Improving PySpark Performance with Cross-Language Optimization

by

Linh-Nga Tran

Electronics, Bachelor of Science
Rhine-Waal University of Applied Sciences, 2018

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	David Bremner, PhD, Computer Science
	Suprio Ray, PhD, Computer Science
Examining Board:	Paul G. Plöger, PhD, Computer Science (Bonn Rhein-Sieg University of Applied Sciences)
	Michael W. Fleming, PhD, Computer Science
	Scott Underwood, PhD, Business

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

August, 2021

Abstract

Big data is a rapidly growing field, and Apache Spark is one of the most commonly used frameworks in this area. Among its APIs, Spark for Python, or PySpark is usually the preferred choice by the data scientist community due to its simplicity and versatility. PySpark is built upon Java Spark, and operates in two runtimes, Python and Java Virtual Machine. However, PySpark is often outperformed by its alternatives in various applications.

From previous experiments, the bottleneck in PySpark is identified as the data marshalling process across the language boundary between Python and Java, specifically in the serialization and deserialization process. This project aims to alleviate this issue by implementing a specialized serializer for PySpark.

The specialized serializer is first built in C++. In addition, to allow for the serialization and deserialization code to be generated dynamically depending on the schema of the dataset, Just-in-Time (JIT) technology is utilized. To be more specific, OMR JitBuilder is used to compile the code at runtime depending on user input. Furthermore, the compiled code can then be assigned to normal functions and be re-used multiple times, hence saving compilation and binding effort.

After building the serializer in C++, the wrappers for Python and Java are implemented. On the Python side, to transfer data between C++ and the source language, ctypes and a header file from the Python development package, Python.h, are used. The ctypes library allows communication from Python to C++, and the header file

enables C++ to access and modify Python objects. On the Java side, Java Native Interface, a tool for Java to pass data and make native calls, is used.

This JitBuilder serializer is then evaluated against other existing serializers, in particular pickle, BSON, and Protocol Buffers. The dataset used is the Part table from the TPC-H benchmark, and it has scale factors of 1, 2, 4, and 8, corresponding to 24.1, 48.4, 96.9, and 194.4 MB. The serializers are tested in different scenarios, which are serialization and deserialization only in the Python runtime, serialization and deserialization only in the JVM, and finally serialization in one language and deserialization of that data in the other language. The results show that this serializer has competitive performance, and in some cases outperforms the other options.

After implementing and evaluating the JitBuilder serializer with the wrappers for Python and Java, a prototype is built where it is integrated into PySpark. To do this, the function calls to serialize and deserialize data in Spark source code are modified. This modified PySpark is shown to have better performance than the original PySpark in common data applications.

Acknowledgements

This research was conducted within the Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick, and the Department of Computer Science, Hochschule Bonn-Rhein-Sieg.

The authors are grateful for the colleagues and facilities of CAS–Atlantic and Hochschule Bonn-Rhein-Sieg in supporting our research. The authors would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

In addition, I wish to sincerely thank my supervisors, Professor David Bremner, Professor Suprio Ray, and my advisor Professor Paul G. Plöger. They have guided me through the project, and provided much needed encouragement as well as constructive criticism.

I want to thank Dr. Mark Stoodley, Mr. Daryl Maier, and Mr. Deverne Jones for always being there to answer many of my questions.

Lastly, I thank my family and friends for support and encouragement.

Table of Contents

Vita

List of Tables

List of Figures

Chapter 1

Introduction

Big data is a rapidly growing field due to the large amount of data generated from various sources, especially from the media, the Internet of Things, and the Web. Among the frameworks developed to efficiently handle big data, Apache Spark is most commonly used. Among the supported APIs, PySpark is usually the most preferred.

Despite its popularity, PySpark is consistently outperformed by its alternatives such as other Spark APIs or other big data frameworks, in various fields. The bottleneck is identified as the data serialization and deserialization process across the language boundary between Python and Java [24]. Therefore, the goal of this research is to improve PySpark's performance by tackling this serialization and deserialization process.

1.1 Motivation

Apache Spark is commonly used to handle big data because it has low latency by allowing in-memory computing [14]. This framework is written in Scala and runs on a Java Virtual Machine (JVM). Spark offers many cross-language APIs, allowing users to write Spark code in Scala, Python, Java, or R.

Among the APIs provided, Spark for Python, or PySpark is usually the preferred choice due to its flexibility and Python's popularity in the data science community [1]. It is built on top of the Spark API for Java, therefore capable of running in both Python and the JVM runtime.

However, PySpark is significantly slower compared to other Spark APIs. For example, Figure 1.1 shows that it is approximately 30 times slower than Scala Spark for training the ridge regression application. By profiling PySpark performance through various experiments [24], the results indicate that the bottleneck lies in the serialization and deserialization across the language boundary between Python and the JVM runtime. When data is transferred between the two runtimes, it needs to be serialized in one runtime, sent to another process or node in the cluster, usually over the network, and deserialized on the other runtime. This generates excessive overhead in term of execution time of the application. This motivates the idea of improving PySpark's performance by tackling the language interoperability issue.

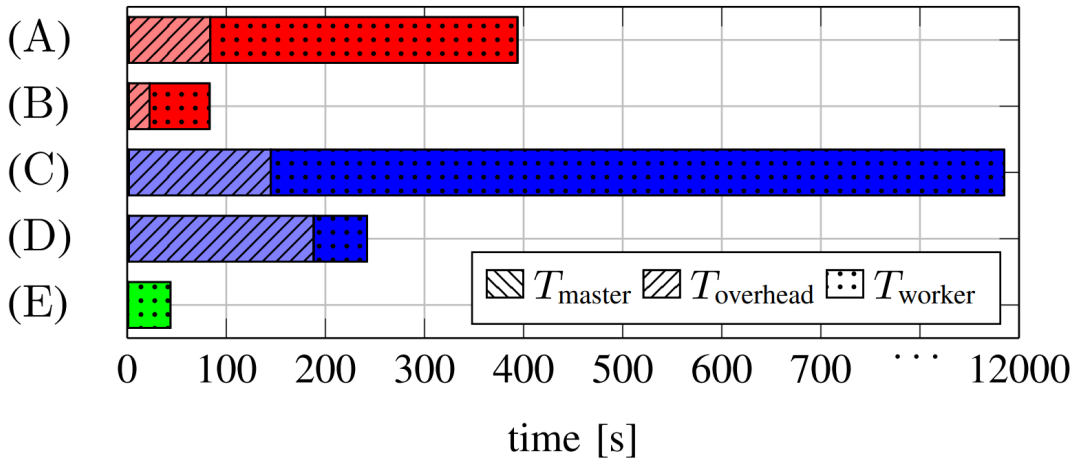

Figure 1.1: Execution time of (A) Scala Spark, (B) Scala Spark with C++, (C) PySpark, (D) PySpark with C++, and (E) MPI in C++ implementations of the COCOA algorithm for training the ridge regression on *webspam* dataset [5]

1.2 Challenges and Difficulties

Since this is an open-ended issue, there are many difficulties faced during the process. The first challenge is to find possible approaches. These options are then evaluated to find the most suitable and reasonable option for this project. Next, the components to implement the selected option need to be evaluated and chosen. Finally, the proposed work is integrated into PySpark. However, Spark is a big library with many components. Therefore, it is a challenge to correctly pinpoint the location of the code that needs to be replaced and accurately replace it with the new serializer.

1.2.1 Choosing an approach to improve PySpark's performance

There are various approaches to handle this language interoperability issue. The first obvious method is to replace pickle [10], which is the current PySpark serializer, with a better serialization library. However, this is not an easy task. The serializers that are flexible and can handle most data types such as JSON are not efficient. On the other hand, the ones that are robust and have a small serialized data size like Protocol Buffers usually require knowing the data schema before compiling and running the application.

Another option is to optimize the program data flow. Currently, PySpark is already highly optimized on the JVM side. However, because this framework also runs in the Python runtime, the cross-language data-flow is not fully investigated. One way to utilize this is to investigate the full program data-flow across Python and the JVM runtime, and find caching opportunities to reduce computation time.

However, both of these options do not suffice. Integrating an existing, yet more efficient, serialization method typically requires a schema that is known beforehand. The cross-language approach seems promising, but when tested, the results did not

yield an improvement.

Nevertheless, the approach of replacing pickle serializer inspires an idea of building a specialized serializer for PySpark. This serializer needs to be more efficient than pickle, and it needs to automatically infer the schema of the data to best serialize it.

1.2.2 Choosing components to build a prototype

From the idea of building a specialized serializer for PySpark, there are a few areas that need to be considered. First, the new serializer needs to be faster than pickle. Currently, pickle is a library written in C. To build a more efficient serializer, the best way is to also use native code, which means C or C++ to implement it. This is because native code is compiled prior to runtime, allowing it to be further optimized by the compiler, making it more efficient. The reason why C++ is chosen over C is because another component for this specialized serializer, OMR JitBuilder, provides documentation and API only in C++.

This leads to a few more challenges, the first one being choosing between using an existing and efficient C++ serializer, or building a serializer from scratch. An example of an existing serializer, bitsery [26] appears to have promising results, as the evaluation shows that it has better performance compared to many other efficient serializers [27]. However, this serializer also requires knowing the schema of the data beforehand. Therefore, the option of building a specialized serializer is chosen, so that it could fulfill these two requirements:

- Being more efficient than pickle.
- Not requiring the user to know the data schema.

  This is important because in PySpark, as the data get passed around between processes and nodes, its format keeps changing. Therefore, the serializer needs to automatically detect the new schema and adapt accordingly.

Choosing to implement a C++ serializer raises an issue that the type of data needs to be known before compile time. It is possible to build an elaborate serializer that handles the general case by implementing an algorithm to traverse the object structure. However, this makes the serializer behave exactly like pickle, therefore not much improvement can be gained. Hence, the specialized serializer is implemented so that it is able to dynamically generate serialization and deserialization code based on the schema of the input data. OMR JitBuilder is selected for this dynamic code generation task. The usage of JitBuilder also justifies the decision of choosing C++ over C, as most of the documentation and examples for this framework are in C++. Once the specialized serializer is built, the wrappers for Python and JVM languages, which could be either Scala or Java, need to be chosen. After some research into the advantages and disadvantages of each language binding, ctypes and JNI are chosen to be the wrapper for Python and JVM respectively, because of its simplicity and efficiency.

Between choosing the JVM language, Java is chosen over Scala to implement the JVM side of the specialized serializer. Even though Scala has the advantage of being the language that most of Spark source code is written in, it has a steep learning curve and not as documented and supported by the community as Java. Therefore, for this project, Java is selected. In the integration stage into PySpark, the Scala source code of Spark can still directly call and access methods from this Java code.

