Lightweight Multi-language Bindings for Apache Spark

Luca Salucci[1(✉)], Daniele Bonetta[2], and Walter Binder[1]

[1] Faculty of Informatics, Università della Svizzera italiana (USI), Lugano, Switzerland
{Luca.Salucci,Walter.Binder}@usi.ch
[2] Oracle Labs, VM Research Group, Lugano, Switzerland
daniele.bonetta@oracle.com

Abstract. Apache Spark has emerged as one of the most prominent frameworks for distributed high-performance data analysis. Among Spark's most appealing features are its bindings for dynamic languages such as Python and R. Despite of the great flexibility of such languages, they often cannot match the performance of statically typed languages such as Java or Scala. However, this limitation is not only due to the intrinsic nature of dynamically typed languages. Largely, the performance gap is caused by the way the language runtimes interact with Spark. In this paper we describe a new approach to integrating Python and R into data-intensive Spark applications. Our approach significantly reduces the performance gap between such languages and their statically typed counterpart, making dynamic languages an attractive alternative for the implementation of big-data applications.

1 Introduction

In the context of big-data frameworks [22], Apache Spark [3] has emerged as one of the most popular solutions for writing complex data-analytics applications. The reasons for this success are manifold, spanning from Spark's convenient high-level programming model, to its rich plugins ecosystem and its very active developer community. Arguably, one of the key aspects of such success is the extensive support for several different analytics techniques and domains: started as a research project targeting data-intensive applications in a distributed cluster [29], Spark has rapidly evolved and is now supporting also other, more complex application domains such as graph analytics [27], stream-based computations [30], and machine learning [24]. Spark is entirely written in Scala and runs on the Java Virtual Machine (JVM). Hence, Spark also considers Java as a first class citizen, and allows developers to write applications in Java, too. Scala and Java, however, are not popular languages in many of the scientific communities targeted by Spark. For this reason, Spark also offers support for other languages, providing native built-in support for Python (i.e., PySpark [12]), and recently also for R (i.e., SparkR [13]). Both languages are very popular among data scientists, as they provide extensive support and abstractions for specifying

P.-F. Dutot and D. Trystram (Eds.): Euro-Par 2016, LNCS 9833, pp. 281–292, 2016.
DOI: 10.1007/978-3-319-43659-3_21

complex data analyses. Python and R are often considered *complemental* to each other, as they offer extensive support for closely related domains, namely, statistical computing (using R) and numerical computing (using Python with popular libraries such as NumPy [10]). Hence, both languages are often *combined* together to develop complex analyses. The support of high-level, dynamically typed languages (such as Python or R) in Spark, however, comes at the expense of performance. The reasons for this performance difference are twofold. Firstly, dynamically typed languages are inherently slower than statically typed ones. Despite of the progress in dynamic compilation techniques [18,26], languages such as Python still cannot match the performance of Java due to the intrinsic nature of their semantics, which requires the language runtime to perform additional checks for common operations such as function calls and property lookups. Secondly, the runtimes of such languages are not executed within the JVM process of the Spark runtime. If a dynamically typed language is used to express an analysis, several independent language runtimes need to be executed in separate processes, imposing significant additional runtime overheads. In particular, the language runtime driving the analysis (i.e., the JVM) has to exchange data between processes executing another language runtime (e.g., the Python engine). Exchanging data between processes is expensive, as it introduces marshalling, unmarshalling, and communication overheads, and because it prevents the just-in-time (JIT) compiler from optimizing the framework code and the data-analysis code together. Moreover, the overhead imposed by mixing multiple language runtimes grows significantly when developers need to use *more than one* dynamically typed language in their analyses. For example, since Spark does not provide any support for integrating Python with R within the same Spark application, developers need to come up with custom, ad-hoc integration solutions. Typically, such solutions are based on very inefficient integration techniques such as file-based data exchange (using HDFS [9]) between language runtimes. The need to implement such custom integration solutions also imposes a significant loss of developer productivity and may severely limit the benefits that could be gained by combining multiple languages in a data analysis.

