guest language features.

In this paper, we introduce ScootR, a novel language integration approach based on an efficient intermediate representation (IR) for both the guest and the host language. We focus on the execution of UDF heavy pipelines – the bottleneck in current state-of-the-art solutions – and provide a dataframe-centric R API for transformation, aggregation, and application of UDFs with minimal overhead. Using a common IR, ScootR avoids the data exchange and serialization overheads introduced by IPC. ScootR extends on STS translation by using the existing compiler infrastructure and back-end of the host language to support a rich set of language features and pre-compiled modules.

ScootR is based on a tight integration of the fastR [24] language runtime with the Java Virtual Machine (JVM) responsible for executing Flink data processing pipelines. fastR is a GNU-R compatible R language runtime based on the Truffle language implementation framework and the Graal dynamic compiler [30, 31] for the JVM. Thus, ScootR efficiently executes a rich set of R UDFs within the same runtime and completely avoids IPC. By harnessing Truffle's efficient language interoperability system, ScootR accesses Flink data types directly inside the R UDFs, avoiding data materialization and unnecessary data copying due to marshalling.

Our experiments show that ScootR achieves comparable performance to source-to-source translation and outperforms IPC based approaches by up to an order of magnitude, while supporting a rich set of language features. Analytics pipelines written in ScootR can either be executed on a single local machine, utilizing multi-threaded execution or distributed in a cluster, using both intra-node multi-threading and inter-node parallelism.

In summary, we make the following contributions:

(1) We present a new integration technique that enables seamless, low-overhead, interoperability between the fastR R language runtime and the Flink dataflow engine. Our approach avoids the overhead of IPC and serialization present in state-of-the-art solutions.

(2) We describe how we enable efficient exchange and access of data structures between fastR and Flink with minimal overhead and why it is necessary to achieve good performance.

(3) We compare our implementation in an experimental study against the current state-of-the-art, as well as native execution in R and fastR.

[1] In Section 5.2, the full benchmark is discussed in detail.

2 BACKGROUND

In this section, we provide the necessary background to the systems used in ScootR. We describe the language interoperability features of Truffle that we use to achieve efficient data exchange between R and the Flink execution engine. Furthermore, we describe the basic concepts behind Flink needed in the following sections.

2.1 Graal, Truffle, and FastR

Truffle [31] is a language implementation framework. It is used to develop high-performance language runtimes by means of self-optimizing abstract syntax tree (AST) interpreters. These ASTs collect profiling information at runtime and specialize their structure accordingly. Examples for such specializations include elision of unnecessary type conversions as well as removal of complex method dispatch logic. Truffle provides interoperability features to efficiently exchange data and access functions between languages build on top of it [11].

Graal [30] is a dynamic compiler that has special knowledge of Truffle ASTs and can produce highly-optimized machine code by means of (automatic) partial evaluation: as soon as a Truffle AST self-optimizes itself by reaching a stable state, Graal assumes its structure to be constant and generates machine code for it. De-optimizations and speculation failures are handled automatically by Graal by transferring execution flow back to the AST interpreter. fastR is a high-performance GNU-R compatible R language runtime implemented using Truffle that relies on the Graal dynamic compiler for runtime optimization. It is open-source, and is available as one of the default languages of the GraalVM multi-language runtime [11, 30]. GraalVM can execute Java applications on top of the HotSpot [19] Java VM, and can execute other Truffle-based language runtimes such as JavaScript, Ruby, Python, and LLVM.

2.2 Apache Flink

Apache Flink is a massively parallel dataflow engine that extends the ideas of the MapReduce paradigm. It combines optimizations known from the database community with the UDF-centric work flow of a distributed execution engine. Written in Java, Flink offers native APIs in Java and Scala based on the *DataSet* abstract data type that represents a distributed collection. This type enables to describe dataflow pipelines by means of transformations on bags, based on second-order operators. A special tuple type (with fixed arity and typing of the fields), together with an extended set of operators, such as join and grouping, enable a more relational specification of the execution pipelines using acyclic graphs. Jobs specified in the DataSet API internally build a logical execution plan, which is optimized. The resulting physical plan is then scheduled for execution by the master node, called *JobManager*. The worker nodes – called *TaskManager* – execute in a shared-nothing architecture.

3 GUEST LANGUAGE INTEGRATION

In this section, we discuss different approaches to call R code from within Java. While we focus on integrating R in a dataflow engine, the presented approaches are applicable to other programming languages as well. In the following examples, we concentrate on the task of evaluating a user-defined function, written in R, within a worker node of a dataflow engine, e.g., Apache Flink or Apache Spark. With that, we present current approaches based on inter-process communication, source-to-source translation, and common intermediate representations for both languages.

3.1 Inter-process communication

The first approach is based on inter-process communication between the Java process that runs the worker node of the dataflow system and an external R process. We provide a schematic illustration of the IPC in Figure 2.

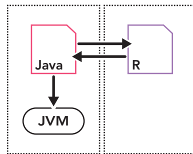


Figure 2: IPC between Java and an external R process.

In the IPC approach, the worker node sends elements to the R process and the function is evaluated with the native R interpreter. Afterwards, the result is sent back to the worker node. The approach introduces three drawbacks: (*i*) The data in Java has to be serialized to a format suitable for exchange and deserialization in R. (*ii*) Additional communication overhead is introduced, as data is exchanged either through a (local) socket or a (memory-mapped) file, shared between the two communicating processes. (*iii*) In resource restricted environments, Java and R have to compete for the available memory, due to their isolated address spaces. Despite the presented drawbacks, IPC is used by several systems [9, 13, 32], as it only requires basic I/O facilities.

3.2 Source-to-Source Translation

Source-to-source translation (STS) tackles the problem from a completely different direction as the previously presented approach based on IPC. Instead of exchanging data between the processes, the execution of R code is avoided altogether by executing a (semantically-equivalent) translation of the UDF to a programming language natively supported by the dataflow engine.

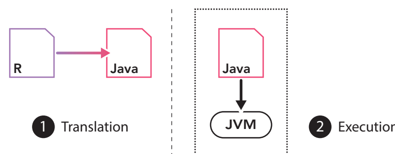


Figure 3: Source-to-source translation of R to Java for execution in native dataflow API.

As an example, Figure 3 shows how the R source code of a user-defined function is translated to equivalent Java source code, *before* the actual execution of the dataflow program takes place. Once the translation is done, there is no interaction with R during program execution and STS translation offers native performance. Nevertheless, extensive support of guest language features essentially requires a full-fledged compiler and yield a huge effort. Thus, STS translation is often restricted to a domain-specific language subset to reduce the implementation effort.

3.3 Hybrid Approach

The R integration in Apache Spark [33], called SparkR [27], builds on a hybrid approach, combining STS translation and IPC. R language constructs that can be directly mapped to Spark's native dataframe API are source-to-source translated, as described in Section 3.2. These constructs are limited to a subset of filter predicates (e.g., >, <, =, etc.), column manipulations and transformations (e.g., arithmetic operators, string manipulations, etc.), and library function calls. For instance, in the following example, an R `filter` function selects all tuples in the dataframe `df` that have the value *"R"* in their *language* column:

```
df <- filter(df, df$language == "R")
```

The R `filter` function can be translated to the following `filter` operator in Spark's native Scala dataframe API, including the user-defined predicate:

```
val df = df.filter($"language" === "R")
```

To run arbitrary UDFs, the user can specify functions on partitions of the dataframe, analogous to the `apply` function in R, and grouped data for aggregation. Here, source-to-source translation cannot be used anymore and SparkR falls back to the previously presented approach based on inter-process communication (Section 3.1). Thus, SparkR represents a combination of both presented approaches. It achieves near native performance for a subset of operators via STS translation, but falls back to IPC in case of general user-defined functions.