1.2.3 Integrating with PySpark

The last difficulty in this project is to integrate the specialized serializer into PySpark. PySpark is a large library with over 100,000 lines of code, containing multiple components. The first challenge here is to locate the files containing the serialization and deserialization code on both Python and Scala. In addition, PySpark is not a well documented API, with its last update on documentation being in 2016 with

many missing subsections [20].

Once the serializer files are located, the JitBuilder serializer has to be carefully placed into the source code. Moreover, some minor adjustments are needed to make this serializer compatible with Spark. Specifically in PySpark, pickle is used to serialize data as well as functions and commands. Therefore, there needs to be some checks to make sure that only the actual data is passed into the JitBuilder serializer, while the commands are still passed to pickle. Furthermore, the specialized serializer is written in Python and Java, but PySpark is written in Python and Scala. Hence, the Scala source code is modified so that it passes data with the correct type to the Java wrapper for the specialized serializer.

1.3 Problem Statement

PySpark is a popular framework for handling big data. However, it is quite slow compared to its alternatives. From previous works, the bottleneck in the framework's performance lies in the cross-language process, specifically in the serialization and deserialization step. To alleviate this issue, a few methods are suggested. After some initial investigations, a solution is proposed, which is to build a specialized serializer in C++ using JitBuilder. A prototype of this serializer being integrated into PySpark is implemented, in the hope to improve PySpark performance.

Chapter 2

Related work

To alleviate the bottleneck in PySpark by building a specialized serializer, a few solutions are suggested, which are analyzing the data-flow of PySpark application across the language boundary, or improving its serialization and deserialization process. This chapter discusses each of these options and their applicability to the scope of this project.

2.1 Cross-language data-flow analysis

One of the possible approaches to enhance PySpark performance is to study the data-flow of the application across the language boundary between Python and the JVM run time. An example that follows a similar approach is Gerenuk [18]. In this project, Navasca et al. represent the data objects within a Spark application by inlining their payloads in native bytes. Furthermore, they also use the natural flow of data to identify statements that need to be transformed so that they can directly operate on these inlined objects.

Figure 2.1 [18] shows the result of the execution time of Scala Spark and Gerenuk. On average, Gerenuk is 1.96× faster than Scala Spark. However, there are a few

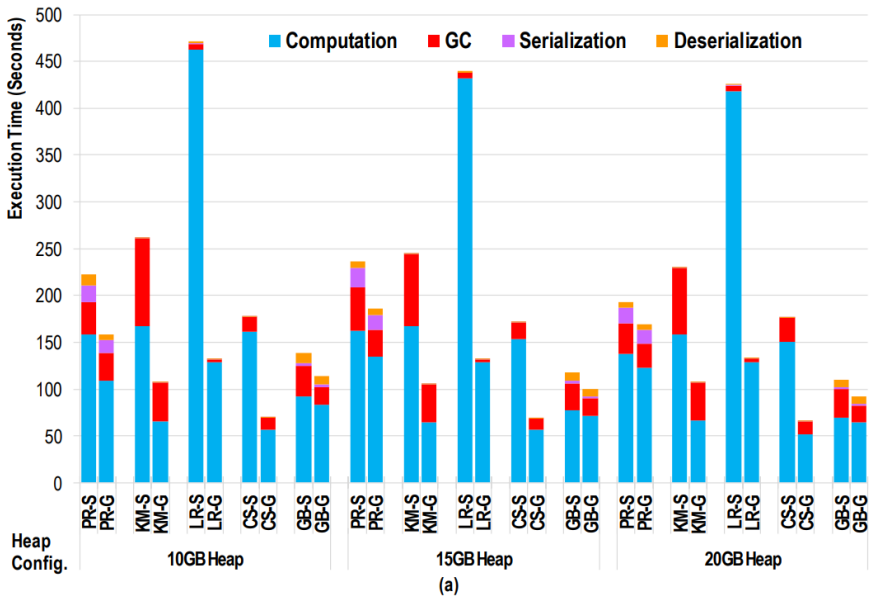

Figure 2.1: Execution time of Scala Spark (the left bar in each pair) against Gerenuk (the right bar in each pair) implementation [18]

issues with this approach. First, the user needs to manually identify the start and end point of the data flow. This might be difficult in practice, since most of the people using Spark do not have an in-depth understanding of the system. Secondly, Gerenuk only operates within a single runtime, which is the JVM. However, PySpark is more complicated, as it runs in both Python and Java runtimes.

Nevertheless, the results from Gerenuk suggest that the data-flow analysis can possibly be utilized to enhance the performance of PySpark, even in a cross-language boundary setting. This idea seems promising, but when tested, the result did not yield an improvement. For example, a small test application is performed, in which the inputs are two datasets, each with 10 rows, generated according to Sanzu [28]. The inputs are then joined and sorted. Figure 2.2 shows the data-flow of this application in the Python runtime. Here, no Resilient Distributed Datasets (RDDs), which is a data type in Spark, are repeated. This trend is observed even for bigger datasets. This means that there is no visible caching opportunity, hence no possible improvement can be made in terms of execution time.

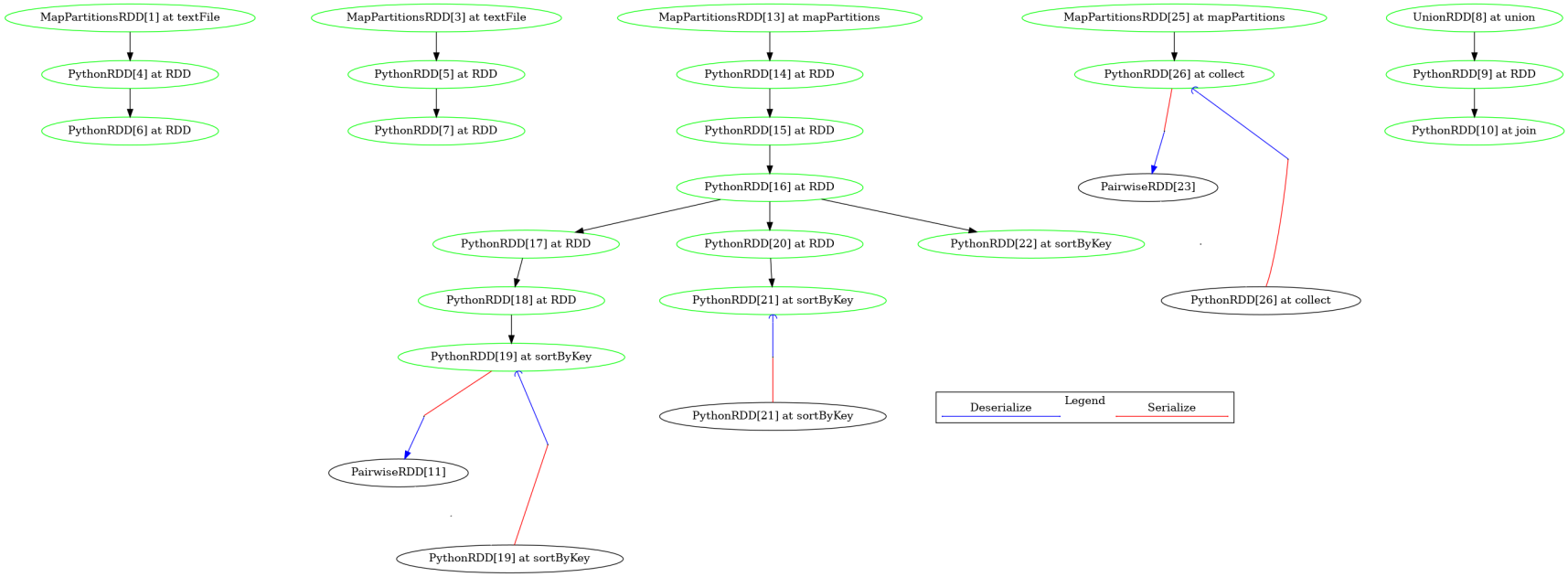

Figure 2.2: Data-flow in the Python runtime of a PySpark application to join and sort datasets

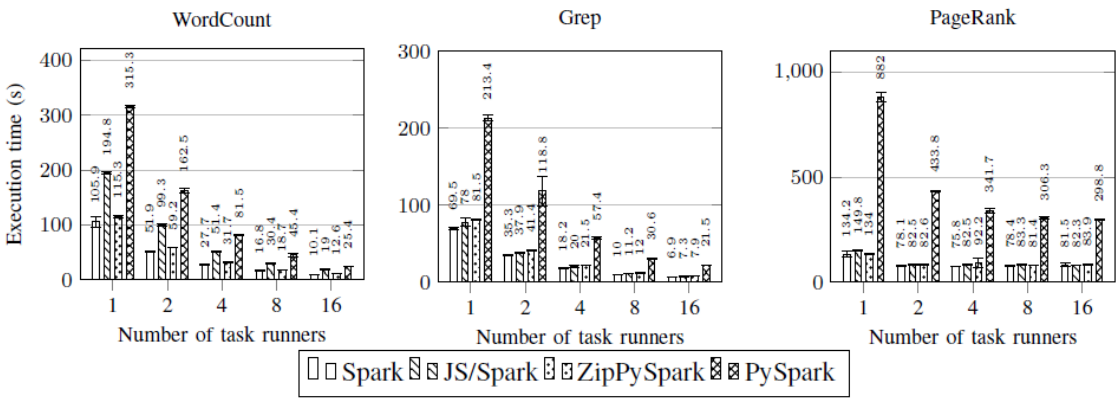

Figure 2.3: Execution times of various tasks for different number of task runners for Spark, Salucci's implementation — JS/Spark and ZipPySpark, and PySpark [21]

There has been another attempt to improve PySpark's performance by eliminating the cross-language boundary issue and putting the entire data-flow in a JVM. This project carried out by Salucci et al. [21] removes the Python runtime by using ZipPy, which is a Java-based implementation of Python. With this, all of PySpark processes now run within a single shared JVM. The results of this work, which is called ZipPySpark, are shown in Figure 2.3.

Even though ZipPySpark has considerable improvement compared to PySpark, there are still some issues with this approach. First, it is unclear if the performance gain comes from removing the cross-language component, or from the optimizations done by Truffle and Graal. (Truffle is a framework to develop high-performance runtimes on the Java Virtual machine, and Graal is a dynamic compiler used to generate highly efficient machine code). The next problem is that by using ZipPy, ZipPySpark can no longer utilize many popular Python libraries such as NumPy or Pandas, because they are all written in C. This significantly limits the usability of this PySpark version.

2.2 Existing serializers

Nowadays, there are multiple off-the-shelf efficient libraries for serializing data. The following subsections examine the details of some of the most popular serializers and

their capability.

