Spark Wasm UDFs

Raymond Huffman

MIT 6.858 Final Project
May 2022

1 Introduction

Apache Spark is a popular system for data processing and analytics. Spark typically runs on a cluster of machines, and enables users to process datasets that may be too large to fit on a single machine. Spark provides a simple interface for writing SQL-like queries, but also supports arbitrary user-defined functions (UDFs). UDFs are treated as black-boxes for functional operations like map, filter, and reduce. Spark has official support for Java, Scala, Python, and R, and UDFs can be implemented in any of those languages.

Consider a cloud computing provider that hosts shared Spark clusters and allows its users to submit jobs. Spark UDFs would essentially allow users to run arbitrary code on the Spark cluster, with the potential to interfere with and access the filesystems of peer jobs on the cluster. Spark does provide some security features, such as authenticated and encrypted RPCs between processes, authentication for requests to the cluster, and encryption for local temporary files. These features, however, are not sufficient to isolate a malicious job with submit permissions on the cluster. Currently, the best way to isolate Spark jobs its to spin up separate clusters, which introduces significant overhead and startup time. At a major data analytics company that uses Spark to power a customer-facing analytics product, initializing a Spark environment could take anywhere from 10 minutes to nearly an hour, depending on the dependencies required.

This project explores a novel solution for isolating untrusted Spark UDFs using WebAssembly. Specifically, it uses a generic UDF wrapper written in Java that serves as an interface between Spark RDDs and a WebAssembly runtime.

2 Design

2.1 WebAssembly

WebAssembly (Wasm) was first introduced by Mozilla in 2017, and enables programs written in any supported language to be compiled for and run in a web browser. Wasm is a binary instruction format and a compilation target for programming languages. Just as a program can be compiled for x86 or for ARM, it can be compiled for Wasm. Wasm programs can execute at native speed, making them significantly faster than equivalent programs written

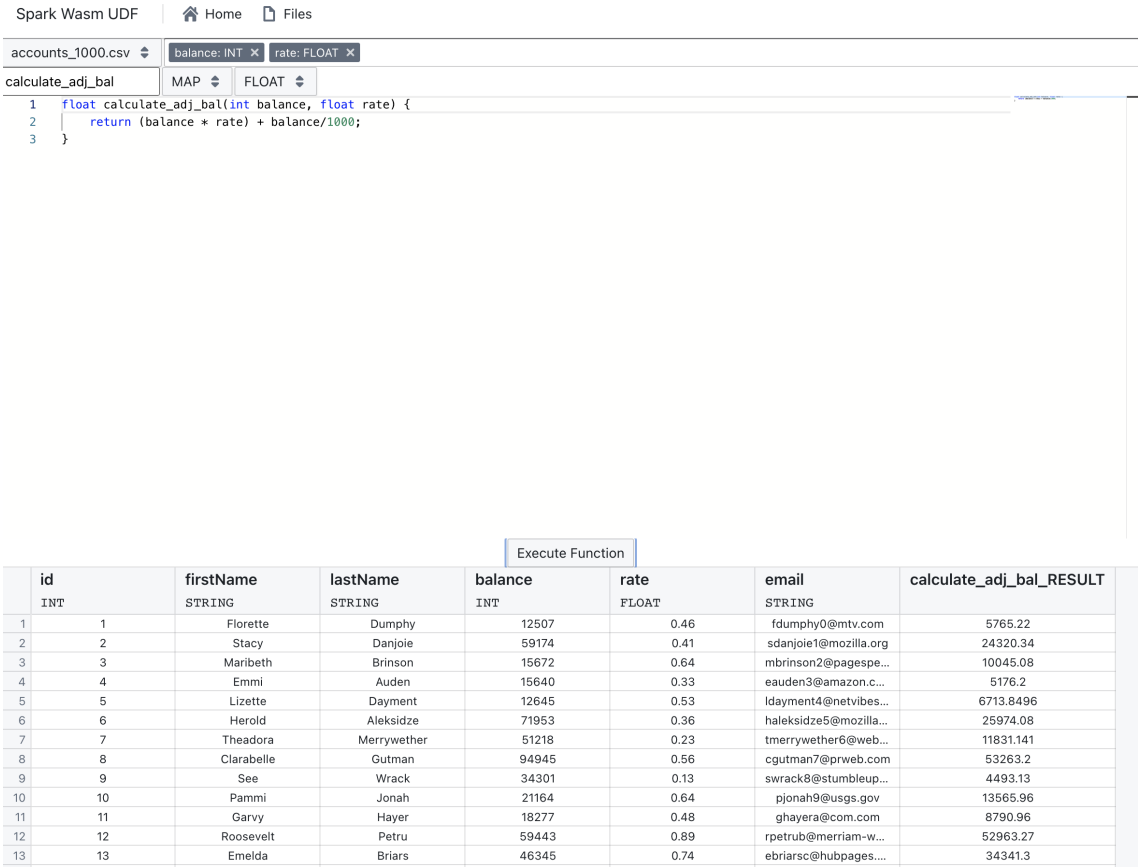
in JavaScript. Additionally, by acting as a common compiler target for any programming language, a programmer can choose the language and packages that best fit their needs.

