Processing Java UDFs in a C++ environment

Viktor Rosenfeld*
Technische Universität Berlin
viktor.rosenfeld@tu-berlin.de

Rene Mueller†
Bern University of Applied Sciences
rene.mueller@bfh.ch

Pınar Tözün
IBM Research - Almaden
ptozun@us.ibm.com

Fatma Özcan
IBM Research - Almaden
fozcan@us.ibm.com

ABSTRACT

Many popular big data analytics systems today make liberal use of user-defined functions (UDFs) in their programming interface and are written in languages based on the Java Virtual Machine (JVM). This combination creates a barrier when we want to integrate processing engines written in a language that compiles down to machine code with a JVM-based big data analytics ecosystem.

In this paper, we investigate efficient ways of executing UDFs written in Java inside a data processing engine written in C++. While it is possible to call Java code from machine code via the Java Native Interface (JNI), a naive implementation that applies the UDF one row at a time incurs a significant overhead, up to an order of magnitude.

Instead, we can significantly reduce the costs of JNI calls and data copies between Java and machine code, if we execute UDFs on batches of rows, and reuse input/output buffers when possible. Our evaluation of these techniques using different scalar UDFs, in a prototype system that combines Spark and a columnar data processing engine written in C++, shows that such a combination does not slow down the execution of SparkSQL queries containing such UDFs. In fact, we find that the execution of Java UDFs inside an embedded JVM in our C++ engine is 1.12× to 1.53× faster than executing in Spark alone. Our analysis also shows that compiling Java UDFs directly into machine code is not always beneficial over strided execution in the JVM.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**; **Query operators**; **Database performance evaluation**; *Online analytical processing engines*;

KEYWORDS

Data management systems, User-defined functions

*Part of this work was done when author was employed at IBM Research - Almaden.
†Work done while author was at IBM Research - Almaden.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3132022

ACM Reference Format:
Viktor Rosenfeld, Rene Mueller, Pınar Tözün, and Fatma Özcan. 2017. Processing Java UDFs in a C++ environment. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3132022

1 INTRODUCTION

A key feature of popular data analytics frameworks of the last decade, e.g., Hadoop [8], Spark [10], and Flink [7], is an execution model centered on *user-defined functions* (UDFs). UDFs are used as arguments to second-order functions such as *map* and *reduce* (e.g., in Hadoop) or within declarative queries (e.g., in SparkSQL [1]). By supporting arbitrary code inside UDFs, these systems achieve a high degree of expressiveness and versatility.

Many popular data analytics frameworks are written in Java or in another language that uses the JVM, e.g., Scala in the case of Spark. The reasons for this state of affairs are partly historical, but Java also offers clear advantages, e.g., the rich ecosystem of languages and tools that are built on top of the JVM, as well as an abundance of developers trained in Java. However, the reliance on the JVM restricts the interoperability of these frameworks with other components of the ecosystem that do not use the JVM.

In this paper, we study the problem of executing scalar UDFs written in Java inside a data processing engine written in a language that is compiled down to machine code, such as C++. In particular, we describe how to extend a data processing engine written in C++ to support arbitrary scalar UDFs encountered in SparkSQL queries. We refer to code that is compiled down to machine code and is directly executed by the CPU without an intervening interpreter or JIT compiler as *native code*.

The major challenge toward the interoperability between JVM-based languages and native code is that there is no straightforward way to link Java bytecode with native code. That is because the JVM abstracts from its internal representation of Java objects and how Java bytecode is executed by the JVM. The Java Native Interface (JNI) [19] provides a mechanism to bridge this gap by allowing Java methods to call native functions and vice versa. Through the JNI, native code can even instantiate an *embedded JVM* to execute Java code. Unfortunately, a JNI call crossing the boundary between native code and the JVM has a considerable runtime overhead compared to a call that does not cross this boundary. Because UDFs are typically written to process an individual tuple or a row at a time, a naive implementation that simply invokes the UDF through JNI for each row incurs this overhead for each row (Section 2).

Previous solutions to combine UDF-centric frameworks with native execution either reimplement the entire software stack in native

code (e.g., TupleWare [11]) or force the user to write UDFs directly in C (e.g., Impala [14]). Consequently, they forego a large amount of existing code and sacrifice interoperability with JVM-based languages. In this paper, we aim to investigate how to preserve interoperability with the JVM and optimize execution of Java-based UDFs inside a native code environment.

A rather obvious remedy is to amortize the overhead of the JNI call over more than one row, i.e., instead of invoking the UDF for one row at a time, a single JNI call processes a batch of rows. In this paper, we call such a batch a *stride* and this form of UDF invocation *strided execution*. Existing systems (e.g., Vertica [15]), require users to program against a specific interface to enable strided execution of UDFs that operate on a single row at a time.

Explicitly exposing the strided nature of execution allows for optimizations that are not possible with tuple-at-a-time execution model, such as SIMD vectorization. However, it also puts an additional burden on the user which we try to avoid. Note that both approaches can be mixed, i.e., the system can optimize UDFs in a tuple-at-a-time execution model by default, and include a batch-aware interface that can be used if further, UDF-specific optimizations are required by users.

In our analysis, we use *just-in-time (JIT) compilation* of a *strided execution wrapper* to move the critical loop that invokes the scalar UDF from the C++ engine into the JVM. Compiling a UDF-specific wrapper at run time allows us to execute scalar UDFs in a strided fashion transparently without exposing a new interface to the user writing the UDF. Additionally, we use Java's *direct byte buffers* to pass data between the engine and the JVM without creating unnecessary copies in many cases. We implement these techniques in the research prototype Wildfire [2] that integrates a C++ columnar engine with Spark. We embed a JVM into the Wildfire engine and compare the resulting execution times of queries that contain scalar UDFs with their execution in Spark.

In this paper, we make the following contributions:

- We describe in detail how to effiently execute scalar Java UDFs inside an embedded JVM in a native code environment using JIT-compilation of a UDF-specific strided execution wrapper (Section 3).
- We find that executing Java UDFs inside an embedded JVM in Wildfire is consistently faster, at least 1.12 times, than executing them directly in Spark. Furthermore, the overhead of a UDF-based execution is small. Evaluating a predicate using a UDF is 1.27 times slower than an evaluating an equivalent SQL predicate without a UDF (Section 4).
- We compare the strided execution of scalar Java UDFs inside an embedded JVM to Java UDFs that are compiled directly to machine code as well as versions hand-written in C++. Our analysis shows that strided execution inside an embedded JVM is generally faster than compiling Java UDFs to machine code and comparable to hand-written code. Overheads compared to hand-written versions are due to constraints of the Java programming model, e.g., that Java strings are immutable (Section 5).

We discuss related work in Section 6 and present our conclusions in Section 7.

2 BACKGROUND

In this section, we briefly discuss how Java UDFs are represented in Spark, quantify the overhead of JNI calls, and describe Wildfire, the research prototype we use for our analysis.

2.1 Scalar UDFs in SparkSQL

Users have to register UDFs with Spark before referring to them in SparkSQL [1] queries, as shown below:

```
var offset = 10
sqlContext.udf.register("add_offset",
                        (i: Int) => i + offset)
sqlContext.sql("SELECT add_offset(i) FROM table").show()
```

Typically, SparkSQL UDFs are specified as Scala lambda functions, which are closures, i.e., they capture any free variables used in their definition. In the example above, the `add_offset` UDF adds to its input the value in the variable `offset` that is specified outside of the UDF. If a captured variable is mutable, such as `offset` above, and its value changes between SparkSQL queries, each query will use the latest value. This property can be used to configure more complex UDFs, e.g., machine learning models.