2.2.1 Pickle

The first serializer that is evaluated is pickle. It is the standard library in Python for serialization and deserialization, and was invented in 1995 [8]. It is the current option used in PySpark. There are two versions of this serializer, pickle and cPickle. They actually implement the same algorithms, but the former is implemented in Python, and latter is implemented in C, making cPickle much faster. In Python 3, cPickle is the default selection when the library is imported. In this project, whenever pickle is mentioned, it actually refers to the C implementation.

```
>>> import pickle
>>> import pickletools
>>> pickle.dumps([4,5,6,7])
b'\x80\x04\x95\r\x00\x00\x00\x00\x00\x00\x00]\x94(K\x04K\x05K\x06K\
   x07e.'
>>> pickletools.dis(_)
0: \x80 PROTO      4
2: \x95 FRAME      13
11: ]    EMPTY_LIST
12: \x94 MEMOIZE    (as 0)
13: (    MARK
14: K        BININT1    4
16: K        BININT1    5
18: K        BININT1    6
20: K        BININT1    7
22: e        APPENDS    (MARK at 13)
23: .    STOP
```

```
highest protocol among opcodes = 4
```

Listing 2.1: Python 3.8 pickle example

This library is one of the more versatile serializers, as not only it can serialize data, but also functions, classes and the states of class instances. In pickle, there are various levels of protocols that can be used for serializing [10].

This serializing library uses two areas to store and interact with data, which are the memo and the stack. The memo is for long-term storage, while the stack is for short-term. The example in Listing 2.1 further illustrates how pickle works. The serialized data contains a set of opcodes, with each opcode followed by an argument. Here, the opcode PROTO and its argument indicate the protocol level used by pickle. In this example, Python 3.8 is used, therefore the default protocol version is 4. FRAME is then used to indicate the beginning of a new frame. Next, the opcode EMPTY_LIST marks the start of an empty Python list and pushes it on the current stack. MEMOIZE then stores the current stack into the memo, with the location of the memo being the corresponding argument. The MARK opcode then marks the start of the list on the stack. Next, the BININT1 opcode parses the binary representation of an integer and pushes it onto the current stack. The pushing operation is stopped when the opcode STOP is reached.

This format is specific for Python, which means that other languages such as Java or Scala are not able to reconstruct it natively. However, there is an implementation of this same serializer for the JVM languages [4] and it can be installed using Maven, which is a build tool for Java projects, as pip is for Python [12].

2.2.2 Protocol Buffers

Protocol Buffers, or protobuf, is developed by Google, and is one of the most popular serializers in the market due to its independence from language and platform and its simplicity and efficiency [13]. This library currently supports Python, Java, C++, Go, Ruby, and many other languages. However, protobuf's biggest downside is that it requires a schema, which needs to be defined and compiled beforehand. In addition, this schema needs to be available and consistent on all machines to ensure correct serialization and deserialization.

```
syntax = ‘‘proto2’’;
message Part {
        required int32 partID = 1;
        required string name = 2;
        required float retailPrice = 3;
        required string comment = 4;
}
```

Listing 2.2: Proto file example

To use Protocol Buffers, first, the data structure needs to be described in a proto file, such as the example shown in Listing 2.2. In this file, the schema is defined with the data type, the field name, and a unique tag for each variable. For each field, a modifier needs to be defined, with three possible values: *optional*, *repeated*, and *required*. While the modifiers *optional* and *required* are self-explanatory; the *repeated* modifier indicates that this specific field could be repeated any number of times, and the repeating order is preserved. Afterwards, this proto file needs to be compiled with protoc. This process generates code that can be invoked to store and load data later.

This library serializes data into a binary wire format. The serializer uses the infor-

mation compiled by protoc to know how many bytes are needed to store the data, and what format to correctly load it in. This makes the size of the serialized data generated by Protocol Buffer very compact. Figure 2.4 shows an example of how the Protocol Buffers library packs the data with the format shown in Listing 2.3.

```
{
        "userName": "Martin",
        "favoriteNumber": 1337,
        "interests": ["daydreaming", "hacking"]
}
```

Listing 2.3: Data to be serialized [15]

This library uses the field tag and data type defined in the proto file to efficiently pack the order and the type of the data so that the result is only 33 bytes. For this same data, both BSON and pickle require 96 bytes.

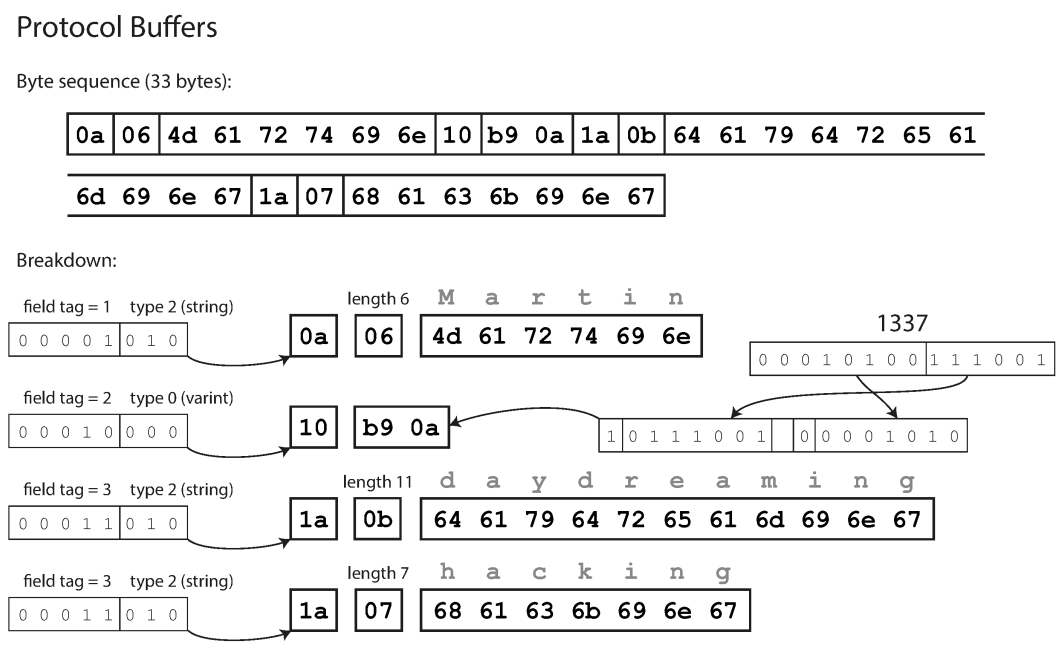

Figure 2.4: Protocol Buffer serialized data for listing 2.3 [15]

2.2.3 BSON

In the scope of this project, JSON is not considered. This is because even though JSON is very popular and versatile, the size of the generated serialized data is too large, and it takes too much time for serialization and deserialization. Therefore, a more efficient version of JSON is investigated, which is BSON. BSON, or Binary JSON, is very similar to JSON. It also supports various languages, some of them being Scala, Java, and Python. This already gives it an advantage over pickle.

Unlike protobuf, BSON does not require a set schema beforehand, therefore it is slightly more flexible. However, it cannot serialize any object like pickle, as it can only serialize key/value pairs.

```
\x16\x00\x00\x00
\x02
hello\x00
\x06\x00\x00\x00world\x00
\x00
```

Listing 2.4: BSON serialized data example

For example, if the BSON serializer receives the data input { *"hello":"world"*}, it would generate the serialized hexadecimal data as shown in Listing 2.4. The first four bytes indicate the total size of the document, which translates to 22 in decimal. The next byte represents the type of the value in the key/value pair, which is type String. The key is shown in the following 6 bytes. The value in the key/value pair is then serialized into a byte array containing the size of the value, the actual value itself, and the null terminator. Finally, at the end of the object, another null terminator is placed to indicate the stopping point. One thing to note here is that the key always need to be a string. Even if the key passed in has another data type, BSON automatically converts it to string type.

Library	Serialize time (ms)	Deserialize time (ms)
bitsery	959	927
boost	9826	8313
cereal	6324	5698
flatbuffers	5129	2142
handwritten	1023	882
protobuf	11966	13919
yas	1908	1217

Table 2.1: Bitsery serializer benchmark [27]

2.2.4 Bitsery

The previous options either officially or unofficially support various languages. However, for the purpose of finding a solution that can at least match pickle performance, some C++ serializers are also investigated. Among the available options, bitsery is one of the more promising ones.

The library is a cross-platform compatible serializer developed by Vinkelis in 2017 [26]. It is quite efficient, and generates compact serialized data. In Table 2.1 [27], the bitsery serializer outperforms most of the other options, and has a competitive result with the handwritten C++ code to pack and unpack serialized data. The results from this table is from a benchmark developed by Vinkelis, with randomly generated data to represent more realistic data format with more complicated nesting relationships. After the data is generated, all the serializers perform serialization and deserialization multiple times on the same objects.

Part of the reason for the speed of bitsery is that it asks the user to pass in a vector to store the serialized data, hence the serializer itself does not need to do any allocations. Bitsery then outputs the data by copying from the source memory address to the target address.

Another reason why this serializer is so fast is that it requires knowing the data structure and data type beforehand. Therefore, the serialized data size is compact,

and it does not need to do any type checking, hence saving execution time. An example of a user-defined structure needed for this serializer can be seen in Listing 2.5. In this example, MyStruct is the format of the input data, and the serialize template function defines how the library bitsery should serialize the format MyStruct.

```
struct MyStruct {
        uint32_t i;
        char str[6];
        std::vector<float> fs;
};


template <typename S>
void serialize(S& s, MyStruct& o) {
        s.value4b(o.i);
        s.text1b(o.str);
        s.container4b(o.fs, 100);
}
```

Listing 2.5: Bitsery user-defined structure example [26]

2.3 JitBuilder

The Eclipse OMR project is an ecosystem which contains multiple components, all for the purpose of building reliable and high performance runtimes [7]. This framework is mostly written in C and C++. Some of the components included are *gc* — a garbage collection framework, *compiler* — a tool to build compiler technology, *JitBuilder* — a collection of components to dynamically generate code.

Within the scope of this project, JitBuilder is the only component that we focus on. It is a simplified interface to utilize OMR Just-in-Time (JIT) compiler technology

for dynamically generating native code. This is done via simple API calls that abstract the details of the underlying Intermediate Language (IL) used by the OMR JIT compiler. Therefore, the solution it provides is simple and elegant, yet very efficient [17].