In this paper we show how the overheads of integrating multiple dynamic language runtimes with Spark can be drastically reduced by hosting them within the same JVM process. In this way, both the dynamic language runtimes (e.g., Python, R) and the Spark runtime execute in the same JVM, taking advantage from the shared memory of the underlying multicore machine. Our approach is based on a modified version of Spark called TruffleSpark, which supports any language implemented with the Truffle [25] framework; in particular, in this paper we are using the Truffle implementations of Python (ZipPy [14]) and R (FastR [5]). Our approach not only reduces the performance gap between dynamically typed and statically typed languages in Spark, but also allows for an efficient integration of *multiple* dynamically typed language runtimes in the *same* JVM process, enabling an efficient execution of multi-language Spark analyses. Thus, we offer Spark programmers the possibility to directly combine functions written in different languages within the same Spark application without sacrificing per-

formance. This paper makes the following contributions to the state-of-the-art in the field of data analytics:

* We present TruffleSpark and show how Truffle-based languages can be efficiently integrated into Spark. TruffleSpark is based on an earlier research prototype discussed in [15], which we extend to support multiple languages in the same Spark application.
* Thanks to TruffleSpark, we enable multi-language support for Spark, allowing the implementation of data analyses in multiple (statically and dynamically typed) languages. TruffleSpark operates on spark's RDD data types, enabling low-overhead access to RDDs from dynamic languages.
* We evaluate TruffleSpark comparing it against the original Spark framework. On the considered benchmarks TruffleSpark always outperforms the Spark bindings for Python (i.e., PySpark), and has performance close to the ones of equivalent analyses written in a statically typed language (i.e., Scala).

2 Background

Our approach is based on a modified version of the Spark runtime that supports the execution of all the languages developed using the Truffle framework. In this section we provide an overview of Spark and Truffle.

2.1 Apache Spark

Apache Spark [3] is a general-purpose framework for large-scale data processing running on the JVM. One of the main programming abstractions in Spark are Resilient Distributed Datasets (RDDs), a fault-tolerant collection of homogeneous data which can be operated on in parallel. Informally, Spark applications based on RDDs consist of subsequent modifications to such data structures. Modifications can operate either on existing RDDs (e.g., by applying an operator to all elements of an RDD), or on newly created ones. Spark supports the creation of RDDs using multiple data formats, e.g., from a file in an external storage system such as a shared filesystem, HDFS, HBase, or any data source supporting the Hadoop input format. Applications that rely on the RDD abstraction can be developed using both statically typed languages (e.g., Scala) and dynamically typed ones. In the rest of this paper, we will focus on applications using RDDs from dynamic languages such as Python.

At runtime, Spark executes as a set of JVM processes deployed on a cluster in a master-slave fashion: one *driver program* orchestrates the computation from the cluster's master node, while other *worker* processes are responsible for performing the actual distributed computation (i.e., one per cluster node). Upon each Spark's job submission, every worker spawns an *executor* process, which executes the tasks submitted to by the master using a thread pool. Once the executor is started, it communicates directly with the driver program, so that worker processes are not effectively involved in the computation. Spark can

be deployed using a distributed and a standalone mode. While in the distributed deployment the driver, workers, and executors are running in distinct JVM processes, in the local standalone deployment they run as threads within a single JVM process. In both configurations, there is always one JVM per cluster node that is responsible for executing all the analyses that have been submitted. Therefore – when running with statically typed languages –, the interactions between the parallel threads running an analysis can happen very efficiently within the same JVM process. This is not the case with dynamically typed languages, as we will discuss in the next section.