Internally, Scala lambda functions are represented as anonymous Java classes. The listing below shows the class definition of the `add_offset` UDF that is generated by the Scala compiler.

```
1  public final class SparkProgramm$$anonfun$run$1
2      extends scala.runtime.AbstractFunction1$mcII$sp
3      implements scala.Serializable {
4    public SparkProgram$$anonfun$run$1(
                              scala.runtime.IntRef);
5    public final int apply(int);
6    ...
7  }
```

We want to point out the following details. Free variables are captured by the UDF by passing them as an argument to the class constructor (line 4). The variable `offset` is represented internally by the Scala compiler as an `IntRef` type because it is mutable and captured by a lambda function. Consequently, changes to `offset` outside of the UDFs are visible when the UDF is evaluated. If `offset` were immutable, it would be passed as a simple primitive `int` type and would be captured by value.

The actual lambda function is executed by calling the `apply` method of the class (line 5). It adds the value that is passed as an argument to the method to the value of `offset` that was captured by the class constructor.

Note that the name of the UDF, `add_offset`, is not stored inside the Java class. The class name has no connection to the UDF's name.

2.2 The Java Native Interface

To facilitate interoperability between Java and code that does not run on the JVM, the Java specification defines the Java Native Interface (JNI) [19]. It provides an API to start a JVM, and manipulate Java objects and call Java methods inside it. However, as Figure 1 shows, executing Java methods via a JNI call incurs a significant overhead. In order to highlight the JNI overhead, we use a Java method that performs the least possible amount of work, i.e., by returning a constant integer. We invoke the method one billion times from a loop in C++. The figure shows that implementing the loop in C++ code and calling the method via JNI (dashed line) is

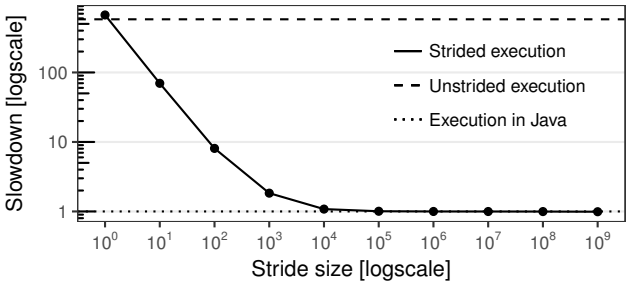


Figure 1: JNI call overhead.

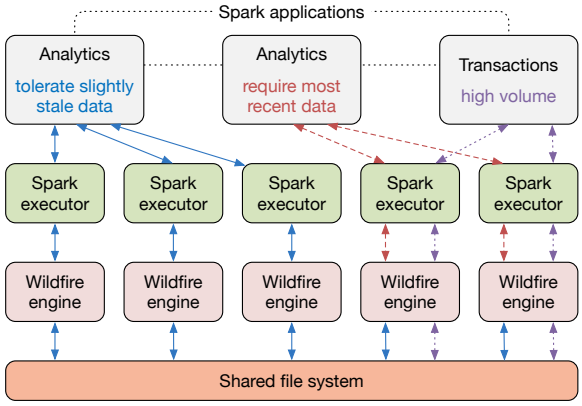


Figure 2: Wildfire architecture.

two orders of magnitude slower than implementing the complete loop in Java and, thus, avoiding the JNI call altogether (dotted line).

In order to reduce this overhead, we must move the loop from C++ into the JVM. To this end, we break up the loop into strides. We loop over the strides in the C++ code and execute one JNI call per stride. Instead of calling the Java method directly, we invoke a wrapper method that loops over the rows of a stride (solid line). As we can see, a comparatively small stride size of 10,000 rows effectively amortizes the JNI call overhead. Given the low computational complexity of the UDF, this value presents an upper bound on the stride size required to amortize the cost of JNI calls.

2.3 Wildfire

Wildfire is a research prototype of a distributed, hybrid transactional and analytics (HTAP) system. It is designed to handle very high rates of OLTP traffic while, at the same time, supporting OLAP queries on the latest data [2].

System architecture. The Wildfire architecture, shown in Figure 2, leverages Spark as a front end for analytical queries. It takes advantage of the existing ecosystem for big data analytics, machine learning, and graph processing. Users can submit analytical queries to Wildfire using SparkSQL or Spark's DataFrame API. The Wildfire engine speeds up analytical queries (solid arrows in Figure 2), handles OLTP traffic (dotted arrows), and provides access to newly ingested data before it is stored in the shared file system through a background process (dashed arrows).

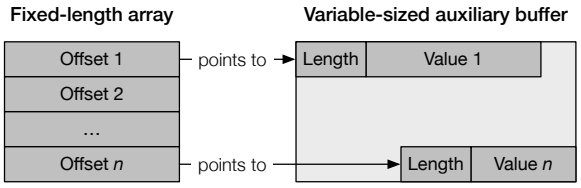


Figure 3: Storage of variable-sized values.

In order to move computation to the data, SparkSQL plans are pushed down to the Wildfire engine as much as possible. The well-defined semantics of SQL make this process fairly straightforward, with the exception of UDFs because they allow the execution of arbitrary code. Executing UDFs outside of the Wildfire engine moves the computation away from the data and leads to costly data transfers. It is, therefore, desirable to also move the execution of UDFs from the Spark front end to the Wildfire back end as part of the plan push-down. However, there's a conflict of implementation languages. SparkSQL UDFs are generally written in Scala whereas the Wildfire engine is implemented in C++.

Query engine. Wildfire uses a columnar query engine similar to DB2 with BLU acceleration [21]. It executes queries in a pipelined [18], block-at-a-time [6] fashion. The operator tree of a query is split into individual pipelines. Each pipeline scans a single input table and consists of operators that perform filters, hash table probes for joins, aggregations, etc. Furthermore, pipelines operate on strides of rows. Each operator consumes strides of rows consisting of one or more input columns and produces one or more output column strides. An input stride is fully consumed by a pipeline before the next input stride is processed.

Data storage and representation. On disk, data processed by Wildfire is stored as Parquet files [9]. The columnar representation of the Parquet format is kept during execution, i.e., the data of each column is stored as a contiguous memory region. Fixed-sized values are simply stored as an array. Variable-sized values are stored using two data structures, as illustrated in Figure 3. A fixed-sized array contains offsets into a variable-sized auxiliary buffer. In this auxiliary buffer, each value is prefixed by its length. We refer to the fixed-sized arrays collectively as *input/output buffers*.

Because the size of input/output buffers does not change during the execution of different blocks of the same query, they are only allocated once at the beginning of the execution of the query and then reused. Wildfire reads the data of each block into the same input buffer. Similarly, the memory region allocated for the output buffer is also reused. However, memory allocated for auxiliary buffers is not guaranteed to be reused because the size of the buffer can change from one stride to the next due to variable-sized values. When a stride's auxiliary buffer has a larger size than the previous one, the memory is reallocated for that buffer instead of reusing the one from the previous stride.

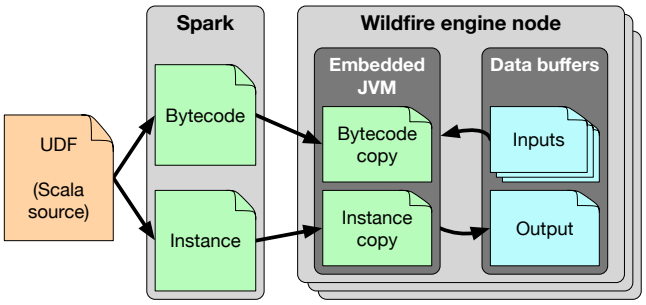


Figure 4: UDF execution inside an embedded JVM.

3 UDF EXECUTION IN AN EMBEDDED JVM

We focus our analysis on executing SparkSQL UDFs inside an embedded JVM on all Wildfire engine nodes. An overview of this process, and some of the necessary steps, are shown in Figure 4. Notably, we have to transfer the UDF's bytecode and its class dependencies to the engine nodes when the UDF is registered in Spark. We inject the bytecode into the embedded JVM, generate and compile a strided execution wrapper for the UDF, and register it under the name of the UDF. When the UDF is called inside a SparkSQL query, we also need to transfer the current instance of the generated UDF class to execute the UDF with the correct state. During the UDF's execution, we need to pass data from the engine's data buffers to the embedded JVM. These steps are described in more detail in the following sections.