The example in Listing 2.6 shows how JitBuilder is used to define and generate a function. In the class *AtomicInt32Add*, the constructor takes in an argument of type *TypeDictionary*, which is a JitBuilder component containing all the data types that it supports. Within this method, it defines the name of the generated function within the scope of JitBuilder, as well as the function's argument and return type, which are both pointers to a 32-bit integer.

```
AtomicInt32Add::AtomicInt32Add(OMR::JitBuilder::TypeDictionary *d)
: OMR::JitBuilder::MethodBuilder(d) {
        DefineLine(LINETOSTR(__LINE__));
        DefineFile(__FILE__);

        DefineName("increment");
        pInt32=d->PointerTo(Int32);
        DefineParameter("addressOfValue", pInt32);
        DefineReturnType(Int32);
}
```

Listing 2.6: Example of JitBuilder constructor to generate code to increment 32-bit integer [6]

In the method *buildIL* of the same class *AtomicInt32Add* shown in Listing 2.7, the JitBuilder code first loads the memory address of the value from the parameter. It then generates the code to perform an atomic add, by incrementing the integer value stored at that address by one. Finally, to return the incremented value, the generated code first loads the address of the value, then loads the actual data with

the data type Int32 at that address, and return that data.

```
bool
AtomicInt32Add::buildIL() {
        IlValue *value = IndexAt(pInt32,
        Load("addressOfValue"), ConstInt32(0));
        AtomicAdd(value, ConstInt32(1));
        Return(LoadAt(pInt32, Load("addressOfValue")));

        return true;
}
```

Listing 2.7: Example of JitBuilder method to generate code to increment 32-bit integer [6]

2.4 Language binding

Since the proposed solution is to implement a serializer that can at least match pickle's performance, and since pickle is written in C, the easiest approach is to develop this specialized serializer in C or C++. The reason why the library can also be written in C++ is because C is a subset of C++, and the two languages have comparable performance.

However, PySpark is written in Python and Scala. Nevertheless, the Scala codes are then compiled into Java byte code. Furthermore, Scala is compatible with Java. Because of these reasons, there is a need to investigate possible methods to communicate data across the language boundary, specifically between Python and C++, and between Java and C++.

2.4.1 Python

Since the reference implementation of Python is written in C, there are many Python bindings developed to support Python interacting with C and C++. From C++ to Python, the best solution is to use Python.h, which is a header file from the Python package. This file is installed when the developer version of Python, python-dev, is installed on the system. One thing to note here is that this package is specific to the Ubuntu operating system. With this, C++ can easily access Python objects via simple native calls [11].

In Listing 2.8, the C++ function takes an argument of type PyObject representing a Python list, and a C++ integer. The function then converts the C++ integer into a Python object, then appends it to the Python list. If the operation is successful, the function returns Python Boolean true, else it returns Python Boolean false.

```
static PyObject * appendPythonList(PyObject *pyList, int val){
        int success = PyList_Append(pyList, PyLong_FromLong(val));
        if (success == -1) {
                Py_RETURN_FALSE;
        }
        Py_RETURN_TRUE;
}
```

Listing 2.8: Example of C++ function appending a C++ element to a Python list

The communication from C++ to Python is done quite easily using this header file. Even though it is not too challenging, but the communication from Python to C++ is slightly more complicated than the other way. Therefore, there have been many libraries developed to support this process.

2.4.1.1 ctypes

The simplest Python binding for C++ is ctypes. This is a standard library for Python, therefore it does not require additional installation or setup. The ctypes library supports low-level control to communicate between Python and C++ by providing native data types as well as allowing calls to C++ functions from Python code [9].

An example of Python calling the C++ function created in Listing 2.8 is shown in Listing 2.9. First, Python uses ctypes to load the shared object, then it can directly access all the functions available in C++ code. However, there is one caveat, that is the argument type and the return type of the C++ function used need to be defined. This is important especially in the case of working with C++ pointers, or else data could easily be lost during the transferring process between these two languages.

This library has an advantage that it is already built into Python, so there is no extra effort needed to install it. In addition, it is very simple and easy to use. And finally, since it is official a part of Python, its documentation is very extensive and well-supported.

```
libname = pathlib.Path().absolute() / "libnative.so"
c_lib = cdll.LoadLibrary(libname)
append_C = c_lib.appendPythonList
append_C.argtypes = [py_object, c_int]
append_C.restype = py_object


py_list = []
success = append_C(py_list, c_int(42))
```

Listing 2.9: Example of Python using ctypes to call a C++ function to append a C++-type value to a Python list

2.4.1.2 Cython

Another very popular option for Python to communicate with C++ is Cython. Cython takes a completely different approach compared to the ctypes library. It uses a Python-like language to define the bindings that the Python code needs. It then generates native code and compiles it into a module.

The code in Listing 2.10 shows how Cython does Python binding. The language Cython uses for this purpose is a combination of C, C++ and Python. The first two lines are used to inform Cython where to look for the native function declaration. The last two lines are normal Python code defining a function that has access to the C++ function.

```
cdef extern from "cLib.hpp":
    float mul_C(int a, float b)


def mul_Py(a, b):
    return mul_C(a, b)
```

Listing 2.10: Example Cython Python-like language (pyx) to create binding

However, this is not the end, as this code still needs to be built and compiled. First, this pyx file, which is the file containing the Python-esque code, is run with Cython to generate a cpp file, which contains the equivalent C++ code. From here, this cpp file is compiled to create a static object that Cython can directly access.

Cython is complex library, hence allowing for high-level concepts such as class and polymorphic subtype. However, the Python-like language is quite difficult at first to figure out where to put the Python and C++ code, as they are fully intertwined in the Python-esque language. In addition, this high-level concept is not needed in this application, as the JitBuilder serializer only needs to make simple function calls across language boundary.

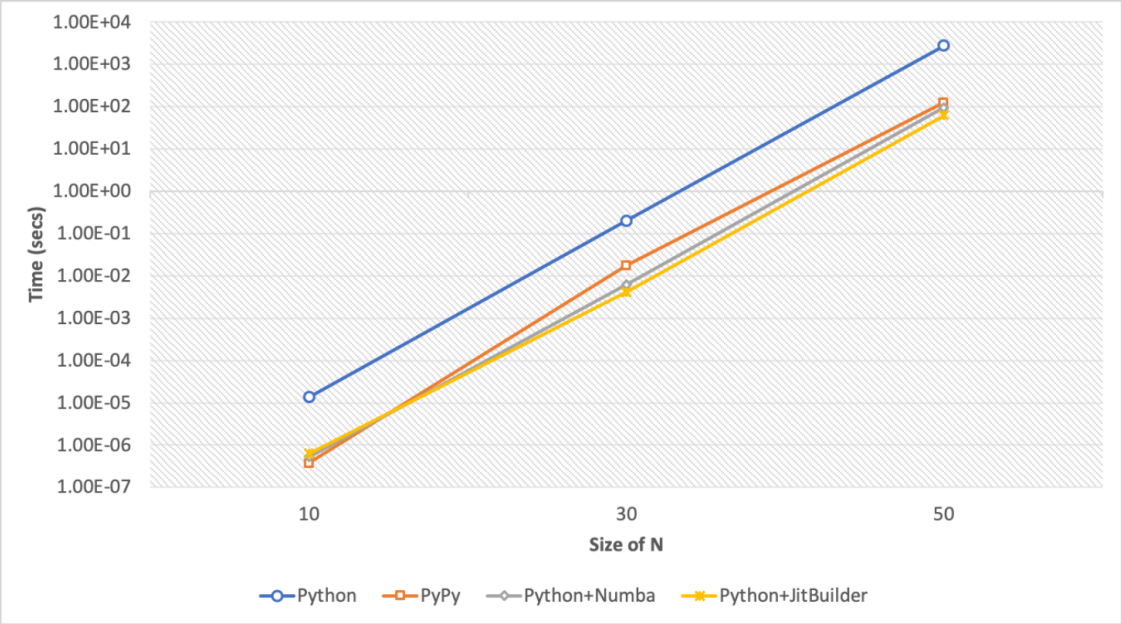

Figure 2.5: Recursive Fibonacci benchmark for JitBuilder's Python API [3]

2.4.1.3 JitBuilder's Python API

Instead of writing a specialized serializer in C++ and implement a wrapper for Python, another option is to directly build this serializer in Python with JitBuilder using its Python API. In his research [3], Allen set out to integrate OMR JitBuilder directly into Python. The results in Figure 2.5 of the recursive Fibonacci benchmark are shown to be very promising. Overall, this implementation of JitBuilder in Python achieves competitive performance compared to the other state-of-the-art approaches such as Numba or Pypy.

2.4.2 Java

With the options for Python binding out of the way, the following subsections examine some of the Java binding possibilities.

2.4.2.1 JNI

To allow Java to call C++ functions, Java native Interface (JNI) is the most well-known option. It is built into standard Java Development Kits. With this tool, the Java application can pass data and make C++ function calls.

To further understand this interface, an example is shown in Listing 2.11. Here, the Java code loads the shared object *libnative*. One key thing to note here is that the prefix *lib* is not included when loading native libraries. Then, it defines a native function *sayHello*.

```
static {
        System.loadLibrary("native");
}
public native void sayHello();
```

Listing 2.11: Example of Java loading native library

Next, this Java class needs to be compiled using javac with the flag *-h*, so that a JNI header file is generated. Afterwards, the C++ file could be created, and the implementation of functions could be done in an almost-C++ manner, plus some Java notations and function calls.

```
JNIEXPORT void JNICALL Java_test_sayHello
(JNIEnv* env, jobject thisObject) {
        std::cout << "Hello␣from␣C++␣!!" << std::endl;
}
```

Listing 2.12: Example of Java binding using JNI

An example of how C++ implements a JNI function can be seen in Listing 2.12. In the argument of the function, the *env* pointer represents a structure that contains the interface to the JVM. This single structure contains all of the functions to access

and modify Java objects [19]. The next argument jobject refers to the Java object that declares this native method

JNI is relatively easy to use, as it closely resembles C++. However, it is very expensive to perform these calls, as the overhead can be as much as three to five times more than the normal Java methods [16].

2.4.2.2 JitBuilder's Java API

Similar to the Python side, it is also possible to skip the extra step of using a Java binding to communicate with the C++ code. Instead, the Java API for JitBuilder could be directly used to implement the specialized serializer in Java. There has been discussion of a prototype of this Java API being built for JitBuilder [22].

2.5 Limitations of previous work

From previous experiments [24], the results have shown that the current bottleneck of PySpark lies in the serialization and deserialization step across the language boundary. Although the pickle library provides a very efficient and versatile serializer for Python, it is not robust for Java.

BSON has competitive performance compared to pickle, but it requires the data to be in a key/value pair format. This requires extensive changes to the source code of PySpark to adapt to this requirement.

Protocol Buffers is a very efficient serializer, with compact serialized data size. However, it requires the user to provide data schema before running the experiment. This is not applicable in PySpark, because as it processes the information, it constantly shuffles, groups and merges columns of data, hence making the schema constantly changes between stages.

Finally, the bitsery library fits various requirements. It is fast and efficient, with small serialized data. However, it also asks the user to create a structure beforehand to know how to serialize and deserialize data. Furthermore, it is written in C++, therefore wrappers for Java and Python are needed so that PySpark can access and make the native calls for serialization and deserialization.

Due to the fact that any serializer written in Python or Java would have a difficult time competing with pickle, a serializer being efficiently implemented in C++, a decision is made to build a specialized serializer for PySpark, with the underlying algorithm also written in C++. However, the issue of unknown data formats still needs to be dealt with.