2.2 Spark Bindings for Dynamically Typed Languages

Spark features language bindings for Python and R, namely PySpark [12] and SparkR [13]. Both languages are supported via an ad hoc runtime (internal to Spark, but different than the one used when executing Scala or Java applications) which has the role of orchestrating the interaction between the Spark runtime and the external language runtimes. Such an ad hoc runtime mimics the original JVM-based runtime, adopting the same driver-worker architecture, with the notable difference that executors do not use a thread-pool to accomplish their tasks, but instead, additionally spawn external Python processes. As a consequence of this runtime architecture, a Python-based analysis in Spark may require the execution of hundreds of independent processes, each one running a dedicated Python runtime, communicating via inter-process communication channels (e.g., using OS sockets). Such runtime architecture can sometimes be inefficient, as the serialization, transmission and reconstruction of objects that take place in the current system are unnecessary and avoidable, as the entities involved run on the same shared-memory machine. Although such socket-based communications may be optimized by the OS and be highly-efficient, they still incur a cost which is considerably higher than the one of simply sharing an object reference among threads sharing a common memory space. On the contrary, an equivalent analysis using Java or Scala involves in the computation only a single JVM process per cluster node, and can benefit from efficient in-memory communication between JVM threads. More precisely, in the case of the standalone deployment, ZipPy/Spark uses a single JVM process, while PySpark uses one JVM process, plus N Python processes (where N is the number of available cores). In the distributed deployment case, ZipPy/Spark uses one JVM process per cluster node, for a total of M JVM processes (with M being the number of nodes in the cluster). PySpark running in a cluster still requires to spawn one JVM per cluster node, and additionally requires N Python processes per node (resulting in M JVM processes plus $M * N$ Python processes deployed across the cluster). The Spark terminology lacks a term to refer to the threads in the executor's pool and to the Python (respectively, R) processes spawned by PySpark (respectively, SparkR). These two entities fulfill the same role in their runtimes, and we will call them *task runners*. Task runners are threads in the Spark runtime, whereas they are Python (respectively, R) processes in the PySpark (respectively, SparkR) runtime.

2.3 GraalVM and Truffle

Our modified version of Spark relies on Truffle and on the GraalVM runtime. Graal [7] is a state-of-the-art JIT compiler for the JVM, written in Java and focused on performance and language interoperability. Graal implements several optimizations such as method inlining, eliding object allocations, and speculative execution, and can apply them to dynamically typed languages developed with the Truffle [25] language development framework. The GraalVM comes with support for multiple Truffle-enabled languages:

* Graal.js: a JavaScript runtime supporting Node.js applications
* FastR: a fast JIT compiler-enabled R language runtime
* RubyTruffle: a Ruby language runtime
* ZypPy: a Python language runtime

Truffle is a language implementation framework that relies on the notion of self-optimizing Abstract Syntax Tree (AST) interpreters [25]. A Truffle language is defined in terms of AST nodes corresponding to the constructs available in the language. A notable characteristic of such nodes is that they use the information gathered during their execution to specialize (i.e., to rewrite themselves) for the types observed at runtime. In this way, when running with GraalVM, a Truffle AST can be compiled to efficient machine code via partial evaluation [25].

3 TruffleSpark

TruffleSpark is our modified version of Spark that supports Truffle-based languages. In this section we provide an overview of its architecture and of its main components, that is, guest function wrappers and the mechanism for data-type conversions between different language runtimes. TruffleSpark enables the lightweight embedding of dynamically typed guest languages (e.g., Python) in applications developed using the Spark standard API (based on Scala or Java). TruffleSpark extends Spark with the following two main capabilities:

* **New Python and R RDD Bindings**. The two dynamic languages are integrated in the Spark runtime at the level of the JVM, that is, without requiring the integration of external language runtimes. As a consequence, TruffleSpark does not need to execute Python or R code in independent processes, and can execute the entire analysis in the same JVM process. This brings notable advantages (in terms of performance), as the overhead due to inter-process communication is substantially reduced. Moreover, this enables the JVM to perform optimizations (e.g., JIT compilation) of Python code together with the Spark runtime code. Finally, this approach has notable advantages in terms of resource utilization, as Java threads are used instead of more heavy-weight processes.
* **Language Interoperability**. Since the Truffle framework and the GraalVM support several languages, our integration allows one to combine multiple

of them in Spark in the same application, allowing developers to share the infrastructure code and classes necessary to execute analyses that combine multiple language runtimes.