3.1 Bytecode extraction and transfer

As a first step, we have to transfer the bytecode of the UDF class and all its dependencies to the Wildfire engine nodes when the UDF is registered in Spark. Given their class names, the bytecode of these classes can be retrieved as resources from the Spark class loader. Once retrieved, we parse the bytecode to enumerate all referenced types, and repeat this process recursively. Rather than computing a complete transitive closure of all referenced types, we stop the recursion when we encounter a Java or Scala common type as we do not want to transfer them. Java types are present in the embedded JVM by default and we also unconditionally load the Scala language library into the embedded JVM.

3.2 Bytecode injection

Once the bytecode of the UDF and its dependencies has been received by a Wildfire engine node, we have to make the embedded JVM aware of it. The JNI provides the function `DefineClass` to inject a class into a JVM. However, once defined, a class with a particular name cannot be changed. This poses a potential problem, as UDFs defined in the Spark front end can reuse the class names of previous UDF classes. For example, the lifecycles of the Spark front end and the Wildfire engine nodes are independent, and a user could change the implementation of a UDF between different Spark jobs. Similarly, multiple users may use different UDFs with the same Java class name.

Fortunately, different classes with the same name can be isolated in a JVM if they are loaded from different class loaders. Class dependencies, and the strided execution wrapper described below, must be loaded from the same class loader as the UDF class itself. We, therefore, inject the bytecode of the UDF and its dependencies into the embedded JVM by initializing a custom class loader with a mapping of class names to byte arrays containing the respective bytecode as described in the previous section. This class loader can define classes from the stored bytecode as they are loaded by the JVM.

To support UDFs with the same name from different Spark front end session, we have to store a reference to the UDF-specific class loader in a session-specific engine catalog. When a session is terminated, or the UDF is otherwise unregistered from the engine, the class loader can be released and garbage-collected. However, in our prototype, we store it in a global session catalog for simplicity.

3.3 Passing data

As described in Section 2.3, input/output buffers are fixed-size, contiguous memory regions. For variable-sized values, another contiguous memory region is used as an auxiliary buffer.

The JNI can wrap memory regions inside a Java direct ByteBuffer object. The contents of this byte buffer can be accessed from Java classes with typed getter and setter methods, e.g., `get` to read a 8-bit byte or `setInt` to write a 32-bit integer. The JVM will make a "best effort" to avoid unnecessary data copies when accessing the contents of direct byte buffers.[1] By wrapping input/output buffers, as well as auxiliary buffers, inside byte buffers, we can pass the data processed by a UDF as opaque memory blocks from the Wildfire engine to the embedded JVM, minimizing JNI call overhead as well as unnecessary copies. Inside the JVM, input buffers that wrap primitive data types are accessed using the typed getter methods and are essentially treated as typed arrays. Buffers that wrap variable-sized values, e.g., strings, need to construct a Java object from the data contained in the auxiliary buffer. Unfortunately, this object construction requires the data to be copied at least once into a temporary Java byte array.

Because Wildfire reuses the input buffers and replaces their contents during the execution of subsequent blocks of a single query, they only have to be wrapped once as a direct byte buffer for each query. Similarly, output buffers only have to be wrapped once. However, as stated in Section 2.3, auxiliary buffers for input and output are not guaranteed to be reused. Therefore, they have to be wrapped within new direct byte buffers for each query block.

3.4 Strided execution

To overcome the overhead associated with JNI calls for each input tuple, we automatically generate the Java code for a custom strided execution wrapper for each UDF when it is registered, compile it using the Janino compiler [23], and inject it into the UDF-specific class loader. We need to distinguish two cases. For fixed-size outputs, only one JNI call per stride to the wrapper is necessary. For variable-sized outputs, multiple calls may be necessary if the output exceeds the allocated size of the auxiliary buffer.

[1] https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html#direct

```
1  public class StridedExecutionWrapper {
2    public static void wrapUdf(
3        UdfClass udfInstance,
4        int numRows,
5        ByteBuffer output,
6        ByteBuffer auxiliaryOutput,
7        ByteBuffer[] inputs,
8        ByteBuffer[] auxiliaryInputs) {
9      for (int i = 0; i < numRows; ++i) {
10       output.putX(udfInstance.apply(
             inputs[0].getX(),...,inputs[N].getX()));
11     }
12   }
13 }
```

Listing 1: Strided UDF wrapper template.

The fixed-size version of the wrapper follows the template shown in Listing 1. It exports a standardized method call, which takes a UDF class instance, the number of rows, and the input/output and auxiliary buffers as parameters. The wrapper iterates over the input rows and calls the `apply` method of the UDF class for each input tuple, unpacking the input byte buffers in the process using the appropriate typed getter methods, i.e., `getX` in line 10. The output is then stored in the output buffer using the appropriate setter method, i.e., `putX` in line 10. Note that the example assumes that input types are primitive; if strings are used, a temporary String object needs to be constructed from the auxiliary buffer. This process is omitted in the template of the wrapper.

The variable-sized version extends the template in Listing 1 to handle multiple invocations, and accept and return internal state for each invocation. The state consists of the index of the last processed row in the current stride and the result of the last UDF invocation wrapped inside a Java object. It is used as a signal that the engine has to resize the auxiliary buffer and is otherwise opaque to the Wildfire engine. In our prototype, we initialize the auxiliary buffer with the size of the largest buffer seen so far since the start of the query, and double its size whenever a stride exceeds it.

Figure 5 illustrates this process using an example where the auxiliary buffer needs to be resized once to fit the results of the UDF. Initially, the wrapper is invoked with an empty state, i.e., passing `null` to it. The wrapper then loops over input rows as

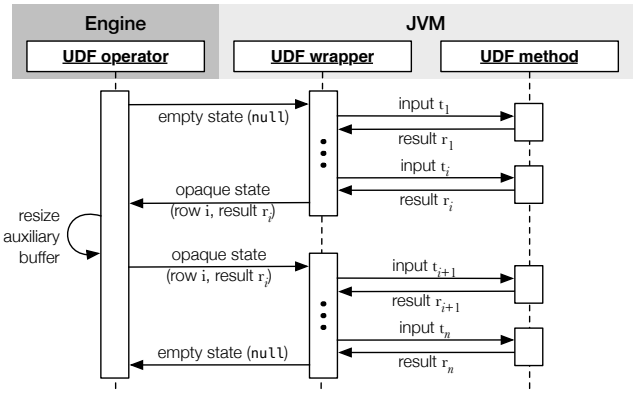


Figure 5: Control flow for variable-sized outputs.

shown in Listing 1. If adding the result of the current row to the auxiliary buffer would exceed its capacity, the row's index and the current result is stored in the state object which is returned to the engine. The engine then extends the memory region containing the output auxiliary buffer, wraps it inside a new direct byte buffer, and calls the wrapper again with the previously returned state, i.e., the previously computed result of the UDF and the index of the corresponding row. The wrapper adds the stored result to the new auxiliary buffer and continues processing of the block after the row stored in the state object. Once all rows have been processed, the wrapper signals the end of processing by returning an empty state.

3.5 Detailed architecture

Figure 6 depicts our architecture to execute UDFs in an embedded JVM in detail. The work flow is broken down into three parts, implemented by three operators evaluated by the Wildfire engine.