This is where JitBuilder plays an important role. It could be used to generate the structure on-the-fly at run time using information that user passes in. With that, it could also use this information to generate specialized serialization and deserialization code for that data schema.

JitBuilder and bitsery could be integrated to build this serializer. However, the underlying logic of bitsery is to convert the data into bytes and copy it from the memory location source into a buffer. Using the same logic, a new serializer could be implemented so that it could both utilize JitBuilder functionality to the fullest, while also being efficient without the need to go through third party.

With that decision in mind, the next step is to choose the language bindings for this specialized serializer. For Python, ctypes is picked because it is built into the standard Python library, it is simple to use, and provides enough control for this specific application. Even though Cython has more capability, is more complex to install and use, and not needed for this simple interface of sending data and making native calls. For Java, the only viable option is JNI. This is because JNI provides enough functionalities, and it can easily be integrated into PySpark later.

The reason why JitBuilder's Java and Python APIs are not used is because, at the

time of this research, they are still in the prototype phase. All of the functionality required are not fully developed or not yet compatible with the current Python or Java versions.

Chapter 3

Solution

This chapter explains the details of the proposed algorithm and its implementation.

3.1 Proposed algorithm

This section describes the approach to tackle the bottleneck in PySpark performance, which is the serialization and deserialization step. After researching and evaluating the advantages and disadvantages of possible solutions to alleviate this issue, the final decision is to implement a specialized serializer for this task.

This serializer is specialized based on the schema of the input. However, in Spark applications in general, and PySpark specifically, the data schema changes throughout the process as the data is marshalled and processed. Therefore, the serializer needs to be able to dynamically adapt to those changes. OMR JitBuilder is utilized to do this task.

The flow of the serializer is as follows:

1. The JIT is initialized

2. The specialized serializer receives the data as well as its schema.

3. The serializer then uses this schema information to determine if it needs to recompile the serialization and deserialization code.

    - If the schema received is different from the previous one, the algorithm recompiles the serialization and deserialization code

      JitBuilder plays an important role in this algorithm due to its ability for efficient dynamic code generation.

    - If the schema is the same, the algorithm proceeds to the next step

4. The data is then serialized or deserialized depending on the given command.

One thing to note is that the schema of the data is also serialized and inserted in the beginning of the byte array, so that the deserialization code in a different process or on a different node can get this schema information and process the data correctly.
This proposed algorithm works well when the application needs to serialize and deserialize a large batch of data instead of a single row. This is because for multiple rows of data, the codes for the serializer and deserializer only need to be compiled once and then can be reused multiple times. JitBuilder has already processed these codes so that they do not have any conditional statements left at runtime, and only need to perform straightforward calls to native functions to create and manipulate objects. This is potentially where the speed-up in performance could come from.
On the other hand, if the serializer needs to handle data with different schemas every time, it would negatively affect its performance. This is because the serialization and deserialization code needs to be recompiled every time the serializer receives a new schema, and these recompilation and binding steps are expensive compared to directly invoking native functions.
For this project, the target application is PySpark, which is a framework built to handle big data. This means that this proposed algorithm would be able to utilize

the specialization of the serialization and deserialization code to speed up PySpark's performance when dealing with a large amount of input.

Once the base serializer in C++ is implemented, the wrappers for Python and Java are also needed to complete the whole pipeline. For Python, this is achieved quite easily. This is because the most commonly used implementation of this language is CPython, which is written in C and Python, and C is largely compatible with C++. The Python standard library ctypes and the header file Python.h in the Python development package are used for this wrapper. On the JVM side, there is also the Java Native Interface (JNI) to allow Java to transfer data and make native code calls.

CPython can be defined as both an interpreter and a compiler as it compiles Python code into bytecode before interpreting it. It has a foreign function interface with several languages, including C, in which one must explicitly write bindings in a language other than Python.

Finally, this serializer needs to be integrated into PySpark by replacing the two files to check for its correctness. In Python, the serializer code needs to be placed in the file python/pyspark/serializers.py. For the JVM side, the code goes into the Scala file core/src/main/scala/org/apache/spark/api/python/SerDeUtil.scala. These paths are valid up to the current version of Spark, which is 3.1.

3.2 Implementation details

This section further elaborates on the details of how the proposed algorithm is implemented in four languages, C++, Python, Java, and Scala.

3.2.1 Specialized serializer

The solution selected to improve PySpark performance is to build a specialized serializer in C++ with the support of JIT technology, specifically OMR JitBuilder. JitBuilder is integrated here so that the serialization and deserialization code can be specialized according to the schema of the dataset during runtime. One thing to note here is that the schema is also being deduced at runtime, and is not required to be specified by the user. It does so by generating native code dynamically based on the user input. For example, the piece of code in Listing 3.1 takes an array of JitBuilder types and an array of field names, and creates a native Struct with the fields corresponding to those types on the fly. However, for the string type, instead of storing the string's content or its current address, the field stores an integer, which is the offset of the string with regards to the address of the current Struct. The reason for this is explained in the following sections.

However, this is only the C++ code. To be able to integrate this JitBuilder serializer in PySpark, the wrappers for Python and Java need to be implemented. The reason why there is a wrapper for each language is because of the different methods C++ uses to access the data types of each of these languages. C++ can directly modify Python objects, as the Python runtime is implemented in C. However, the situation is different for Java. Even though OpenJ9 is also built upon C++, the effort to integrate PySpark, and then the JitBuilder serializer into it is not within the scope of this project. Therefore, another solution to transfer data between C++ and Java, JNI, which is an interface so that Java can use native methods, is used. The following sections go further into the details of the wrapper built for Python and Java.

```
resultTypeDictionary::resultTypeDictionary(
TypeDictionary type, vector<char *> vec_str,
vector<IlType *> vec_types)
: OMR::JitBuilder::TypeDictionary() {
```

```
        DefineStruct("MyStruct");
        for(int i = 0; i < vec_str.size(); i++)
        {
                if (vec_types[i]->primitiveType(&type) ==
                    type.Address){
                        DefineField("MyStruct", vec_str[i], Int64);
                }
                else {
                        DefineField("MyStruct", vec_str[i],
                                    vec_types[i]);
                }
        }
        CloseStruct("MyStruct");
}
```

Listing 3.1: Example of JitBuilder code create a Struct on the fly based on the user input

3.2.1.1 General

Before discussing the specifics of the wrappers for the two languages, there is some code that is shared between these APIs for both Python and Java. The first common part is the initialization code for the relevant JitBuilder components.

In C++, a function is defined to perform the initialization of the JIT, then compile all the specialized serialization and deserialization methods, and finally assign them to normal native functions so that they can later be called like any other C++ functions.

But even before that, the source language, which is either Python or Java, receives the input data, and performs a type check routine. The output of this process is a

Data	[	8429,	(	31.49,	"hello"	)	]
Type list	4	1	5	2	3	6	7

Table 3.1: Example of the list of integers representing the data types of a sample input data

list of integers, with each value being mapped to a specific data type as follows:

- 1 - Long
- 2 - Double
- 3 - String
- 4 - Beginning of list
- 5 - Beginning of tuple
- 6 - Ending of list
- 7 - Ending of tuple

For this prototype, the JitBuilder serializer only supports long, double and string to simplify the process, under the assumption that all numeric values fit in 64 bits. The reason why long and double types are used instead of integer and float types is because they can cover a broader range. To be specific, on a 64-bit machine, long is the same as int64, and can cover a range from -9223372036854775808 to 9223372036854775807. Double can have a value within the range of 1.7^{-308} to 1.7^{308} with a precision of 15 digits. This implementation is chosen because for the JitBuilder serializer, all the data that belong to the same column occupy the same amount of space. This is done to eliminate the process of checking that every column of the data can fit into a smaller data type than 64 bits. For example, if the data on the first row can fit into 32 bits, but the data in the later rows does not, this could potentially produce garbage data or even crash the system. However, this approach

has the downside of using more space than necessary, but later experimental results show that it does not greatly affect performance.

Coming back to the list of integers, these values are then used in the initialization step for JITed code. First, the serialization and deserialization code specialized for this exact schema from the list of integers is compiled based on the JitBuilder codes. These JITed codes are then assigned to pointers of normal native functions. Later, these native functions can be directly invoked in C++.

Listing 3.2 shows a piece of the initialization function. In the first stage, the type list, which is a list of integers, is cleaned up so that the only data type left is of primitive types. The cleaned up list is then passed through a function so that a struct type can be created with the exact schema, like the data after being flattened using the generated code. With this information, the serialization code specialized for this schema is created and compiled in step three. Finally, the newly generated function is assigned to a normal native function so that it can be used just like any other C++ function.

```
extern "C" void init(int* typeValue, int len){
        printf("Step␣1:␣define␣type␣dictionary\n");
        OMR::JitBuilder::TypeDictionary type;
        std::vector<OMR::JitBuilder::IlType *> typeList;
        std::vector<char *> fieldList;
        int curValidDataIndex = 0;

        for (uint32_t counter=0; counter < len; counter++){
                char *name = new char[10];
                if (typeValue[counter] == 1){
                        std::sprintf(name, "f%d", cur);
                        fieldList.push_back(name);
```

```
                        curValidDataIndex ++;
                        typeList.push_back(type.toIlType<int64_t>());
                }

                if (typeValue[counter] == 2){
                        std::sprintf(name, "f%d", cur);
                        fieldList.push_back(name);
                        curValidDataIndex ++;
                        typeList.push_back(type.toIlType<double>());
                }

                if (typeValue[counter] == 3){
                        std::sprintf(name, "f%d", cur);
                        fieldList.push_back(name);
                        curValidDataIndex ++;
                        typeList.push_back(type.toIlType<char *>());
                }
        }

        resultTypeDictionary methodTypes(&type, fieldList, typeList);

        printf("Step␣2:␣compile␣serPyMethod␣builder\n");
        SerializePyObjectMethod serPyMethod(&methodTypes, fieldList,
typeValue);
        void *serPyEntry;
        int32 rc = compileMethodBuilder(&serPyMethod, &serPyEntry);
        if (rc != 0) {
```

```
                fprintf(stderr,"FAIL:␣compilation␣error␣%d\n", rc);
                exit(-2);
        }

        printf("Step␣3:␣assign␣serPyMethod␣to␣function␣pointer\n");

        serPy = (SerializePyObjectFunctionType *) serPyEntry;
}
```

Listing 3.2: A part of the JitBuilder serializer intialization function

The next part that the wrappers for Python and Java share is the structure dynamically generated by JitBuilder. This structure is needed so that it is possible to easily assign the order of each element in the row and the total space needed for each row without much effort. This is the Struct that is created in Listing 3.2.
Even though the details are slightly different, the algorithm behind the serialization and deserialization code of the JitBuilder serializer share the same logic. The concept behind the serialization code is described as follows:

- In the serialization code, as the JITed code walks through each element in the row, it puts the data value in the correct structure field under the correct data type.