The embedding of guest language functions in the runtime is achieved by means of *function wrappers* (called also *wrappers* in the shorthand form); these wrappers provide the functionality for parsing and compiling the Python or R functions (by interacting with the GraalVM), as well as for invoking them once parsed. Since wrappers are Java objects, they can be optimized together with the Spark runtime and the function they wrap by the JVM's JIT compiler. Being able to execute Python or R code is a necessary but not sufficient condition for integrating foreign languages in Spark. As the dynamically typed language runtime is hosted on the same JVM where Spark runs, it is possible to directly exchange object references between it and the Spark runtime. Exchanged objects, however, have different data type representations and APIs (e.g., when accessed from Python or Java). To support operating on data types belonging to different language runtimes, we provide *object adapters* (called also *adapters*) for each of such objects. In this way, data structures that offer the API enforced by the host language (i.e., Scala or Java) are backed by the original guest object (e.g., in Python). The same approach is used also for objects that are passed from Spark to the dynamic language runtime, with different adapters enclosing a Java object while offering the API expected from, e.g., a Python object. This approach avoids deep copying of the objects exchanged between language runtimes, and greatly reduces overhead.

An example TruffleSpark application combining multiple Truffle languages is depicted in Fig. 1. The application corresponds to a typical word-count benchmark, expressed by combining Java, Python, and R. In the example, each language is used to express a different part of the computation. Java is used to specify how the RDDs are created, and what operations have to be applied to them (e.g., `flatMap`). Python and R are used to express the functional part of the analysis, that is, to split the input and to count words. Like with normal Spark applications, it is the responsibility of the programmer to choose the proper interface for the RDD data types involved in the analysis (e.g., `JavaPairRDD` of strings). Different than plain Spark, expressions in such languages can be directly embedded in the analysis code. To this end, we provide bindings for Truffle-based languages using a function wrapper (e.g., `PyFunction` in Fig. 1). Such wrappers enable the execution of the guest language (e.g., Python) in the same JVM process running Spark, as they implement the methods used internally to construct, serialize, restore, and invoke the guest language function.

3.1 TruffleSpark Implementation

The TruffleSpark framework builds upon the Sparks' Java RDD API, which has been extended to support dynamically typed languages. The two main elements that compose our TruffleSpark framework are guest language wrappers and the object adapters used to perform data-type conversion between the *guest* languages (i.e., Python and R), and the *host* language (i.e., Java).

```
1 // A file is read from the disk and stored into an RDD
2 JavaRDD<String> lines = ctx.textFile("file:///wordcount/big-input.txt");
3 // The first part of the computation is expressed in Python
4 JavaRDD<String> words = lines.flatMap(new PyFlatMapFunction<String, String>(
5         "def splitLine(l) : return l.split();");
6 // The second part of the computation is expressed in R
7 JavaPairRDD<String, Integer> ones = words.mapToPair(
8     new RPairFunction<String, String, Integer>(
9         "tup <- function(x) { return(list(x, as.integer(1))) }"));
10 // Python is used again to accumulate the final result
11 JavaPairRDD<String, Integer> counts = ones.reduceByKey(
12     new PyFunction<Integer, Integer, Integer>(
13         "def reduce(c1,c2) : return c1 + c2;"));
```

Fig. 1. An example word-count computation expressed in TruffleSpark, combining Python, R, and Java code.