3.4 Common Intermediate Representation

To avoid IPC while supporting holistic optimizations for a rich set of language features, one can define a common intermediate representation (IR) both languages are translated to.

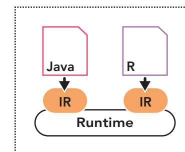


Figure 4: A common runtime for the intermediate representations of both languages.

The IR is then interpreted (and/or just-in-time compiled) on a common runtime or compiled to machine code, as depicted in Figure 4. Implementing such a compiler is a huge implementation effort. Translating high-level languages to an existing compiler

infrastructure reduces this implementation effort, increases portability, and facilitates the reuse of compiler back-end components, e.g., to generate efficient machine code through a Virtual Machine (VM). Prominent examples are the Java Virtual Machine (JVM), which uses byte code as IR, and LLVM, which uses e.g., bitcode as IR.

Weld [20] provides an IR based on *linear types* that is optimized for multi-threading by loop tiling and vectorization. Implementations for several important libraries exist, including Python's Pandas and NumPy, which are evaluated lazily to build a Weld Abstract Syntax Tree (AST). Thereby, it avoids the creation of intermediate results, e.g., by fusing dataframe transformations followed by a NumPy sum function.

As described in Section 2.1, GraalVM provides Truffle, a language implementation framework. Languages implemented in Truffle are automatically optimized and JIT compiled by the Graal compiler. In contrast to Weld, which provides its own runtime, GraalVM runs on the HopSpot runtime and therefore, can run and access Java seamlessly. In the next section, we describe how ScootR uses GraalVM to provide efficient execution of R code within the worker nodes of dataflow systems.

4 SCOOTR

In this section, we describe our approach to execute R code in Apache Flink. We first provide an overview of all the components in general, before we discuss each step in detail. We focus on the efficient execution of user-defined functions, as they introduce a big performance overhead in currently available solutions (Section 3).

4.1 Overview

We base our approach on fastR, the R implementation on top of the GraalVM. As introduced in Section 2, GraalVM is a language execution runtime capable of running *multiple* languages – including R and Java – in the same virtual machine instance. GraalVM enables seamless interoperability between all of its supported languages, and provides efficient *language interoperability* [11] capabilities. ScootR builds on such capabilities to expose Flink's internal data structures to the fastR engine. ScootR distinguishes between two main phases: the *plan generation phase* and the *plan execution phase*.

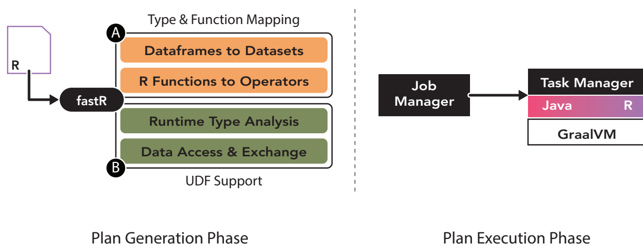


Figure 5: The two main phases in ScootR.

Figure 5 details the components of each phase. In the plan generation phase, described in Section 4.2, ScootR builds a Flink operator plan from R source code, which is later executed by the dataflow engine. Similar to Flink's native APIs, the dataframe API of ScootR is evaluated lazily. Calls to the API trigger no execution, but build a Flink operator graph until a *materialization* point – a sink in the graph – is reached. Section 4.2 Ⓐ explains the steps necessary for *Type and Function Mapping*. ScootR defines the correct mapping of R dataframes to Flink's DataSet abstraction. Based on this mapping, R functions are translated to their corresponding Flink operators as defined by ScootR's function implementations in fastR. We detail the necessary steps to enable efficient execution of UDFs in Section 4.2 Ⓑ (*UDF Support*). First, we show how ScootR determines the result types of UDFs via runtime type analysis. Second, we describe how ScootR achieves efficient data exchange between Java and R and why it is necessary to provide access to Flink's underlying data structures.

After the operator plan is created, it is deployed on the Flink cluster and executed during the plan execution phase (Section 4.3). R UDFs are executed in parallel by each worker node. ScootR's integration with the GraalVM ensures that each UDF is optimized by the Graal JIT compiler, automatically.

Running Example. Listing 1 gives an example of an R application, which makes use of the ScootR dataframe API. We use it as running example throughout the rest of this Section. In Lines 1 – 2, we specify the Flink cluster we execute on and its degree of parallelism. In Lines 4 – 8, we read an input file and convert it to a dataframe. Next, we project the `flight_id` and `miles` columns (Line 10) and create a new column `km` by applying the UDF in Line 11. Finally, we retrieve the first 5 entries of the dataframe in Line 12.

```
1  flink.init(host, port)
2  flink.parallelism(dop)
3
4  df    <- flink.readdf(
5           "hdfs://some/input/file",
6           list("flight_id", "distance", ...),
7           list("integer", "integer", ...)
8         )
9
10 df    <- flink.select(df, flight_id, miles)
11 df$km <- df$miles * 1.6
12 df$head(5)
```

Listing 1: Code snippet for the running example in ScootR.

4.2 Plan Generation Phase

In this section, we explain each necessary step to translate programs defined in ScootR to their corresponding dataflow pipelines. We first detail the mechanics of the type and function mapping in Ⓐ, before we describe the introduced optimizations to increase the performance of R UDFs in Ⓑ.

Ⓐ — In the following, we detail the type and function mapping between R and Java and describe the translation process for functions without user-defined code.

Mapping R Data Frames to Flink DataSets. Dataframes are a popular abstraction to represent tabular data in languages such as Python and R and used in many libraries. As ScootR's API is build around dataframes, it is crucial to provide a valid and efficient mapping from an R dataframe to a data type suitable for processing in Flink. While Flink can work with arbitrary Java data types, it

provides special facilities for instances of its Tuple$N$ type, where N specifies the tuple's fixed arity. The fields of a tuple are typed and can be compared to a row in a database table or an entry in a dataframe. Thus, we can define a mapping from an R dataframe df with N columns and types $t_1, t_2, ..., t_N$ to a Flink dataset ds with element type Tuple$N$<$t_1, t_2, ..., t_N$>. As individual dataframe columns can be accessed either by index or name, we maintain a mapping of the dataframe column names to their respective tuple indexes in our dataframe wrapper for the dataset.

Defining R Functions for Flink Operators. During lazy evaluation, an R program using ScootR's API is translated to a Flink operator plan. To generate such a plan from the R source code, ScootR introduces new Truffle AST nodes (called *RBuiltinNode*) that correspond to new built-in functions available to the R user. Some important functions in ScootR are depicted in Table 1. Listing 2 shows a snippet for the `flink.select` built-in function used in Line 11 of our running example in Listing 1. The specification of Truffle nodes relies on annotations, while the actual code for the AST nodes is generated during compilation by the Truffle framework. The `flink.select` built-in expects a dataframe df and a variable length argument (indicated by three dots) representing the columns to project (Line 2). The behavior of the node is defined by methods annotated with *@Specialization*. For example, the behavior of the `flink.select` node in our snippet is defined in the `select` method in Line 5. Based on the arguments, it extracts the projected columns and adds the according Flink `project` operator to the execution graph.

```
1  @RBuiltin(name           = "flink.select",
2            parameterNames = {"df", "..."})
3  abstract class FlinkSelect extends RBuiltinNode.Arg2 {
4    @Specialization
5    DataFrame select(DataFrame df,
6                     RArgsValuesAndNames fields) {
7      // determine projected columns
8      // add Flink `ProjectOperator` to Execution Plan
9    }
10 }
```

Listing 2: Simplified Snippet of the `flink.select` RBuiltin.