- The data type string needs to be handled more carefully than long or float. This is because in C++ and JitBuilder, when a string, which is of the type const char*, is created, the value saved in the variable is the address pointing to the start of this string.

  Therefore, with the JitBuilder serializer, in the case of string, the value saved in the field of the Struct is the offset of the string with respect to the start of

the structure. Then, when serializing data, the absolute address of the string within the serialized buffer is calculated based on this offset, and the string value is copied to this destination.

This is done because in PySpark, the serialized data is often moved across processes or even nodes in a cluster. Storing just the original address of the string in the buffer would result in memory corruption when the deserializer tries to process the data, as the string does not exist in that memory space.

The reason why the offset is stored in the Struct field instead of the actual string itself is to ensure that all of the structures have the same format and the fields that are accessed by name are in predictable places. The only variability is the strings, and they are all placed at the end of the struct.

- Finally, the whole structure with all the data values stored in it and all the relevant strings stored at the end of the structure is converted to 8-bit values.

The deserialization code works similarly to the serialization code, just in the reverse order. First, it converts the byte array back into the structure with data values under the correct data format at each field. Then, depending on if the source code is Python or Java, it puts the data values from the structure into the correct data format.

The final part of code that the Java and Python wrappers share is the code to shutdown JitBuilder. This is done via a simple call shutdownJit() in the C++ code.

3.2.1.2 Python

When the developer version of Python is installed on the system with *sudo apt-get install python3-dev*, a package including all the tools to build Python modules is installed. One thing to note here is that this command is specific to the Ubuntu operating system. Within this package, there is a file called Python.h. From C++,

this header file can be imported to allow native code to access and modify Python data.

The pipeline on the Python side starts with the data getting passed in from the user. It could be a list (row), or a list of lists (the whole dataset). From the user input, Python first goes through the row, or the first row of the dataset, to get the data schema and represents it as a list of integers. The reason why this operation is done on the Python side is because it is easier to write in Python, and this does not affect the overall performance of the specialized serializer. This is done under the assumption that every row in the dataset has the same format, and currently there is no check for the case where some rows have different formats.

Once the list of integers representing the schema of the dataset is generated on the Python side, it is passed to the C++ code using ctypes. The reason ctypes library is used instead of other Python bindings for C++ is because it is built into the standard Python library. In addition, it uses low-level concepts, which makes specializing the input and output type of functions between these two languages quite easy. The only disadvantage of ctypes is that it does not scale well with the complexity of the task, but since our requirement is just passing data and making function calls, this is not an issue. On the C++ side, once it receives the list of integers from Python, it compiles and assigns the specialized serialization and deserialization code to normal function pointers for later usage.

The last piece of this Python wrapper is the generated code for serialization and deserialization built specifically to handle Python objects. For both of these functions, when they are compiled, they also receive the list of integers representing the dataset schema. From this list, for the serialization code, it generates a specialized native code which knows the schema of the data to extract the Python object and correctly convert it to its corresponding C++ type. To do this, the JitBuilder code needs to make multiple API calls provided by the CPython interpreter, which are

PyList_GetItem, PyTuple_GetItem, PyLong_AsLong, PyFloat_AsDouble, and PyUnicode_AsUTF8. Once the generated JitBuilder code gets the correct value in the correct C++ format, it puts it in the structure at the corresponding field. After the whole structure with all the data values is cast to a byte array, another call to the native function PyBytes_FromStringAndSize is done to convert this array into a Python bytes object.

This process of understanding the data schema and going through multiple conditional loops to check for data types, or finding the length of a list or tuple, is only done once at compilation and binding time. Once the JITed function is created, when it is called, it has unrolled all the loops and only performs simple straightforward native calls. This is the reason why the specialized serialization process would gain a speed advantage over other serialization methods. The actual function accepts the argument as a Python object, which can be either a list or a tuple, and returns a Python object, which is a byte array in the data type *byte*.

A small snippet of the JitBuilder code to generate the specialized serializer is shown in Listing 3.3. First, if the data passed in is a tuple, the JITed code gets the item out of the Python tuple, and if the data is in a list, the item is taken out of the list. This variable is stored in the variable *pyObject*. Then, it checks in the schema format to see what data type the Python object is, so that it can convert it to the correct format. Finally, the generated code stores the data into the C++ struct at the correct field, so that the Struct can be cast to a byte array at the end. This is a simplified version of the inner loop body of the JitBuilder serializer.

For the JITed code to call native functions, it follows the format Call("<function name>", number_of_args, args). For example, Call("PyTuple_New", 1, 5) means make a call to the native function "PyTuple_New". This native function takes in one parameter, which is the desired length of the tuple, in this case 5, and returns the pointer to this tuple.

As previously discussed, in the special case of strings, the offset of the string relative to the beginning of the Struct is stored in the Struct field instead of storing the actual string. The reason for this approach is that if the actual string is stored in the buffer, the location of the subsequent values are shifted according to the previous string's length. This makes traversing the serialized data harder, as the address of the subsequent fields depends on the varying length of the prior string.

```
if (isTuple){
        pyObject = Call("PyTuple_GetItem", 2, data,
                        ConstInt32(index));
}
else {
        pyObject = Call("PyList_GetItem", 2, data,
                   ConstInt32(index));
}


if (typeValue[index] == 1) {
        value = Call("PyLong_AsLong", 1, pyObject);
        StoreIndirect("MyStruct", fieldList[index], myStruct, value);
}
else if (typeValueList[index] == 2) {
        value = Call("PyFloat_AsDouble", 1, pyObject);
        StoreIndirect("MyStruct", fieldList[index], myStruct, value);
}
else if (typeValueList[index] == 3) {
        value = Call("PyUnicode_AsUTF8", 1, pyObject);
        stringLen = Add(ConstInt64(1), Call("strlen", 1, value));
        StoreIndirect("MyStruct", fieldList[index], myStruct,
```

```
   stringOffset);
        Call("memcpy", 3, Add(myStruct, stringOffset), value,
   stringLen);
        stringOffset = Add(stringOffset, stringLen);
        totalStringLen = Add(totalStringLen, stringLen);
}
```

Listing 3.3: A piece of JitBuilder code to extract data from a Python tuple and store it in a C++ Struct data type

As shown in Listing 3.4, the deserialization works backwards from the data generated by the serialization code. Initially, Python passes a byte array into the native call. The deserializer then converts this Python object to a byte array in C++ using the native call PyBytes_AsString. Then, it casts this buffer into the C++ Struct previously generated by JitBuilder specialized for the current data schema. Afterwards, the deserialization code uses the information from the list of integers representing the data schema to know what field is to get what type of data. This information is utilized to convert these data values into the correct Python format, and then put them in a Python object, which could be either a list or a tuple. To perform these operations, the deserializer makes multiple native calls to access and modify Python objects, some of them being PyList_SetItem, PyTuple_SetItem, PyLong_FromLong, PyFloat_FromDouble, and PyUnicode_FromString.

```
value = LoadIndirect(``MyStruct'', fieldList[type], myStruct);
if (typeValue[index]==1){
        pyObject = Call("PyLong_FromLong", 1, value);
        Call(``PyTuple_SetItem'', 3, resTuple, ConstInt32(index),
   pyObject);
}
```

```
else if (typeValue[index]==2){
        pyObject = Call(‘‘PyFloat_FromDouble’’, 1, val);
        Call(‘‘PyTuple_SetItem’’, 3, resTuple, ConstInt32(index),
   pyObject);
}
else if (typeValue[index]==3){
        pyObject = Call(‘‘PyUnicode_FromString’’, 1, Add(myStruct,
   value));
        Call(‘‘PyTuple_SetItem’’, 3, resTuple, ConstInt32(index),
   pyObject);
}
```

Listing 3.4: A piece of JitBuilder code to deserialize data and store it in a Python tuple

This process of checking data type and data schema only happens once, at the compilation and binding time of the initialization step. When the function for deserialization is actually called, it only performs simple native calls without the need to do any conditional loops for any type of checking. At the end, this JITed code returns a Python object with the correct format as the data before serialization

3.2.1.3 Java

The Java side has similar challenges as the Python side. Because of JNI, Java can also write native code that can easily convert from Java types to C++ types. For the serialization process, the Java code initially receives data from the user. Again, the input could be a list (representing a row), or a list of lists (representing the dataset). From this, Java goes through the data in the row, or in the first row of the dataset, to get the schema, under the same assumption that this format is applied in the entire dataset. This schema of the dataset is also represented as a list of integers

with the same mapping as previously mentioned.

Next, the Java code passes this list of integers to a native call to compile and bind the specialized code for serialization and deserialization. However, unlike Python, there are no APIs available for C++ to directly understand Java objects, hence it is not possible to directly pass the data from Java into the JitBuilder generated function. Hence, the data needs to be converted into a C++ data type, then passed as an array of void pointers into the function generated by JitBuilder. This process can be seen in Listing 3.5. The conversion between Java and C++ is done via making JNI calls. Here, the input for this serialization function is a list of Java objects. However, there is no direct way to get the data out of the Java object into any other type besides string. Therefore, the string of the data from the Java object is extracted, then converted to the correct type.

```
for (int index=0; index < listLen; index++) {
        jobject objectJava = env->GetObjectArrayElement(listJava,
   index);
        jstring stringJava = (jstring)env->CallObjectMethod(
   objectJava, getString);
        if (typeValue[index] == 1){
                jlong a = env->CallLongMethod(javaLong, getLong,
   stringJava);
                data[index] = new long(a);
        }
        else if (typeValue[index] == 2){
                jdouble b = env->CallDoubleMethod(javaDouble,
   getDouble, stringJava);
                data[index] = new double(b);
        }
```

```
        else if (typeValue[index] == 3) {
                const char *c = env->GetStringUTFChars(stringJava,0);
                data[index] = new const char*(c);
                stringSize += strlen(c) + 1;
        }
}
```

Listing 3.5: A piece of JNI code to convert Java data into C++ data and store it in a void pointer array

There is one caveat with this approach though. C++ does not allow for an array containing various types, but PySpark allows this concept. This makes passing data between Java and C++ difficult, because there is no easy way to directly pass a Java array of objects of different types into C++. Therefore, the data first needs to be converted into a void pointer array to pass into the JitBuilder generated serialization code. Similarly, when JitBuilder deserialization code processes the data, the result is returned in the format of a void pointer array, and this array needs to be cast to the correct C++ types, and then Java types.

3.2.2 Integration into PySpark

Once the JitBuilder serializer is implemented and its wrappers for Java and Python are built, it needs to be integrated into PySpark. To do this, the two main files in the Spark source code that need to be modified are python/pyspark/serializers.py for the Python side, and core/src/main/scala/org/apache/spark/api/python/SerDeUtil.scala for the Scala file. These two paths are correct up to the most recently released version of Spark, which is 3.1.