Guest Language Function Wrappers. The TruffleSpark runtime is composed of multiple AST interpreters and a single JIT compiler (i.e., Graal [7]), shared among all language runtimes. Each thread in the Spark runtime holds a thread-local instance of a dynamically typed language parser, to avoid data races at the language runtime level. During parsing, an AST for the code provided as input is produced. The AST is implemented as a Java class, and can be used directly from Java by invoking its methods and the API it provides. As described in Sect. 2, when the type for the nodes in the AST stabilizes, the compilation of the AST to machine code is triggered by the Graal compiler, which is shared between threads. The guest language functions are hosted and *embedded* in Java by means of function wrappers that provide the runtime support in order to execute the computation expressed in a dynamically typed language. By embedding we mean that at runtime they resemble standard Java methods, and are therefore optimized by the JVM together with the Spark runtime. Wrappers also implement the parsing and interpretation of guest language functions. This is achieved by interacting with the GraalVM parser and compiler in the background.

Guest-to-host Datatype Conversions. Enabling the execution of guest language code is not sufficient for completing the integration of the different runtimes. The integration of guest languages in TruffleSpark does not require the expensive serialization and reconstruction of objects: objects can be passed from one language to the other freely, since they operate in the same address space. This helps avoiding the transmissions of objects over OS pipes, or other means of IPC, among processes running on the same shared-memory machine. Despite being available and accessible by the other language runtime, such objects cannot be used as they are out of their native context. It is necessary to make possible for the *foreign* language (e.g., Python) to operate on objects generated by another runtime (e.g., R). This has been achieved with a solution that involves object adapters for the foreign language objects, which can consequently be used in the host runtime as if it was a native object. This solution avoids deep copying of the object to its closest equivalent in the target language. In this way, the adapters offer the expected API to the hosting runtime while being backed by

the original object. Thus, with respect to the original system which involves the serialization of the data, we only pay for the allocation of an adapter.

Function Wrappers and Guest Object Serializability. In order to be executable by the Spark runtime, wrappers need to be serializable, as it is necessary, together with the input data, to transmit the code to the Spark workers. To meet this requirement we made the AST stored in a function wrapper `transient`, according to the Java terminology (which causes the field and the object contained in it, to be omitted in the serialization process). We transmit only the code of the function over the network. Once restored on the remote node, the remote instance of the GraalVM will translate the code into its AST equivalent and store it back in the newly constructed wrapper, thus restoring the original object. Moreover in Spark, whenever a type is returned as the output of an RDD transformation and has to be stored in the dataset, it is required to be serializable. As in the case of the guest functions, the guest objects passed among the different stages of the computation, could eventually be transmitted over the network if the framework runs in a distributed deployment. To this end, we modified some of the guest language types implementations (e.g., R tuples and lists or vectors, which are frequently used as intermediate objects between stages of the computation), to make them serializable.

4 Evaluation

We performed an initial experimental evaluation of TruffleSpark with the main goal of highlighting the performance of dynamic languages executed using our approach. To this end, we compared our TruffleSpark framework against Spark on common applications that use dynamic languages.

We first focused our evaluation on the Python bindings for Spark, comparing the performance of some well-known Spark benchmarks (included in the Spark distribution) against ZipPy/Spark. The performance of TruffleSpark in this setup is expected to outperform Spark's native Python bindings (i.e., PySpark). We then focused on a single popular benchmark for data-intensive computations (i.e., WordCount), and executed it with different configurations. Specifically, we have re-implemented the benchmark using R, and using a combination of R and Python. In this way, we can highlight the benefits of combining multiple languages in a data-intensive benchmark. We use Spark 1.5.1, PySpark with Python 2.7.3, we build ZipPy using Java 8 and execute on GraalVM 0.9 (the release provided on the OTN website, which includes FastR). All experiments are executed on a server-class machine running Ubuntu 14.04 LTS equipped with two 8-core Intel Xeon CPUs E5-2680 (2.70 GHz) with 2 MB of L2 cache, 20 MB of L3 cache, and 128 GB of RAM. Hyper-threading is enabled on the cores, Intel Turbo Boost Technology is disabled, and the CPU driver is Acpi-cpufreq.