While WebAssembly was designed primarily to run in web browsers, Wasm binaries can also be run in non-web containers. Wasmtime and Wasmer are two competing implementations of non-web runtime environments for Wasm. For this project, Wasmer was selected as it provides an officially supported package for interacting with Wasm modules. Wasmer provides an isolated execution environment that can be initialized in less than a second.

The isolation provided by the Wasmer runtime is sufficient to allow us to safely run untrusted code on a shared Spark cluster. As configured, Wasmer provides no access to the filesystem, the network, or system calls. Wasm's memory model also restricts programs from escaping Wasmer's allocation, preventing a malicious application from corrupting the memory of the Spark cluster.

2.2 Frontend

A simple frontend application was implemented using React. The user first selects a dataset from a list of uploaded files. Next, they select the columns from the dataset on which their UDF should operate. Then, they implement their function in C using an embedded code editor. Lastly, they specify the Spark return type of their function, whether the function is a Map function or a Filter function, and click "Execute Function" to submit the request to the server.



	id	firstName	lastName	balance	rate	email	calculate_adj_bal_RESULT
	INT	STRING	STRING	INT	FLOAT	STRING	
1	1	Florette	Dumphy	12507	0.46	fdumphy0@mtv.com	5765.22
2	2	Stacy	Danjoie	59174	0.41	sdanjoie1@mozilla.org	24320.34
3	3	Maribeth	Brinson	15672	0.64	mbrinson2@pagespe...	10045.08
4	4	Emmi	Auden	15640	0.33	eauden3@amazon.c...	5176.2
5	5	Lizette	Dayment	12645	0.53	ldayment4@netvibes...	6713.8496
6	6	Herold	Aleksidze	71953	0.36	haleksidze5@mozilla...	25974.08
7	7	Theadora	Merrywether	51218	0.23	tmerrywether6@web...	11831.141
8	8	Clarabelle	Gutman	94945	0.56	cgutman7@prweb.com	53263.2
9	9	See	Wrack	34301	0.13	swrack8@stumbleup...	4493.13
10	10	Pammi	Jonah	21164	0.64	pjonah9@usgs.gov	13565.96
11	11	Garvy	Hayer	18277	0.48	ghayera@com.com	8790.96
12	12	Roosevelt	Petru	59443	0.89	rpetrub@merriam-w...	52963.27
13	13	Emelda	Briars	46345	0.74	ebriarsc@hubpages....	34341.3

2.3 Backend

2.3.1 Web Server

The frontend is served by a Flask web server. It serves requests from the frontend, including listing the available datasets, getting the schema of a dataset, executing a UDF, and retrieving the results. When the user clicks "Execute Function," an ExecutionRequest is created.

```
interface IExecutionRequest {
  program: string; // text of c program
  data: string;  // name of input file
  operation: "MAP" | "FILTER";
  function_name: string;
  input_column_names: string[];
  output_column_name?: string;
  output_column_type?: "INTEGER" | "FLOAT" | "BOOLEAN" | "STRING";
}
```

The program is written to disk on the server, and then compiled to a Wasm module using Emscripten. Then, the Spark Runner program is executed.

2.3.2 Spark Runner

The Spark runner is implemented in Java and is responsible for setting up a Spark session, loading the Wasm module, and running the Wasm module as a UDF. For this initial implementation, a local Spark session was used, but the initialization logic could easily be modified to connect to a Kubernetes cluster or other Spark environment.

Once the Spark session is initialized, the Wasmer JNI is used to load the freshly compiled Wasm module, creating an object of type `org.wasmer.Instance`. This instance must then be passed to the Spark executor to be used in a UDF. Unfortunately, a Wasmer Instance is not serializable, so it cannot be passed directly to Spark.