Operator 1: Register UDF. The first operator is invoked by the Spark front end when the UDF is registered. On the Spark front end, we retrieve the UDF bytecode from the Spark class loader, parse it for dependencies, retrieve the bytecode of dependent classes, and transfer it over the network to the Wildfire engine nodes. The `Register UDF` operator accepts the bytecode of all involved classes and injects them into the embedded JVM using a UDF-specific class loader. It then compiles a strided execution wrapper for the UDF and injects it into the JVM. Finally, a reference to the strided execution wrapper is stored under the UDF's name in the engine's catalog.

Operator 2: Register UDF instance. The `Register UDF instance` operator is invoked by the Spark front end when a UDF is encountered in a SparkSQL query. On the Spark front end, we serialize the UDF instance and all captured variables using the Java serialization API. The serialized data is then transferred over the network to the Wildfire engine nodes. The Wildfire operator deserializes the UDF class instance using the UDF-specific class loader and stores it under the UDF's name in the engine's catalog.

Operator 3: Evaluate UDF. Finally, the `Evaluate UDF` operator is inserted into the operator tree constructed by the Wildfire Catalyst component to evaluate a SparkSQL query. When it is invoked on the first query block, it wraps the engine's input/output buffers as Java direct byte buffers. It also wraps auxiliary buffers for each block if necessary. The operator retrieves the UDF class instance and the UDF-specific strided execution wrapper from the engine's catalog and calls the wrapper, passing the UDF class instance and the input/output and auxiliary buffers as required. For variable-sized outputs, the wrapper is called until the input has been processed entirely.

3.6 User-defined types

In our prototype, we support UDFs with primitive types and strings as parameters. As stated in Section 3.3, primitive types can be accessed from Java's direct byte buffers without making an extra copy. However, to process string parameters, we need to construct a Java String object and also copy the data into a Java byte array.

Similarly to string parameters, user-defined types (UDTs) also require us to create an object of that type. If the UDT can be decomposed into a fixed number of primitive types or strings, constructing

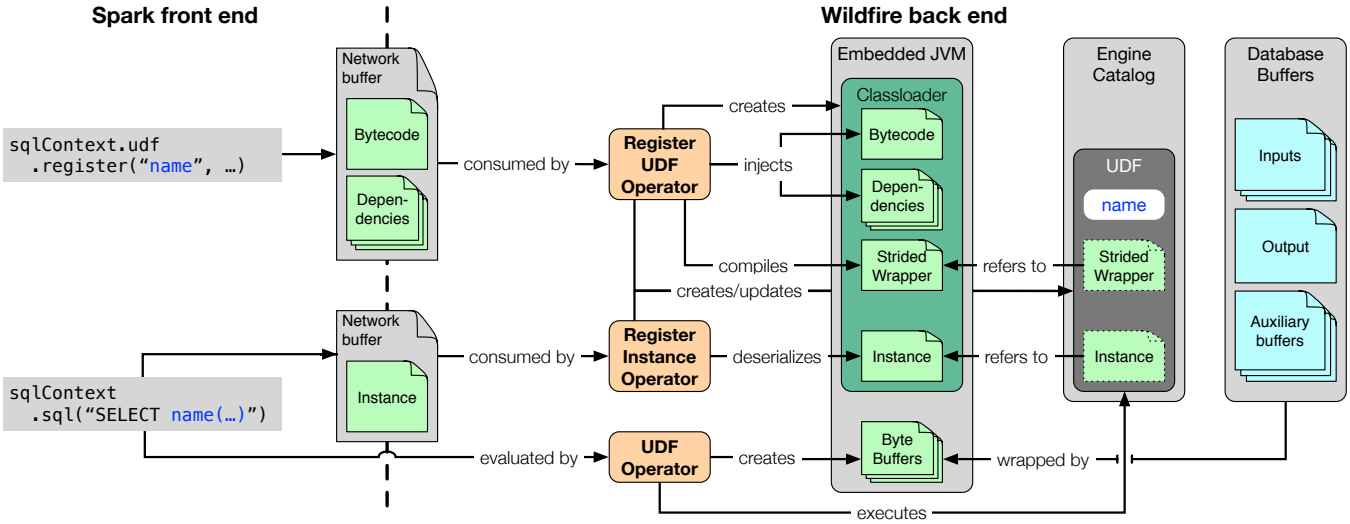


Figure 6: Embedded JVM architecture.

it is a straightforward task that can be accomplished by calling an appropriate constructor. Thus, the cost of supporting such simple UDTs is similar to supporting strings, i.e., creating the object and copying the data internally in the constructed instance.

A UDF parameter can also be a more complex nested data structure, containing optional and/or repeatable components. The Parquet format, which Wildfire uses as its storage scheme, supports nested structures through so-called *repetition* and *definition levels*, which are stored as additional columns alongside the decomposed components [17]. However, supporting UDTs and nested structures as UDF parameters is outside the scope of this paper.

3.7 Security considerations

Database management systems that support UDFs generally have to take measures so that buggy or malicious UDFs cannot crash or otherwise harm the database process. A popular technique is fencing, whereby UDFs run in separate processes and communicate with the database through some form of interprocess communication. Running UDFs inside an embedded JVM can provide a similar mode of separation. We assume that the JVM implementation itself is robust, such that a UDF cannot crash it, and, therefore, also not the process that embeds it. Instead, errors such as dereferencing null values or buffer overflows will throw an appropriate exception. These errors can then be handled gracefully by the calling code.

It is indeed possible to purposefully crash a JVM through the JNI interface, e.g., by calling JNI functions with parameters of the wrong type. However, we can reasonably guard against this contingency since we only call our runtime-compiled UDF wrapper, which has a fixed signature, and not arbitrary methods.

Another consideration is the execution of UDFs that do not crash the database process but are otherwise malicious, e.g., by accessing the file system or opening a network socket. To guard against such UDFs, we can use Java's security manager mechanism[2] to restrict access to specific Java APIs. A UDF that exceeds the permissions set by the security manager will throw an exception which can be handled by the calling code. However, such behavior is currently not implemented in our prototype.

[2]https://docs.oracle.com/javase/8/docs/api/java/lang/SecurityManager.html

4 EVALUATION

In this section, we evaluate the techniques described in Section 3 using a set of queries containing scalar UDFs representative of different use cases and resource requirements.

Our analysis focuses on the effect of UDF execution on query performance, and excludes other influences, such as disk I/O costs. This is a fair evaluation since many data analysis workloads consist of multiple Spark tasks and intermediate results are kept resident in memory to improve performance [26]. Therefore, we devise a number of microbenchmarks that model UDFs with different computational requirements, ranging from simple UDFs that are bound by data movement to more complex UDFs that are compute-bound. We also investigate the effect of the Java type system, i.e., using primitive types vs. Java objects, on UDF performance.

Specifically, we use a simple *range predicate* to evaluate the cost of UDF invocation and UDFs that are bandwidth-bound; a *fixed-point iteration* to evaluate compute-heavy UDFs; a *word length* UDF that takes a variable-sized Java object as input; and an *upper case* UDF, that additionally produces a variable-sized Java object as output.