Functions without User-Defined Code. R functions that do not involve user-defined code are directly mapped to their counterpart operators defined by the Flink dataset API. For instance, the `flink.select` function from the running example (Listing 1, Line 10) is directly mapped to the `project` operator from Flink's DataSet API, as described in the previous paragraph. ScootR performs the entire mapping of R functions without user-defined code during the plan generation phase and, therefore, they introduce no runtime overhead.

B — In the following, we detail how ScootR enables efficient execution of R UDFs. First, we describe how ScootR determines the correct result types of the executed UDFs. Second, we detail how to efficiently exchange data between Java and R for the input and R and Java for the output of the UDF.

Runtime Type Analysis. Explicit type ascription is not required for UDFs specified in R. In contrast, Flink requires the input and output types of operators when the operator plan is built. While the *container* type is fixed to Tuple$N$, the arity N and the types of the fields may change when the UDF is applied. Thus, ScootR needs to execute the R UDF to determine its result type before the corresponding Flink operator, calling the function at runtime, is created. For performance considerations, we avoid taking a sample of the actual data to determine the initial data types, since the data might reside in a distributed file system such as HDFS [22]. Therefore, the current implementation requires to specify the data types in the R API when reading files (as in Listing 1, Line 7). The result types of all other operators in the pipeline are determined automatically by ScootR. The result type of non-UDF operators is defined by their semantics. In case of UDF operator, the R function is executed during the plan generation phase, while the operator graph is created. We instantiate temporary tuples with field values based on the runtime type inference of the previous operator, call the function with them, and thereby determine the result type of the UDF. In case the UDF does not return results for the temporary tuple used (e.g., it requires a specific number range), ScootR throws an exception during compilation and requests an explicit type annotation. Thus, ScootR keeps track of the current tuple type until the operator graph is built.

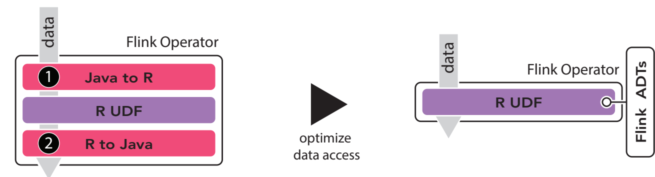


Figure 6: Schema of a Flink operator calling an R UDF without (left) and with (right) applied optimizations.

Data Access and Exchange. An important aspect in ScootR is the efficient access to Java data types in R and vice versa. As we operate in the context of dataflow engines, the R UDFs are on the hot path and get called for each processed data item in the worst case, e.g., for the `map` operator. Thus, efficient data exchange *and* access between Java and R is crucial. Figure 6 depicts the data flow during the execution of a Flink operator. The unoptimized data flow is shown on the left side of Figure 6. Even though ScootR avoids data exchange due to the shared runtime, it still has to apply type conversion and introduces materialization points. On the right side, the optimized version is depicted. Due to the direct access of Java types in R (and vice versa), as well as access to Flink's abstract data types, we avoid type conversion and materialization. In the next paragraphs, we show how ScootR achieves these optimizations.

1 — In the context of dataframes, efficient access to the processed elements means fast access to Flink Tuples (representing *rows*) and their fields (representing *columns*). ScootR distinguishes operators by their expected input – single or multiple tuples per function call:

(*i*) The first case are tuple-at-a-time operators, e.g., `map` or `flatmap`. A naive solution is to determine the columns that are accessed

Table 1: Examples from the ScootR API.

Function	Example	Description
flink.select	`flink.select(df, x = COL1, COL2)`	Project (and rename) the specified set of columns
←	`df$new ← (COL1 / COL2) * 0.1`	Create (Override) column by applying the UDF on each row
flink.apply	`flink.apply(df, func)` `flink.apply(df, key = COL1, func)`	Apply `func` on each row. Group by `COL1` column and apply `func` on each group
flink.groupBy	`max ← flink.groupBy(df, 'COL1')`	Group by `COL1` for further aggregation, e.g., `max`
flink.collect	`fastr_df ← flink.collect(df)`	Collect a distributed dataframe `df` on the driver

in the UDF and to expose them as explicit function arguments. This is achieved, by *wrapping* the UDF in a R function which expects the column values required by the UDF as arguments. For example, the ← `apply` function from Listing 1, Line 11, expecting one argument for the values of the *miles* column, is wrapped into following function: `function(miles) miles * 1.6`.

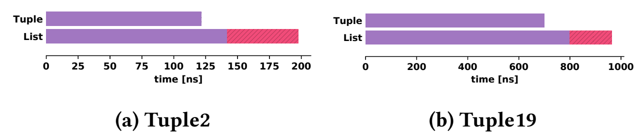
In general, multiple columns are accessed in the UDF and their values have to be extracted in a loop before being passed to the R function in the naive solution. To avoid this scenario and be able to call the function directly with the tuple instance, ScootR makes use of the Truffle language interoperability features, a message-based approach to gain access to foreign objects internals, called *dynamic access* [11]. It enables a guest language (e.g., R) to efficiently access objects from another language (e.g., Java) running in the same GraalVM instance. ScootR integrates the tuple type in the Truffle framework and thus, directly passes the tuples as arguments to the function to access fields as they would be dataframe columns.

(*ii*) The second case are operators that expect multiple tuples per function call, e.g., a `mapPartitions` operator. Flink provides access to all tuples expected by the function (e.g., all tuples contained in a partition) by an iterator. Using the aforementioned interoperability features, we provide direct access to the iterator in the R UDF. As the iterator itself returns tuples, ScootR can access them directly as described before. Without this optimization, ScootR would need to materialize all tuples contained in the partition before passing them, e.g., as an RList to the UDF. Therefore, it would introduce a pipeline barrier in the normally streaming execution, as all tuples have to be materialized before the R function can be called.

❷ — Likewise, an R UDF returns results that are passed back to the calling Flink operator for further processing in the operator pipeline. Therefore, ScootR also needs an efficient mechanism to access results of an R UDF in Flink. The R return type has to be handled differently depending on the higher-order operator that calls the R function:

(*i*) The simplest type is a `map` operator that returns exactly one value per input tuple. ScootR guarantees this case by the semantics of the ← `apply` function (Table 1). In this case, a new tuple is created after the R function execution, either appending a new column or replacing an existing one with the new values.

(*ii*) In the general `apply` function (Table 1), the UDF returns a vector of length N. Since fastR provides wrappers for all primitive type lists in R, the result vector can be accessed with the same methods as the Java Vector class[2]. While this grants access to the values in Java, we still have to convert the R vector to a Tuple$N$ for further processing in Flink. To avoid this type conversion, ScootR provides built-in functions (see Section 4.2 Ⓐ) to create Flink tuples directly in the R function. Thus, instead of returning an R vector, the function is rewritten to create and return instances of the corresponding tuple type directly using the built-in functions. Figure 7 shows the execution time of a general `apply` function that does nothing except returning (*a*) a small and (*b*) a large tuple. We can observe that the function execution (purple bars) itself is about 15 percent faster when we create the Flink tuples directly in the R function. In addition, when an R list is returned, it still has to be converted into the equivalent tuple class, indicated by the pink bars in Figure 7. Overall, ScootR achieves 1.75× better performance by instantiating the tuples directly in the R UDF in this micro-benchmark.