One option is to use Wasmer's serialization feature, which serializes the module to a `byte[]`, which can be passed around by Spark. However, this byte array must be re-instantiated within the UDF, meaning that a new `org.wasmer.Instance` is created for each row. This worked with small datasets but quickly caused out-of-memory exceptions when used with larger datasets.

```
Module module = new Module(Files.readAllBytes("module.wasm"));
byte[] wasmBytes = module.serialize();
Instance instance = Module.deserialize(wasmBytes).instantiate();
```

An alternative approach is to create a wrapper class that holds the `org.wasmer.Instance` as a static member. With this solution, the instance must be created once per JVM (once per cluster node) instead of once per row, a significant improvement.

```
public class InstanceWrapper {
    private static Instance instance;
    public static Instance get(){
        return instance;
    }
    public static void set(Instance instance) {
        InstanceWrapper.instance = instance;
    }
}
```

```
Module module = new Module(Files.readAllBytes("module.wasm"));
InstanceWrapper.set(module.instantiate());
```

Next, the `UserDefinedFunction` is created, which makes a call to the Wasmer Instance.

```
UserDefinedFunction myUdf = org.apache.spark.sql.functions.udf(
   (a1) -> {
    return InstanceWrapper
                .get()
                .exports
                .getFunction(functionName)
                .apply(a1)[0];
   });
```

Last, the UDF is invoked on the Spark dataset, creating a new column when using a Map function, or filtering the rows when using a Filter function.

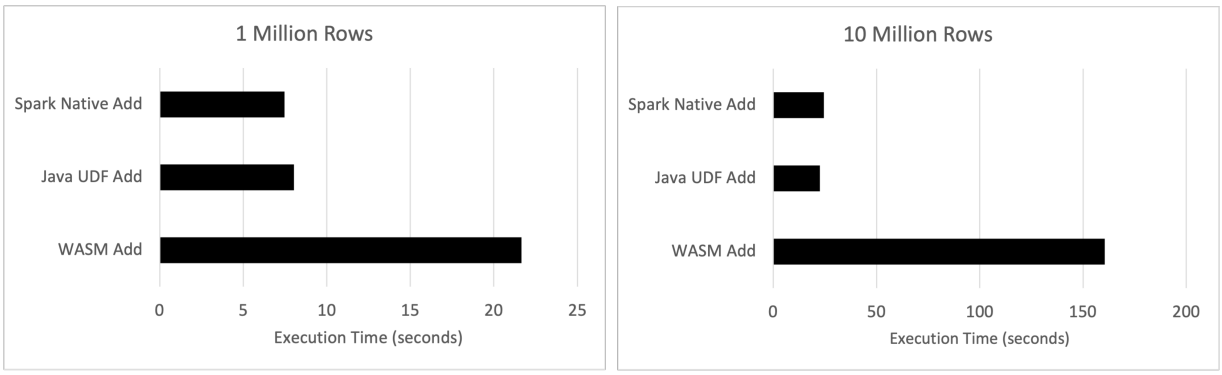
```
dataset.select(
   col("*"),
   myUdf.apply(columns(inputColumnNames)).as("RESULT"));
```

```
dataset
   .select(col("*"))
   .filter(myUdf.apply(columns(inputColumnNames)))
```

3 Results

The current implementation of this system works as described for any numeric Map or Filter function. The compilation of the Wasm module and instantiation of the Wasmer runtime do not add significant overhead to the processing of the Spark job. However, using a Wasm UDF did introduce a significant increase in processing time when compared to a native

implementation of the same operation. On a 1 million row dataset, the Wasm UDF took 2.5 times longer to execute, and on a 10 million row dataset, the Wasm UDF took 7 times longer. These experiments were run with Spark in local mode on a 24-core machine with 96GB of RAM.



4 Conclusion

This project has successfully demonstrated that it is feasible to use a Wasm runtime to safely execute untrusted code as a Spark User-defined function. While the calls to Wasmer do introduce significant overhead, this increase in execution time may be acceptable if it enables operations that would be otherwise impossible using native Spark, or if the increased execution time is outweighed by the significantly decreased initialization time. For ad-hoc analytics workflows, a decreased initialization time may be preferred as it allows users to more quickly compute their first results.

5 Future Work

This implementation does not handle strings or more complex data types. Support for strings could be added by calling the Wasm module's `malloc` function and copying a string from Java memory to Wasm memory.

This implementation has not been tested on a multi-node Spark cluster. Further experimentation should be done on a more realistic Spark cluster, and some modifications to the initialization code will be necessary to properly load the Wasmer Instance object.