4.1 Test system and setup

Before discussing the evaluated UDFs in detail, we want to describe our test environment and experimental setup. Our test machine runs Ubuntu 16.04 LTS and the Oracle JDK 112. Every experiment is executed on a system with four Intel Xeon X7560 processors running at 2.27 GHz. Except for a thread scaling experiment described in Section 4.6, our experiments execute on a single thread. The machine contains 512 GB of memory. On disk, the data is stored in

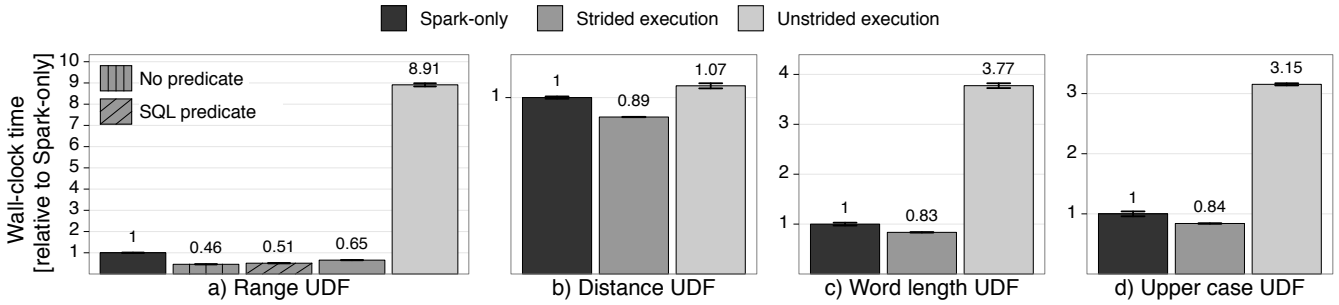


Figure 7: Run times of tested UDFs in different environments, single-threaded execution.

uncompressed Parquet files, using a RLE/PLAIN encoding. Note that on-disk data is accessed from the Linux buffer cache. We observed virtually no data being read from the disk during the execution of the experiments. Also note that we use full table scans and do not build any indexes on the data.

We run each experiment ten times and discard the first result. We report the mean value of the run times of the remaining nine results. We also show one standard deviation as error bars. We measure the wall-clock time required to issue the SparkSQL query and materialize the result in the Spark front end. In particular, we include the time required to transfer the UDF class instance from Spark to Wildfire, including serialization and deserialization, because this cost is incurred for each query using the UDF. However, we do not include the time required to transfer the bytecode of the UDF class and its dependencies, nor the compilation of the strided execution wrapper, as these two steps are only performed during the registration of the UDF. We report normalized wall-clock time relative the execution of the queries in Spark since this provides the JVM-only baseline for comparison.

For each UDF, we have determined the minimal required stride size, so that doubling it does not yield a further improvement. We list these stride sizes for each UDF in Table 1. Note that determining the correct stride size for a query does not only depend on the UDF but also on other aspects of the execution environment, such as cache sizes and tuple width. Such an analysis is outside of the scope of this paper.

The results of our experiments are shown in Figure 7. In general, we show the run times of the strided execution described in Section 3 (medium grey bar) and compare it against the unstrided execution where we invoke the original UDF method for each tuple through JNI (light grey) and the execution of the SparkSQL query in Spark 2.0.2 (dark grey).

Table 1: UDF stride sizes.

UDF	Stride size
Range	4096
Distance	512
Word length	1024
Upper case	1024

4.2 Range UDF

The first query uses a range predicate on an integer column. Such a predicate can be expressed natively in SQL, allowing us to evaluate the overhead of formulating it as an UDF instead. Furthermore, by setting the range so that all values are selected, we can also measure the total cost of UDF invocation.

Consequently, we evaluate three versions of the query on Wildfire which are shown below. The first contains no predicate, i.e., it selects all rows from the table. The second version formulates the range query as a standard SQL predicate. The third version encapsulates the range predicate inside a UDF. Each version of the query has a selectivity of one since the first version does not contain any predicate. Therefore, in order to minimize the amount of data that is transferred from the Wildfire engine nodes back to the Spark front end, and highlight the cost of the UDF, we wrap the results of each query in an aggregation function.

```
-- Version (1)
SELECT sum(a), count(a) FROM R

-- Version (2)
SELECT sum(a), count(a) FROM R WHERE min < a AND a < max

-- Version (3)
-- sqlContext.udf.register("filter",
--                         (a: Int, low: Int, high: Int)
--                         => low < a && a < high)
SELECT sum(a), count(a) FROM R WHERE filter(a, min, max)
```

This UDF is designed to measure UDF invocation overhead. It has a low computational load, i.e., the influence of UDF invocation is maximized compared to the actual computation. The difference between the run times of the first and second query yields the cost of applying the predicate as a native SQL expression, whereas the difference between the second and third query yields the cost of formulating the predicate as a UDF compared to native SQL.

We execute the queries on a 10 GB table with one column containing approximately 2.5 billion random integers. For the strided execution we use a stride size of 4,096, which is a conservative value. The results are shown in Figure 7a.

As stated, the UDF is computationally lightweight, i.e., the difference between the query without a predicate (horizontal pattern) and with a SQL predicate (diagonal pattern) is small. Consequently, there is an order of magnitude difference between strided and unstrided execution. Evaluating the predicate inside a Java UDF (medium grey

bar) is 1.27× slower than evaluating it directly in SQL. However, evaluating the UDF inside an embedded JVM in Wildfire is 1.54× faster than evaluating it in Spark directly.

4.3 Distance UDF

The second query, shown below, contains an example of a computationally heavy UDF. It uses Vincenty's formulæ [25] to compute the distance between two points on an oblate spheroid. Vincenty's formulæ are an algorithm that contains several trigonometric functions and is based on a fixed-point iteration. While this algorithm can be implemented directly in SQL using recursive common table expressions, it is much more straightforward to implement it inside a UDF.

```
-- distance: UDF implementing Vincenty's formula
SELECT sum(lat), count(lat) FROM R
WHERE distance(lat, lon, 0, 0) > 100000;
```

We evaluate the UDF on a 1 GB table containing approximately 64 million random points uniformly distributed on a sphere. The UDF computes the distance between each point and a point we arbitrarily choose at latitude and longitude 0°. On average, the UDF requires five iterations to converge. To eliminate data transfer overheads, we again wrap the results in an aggregation function, and minimize its overhead by chosing the selectivity of the query so that no point is actually selected. Note that the distance UDF processes forty times fewer rows and ten times less data than range UDF. For the strided execution we use a stride size of 512.

The results of this experiment are shown in Figure 7b. Contrary to the range query, the JNI call overhead is dwarfed by the computational complexity of the UDF itself. Consequently, the difference between strided and unstrided execution is much smaller, i.e., a factor of 1.2. In general, the necessity of strided execution to achieve good performance diminishes as the computational load of the UDF increases. The UDF runs 1.12× faster in Wildfire than in Spark.

4.4 Word length UDF

The third query, shown below, returns the length of an input string.

```
-- def length(s: String) = s.length
SELECT sum(length(word)), count(word) FROM R
```

The query evaluates the execution of UDFs that operate on Java objects as input, using variable-sized strings as an example. Unlike primitive types, which can be read directly from the direct byte buffers used to pass data between the engine and the embedded JVM, Java objects used as inputs to the UDF first need to be constructed. This involves copying the data from the direct byte buffer into the data structures that back the Java object which increases the data movement cost of the UDF.

We evaluate the UDF on a data set of about 80 million randomly generated variable-sized words. The word length follows a Poisson distribution with $\lambda = 9.7$, the average length of distinct English words [22]. The approximate size of the data set is 1 GB. For the strided execution we use a stride size of 1,024.

The results of this experiment are shown in Figure 7c. As the UDF is also computationally lightweight, there is a significant difference, about a factor of 4.5, between strided and unstrided execution. However, due to the memory access entailed in constructing String objects, which is required in both versions, the difference is not as big as for the range query. The UDF runs 1.20× faster in Wildfire than in Spark.

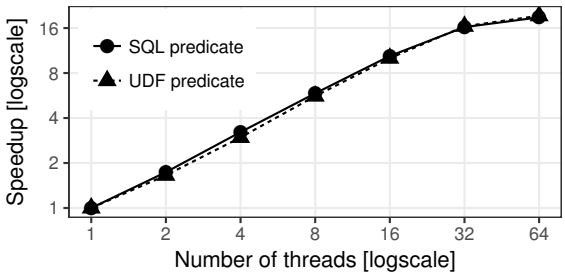


Figure 8: Scaling with multiple threads.