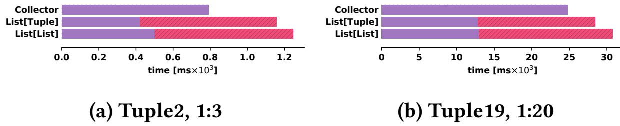
(a) Tuple2 (b) Tuple19

Figure 7: Creating a Flink Tuple in the R UDF vs. Creating the Tuple from an R List in Java. Purple depicts the time spent in the function call, pink the time for type conversion.

(*iii*) Finally, ScootR needs to handle the case where higher-order functions return multiple values per input tuple, e.g., the `flatmap` operator. To this end, Flink provides an additional `Collector` class as argument, which *collects* the results of the function. Again, we provide direct access to the Collector from R. This avoids returning a list containing the results of the UDF, which would require an additional pass over the results to insert the values into the `Collector` in Java. Figure 8 shows the time to execute a flatmap operator returning a List of Lists (the inner lists representing the tuples), a List of tuples, and finally directly using the Collector class in the R function. The function just returns (*a*) 3 tuples with arity 2 and (*b*) 20 tuples with arity 19 for each input tuple. We can observe that ScootR achieves 1.3× speedup when using the Collector directly. Interestingly, the function call takes almost twice as long using the Collector. This is due to increased complexity, as the collector

[2] R lists are backed by an Java array and provide constant time random access.

stores the output using Flink's internal buffer management in the function call. Returning a list, the tuples have to be inserted after the function execution as depicted by the pink bars.



(a) Tuple2, 1:3 (b) Tuple19, 1:20

Figure 8: Flatmap using Flink's `Collector` directly, returning an RList of RList elements, and returning a RList of Tuples. Purple depicts the time spent in the function call, pink the time for type conversion.

Illustrative Rewrite for the Running Example. Figure 9 shows the succession of R functions used and their corresponding Flink operators. Only the `apply` function includes user-defined code which has to be called and executed at runtime. All other functions can be replaced with the corresponding Flink operators during the plan generation phase. In the following, we describe the necessary modifications to the ← `apply` function before job execution.

Since the UDF is executed on every row in the example dataframe, a Flink `map` operator is generated. To determine the result type of the function, we execute it during the plan generation phase with a $Tuple2_{in} : (long, long)$ instantiated with random instances $(1.1, 0.3)$, based on the field types defined by the previous operator. The operator then calls the R function during the execution of each input tuple and has the following signature:

$$Tuple2_{in} : (long, long) \mapsto Tuple3_{out} : (long, long, long)$$

The additional field in the return value results from the extension of the dataframe with the *km* column (Line 11 in Listing 1). Furthermore, given the mapping from column names to tuple indexes, the access to the `miles` column is replaced with a tuple index access[3]:

```
function(tuple) tuple[[2]] * 1.6
```

4.3 Plan Execution Phase

After ScootR successfully generated the operator graph for the pipeline, it forwards it to the JobManager, which schedules its execution. During this process, the JobManager also sends the *serialized* R code to the responsible TaskManagers. The ScootR operator implementations evaluate the R UDFs upon their first use. Since a TaskManager can execute the same function simultaneously in its available task *slots*, ScootR caches the code and shares it between executions. Initially, the UDF is interpreted, however, as it gets *hot*, the JIT compiler will produce an efficient compiled version of it, which is executed for every subsequent call. The result of the job can either be directed to an output file or the user can collect it on the driver node via the `flink.collect` operator. If it is collected, ScootR passes the result as a dataframe to fastR, which can then be used for further local processing.

[3]The tuple fields indexes are 1 based in R.

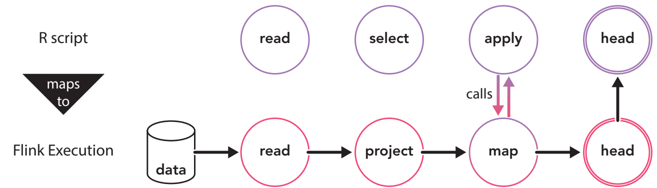


Figure 9: Mapping from an R script (Listing 1) to the corresponding Flink execution plan.

4.4 Implementation

We implemented ScootR in Flink without any changes to its existing code base. Thus, it benefits from new features introduced by new versions of Flink. All functions of ScootR's API are represented via RBuiltin nodes. In addition, all internal data structures that are accessible inside R UDFs are provided as *TruffleObjects*. This enables, e.g., direct access to the Java tuples and their fields in R, without data exchange, as described in Section 4.2.

Library Support. R packages are used very frequently. Therefore, it is important to support R packages. Many of the packages call C implementations internally to achieve good performance. fastR implements the C API of GNU-R and therefore can execute such packages and external libraries seamlessly. While this works for most of the popular packages, some rely on GNU-R internals, which complicates the integration in fastR. fastR is continuously improved and more packages are added, which are then directly available in ScootR too.

5 EVALUATION

In this section, we compare ScootR against the previously presented approaches by evaluating both micro-benchmarks and operator pipelines using real-world datasets.

5.1 Experimental Setup

Cluster Setup. We conducted our experiments on a four-node cluster. Each node features an Intel E5530 processor (2.4GHz, 8 cores), and 24GB main memory. The nodes are connected via a 1GBit Ethernet connection. We used Spark v2.2.0, Flink v1.3.1, and Hadoop v2.7.1 for our distributed experiments. Furthermore, we use GNU-R v3.2.3 [21], the latest versions of fastR[4] and Graal[5] available while conducting the experiments, and JVMCI v0.33, based on the JDK v1.8.0_141. We execute each experiment 7 times and report the median time with error bars.

Datasets. We used two real-world datasets for our evaluation. The first dataset is the *Airline On-Time Performance Dataset*[6], which is also used to evaluate SparkR [27] and dplyr [29]. It contains JSON-formatted arrival data for flights in the USA with detailed information such as departure time, origin and destination, etc. We cleaned the data and reduced it to 19 columns per record (many of the original dataset columns contain no entries for 99.9% of the rows). As parsing JSON infers a high overhead in dataflow systems [16],

[4]https://github.com/graalvm/fastr, commit: 72b868a
[5]https://github.com/graalvm/graal, commit: 7da41b3
[6]https://www.transtats.bts.gov/Tables.asp?DB_ID=120

we converted the dataset to the CSV format. The resulting file size, containing data from the years 2005 – 2016, is 9.5GB. The second dataset is the *Reddit Comments*[7] dataset, which consists of line-separated JSON entries. Each entry represents a comment on the news aggregator website *www.reddit.com*. In addition to the actual text of the comment, it contains further meta-information, such as the author name, up and down votes, category, etc. Similarly to the first dataset, we cleaned and converted the data to CSV in a preprocessing step. The raw data is provided as separate files for each month and we use the first 4 consecutive months starting from 2016. Each month amounts to roughly 33GB of raw data, resulting in about 14GB per month for the CSV used as input.

Benchmark Overview. We evaluated our approach for single and multi-node execution, comparing against: native GNU-R, fastR, and SparkR. First, we conducted a set of micro-benchmarks for *(i)* operators without user-defined code (e.g., `select`), and *(ii)* operators with user-defined code (e.g., `map` and `flatmap`). Here, we also compare the execution of SparkR with source-to-source compilation against the IPC approach. The goal of this set of micro-benchmarks is to highlight the benefits deriving from the efficient execution of UDFs in ScootR.