Python Performance. In evaluating the performance of Python, we considered four popular Spark benchmarks: WordCount, a program that counts the distribution of words in a text file; Grep, an analysis that scans a file searching for

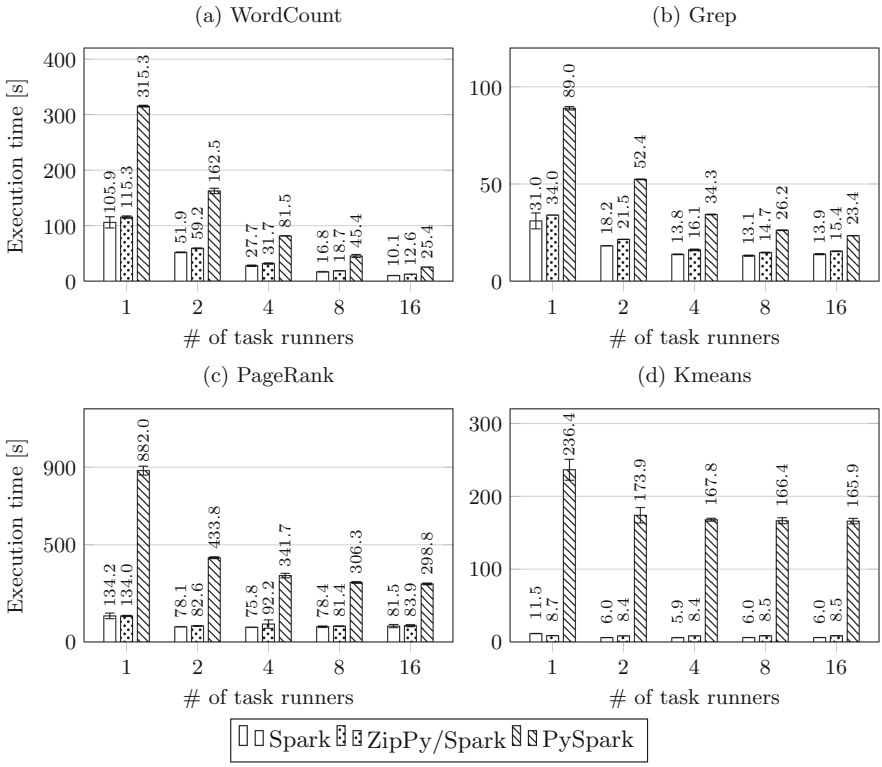

Fig. 2. Python performance. (a) WordCount and (b) Grep use an input file of size 2.7 G; (c) PageRank uses an input size of 42 M (d) KMeans uses a file size of 400 K ($k = 10$)

given patterns; PageRank, a famous graph algorithm [23]; and KMeans, a popular vector quantization method [20]. The performance of TruffleSpark is depicted in Fig. 2, where the average execution time for each benchmark is reported for an increasing number of parallel task runners. As the figure clearly shows, TruffleSpark always outperforms the equivalent implementation in PySpark. Moreover, the performance of Python running with TruffleSpark (i.e., ZipPy/Spark) is close to the same analysis written in Scala. This result suggests that our approach makes Python an efficient alternative to Scala.

Combining R and Python. The R language has performance that often cannot match Python. Still, the language offers very convenient data-analysis functionalities, making it a good choice for certain applications. With the goal of showing the potential of a multi-language solution, we have adapted one of the previous benchmarks (i.e., WordCount) to use R for parts of the computation. The performance for this experiment is depicted in Fig. 3, where the benchmark is executed with different input files. As the picture shows, the performance of