4.5 Upper case UDF

The final query, shown below, transforms an input string into upper case.

```
-- def upper(s: String) = s.toUpperCase
SELECT count(upper(word)) FROM R
```

This query evaluates the generation of variable-sized auxiliary output buffers. Since the UDF is opaque to the Wildfire engine, the size of the auxiliary buffer is not known when processing of a block starts. The engine needs to resize the buffer when the actual output exceeds the size estimation. As stated in Section 3.4, we use a simple strategy that always allocates the size of the largest output block seen so far since the start of the query and double the size estimation if a block exceeds it.

We evaluate the UDF on the same data set used in the word length UDF and show the results in Figure 7d. Compared to the word length UDF, the upper case UDF requires two additional passes over the string, one to transfer its content into upper case and a second to write the string as an array into the output auxiliary buffer. Consequently, the UDF is dominated by String data copies, and each of the strided, unstrided, and Spark versions take about twice as long as the previous UDF (not visible due to normalization in Figure 7d).

4.6 Thread scaling

So far, we have run all UDFs in a single-threaded environment. In a final experiment, we want to evaluate how our embedded JVM approach scales with multiple threads. Our test system is a four-way NUMA machine with eight physical cores per socket that support hyper-threading. In total, there are 64 logical cores in the system. We run the range UDF query from Section 4.1 in Wildfire and increase the number of threads from one to 64.

We show the results of two versions of the range query in Figure 8. Dots represent the query where the predicate is evaluated directly in SQL and triangles represent the query with a UDF predicate. For each version we show the speedup compared to single-threaded execution. As we can see, the system scales almost linearly with the number of threads. In particular, the speedup of the query with the UDF predicate closely follows the query with the SQL predicate, indicating that the embedded JVM does not introduce overheads that limit parallelization.

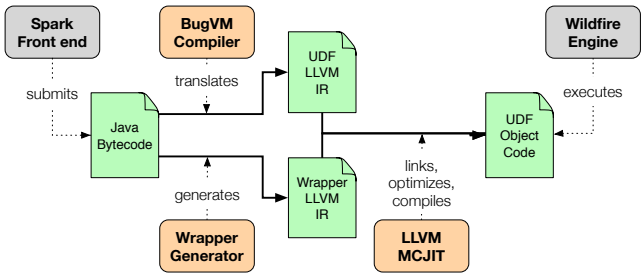


Figure 9: Compiling Java UDFs to machine code.

The drop-off in the curve between 32 and 64 threads is caused by the way data is partitioned on disk into Parquet row groups. During import, Wildfire partitioned the 10 GB into 161 row groups. Most of these row groups are 64 MB large, except for the first and the last which are smaller. Each row group is processed by a different thread in a work-stealing fashion. Since 161 is almost evenly divisible by 32 but not by 64, some threads are starved of data when we use 64 threads. Half of the threads process three row groups whereas the other half only process two.

4.7 Summary

Our approach of embedding a JVM inside the Wildfire engine nodes can effectively bridge Java code and native code. A naive execution strategy, which uses a JNI call to invoke the UDF for each tuple, incurs a large overhead for computationally lightweight UDFs. The key to overcome this overhead is to compile a UDF-specific strided execution wrapper at run time and move the critical loop from outside the JVM into it. The strided execution wrapper has the additional benefit of allowing us to pass opaque data buffers between the Wildfire engine and the embedded JVM using direct byte buffers, further reducing JNI call overhead to handle UDF arguments. The benefit of strided execution diminishes as the computational complexity and memory bandwidth of the UDF increases.

5 COMPILATION TO MACHINE CODE

The range query shows that the execution of a UDF inside an embedded JVM still incurs a small but significant overhead compared to the execution of the semantically equivalent SQL query. In this section, we investigate whether it is possible to improve the performance of Java UDFs by compiling them to machine code at run time. The approach is sketched in Figure 9. When the Spark front end registers a UDF, we first translate the Java bytecode into LLVM [16] intermediate representation (IR). For this step, we use the BugVM compiler [20], which is described in more detail below. We also generate LLVM IR for a wrapper function that calls the UDF for each tuple. We then link, optimize, and compile the UDF and wrapper IR fragments to object code using LLVM's MCJIT. The object code is dynamically loaded by Wildfire and the wrapper is executed for each block.

5.1 BugVM

BugVM is an ahead-of-time compiler for JVM-based languages [20]. It provides a compiler that translates Java bytecode to LLVM IR and a custom implementation of the Java runtime environment. BugVM leverages LLVM to produce machine code from the generated IR. This machine code is linked against the custom Java runtime resulting in a fully contained native binary.

The BugVM compiler uses Soot [24] to translate Java bytecode into a typed three-address IR in static single assignment (SSA) form. Instead of following the JVM's stack-based semantics, this IR is already register-based and can be easily translated to LLVM IR. Special JVM bytecode constructs, e.g., `invokevirtual` to call a polymorphic method, are implemented as functions for which BugVM provides the implementation. These special functions are inlined by the LLVM optimizer when appropriate to improve performance.

The BugVM JVM uses the Boehm-Demers-Weiser garbage collector [4, 5] for memory management.

5.2 JIT-compiled UDF performance

In this section, we compare UDFs that are JIT-compiled from Java bytecode to machine code against strided execution inside an embedded JVM, and also against versions of the UDF hand-written in C++ that are statically linked with the Wildfire engine. To allow comparison with the measurements reported in Section 4, we again normalize the wall-clock time to the execution time of the queries in Spark. We exclude the compilation time from our analysis because it is a one-time cost incurred when the UDF is registered with the engine. Note that each execution strategy evaluates the UDF in a strided fashion, using the stride sizes reported in Section 4.1 for each UDF.

The results, shown in Figure 10, clearly indicate overheads in the embedded JVM approach, as the UDFs hand-written in C++ outperform the Java UDFs executed inside the JVM. The word length UDF shows the biggest difference, a reduction by a factor of 2.1. However, as we will see below, the hand-written version exploits knowledge about the specific task performed by the UDF, as well as the storage scheme used by Wildfire, to achieve this speedup.

The JIT-compiled version of the computationally heavy distance UDF is as fast as the hand-written version and 1.75× faster than executing the UDF in an embedded JVM. In contrast, the hand-written versions of the range UDF and the upper case UDF are only marginally faster than executing the UDF inside the embedded JVM, whereas JIT-compiled versions are considerably slower. We discuss the reasons for this slowdown and possible remedies in the next section.

5.3 String construction optimizations

We now describe optimizations that we can apply to JIT-compiled machine code UDFs, using the word length UDF as a case study. The wrapper executing the UDF on a single stride is shown in Algorithm 1 in pseudocode.

A few operations deserve scrutiny. First, for each input string, we need to construct a Java String object (line 2). In Java, each string is represented by its own immutable String object, which in turn

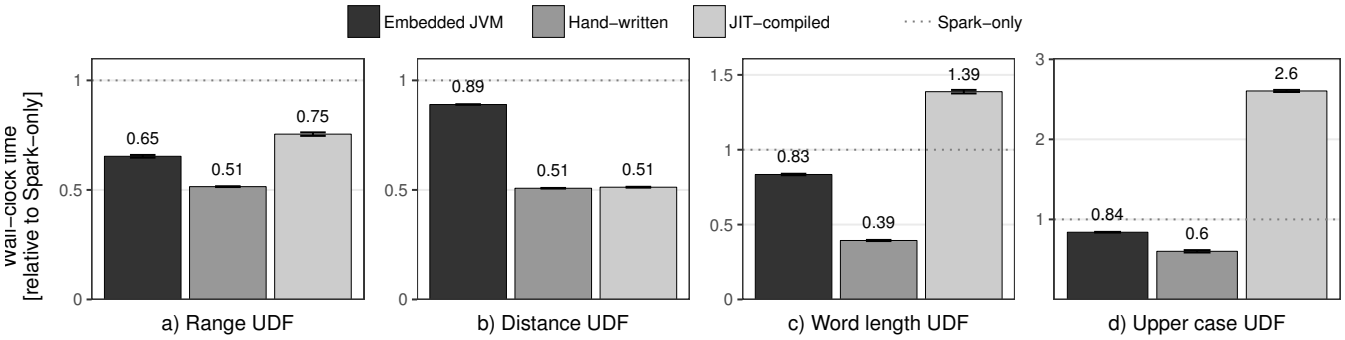


Figure 10: Performance of Java UDFs that are JIT-compiled to machine code.

contains a reference to a Java Char array. Thus, to create a Java string from a string in a Wildfire input buffer, a data copy is needed in addition to two object allocations, one for the String object and one for the referenced Char array, as shown in Figure 11a. Second, after the UDF has been called, we need to check if the UDF raised a Java exception (line 4). Finally, we have to explicitly release the Java string object that was allocated outside of the JVM, otherwise it cannot be reclaimed by the garbage collector (line 5).