Second, in order to show the relevant performance impact of efficient UDFs in the context of real-world applications, we evaluated benchmarks consisting of operator pipelines. To this end, we chose to evaluate an *extract-transform-load* (ETL) or preprocessing pipeline on the airline dataset proposed by Oscar D. Lara et al.[8] [32], a MapReduce-style aggregation pipeline, and a mixed pipeline with a multi-threaded ETL phase and successive, single-threaded model training in R.

5.2 Micro-benchmarks

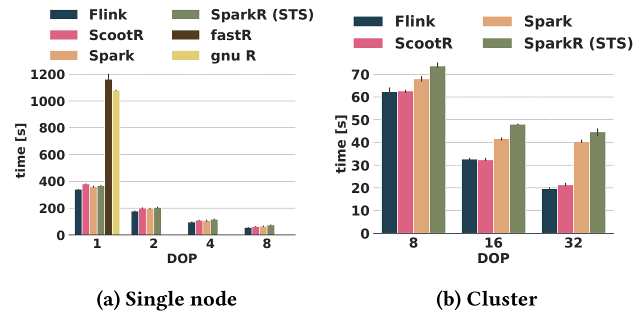
In this section, we present micro-benchmarks for several operators in isolation. For non-UDF operators, both ScootR and SparkR achieve almost native performance compared to the native dataflow API. This is expected, as the operators can be translated before execution. Compared to standalone GNU-R and fastR, SparkR and ScootR are up to 20× faster on a single node (using 8 cores) and up to 46× for distributed execution (4 nodes × 8 cores).

The micro-benchmarks for operators with user-defined code show that ScootR and SparkR with STS translation achieve almost equal performance compared to their respective native API. It is important to note that ScootR achieves this even though the R UDF is executed. Benchmarks using IPC in SparkR, and thereby executing the R UDF, reveal its performance drawbacks as it fails to execute on the airline dataset within the set experiment timeout of 4 hours. Experiments on 10% of the original data show up to an magnitude slower execution times for SparkR with IPC compared to ScootR. The benchmarks also show the importance of direct access to internal data-structures to avoid additional cost due to materialization barriers. The benchmarks for 1:N output operators, e.g., `flatmap`, verify our assumptions that direct access to the Flink collector class in R yields comparable performance to native execution. In the following paragraphs, we describe each benchmark in detail.

[7] http://files.pushshift.io/reddit/comments/

[8] The pipeline reports the maximum arrival delay per destination for flights from NY.

Non-UDF Operator. In this first micro-benchmark, we project three columns from the airline dataset and write the result to a file. Figure 10 depicts the results for this benchmark for (a) a single node and (b) execution on the four nodes cluster. *SparkR (STS)* reflects the result for SparkR with source-to-source translation.



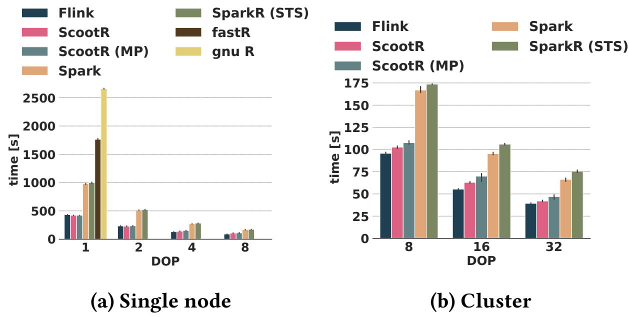
(a) Single node **(b) Cluster**

Figure 10: Micro-benchmark for a single `select` function.

As expected, SparkR (STS) and ScootR achieve almost native performance as the R `select` function is mapped to the `project` operator of the native APIs of the benchmarked systems. SparkR (STS) is about 1.15× slower than native Spark and ScootR about 1.13× slower than Flink. In contrast to GNU-R and fastR, which materialize the input data in memory before applying the `select` function, Flink and Spark stream the input data directly to the `project` operator. This results in a speedup of about 3× for both SparkR and ScootR compared to GNU-R for single-threaded execution. With increasing degree-of-parallelism (DOP), the speedup increases further to about 20× on a single node with DOP 8 (Figure 10 (a)) and up to 46× for the fully distributed execution on 32 cores (Figure 10 (b)). This result is expected, as the project operator is embarrassingly parallel. Interestingly, fastR is by a factor of 1.06× slower than GNU-R. We attribute this behavior to a more efficient implementation of the `read.csv` function in GNU-R.

UDF Operator with 1:1 Output. In this micro-benchmark, we compare the execution of an ← *apply* function similar to the one in the running example (Line 11 in Listing 1). It multiplies the *distance* column by a constant factor and appends the result as new column to the dataframe. The function is executed in ScootR via a `map` operator, as detailed in Section 4.2. SparkR (STS) uses source-to-source translation.

Figure 11 (a) depicts the result of the benchmark on a single node. Both SparkR (STS) and ScootR achieve almost the performance of their respective implementations in the native APIs. ScootR is at most 1.15× slower than Flink, while SparkR (STS) is about 1.07× slower respectively. These results are expected for SparkR (STS), as the UDF is translated to the native API. For ScootR, these results validate our expectations that we can achieve comparable performance to the native API even though the R function is executed in a `map` operator. GNU-R is outperformed by fastR (1.5×), and by both SparkR (STS) (up to 15×) and ScootR (up to 25×). Again, this is mainly due to the *streaming* facilities in Spark and Flink. In contrast to the previous benchmark, fastR is able to outperform GNU-R due to the more efficient execution of the `apply` function.



(a) Single node (b) Cluster

Figure 11: Micro-benchmark for the apply function from Listing 1, Line 11.

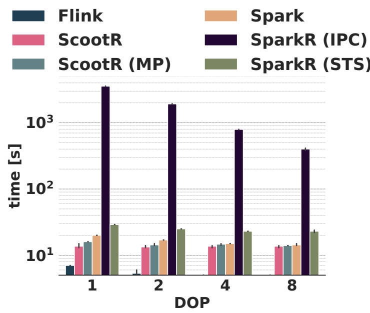


Figure 12: Single node micro-benchmark for the apply method from Listing 1, Line 11 with a 10% sample of the original data. The y-axis is in log scale.

Figure 11 (b) depicts the same experiment, but now we distribute the computation on up to 4 nodes. Again, ScootR (1.1× for a DOP of 32) and SparkR (STS) (around 1.08× for a DOP of 32) introduce only a small overhead compared to their respective native APIs.

To determine the cost of IPC, we implemented the UDF using the `dapply` function of SparkR, which internally executes the UDF in a `mapPartitions` operator. For a fair comparison, we implemented the UDF using the general `apply` in ScootR, shown as *ScootR (MP)*, which internally also uses a `mapPartitions` operator. SparkR (IPC) failed to execute the function within the set experiment timeout of 4 hours for DOPs up to 32. In comparison, we observe that ScootR (MP) is competitive (around 1.1× overhead) to the ← `apply` function, due to direct access to Flink's data structures.