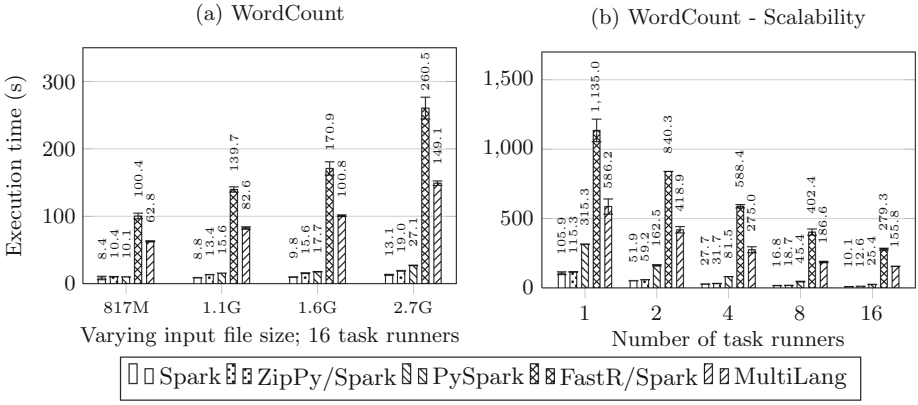

Fig. 3. WordCount performance using multiple languages. In (a) the number of task runnners is fixed at 16. In (b) the input file size is 2.7 G.

R cannot match pure Scala or Python. Nevertheless, it is possible to express the most CPU-demanding part of the computation in another language (Multilang, in the figure). By using Python, the performance of the data analysis improves significantly. As the figure suggests, by combining multiple languages it is possible to mitigate the performance impact of certain (slow) computations by selectively replacing them with faster implementations. Combining languages in this way is not possible in the current Spark framework, where users who want to use R for their analysis must accept its high runtime overhead.

5 Related Work

Other frameworks exist that provide functionalities similar to Spark. Two notable examples are Apache Flink [16], and Google Cloud Dataflow [6]. Both frameworks offer bindings for multiple languages: Flink, for example, enables the development of analyses in Scala and Java, and supports the integration of other languages via inter-process communication. Another example of multi-language integration is Apache Hadoop [1], a well-known open-source implementation of MapReduce [19]. Hadoop's standard way to express computation is to provide map and reduce operations implemented in Java. To enable multi-language integration, Hadoop also enables to execute map and reduce operations as external processes via Hadoop Streaming [8], allowing programmers to express map/reduce jobs using potentially any programming language with basic input-output capabilities. In such configuration, the map and reduce functions are executed by language runtimes that read the input from the standard input and produce results by writing it to the standard output. Another relevant approach is the one of Apache Pig [2], a high-level platform for creating MapReduce programs on top of Hadoop that enables guest language integration through IPC. Multi-language integration is enabled through a language called Pig Latin, which abstracts the

Java map/reduce idiom into a form similar to SQL. Users can extend Pig Latin by writing user defined functions (UDFs in the shorthand form), using Java, Python, Ruby, or other scripting languages, and then call them directly from Pig Latin. Support for such UDFs requires a light-weight serialization/deserialization layer with bindings in the supported languages. Unlike all such frameworks, SparkSQL [17] exemplifies an alternative approach for achieving external language integration, which overcomes IPC and serialization. Built on the previous experience with Shark [28], Spark SQL is the state-of-the-art API for querying structured data in Spark. SparkSQL features a highly extensible optimizer (i.e., Catalyst [4]) and offers much tighter integration between relational and procedural processing. Thanks to Catalyst, Spark SQL uses a code generation approach that involves the translation of SQL queries into equivalent Java bytecode, using a domain-specific language (DSL) similar to R data frames [21] and Python Pandas [11]. SparkSQL corresponds to a relevant improvement towards the integration of foreign languages in Spark. Unlike our approach, it is focused at structured data types (as opposed to RDDs), and supports only a subset of the Python language. Unlike SparkSQL, TruffleSpark supports RDDs and aims at supporting the entire Python language.

6 Conclusion

In this paper we have introduced TruffleSpark, a modified version of the Spark runtime that supports the execution of Truffle-based languages. Thanks to TruffleSpark, it is possible to develop data analyses in Spark that make extensive use of dynamically typed languages, with performance comparable to the ones of statically typed languages. TruffleSpark is based on the tight integration of Truffle-based languages with the Spark runtime, and enables the execution of data analyses that combine more than a single language. Our performance evaluation suggests that dynamically typed languages such as Python or R are a valid alternative to Java or Scala for developing Spark applications.