The hand-written UDF is much simpler. Most conveniently, the length for each word is already stored in the auxiliary buffer, as shown in Figure 3 in Section 2. Thus, the Wildfire engine can evaluate the UDF without accessing the contents of the string at all. Wildfire also does not have to do any input-specific error handling. In other words, lines 2 to 5 are replaced by just copying the word length value from the auxiliary buffer to the output buffer.

In order to address the overheads in the generated wrapper, we investigate four optimizations: (1) Reusing String objects to reduce object construction; (2) Modifying the JVM's String objects to wrap data from the engine's input buffers to minimize data copies; (3) Relaxing the JVM's exception handling; and (4) Removing a memory fence normally required by the Java memory model.

Eliminate String object construction. Since the word length UDF does not store a reference to the input string, we can reuse the existing String object between UDF calls and simply replace the internal Char array. This optimization saves one object allocation as shown in Figure 11b, and, therefore, puts less pressure on the garbage collector. Note that this optimization requires knowledge about the inner workings of the UDF and cannot be applied in general. In particular, if the UDF were to store (a reference to) the input string internally, the contents of the stored string would change in-between each UDF call, violating the immutability of Java strings.

Input : Input buffer *input* with strings.
Output: Output buffer *output* of string lengths.
1 **for** $i \leftarrow 1$ **to** *size of input stride* **do**
2 $\quad$ *javaString* $\leftarrow$ `CreateJavaString`($input_i$);
3 $\quad$ $output_i \leftarrow$ `WordLengthUdf`(*javaString*);
4 $\quad$ `CheckForJavaException`();
5 $\quad$ `ReleaseJavaObject`(*javaString*);
6 **end**

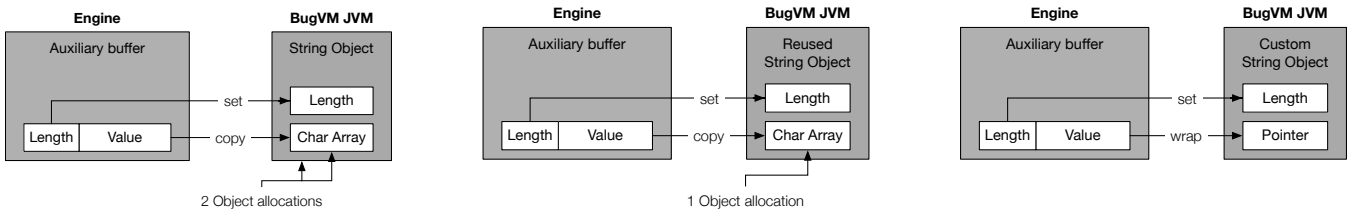
Algorithm 1: Word length UDF wrapper.

Eliminate data copies. To eliminate the object allocation of the internal Char array, we can implement a custom String class that wraps a string stored in a memory region outside of the JVM by storing a pointer to it, as shown in Figure 11c. This also removes the need to copy the string contents from outside the JVM into it. However, we now have to take care of distinguishing String objects created inside Java and those that wrap a memory region outside of the JVM. The latter are owned by the engine and cannot be automatically reclaimed by the garbage collector.

Relax exception handling. In general, Java UDFs submitted by Spark can have side effects, such as changing global state or performing I/O. Consequently, we have to check for error conditions after each UDF invocation, otherwise side effects can continue to occur after an error, resulting in inconsistent state of the application.

In Java, errors are typically signaled by raising an exception. If an exception was raised inside a JNI call, it will remain active until it is explicitly cleared. For a general UDF, we, therefore, have to check for the presence of an exception, take appropriate measures, and clear it. However, we know that the word length UDF is free of side effects. Furthermore, given our previous optimizations, we can also rule out an out-of-memory error since there are no more object allocations. Hence, we can move the exception handling code out of the loop.

Remove memory fence. The Java memory model guarantees that final fields of objects will be correctly initialized after an object is constructed [12, Section 17.5]. This guarantee, which restricts possible variable assignment reorderings by the compiler, applies to the internal Char array of the String object. To satisfy this guarantee in a multi-threaded environment, the compiler has to insert a memory fence instruction after the String object has been constructed. However, since the String object is not referenced by other code (and, therefore, also not by other threads), we can remove the memory fence from the generated code without violating this guarantee.



(a) Construction of String objects. **(b) Reuse of String objects.** **(c) Wrapping of engine buffer.**

Figure 11: Java String creation and optimizations.

5.4 Effect of optimizations

Figure 12 shows the effect of the optimizations described above, with the original JIT-compiled UDF without optimization and the UDF hand-written in C++ as a reference. The measured wall-clock times are again normalized to execution time in Spark. Each optimization is applied on top of the previous optimizations in the order they are mentioned above. The fully optimized UDF is almost as fast as the hand-written version. Eliminating object allocation and data copies has the biggest effect on performance. Further optimizations have diminishing returns.

While the exception handling code has a small influence on the performance of the word length query, for the range query it is precisely responsible for the gap between the hand-written and the JIT-compiled version of the query in Figure 10. In fact, if we eliminate this check, LLVM's JIT compiler produces exactly the same machine code as for the hand-written version, which is inlined and makes use of the processor's SIMD instructions. For the distance UDF, there is virtually no difference between the execution time of the JIT-compiled and hand-written version because the time required for exception handling is neglible compared to the time spent in the UDF itself.

As we have seen, when we apply all of the proposed optimizations to the JIT-compiled word length query, it is almost as fast as hand-written code. Unfortunately, we have broken Java in the process because we have changed core language semantics.

5.5 Applicability of optimizations

The Java language specification mandates that strings are immutable. Three of the optimizations violate this critical guarantee. The issue arises when a UDF *leaks* the string reference, i.e., if it stores the reference in a global variable as part of state it maintains across invocations. For regular Java strings, such leaking is unproblematic because strings are immutable. However, if we reuse the String object across UDF invocations and exchange the underlying Java Char array, the leaked string will change from one invocation to the next. Similarly, if we use a custom String object that wraps a memory region owned by the engine, the leaked string also changes when the engine modifies the memory region. Finally, if we remove the memory fence and the UDF passes the String reference to a different thread, this thread could see the uninitialized contents of the String object.

For the word length UDF, we can apply these optimizations safely since we know that no String reference is leaked. In general, these optimizations are safe if the UDF is free of side effects.

UDFs that keep state across invocations may lead to unpredictable behavior in Spark. There is no way to directly exchange state information between Spark executors. Furthermore, the result depends on how the Spark job is distributed across executors. The same applies to the embedded JVMs in the Wildfire engine nodes.

Even without knowledge of the inner workings of the UDF, it is possible to apply many optimizations with the help of static and dynamic code analysis. For example, by checking the reference count of the String object after the UDF has finished, we can verify that no String reference has been leaked and reuse the object for the next iteration.