To integrate this serializer into PySpark for the Python runtime, it is not possible to just add the Python files to the Spark source code. This is because the JitBuilder

serializer requires an additional shared object in order to access the native functions. However, PySpark needs to build the source code into a zip file so that all the Python workers can access other Spark components easily. This makes placing the shared object in the correct location difficult.

Therefore, the Python module for the JitBuilder serializer is packaged and published on the platform Test Python Package Index, or TestPyPI. This is a separate instance of Python Package Index, and it allows the user to distribute packages for testing and experimentation without affecting the real PyPI index. This makes installing the module and using it in PySpark much easier. This package can be installed using pip, and it includes the shared library previously mentioned.

However, there are a few downsides to using this method, the first one being that whenever there is a change in the code, the package needs to be re-uploaded as a new version, and reinstalled on the system. Another downside is that TestPyPI is a platform for prototyping, so the package is frequently deleted to make space for other prototypes. However, this is not an issue, as the source code can be easily uploaded again if needed.

The integration into the JVM runtime language is slightly different compared to Python. Since the support for Scala and Java is much more extensive, the Java file can be directly placed into the source code, and the Scala code can import and make calls to the Java methods. However, since the current Java JitBuilder serializer only supports a list of Java objects, or a list of lists of Java objects, some Scala objects need to be converted. An example of this can be seen in Listing 3.6.

```
cleaned.grouped(batchSize).map(batched =>
        dumps(batched.map(x => x.asInstanceOf[Array[Object]]).asJava))
```

Listing 3.6: A piece of the Scala code to convert Scala object to suitable Java object for JitBuilder serializer

3.3 Performance debugging

Besides making sure that the serializer performs correctly, as in the serialization code serializes data into the correct format, and the deserialization code can deserialize the value and put it back into the correct form, the JitBuilder serializer also needs to have better performance than the current serializer being used in PySpark, which is pickle. This leads to an important aspect of implementing this serializer, which is performance debugging. This is done to ensure that as much unnecessary work as possible can be removed to enhance the performance.

The first performance inefficiency that is alleviated is the need to re-compile the code for the same schema more than necessary. Recompiling the JITed methods is an expensive process, where all the JitBuilder specialized code is compiled and bound. To reduce the number of times this initialization function is called, every time the serializer receives data to be processed, its schema is checked. If the data schema is the same as the one used last by the serializer, it means that reusing the specialized code is possible, hence reducing the excessive compilation and binding.

The second performance issue comes from the process of transferring data between Python and C++. The initial approach is to build a structure in Python, then generate the fields according to the schema. Once the structure is ready, the data from each row can be put into the corresponding fields of the structure. Next, the instance of this structure is cast to a C++ pointer using ctypes. Then, on the C++ side, the data in this structure is copied into a common structure, so that both Python and Java can understand it. Finally, the common structure is serialized. The deserialization process is quite similar, just in reverse order.

A simplified version of this approach is shown in Listing 3.7. First, a class in Python mimicking a C++ Struct is created, with the name fields and data types defined. Then, to pass this object to C++, the object needs to be cast to a pointer data type.

```
class POINT(Structure):
    _fields_ = ("x", c_int), ("y", c_int)


libname = pathlib.Path().absolute() / "libnative.so"
c_lib = cdll.LoadLibrary(libname)
printPOINT = c_lib.printPOINT
printPOINT.argtypes  = [POINTER(POINT)]


p1 = POINT(1, 2)
printPOINT(pointer(p1))
```

Listing 3.7: Example of creating and passing C++ Struct in Python [9]

There are many issues with this approach. The first problem is that the data needs to be put in the correct field, but the number of fields or even each field's name are not known beforehand. Secondly, the Python constructor for the class POINT only accepts each argument directly as the value for each field. There is no easy way to pass in a list of data and have that list mapped to the field of this structure. This makes putting data into the structure quite challenging.

The next issue that negatively impacts the performance is that the data actually needs to be traversed twice, once when it is put in the structure, and another time when it is copied from this structure to the common structure shared between Python and Java. This is unnecessary as C++ can actually understand and work with Python objects directly.

An improved version of this approach is to directly pass a list of objects from Python to C++. Then, the C++ code can convert the list of Python objects into a void pointer array, and pass it to the same code generated by JitBuilder for Java. The same is done for the deserialization. The byte array is passed into the same deserial-

izer generated for Java, and it returns a void pointer array. This pointer array is then traversed, and each element in this array is copied and placed in a list containing Python objects.

This approach eliminates the need for an additional structure, but it still traverses the data twice at serialization and deserialization steps. However, this leads to the idea of directly unpacking the data in the JitBuilder generated code. This approach is explained in detail in the previous section. The advantage of this approach over the other two options is that the serialization or deserialization code only has to go over the data once, as the data from the Python object list is directly unpacked and put in the common structure, or vice versa. In addition, this is done within the JitBuilder generated code, meaning that all the conditional statements and type introspection calls are already resolved at compile time. Therefore, at run time, the code only has to perform simple straightforward native calls.

Another improvement to enhance the performance of the JitBuilder serializer is by reducing the JNI calls made in Java. Since performing these calls is quite expensive, the specialized serializer includes functions that allow batch serialization or deserialization. With this, the source language can pass in a batch of rows from the dataset instead of a single row at a time, and in return it receives the serialized data of the entire batch.

Chapter 4

Methodology

This chapter explains how the specialized serializer is evaluated against other existing serializers. It also explains about the dataset, which is a part of the TPC-H dataset, used to compare these serializers. Finally, it discusses the approach to build a prototype by integrating this JitBuilder serializer into PySpark.

4.1 Evaluation setup

This section explains further about the dataset chosen to evaluate the JitBuilder serializer against other existing serializers, and the modified PySpark against the original PySpark. It also discusses the other serializers chosen to be evaluated against.

4.1.1 Dataset

The TPC-H benchmark consists of multiple queries and datasets. This particular benchmark is selected because its databases include all data types, and the data size can be easily scaled up [23]. For this evaluation, the dataset selected out of TPC-H is the Part table. It is chosen because it consists of most of the basic data types, which are integer, float, and string. To see the scaling affect of the serializers, different scaling factor for this table is also selected, which are 1, 2, 4, 8, corresponding to 200,000,

400,000, 800,000, and 1,600,000 rows respectively. The Part table is selected and not the other tables, such as Order or Customer, because even though the Part table contains most basic data types, it does not include nested relationships between objects, which is something the JitBuilder serializer is not capable of yet. Each row of this table follows the schema: Integer|String|String|String|String|Integer|String|Float|String.

4.1.2 Serializers

After implementing the specialized serializer with JIT technology, its performance needs to be evaluated. The best way to properly measure its performance is to compare it with other serializers. The first serializer option is pickle, as it is the default for PySpark. By comparing between these two serializers, an estimate could be made of how much of an impact this JitBuilder serializer will have on PySpark's performance.

Besides pickle, a few other options are selected. Some of the popular serializers are considered, which are BSON, Java default serialization, JSON, and Protocol Buffers. The only criterion for choosing the serializers is that it must have API for both Python and Java. This is because PySpark runs in both Python and the JVM runtimes, therefore the serializer needs to be able to operate in these environments also.

With this criterion, Java default serialization is eliminated, because there is no easy API to perform serialization and deserialization of data to this format on the Python side. The next serializer being eliminated is JSON. Even though it is a popular serialization standard, it is not chosen due to its much lower speed compared to all the other alternatives. The remaining options, which are BSON and Protocol Buffers are therefore selected. They provide efficient method to serialize data while maintaining small generated serialized size.

4992	beige thistle misty blue turquoise	Manufacturer#5	Brand#54	MEDIUM ANODIZED TIN	6	LG BOX	1896.99	ct carefully silent
4993	red light metallic linen khaki	Manufacturer#5	Brand#54	STANDARD PLATED STEEL	28	LG BAG	1897.99	packages
4994	linen plum cyan white navajo	Manufacturer#3	Brand#33	SMALL BURNISHED STEEL	27	JUMBO PKG	1898.99	final re
4995	blanched dim powder orchid deep	Manufacturer#2	Brand#25	LARGE BRUSHED TIN	5	JUMBO CAN	1899.99	wake slyly above t
4996	orchid medium dodger cornsilk royal	Manufacturer#2	Brand#22	LARGE POLISHED TIN	18	LG PKG	1900.99	lyly ironic pinto b
4997	navajo turquoise peru dodger linen	Manufacturer#3	Brand#33	STANDARD ANODIZED NICKEL	40	MED CASE	1901.99	lly special request
4998	honeydew deep goldenrod chartreuse almond	Manufacturer#3	Brand#32	ECONOMY BRUSHED TIN	25	WRAP JAR	1902.99	slyly
4999	dim thistle deep forest medium	Manufacturer#3	Brand#32	SMALL BRUSHED TIN	9	MED BOX	1903.99	olites along th
5000	lace antique light cream grey	Manufacturer#3	Brand#34	MEDIUM BRUSHED BRASS	31	LG JAR	905.00	ow accounts.

Table 4.1: Example of Part table data

4.1.3 PySpark integration

Beside evaluating the standalone serializer, the integration of this JitBuilder serializer into PySpark also needs to be evaluated. This modified version is compared to the original PySpark to investigate if there is any performance gain.

4.2 Evaluation experimental design

The following subsections elaborate on the detail of the experiments, specifically the scale of the dataset used, and the scenarios of each experiments.

4.2.1 Serializer

The dataset used for this evaluation is the Part table from the dataset TPC-H. The data is generated at four different scale factors, which are 1, 2, 4, and 8, corresponding to 24.1, 48.4, 96.9, and 194.4 MB. One important thing is that among the serializers, the JitBuilder serializer and pickle do not require schema, therefore the data can be serialized and deserialized per row as a list of objects with various data types. The BSON serializer needs the data to be in a key-value pair format, but it does not require prior knowledge about the data type of each element. Lastly, protobuf requires an external file to declare the data structure beforehand, as well as the data type for each value.

The two criteria used to compare the serializers are the size of the serialized data, and the execution time. For the size of the serialized data, after the serialization of each row, the byte array is dumped into a binary file. The size of the final file is measured as the size of the serialized data.

Because in PySpark, the serializer needs to run in both Python and the JVM runtime, the execution time is measured in six different scenarios:

- The serialization time of the serializer in the Python runtime.

- The deserialization time of the serializer in the Python runtime.
- The serialization time of the serializer in the JVM.
- The deserialization time of the serializer in the JVM.
- The total time to serialize the data in Python, transfer it over the network, and deserialize it in Java.
- The total time to serialize the data in Java, transfer it over the network, and deserialize it in Python.

These scenarios list all the possibilities that the serializer could be used in PySpark. The last two also take into account the time taken to transfer data between processes, which is directly affected by the serialized data size. With these experiments, the big picture of the performance of the JitBuilder serializer could be obtained and evaluated against its alternatives at any possible usage.