To obtain results for SparkR (IPC), we sampled the airline dataset from 9.5GB down to roughly 100MB. Figure 12 shows the results for single node execution with increasing DOP using the downsampled airline dataset. For single-thread execution, SparkR (IPC) takes ∼50 minutes to complete the task compared to 30 seconds for Spark (STS). Using the 8 available cores, SparkR (IPC) executes in ∼7 minutes. Both versions of ScootR are about 1.8× slower than native Flink, while SparkR (IPC) is about 170× slower than native Spark. This performance overhead is due to the drawbacks of IPC discussed in Section 3.1, namely *serialization* and *data transfer*. In addition, the `dapply` function in SparkR (IPC) uses Spark's

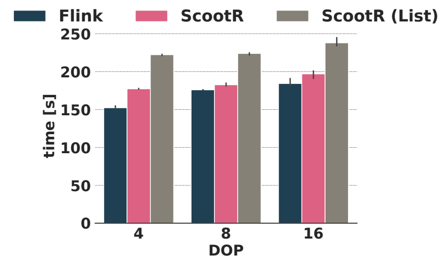


Figure 13: Cluster micro-benchmark for calculating the N-grams in the *body* column of the Reddit comments dataset. The data is scaled according to the number of used nodes.

`mapPartitions` operator to execute the UDF. The operator provides all tuples contained in the partition via an iterator to the UDF. As SparkR cannot access the iterator, all tuples in the iterator have to be materialized and are provided as dataframe to the UDF. This introduces a materialization barrier in the streaming execution and causes additional performance overhead. ScootR (MP) also uses the mapPartitions operator of Flink, but has access to the iterator via the language interoperability features described in Section 4.2 (B). Thus, ScootR does not have to materialize and can directly access the tuples in a streaming fashion via the iterator in the R UDF.

UDF Operator with 1:N Output. The next micro-benchmark executes a more complex UDF, where we generate all *2-grams* within the *body* column of the Reddit comments dataset. Compared to the previous benchmarks, the UDF is compute-heavy and second, the function is called within a `flatmap` operator. As the body has N 2-grams per comment, the function may emit $0, 1, \ldots, N$ elements per input tuple. The ScootR function used in the experiment is detailed in Listing 6 in the Appendix. As described in Section 4.2, ScootR has direct access to the Flink `Collector` class, which collects the output directly in R UDF.

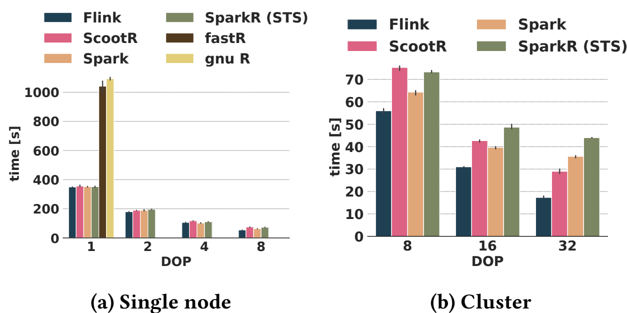
Figure 13 depicts the result for the benchmark. The data size is increased alongside with the number of nodes and we use 1, 2, and 4 months of data. We observe that ScootR is only about a factor of 1.15× slower than Flink. As we can access the collector and create the Flink tuples directly inside the R function, we avoid the materialization and type conversion of the returned result. We report the execution times without access to the collector as ScootR (List) to show the benefit of direct access to Flink's data structures. As discussed in Section 4.2 (B), the necessary additional pass over the List and the type conversion results in 1.2× slower execution compared to ScootR with direct access. SparkR with IPC failed to execute within the set experiment timeout of 4 hours. The UDF cannot be expressed in SparkR with STS translation.

5.3 Results for Operator Pipelines

In this section, we provide benchmarks for operator pipelines. The first benchmark shows a ETL pipeline composed of several operators. It validates the results from the previous micro-benchmarks and shows near native performance for ScootR and SparkR compared to their respective system's native APIs. Both outperform GNU-R and fastR by up to 2.5× for single-threaded and up to 20× for distributed execution. Again, while SparkR uses STS translation,

ScootR achieves this while executing the UDFs. The second benchmark shows a classical MapReduce pipeline. ScootR and SparkR execute in near native performance. The third benchmarks shows a mixed pipeline combining preprocessing and model training. It shows the benefits of the seamless integration of ScootR, as we collect the distributed data for further local processing in the same R script (depicted in Listing 5 in the Appendix). Thereby, we can achieve up to 12× performance improvement compared to executing the complete pipeline in fastR as the majority of the time is spent for preprocessing. In the following paragraphs, we describe each benchmark in detail.

ETL Pipeline. In this experiment, we execute the pipeline described by Oscar D. Lara et al. [32]. The ScootR code for the pipeline is depicted in Listing 3 in the Appendix. The goal of this pipeline is categorizing the delay of flights by two carriers in the airline dataset. The operators used in the pipeline are embarrassingly parallel.



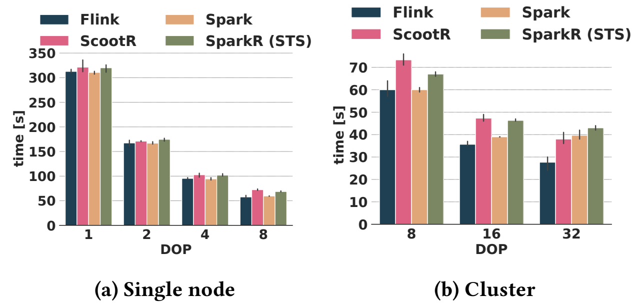
(a) Single node (b) Cluster

Figure 14: Benchmark for the ETL pipeline shown in the Appendix, Listing 3.

Figure 14 depicts the results for the execution on (a) a single node and (b) the four nodes cluster. SparkR (STS) is around 1.2× slower than Spark and ScootR is up to 1.4× slower than Flink in the worst case. Both outperform GNU-R and fastR by 2.5× for single-threaded and up to 20× for distributed execution. GNU-R is only 1.05× slower than fastR. This is mostly due to high selectivity of the `filter` operator at the beginning of the pipeline, which significantly reduces the amount of data. Thus, the data that has to be processed by the two successive UDFs is reduced significantly.

Map-Reduce Pipeline. So far, the benchmarks did not involve aggregations. Therefore, we designed a MapReduce-style aggregation pipeline to determine the largest arrival delay per destination for flights that started from New York. The ScootR code for the pipeline is depicted in Listing 4 in the Appendix.

Figure 15 (a) and (b) depict the results for the benchmark on a single node and the four nodes cluster. ScootR is up to 1.3× slower than Flink and SparkR (STS) up to 1.15× than Spark. While both translate the aggregation function to a native API call, ScootR directly executes the R predicate for the `filter` function. Even though the data size is reduced significantly by the filter operation, the aggregation, due to the necessary shuffle step, together with the initial reading of the data is still responsible for the majority of the execution time.



(a) Single node (b) Cluster

Figure 15: Benchmark for the MapReduce pipeline shown in the Appendix, Listing 4.

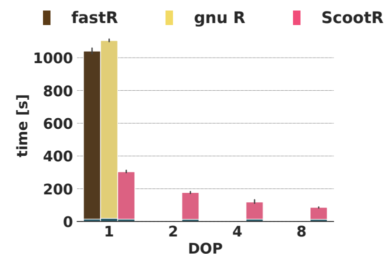


Figure 16: Benchmark for the mixed pipeline shown in the Appendix, Listing 5. The fraction of time spent for the `glm` function is indicated in dark blue.

Mixed Pipeline. In this experiment we evaluate a mixed pipeline. We use the dataflow system to perform the initial data preprocessing before it is gathered on the driver node for further analysis locally as in-memory dataframe. Specifically, we train a generalized linear model with the `glm` function provided in R and show the model description with the `summarize` function. The ScootR code for the pipeline is depicted in Listing 5 in the Appendix.