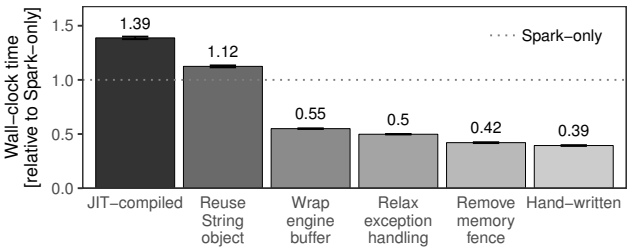


Figure 12: Effect of optimizations on JIT-compiled code for the word length UDF.

5.6 Summary

As we have seen, computationally heavy UDFs that do not create objects can benefit from JIT-compilation to machine code, leading to a performance increase by a factor of 2.1. While the Java programming model limits many possible automatic optimizations related to object creation overheads, it is still possible to apply these optimizations through static and dynamic code analysis. Thus, it is not surprising that the Oracle HotSpot VM is quite good at dynamically optimizing Java code. Consequently, UDFs evaluated inside an embedded JVM in a strided fashion achieve a performance that is comparable to machine code for many use cases.

6 RELATED WORK

UDFs are very common in database systems. Most commercial database vendors, such as IBM, Oracle, Vertica, and Cloudera, also support Java UDFs, but their manuals discourage the use of Java UDFs in favor of faster native implementations that are dynamically loaded at run time.

Impala [14], Cloudera's Hadoop SQL engine with a C++ back end, supports Hive UDFs written in Java to minimize the effort for users that migrate from Hive to Impala. In fact, Hive UDFs can run unmodified inside a JVM embedded by the Impala engine. The UDF is executed through a JNI call one row at a time, thus, incurring a significant overhead. The call overhead is minimized by preallocating memory buffers that are managed by the JVM. In this paper, we analyze an approach that further reduces the call overhead by a strided execution of the UDF over a number of rows through a wrapper function which we generate automatically on the fly. The Impala manual recommends C++ UDFs for performance reasons, stating that compiled C++ UDFs yield execution speeds 10× faster than equivalent Java UDFs.[3]

Vertica [15] supports so-called *User-defined Transform Functions* that can process multiple rows per invocation and, thereby, reduce the impact of the JNI call. Users need to implement a `TransformFunction` class, in which they get access to an unspecified number of rows via an iterator. In our analysis, we show that this iterator-based strided execution wrapper can be generated automatically.

Impala also supports UDFs that are provided in LLVM IR form although this feature is undocumented. Providing UDFs in LLVM IR allows the LLVM JIT compiler to inline the UDF code with engine code and optimize them together. We use the same approach when we move the critical loop calling the Java UDF from the engine to the JVM in our strided execution wrapper. Here, Java bytecode is the common IR and is optimized holistically by the Oracle HotSpot JIT compiler. This holistic approach is identical to our scenario in which we compile UDFs to LLVM via BugVM together with the generated wrapper. The largest gains are possible when the LLVM IR of the UDF is combined with the IR of an entire query plan.

Injecting code in IR form is a common theme in other LLVM-based runtimes as well. TupleWare [11] and HyPer [13], or even HPC environments such as Julia [3], also take advantage of the common intermediate representation when injecting external user code. LLVM's intermediate representation allows the design of polyglot systems that support a variety of different language front ends.

7 CONCLUSION

In this paper, we analyze the overheads of executing Java UDFs within a database engine written in native code, and which does not use the JVM. Contrary to common belief, we find that using an embedded JVM does not necessarily lead to disastrous performance. As expected, a strided execution pattern effectively minimizes call overheads commonly associated with JNI calls, and the strided execution wrapper can be generated automatically by the engine making the entire process transparent to users. The necessity of strided execution diminishes as the computational load of the UDF increases.

By using Java direct byte buffers, we can pass entire data blocks from the engine to the embedded JVM. If the UDF uses primitive types in its interface, the data encapsulated in these buffers can be accessed directly without incurring an additional copy. However, Java objects in the UDF interface require the data to be copied from input buffers to the Java object's data structures, or from the Java object to output buffers, which dominates UDF execution. Consequently, primitive types should be used whenever possible.

We also show that running UDFs inside an embedded JVM compares well with UDFs hand-written in C++. JIT-compiling computationally heavy UDFs, that do not use Java objects, to machine code, can improve performance by about a factor of two. Other UDFs do not benefit from JIT-compilation to machine code because of the overheads associated with the guarantees of the Java language, namely the immutability of Strings, exception handling, and a particular memory model.

We were able to apply a number of optimization to the generated machine code because our evaluated UDFs were free of side effects. We would argue that it is good practice to use side effect-free functions for scalar UDFs in any case. However, we also advocate to integrate these optimizations inside an existing JVM's JIT compiler instead of essentially duplicating the HotSpot VM.

[3] https://www.cloudera.com/documentation/enterprise/5-8-x/topics/impala_udf.html

REFERENCES

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[2] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, Rene Mueller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam Storm, Yuanyuan Tian, Pınar Tözün, et al. 2017. Evolving Databases for New-Gen Big Data Applications. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR '07)*. www.cidrdb.org. Retrieved August 8, 2017 from http://cidrdb.org/cidr2017/papers/p123-barber-cidr17.pdf

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2014. Julia: A Fresh Approach to Numerical Computing. *Computer Research Repository (CoRR)* abs/1411.1607 (2014). Retrieved August 8, 2017 from https://arxiv.org/abs/1411.1607

[4] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly Parallel Garbage Collection. *SIGPLAN Not.* 26, 6 (May 1991), 157–164. https://doi.org/10.1145/113446.113459

[5] Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820. https://doi.org/10.1002/spe.4380180902

[6] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-pipelining Query Execution. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '05)*. www.cidrdb.org, 225–237. Retrieved August 8, 2017 from http://cidrdb.org/cidr2005/papers/P19.pdf

[7] Apache Flink Committee. [n. d.]. Apache Flink. ([n. d.]). Retrieved August 8, 2017 from https://flink.apache.org

[8] Apache Hadoop Committee. [n. d.]. Apache Hadoop. ([n. d.]). Retrieved August 8, 2017 from https://hadoop.apache.org

[9] Apache Parquet Committee. [n. d.]. Apache Parquet. ([n. d.]). Retrieved August 8, 2017 from https://parquet.apache.org

[10] Apache Spark Committee. [n. d.]. Apache Spark. ([n. d.]). Retrieved August 8, 2017 from https://spark.apache.org

[11] 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.

[12] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2015. The Java Language Specification (Java SE 8 Edition). (2015). Retrieved August 8, 2017 from https://docs.oracle.com/javase/specs/jls/se8/html/index.html

[13] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th IEEE International Conference on Data Engineering*. IEEE, 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[14] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proceedings of the 7th Biennial Conference on*

Innovative Data Systems Research (CIDR '15). www.cidrdb.org. Retrieved August 8, 2017 from http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf

[15] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1790–1801.

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[17] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2011. Dremel: Interactive Analysis of Web-scale Datasets. *Commun. ACM* 54, 6 (June 2011), 114–123. https://doi.org/10.1145/1953122.1953148

[18] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[19] Oracle Corperation. [n. d.]. Java Native Interface Specification. ([n. d.]). Retrieved August 8, 2017 from https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

[20] BugVM Project. [n. d.]. BugVM. ([n. d.]). Retrieved August 8, 2017 from https://github.com/bugvm/bugvm

[21] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proceedings of the VLDB Endowment* 6, 11 (Aug. 2013), 1080–1091.

[22] Reginald Smith. 2012. Distinct word length frequencies: distributions and symbol entropies. *Glottometrics* 23 (2012), 7–22.

[23] Arno Unkrig et al. [n. d.]. Janino Compiler. ([n. d.]). Retrieved August 8, 2017 from https://janino-compiler.github.io/janino

[24] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot – A Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM Press.

[25] T. Vincenty. 1975. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review* 23, 176 (1975), 88–93. https://doi.org/10.1179/sre.1975.23.176.88

[26] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2.