Acknowledgments. Our research has been supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200021 153560). We thank the VM Research Group at Oracle for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

1. The Apache Hadoop distributed system. http://hadoop.apache.org
2. Apache Pig, high-level platform for MapReduce. https://pig.apache.org/
3. The Apache Spark engine. https://spark.apache.org
4. Catalyst: A Query Optimization Framework for Spark and Shark. https://github.com/apache/spark/tree/master/sql/catalyst
5. FastR, an high performance R runtime. https://bitbucket.org/allr/fastr/overview

6. Google Cloud Dataflow. http://cloud.google.com/dataflow
7. The Graal project. http://openjdk.java.net/projects/graal/
8. Hadoop Streaming. https://hadoop.apache.org/docs/r1.2.1/streaming.html
9. HDFS distributed file system. https://hadoop.apache.org/docs/r1.2.1
10. NumPy, scientific computing with Python. http://www.numpy.org/
11. Pandas, Python Data Analysis Library. http://pandas.pydata.org/
12. PySpark. https://cwiki.apache.org/confluence/display/SPARK
13. Spark on R. https://spark.apache.org/docs/1.6.0/sparkr.html
14. ZipPy, a fast and lightweight Python implementation. https://bitbucket.org/ssllab/zippy
15. Efficient Embedding of Dynamic Languages in Big-data Analytics. In: Proceedings of the 36th International Conference on Distributed Computing Systems Workshops. DCPerf 2016, IEEE (2016)
16. Alexandrov, A., Kunft, A., Katsifodimos, A., Schüler, F., Thamsen, L., Kao, O., Herb, T., Markl, V.: Implicit parallelism through deep language embedding. In: Proceedings of SIGMOD, pp. 47–61 (2015)
17. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., et al.: Spark SQL: relational data processing in spark. In: Proceedings of SIGMOD 2015, pp. 1383–1394. ACM (2015)
18. Bolz, C.F., Cuni, A., Fijalkowski, M., Rigo, A.: Tracing the Meta-level: PyPy's tracing JIT compiler. In: Proceedings of ICOOLPS, pp. 18–25 (2009)
19. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
20. Hartigan, J.A.: Clustering Algorithms. Wiley, New York (1975)
21. Ihaka, R., Gentleman, R.: R: a language for data analysis and graphics. J. Comput. Graph. Stat. **5**(3), 299–314 (1996)
22. Nothaft, F.A., Massie, M., Danford, T., Zhang, Z., Laserson, U., Yeksigian, C., Kottalam, J., Ahuja, A., Hammerbacher, J., Linderman, M., Franklin, M.J., Joseph, A.D., Patterson, D.A.: Rethinking data-intensive science using scalable analytics systems. In: Proceedings of SIGMOD 2015, pp. 631–646 (2015)
23. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Technical report 1999–66, November 1999
24. Shanahan, J.G., Dai, L.: Large scale distributed data science using apache spark. In: Proceedings of KDD, pp. 2323–2324 (2015)
25. 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: Proceedings of Onward! 2013, pp. 187–204. ACM (2013)
26. Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C.: Self-optimizing AST interpreters. SIGPLAN Not. **48**(2), 73–82 (2012)
27. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: GraphX: a resilient distributed graph system on spark. In: Proceedings of GRADES, pp. 2:1–2:6 (2013)
28. Xin, R.S., Rosen, J., Zaharia, M., Franklin, M.J., Shenker, S., Stoica, I.: Shark: SQL and rich analytics at scale. In: Proceedings of SIGMOD 2013, pp. 13–24. ACM (2013)
29. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of NSDI 2012, p. 2 (2012)
30. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of SOSP, pp. 423–438 (2013)