Figure 16 depicts the results for the described pipeline. We can observe that most of the execution time is spent in the ETL pipeline, which reduces the initial Airline dataset from 9.5GB to approximately 90MB. While all of the systems spend the majority of the time in the preprocessing phase, we can decrease the duration significantly by using ScootR, even in the single-threaded execution case. Compared to GNU-R, ScootR is about 3.6× faster for single-threaded execution, and 12.3× faster when using 8 cores. The performance drawbacks of fastR and GNU-R result from the initial dataset reading and materialization costs.

5.4 Discussion

The main goal of our evaluation was to highlight our two main contributions: (i) The integration of an R language API based on a common runtime to avoid data exchange, while supporting a rich set of language features. (ii) The necessity of our applied optimizations to share data structures between the guest- and the host language to provide fast data access and avoid type conversion.

To this end, we conducted benchmarks comparing ScootR, SparkR, fastR, and GNU-R for both single operators and operator pipelines

for single node and cluster configurations. The non-UDF micro-benchmark functions clearly show that ScootR and SparkR provide reliable mapping of R functions to their respective native API calls, with below 1.2× overhead. For functions calling R UDFs, ScootR can compete with SparkR's source-to-source translation approach, even though ScootR executes the R function in the fastR language runtime. In contrast, when SparkR has to fall back to inter-process communication, its performance degrades by an order of magnitude compared to ScootR. The benchmarks for 1:N operators show that direct access to data structures is necessary to avoid data materialization and therefore achieve comparable performance to the native execution. The benchmarks for operator pipelines, validate the assumptions behind the micro-benchmark experiments, and show very small performance overheads of up to 1.2× for SparkR (STS) and 1.4× for ScootR. Both SparkR and ScootR outperform GNU-R and fastR, even for single-threaded execution on a single node.

6 RELATED WORK

In this Section, we discuss related work on DSL language compilers, parallelization based on existing dataflow systems, and parallelization packages for the R programming language itself.

Compiler-based Approaches. Weld [20] offers a functional intermediate representation based on nested parallel loops and *builders* that specify what should be computed. Libraries and functions represent their operations using this IR to avoid materializing/copying of intermediate results, which is normally required when data is passed from one library to another. Weld applies optimizations such as loop tiling and vectorization, and generates code for diverse processors including CPUs and GPUs. Tupleware [8] is a distributed analytics system that focuses on the efficient execution of UDF-centric workflows. It provides an IR based on the LLVM compiler framework [15], which can be targeted by any language that emits LLVM code. Tupleware applies high-level optimizations, such as predicate pushdown or join reordering, as well as low-level optimizations, such as reordering of the program structure and vectorization. Pydron [18] parallelizes sequential Python code to execute on multi-core, cloud, and cluster infrastructures. It is based on two Python decorators, one to mark functions to parallelize and another one to mark side-effect free functions. Functions annotated for parallelization are translated to an intermediate representation. Pydron applies several optimizations based on this IR, including control flow and scheduling decisions for it's parallelization.

All of the mentioned systems provide a familiar interface to the programmer, while they achieve efficient execution by carefully applied optimizations or parallelized execution. ScootR shares this goal, but incorporates R into an existing dataflow system *without* changing it. It achieves this by relying on the GraalVM, a JVM-compatible language runtime that enables multi-language execution. The approach is not restricted by an IR specially designed for the systems optimization goals. Thus, ScootR profits directly from the ongoing efforts to advance the performance of Graal, and can be easily extended with support for new languages and diverse processors, e.g., GPUs [10].

Parallelism based on Dataflow engines. Hadoop's *Streaming* utility is used as a common basis for IPC in several frameworks. It allows to specify executables and scripts that are called in the map and reduce functions. Here, scripts receive data via *stdin* while results are emitted via *stdout*. RHadoop is a collection of tools to work with the Hadoop ecosystem within R. Likewise, R Revolution, now called Microsoft R Open and Server (commercial version), provides the option to run R on top of Hadoop. All the presented systems inherit the drawbacks of IPC, namely communication overhead, (de)serialization and data-processing pipeline disruption, as discussed in Section 3.1.

RHIPE [12] is also based on Hadoop and uses IPC while exchanging data via Google's ProtocolBuffers, a language- and platform-neutral mechanism for serializing structured data. RHive allows for easy use of HSql, the query language of Hive [25], in R. In addition, UDFs and user-defined aggregate functions (UDFAs) can be specified in R, which are executed via IPC with an external R process. RHIPE has a more efficient data exchange format compared to Hadoop Streaming, but it still inherits the drawbacks of IPC, as the executables run in separated processes.

Ricardo [9] was developed by IBM and executes Jaql [5] queries on top of Hadoop. Beside Jaql functions that do not involve user-defined code, users can specify R UDFs, which are executed in an external R process. Thus, it provides a hybrid approach as discussed in Section 3.3. Ricardo inherits the drawbacks from IPC when executing user-defined functions, but it provides so-called *trading* that allows for mixed R and Hadoop execution. Preprocessing can be executed in Hadoop before the results are fetched in R and can be used as input to the manifold libraries in R. ScootR is influenced by the trading concept, but does not have to fall back to IPC in case of user-defined functions.

Big R [32] is based on IBM BigInsights and uses a restricted, overloaded set of R operations and transformations specified in Jaql that can be executed on top of Hadoop. The results are returned as a dataframe, which is used for further processing in R. In contrast to Big R, ScootR is not restricted to a limited set of operators and executes arbitrary R functions.

SparkR [27] provides a dataframe-centric programming abstraction in R. As described in Section 3, SparkR avoids IPC by applying source-to-source translation for a subset of operations and library calls. In case the source-to-source compiler cannot translate the R program, SparkR falls back to use an external R process using inter-process communication. This fall back causes large performance penalties. ScootR builds upon the ideas of SparkR for non-UDF operators, while improving execution time for arbitrary UDFs.

Spark also provides a programming abstraction for Python, called PySpark. While the underlying concepts are the same as in SparkR, there is an ongoing effort to integrate Apache Arrow [1]. Apache Arrow's goal is to provide a common in-memory data representation that provides efficient access and APIs in Python, C, and Java. Therefore, it improves data exchange between Spark and Python, while also providing more efficient access for the popular Python *pandas* dataframes. While Arrow looks promising, data needs to be serialized to and de-serialized from the binary format of Arrow.

SystemML [6] is a system for the efficient execution of linear algebra programs on Apache Spark written in a DSL based on R's matrix abstraction. While its focus is clearly on linear algebra, it provides basic facilities to transform input data with a restricted set of operations and predefined functions. As SystemML focus is

clearly on linear algebra, it is orthogonal to our goal of providing efficient execution of UDF-centric workflows.

SciDB [7] is an array database that focuses on efficient execution of array manipulation and linear algebra. SciDB provides an R abstraction in addition to its native API. As SystemML, its focus is not UDF support and therefore orthogonal to our goals.

R Parallelization Packages. There are several *explicit* parallelization packages, such as Snow [26] and Snowfall [14], with parallel versions of the common *apply** methods. In addition, there are packages based on parallelized versions of the *foreach* construct [17, 28] for different back-ends such as Socket, MPI, PVM, and NetWorkSpaces. These packages focus on parallelizing computation heavy, splittable tasks, but not on large amounts of data. In consequence, they offer no facilities to read distributed files and reflect the scatter/gather model from MPI. In contrast, ScootR focuses on parallelizing computations on large amounts of data.

7 CONCLUSION

In this paper, we presented ScootR, a novel approach to execute R user-defined functions in dataflow systems with minimal overhead. Existing state-of-the-art systems, such as SparkR, use source-to-source translation to the systems' native API to achieve near native performance for a restricted subset of user-defined functions. When running arbitrary UDFs, their performance may degrade by a factor of up to 170×, as they have to fall back to inter-process communication. This overhead is due to the necessary data serialization and data transfer imposed by IPC. ScootR avoids such overheads by tightly integrating the dataflow engine with the R language runtime, using the Truffle framework and the Graal compiler. By making Flink abstract data types accessible to the R user-defined functions, ScootR avoids type conversion as well as intermediate results duplications and copies. Our experimental study shows that ScootR achieves comparable performance to systems based on source-to-source translation, even though ScootR executes the UDF in an R language runtime. When SparkR has to fall back to inter-process communication, ScootR has up to an order of magnitude higher performance.

Future Work. The techniques and approaches presented so far are general and are applicable to other dataflow systems as well. With possibly few exceptions, most of the existing systems provide a relational-style API based on typed, fixed-length tuples. For instance, one could provide a similar abstraction implemented on top of the *Spark* dataset and/or dataframe abstraction, following the approach outlined in this paper. Another interesting extension would be the integration of other Truffle-based (dynamic) languages such as JavaScript or Python. To this end, a small language agnostic and data-processing centric Truffle API could be defined, which could be used as common abstraction by different language runtimes.

APPENDIX

```r
df <- flink.readdf(...)
df <- flink.filter(df, df$cancelled == 0 &&
  df$dep_delay >= 10 && df$carrier %in% c("AA", "HA"))
df <- flink.select(df,
  carrier, origin, dest, dep_delay, arr_delay)
df$avgDelay <- (df$arr_delay + df$dep_delay) / 2

df$delay <-
  if (df$avgDelay > 30) "High"
  else if (df$avgDelay < 20) "Low"
  else "Medium"

df$head(5)
```

Listing 3: Data transformation pipeline proposed in [32].

```r
df  <- flink.readdf(...)
df  <- flink.filter(df, df$origin == 'JFK')
grp <- flink.groupBy(df, 'dest')
max <- grp$max('arr_delay')
cat(max$head(5))
```

Listing 4: Calculating the maximal arrival delay per destination for flights starting from New York.

```r
df <- flink.readdf(...)
df <- flink.filter(df, df$cancelled == 0 &&
  df$dep_delay >= 10 && df$carrier %in% c("AA", "HA"))
df <- flink.select(df,
  carrier, dep_delay, arr_delay, distance)
fastR_df <- flink.collect(df)

model <- glm(
  arr_delay ~ dep_delay + distance,
  data = fastR_df,
  family = "gaussian")

summary(model)
```

Listing 5: A mixed pipeline. The ETL part is executed on Flink, before the data is collected on the driver to train a generalized linear model in fastR.

```r
df <- flink.readdf(...)
ngrams <- function(tpl, collector) {
    splits    <- strsplit(tpl$body, " ")[[1]]
    numSplits <- length(splits)

    srtIdx <- 1
    endIdx <- 2
    while (endIdx <= numSplits) {
        twoGram <- paste(splits[srtIdx:endIdx],
                         collapse = " ")
        srtIdx <- srtIdx + 1
        endIdx <- endIdx + 1
        collector$collect(flink.tuple(twoGram, 1))
    }
}

df <- flink.apply(df, ngrams)
flink.writedf(df, outFile)
flink.execute()
```

Listing 6: Calculating the 2-grams of the *body* column in the Reddit comments dataset.

REFERENCES

[1] 2017. Apache Arrow. (2017). https://arrow.apache.org/ Accessed: 2018-8-27.

[2] 2017. IEEE Spectrum, The 2017 Top Programming Languages. (2017). https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages Accessed: 2017-10-23.

[3] Alexander Alexandrov et al. 2014. The stratosphere platform for big data analytics. *The VLDB Journal—The International Journal on Very Large Data Bases* 23, 6 (2014), 939–964.

[4] Alexander Alexandrov et al. 2015. Implicit parallelism through deep language embedding. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 47–61.

[5] K Beyer, Vuk Ercegovac, Jun Rao, and Eugene J Shekita. 2011. JAQL: Query Language for JavaScript (r) Object Notation (JSON). (2011).

[6] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *VLDB* 9, 13 (2016), 1425–1436.

[7] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*. ACM, 963–968.

[8] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An architecture for compiling udf-centric workflows. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1466–1477.

[9] Sudipto Das, Yannis Sismanis, Kevin S Beyer, Rainer Gemulla, Peter J Haas, and John McPherson. 2010. Ricardo: integrating R and Hadoop. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 987–998.

[10] Juan José Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation.. In *VEE*, Vol. 17. 60–73.

[11] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 78–90.

[12] Saptarshi Guha. 2010. Computing environment for the statistical analysis of large and complex data. (2010).

[13] Saptarshi Guha, Ryan Hafen, Jeremiah Rounds, Jin Xia, Jianfu Li, Bowei Xi, and William S Cleveland. 2012. Large complex data: divide and recombine (d&r) with rhipe. *Stat* 1, 1 (2012), 53–67.

[14] Jochen Knaus. 2015. *snowfall: Easier cluster computing (based on snow).* https://CRAN.R-project.org/package=snowfall R package version 1.84-6.1.

[15] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[16] Yinan Li et al. 2017. Mison: A Fast JSON Parser for Data Analytics. *PVLDB* 10, 10 (2017).

[17] Microsoft Corporation and Stephen Weston. 2017. *doSNOW: Foreach Parallel Adaptor for the 'snow' Package.* https://CRAN.R-project.org/package=doSNOW R package version 1.0.15.

[18] Stefan C Müller, Gustavo Alonso, Adam Amara, and André Csillaghy. 2014. Pydron: Semi-Automatic Parallelization for Multi-Core and the Cloud.. In *OSDI*. 645–659.

[19] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The java hotspot TM server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association, 1–1.

[20] Shoumik Palkar et al. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[21] R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org

[22] Konstantin Shvachko et al. 2010. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 1–10.

[23] David Smith. 2017. R, Then and Now. (2017). useR!, Brussels, 2017.

[24] Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. 2016. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages*. ACM, 84–95.

[25] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.

[26] Luke Tierney, A. J. Rossini, Na Li, and H. Sevcikova. 2016. *snow: Simple Network of Workstations*. https://CRAN.R-project.org/package=snow R package version 0.4-2.

[27] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, et al. 2016. Sparkr: Scaling r programs with spark. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1099–1104.

[28] Stephen Weston. 2017. *doMPI: Foreach Parallel Adaptor for the 'Rmpi' Package.* https://CRAN.R-project.org/package=doMPI R package version 0.2.2.

[29] Hadley Wickham et al. 2017. A Grammer of Data Manipulation. (2017). https://cran.r-project.org/web/packages/dplyr/dplyr.pdf CRAN.

[30] Thomas Würthinger et al. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. ACM, 187–204.

[31] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST interpreters. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 73–82.

[32] Oscar D Lara Yejas, Weiqiang Zhuang, and Adarsh Pannu. 2014. Big R: large-scale analytics on Hadoop using R. In *Big Data (BigData Congress), 2014 IEEE International Congress on*. IEEE, 570–577.

[33] Matei Zaharia et al. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10, 10-10 (2010), 95.

[34] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization optimizations for feature selection workloads. *TODS* 41, 1 (2016), 2.