4.2.2 PySpark integration

The same dataset with the same scale is also used for the evaluation of the integration of the JitBuilder serializer into PySpark. The metric used here is the execution time of both PySpark versions on specific tasks.

For these experiments to be more realistic, some common applications are selected for the evaluation. The first task is to sort the input data based on its ID. The second one is a simple lookup task to find the information of this specific part with ID 10000. Finally, the two versions of PySpark is supposed to filter the input data and returns a list of parts in which each part's ID must be a multiple of 10.

Even though these tasks can help evaluating the basic functions of PySpark, a scenario mimicking real-world application is also used for a more thorough assessment. The code for this task is shown in Listing 4.1. In this task, the first dataset is also a

Part table, but at the scale of 0.1, which has 20,000 rows and is 2.4 MB. The second dataset is the same one used in all the previous experiments, and is also varied at the scale 1, 2, 4, and 8. For this task, the second dataset is first filtered for the part with ID being a multiple of 100. Then, the first dataset and the filtered second dataset are joined by the part ID, and then sorted.

```
def parse_rdd(line):
    values = [x for x in line.split('|')]
    return int(values[0]), values[1], values[2], values[3],
   values[4], int(values[5]), values[6], float(values[7]), values[8]


data1 = sc.textFile("/home/ngatran/Desktop/Data/0.1/part.tbl")
rddParsed1 = data1.map(parse_rdd)
print(rddParsed1.collect())


data2 = sc.textFile("/home/ngatran/Desktop/Data/2/part.tbl")
rddParsed2 = data2.map(parse_rdd)
rddParsed2 = rddParsed2.filter(lambda x: x[0]%100==0)
print(rddParsed2.collect())


rdd_joined = rddParsed.join(rddParsed2)


rdd_sorted = rdd_joined.sortByKey()
print(rdd_joined.collect())
```

Listing 4.1: PySpark listing for the combination task

Chapter 5

Evaluation

This chapter expands on the details of the experimental implementation and the metrics used to compare the JitBuilder serializer with other existing serializers.

5.1 Experimental implementation

All the following experiments are performed on the machine Legion Y740. The processor is 9th Generation Intel® Core™ i7-9750H with eight cores, and 16 GB memory. The operating system is Ubuntu 20.04.2 LTS, and the applications are run on the bare metal. All the tasks are executed in standalone mode.

5.1.1 Serializers

To evaluate the performance of the standalone JitBuilder serializer, it needs to be compared to other existing serializers. For this set of experiments, the JitBuilder serializer is evaluated against pickle, BSON, and Protocol Buffers.

5.1.1.1 Size

The first metric to compare these options is the size of the serialized data. For this experiment, each serializer receives the same input, which is a Part table from

the dataset benchmark TPC-H, at four different scales 1, 2, 4 and 8. These scales correspond to 200,000, 400,000, 800,000, and 1,600,000 rows, and 24.1, 48.4, 96.9, and 194.4 MB. Each serializer then writes the serialized data into a binary file. The sizes of these files are recorded and compared.

5.1.1.2 Execution time

The second metric measured is the execution time to serialize and deserialize data. There are a few scenarios that the JitBuilder serializer, pickle, BSON and Protocol Buffers are tested in.

- The serialization and deserialization time when running in Python.
- The serialization and deserialization time when running in Java.
- The serialization and deserialization time when running across the language boundary.
  - Serialization in Python and deserialization in Java.
  - Serialization in Java and deserialization in Python.

The execution times of these processes are measured in seconds. These experiments also examine if the serializer can scale up as the size of the data increases.

5.1.2 PySpark integration

The prototype of PySpark with the JitBuilder serializer is implemented and evaluated against the original PySpark. The metric to compare these two versions is the execution time, measured in seconds.

5.2 Measurements

For each experiment, the application is run five times on the same machine to collect its execution time. The results are averaged and their standard deviations are calculated using NumPy functions. An example of the measurements of execution time for the scenarios where serializers serialize data in Python runtime is shown in Table 5.1.

The graphs are plotted as below using Matplotlib. The plot is of the performance of the JitBuilder serializer against pickle, BSON and Protocol Buffers serializers, for the Part table with different input data sizes, which are 200,000, 400,000, and 800,000, and 1,600,000 rows. The metrics measured here are serialized data size, and execution time. The size is measured in megabytes, and the time is measured in seconds.

In the plots, the standard deviations of each value are also graphed. However, since the measurements are performed on the same machine, in the same environment, and back-to-back, the values are very precise. This makes the error bars too small to be visible in the graphs.

5.2.1 Serializers

The following figures show the plots of the measured data for the serializer microbenchmarks.

5.2.1.1 Size

Figure 5.1 shows the serialized data size generated by JitBuilder serializer, pickle, BSON and protobuf for Part table, at scale factors 1, 2, 4, and 8.

Scale factor	Serializer	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	JitBuilder serializer	0.096652	0.094633	0.094782	0.094751	0.098188	0.095801	0.001575
	pickle	0.249755	0.248106	0.245237	0.247336	0.242796	0.246646	0.002697
	BSON	3.381817	3.538957	3.419001	3.331820	3.421850	3.418689	0.076453
	protobuf	1.121814	1.093831	1.082954	1.123939	1.091963	1.102900	0.018709
2	JitBuilder serializer	0.188964	0.186994	0.188480	0.199636	0.190079	0.190831	0.005045
	pickle	0.477108	0.510771	0.498426	0.512408	0.507315	0.501206	0.014515
	BSON	6.913250	6.690805	6.648373	6.749403	6.838419	6.768050	0.108019
	protobuf	2.155207	2.278880	2.191943	2.270488	2.094532	2.198210	0.078054
4	JitBuilder serializer	0.382670	0.381811	0.388979	0.372884	0.380794	0.381428	0.005747
	pickle	0.968891	0.997727	0.985470	0.964530	0.943792	0.972082	0.020643
	BSON	13.664445	13.391006	13.689945	13.600316	13.530872	13.575317	0.120056
	protobuf	4.428319	4.555102	4.426321	4.348307	4.274978	4.406605	0.104446
8	JitBuilder serializer	0.959689	0.941882	0.936583	0.933059	0.938211	0.941885	0.010446
	pickle	2.052281	1.954706	2.049639	2.045745	1.974263	2.015327	0.046982
	BSON	27.212036	27.114983	27.011450	27.340593	27.139122	27.163637	0.122255
	protobuf	8.981423	8.648605	8.455931	8.961354	8.826438	8.774750	0.222354

Table 5.1: Execution time, average and standard deviation of different serializers when serializing data in Python runtime at scale factors 1, 2, 4, and 8

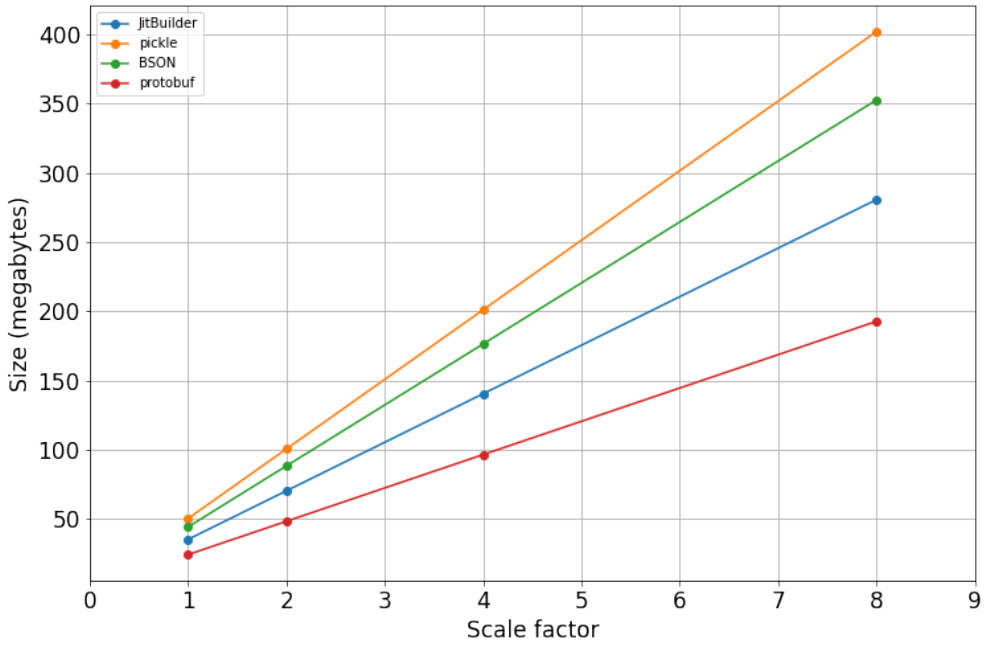

Figure 5.1: Serialized data size generated by JitBuilder serializer, pickle, BSON and protobuf for Part table, at scale factors 1, 2, 4, and 8

5.2.1.2 Execution time

Figures 5.2 – 5.7 are the measurements of the execution time of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 for various scenarios.

5.2.2 PySpark integration

Figures 5.8 – 5.11 are the plots of the execution time of the two PySpark versions for Part table, at scale factors 1, 2, 4, and 8.

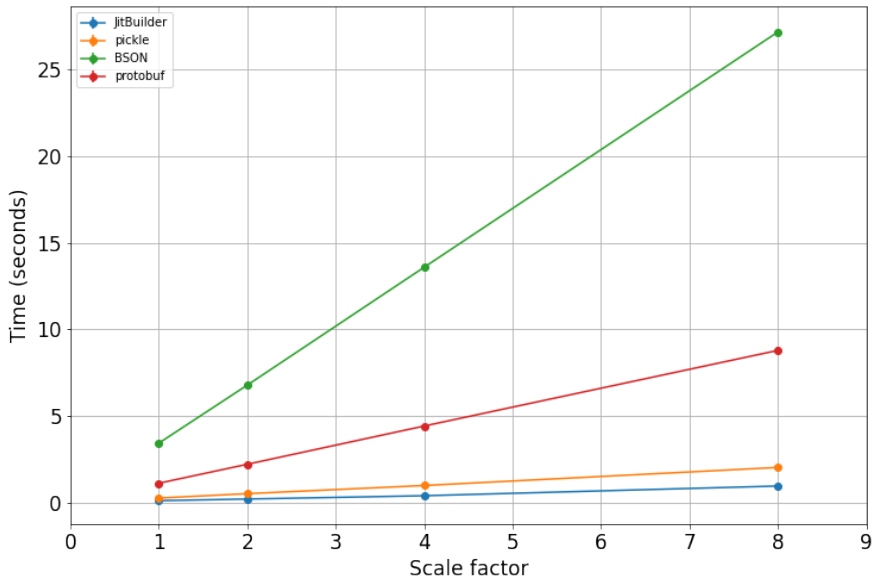

Figure 5.2: Execution times of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when serializing data in the Python runtime

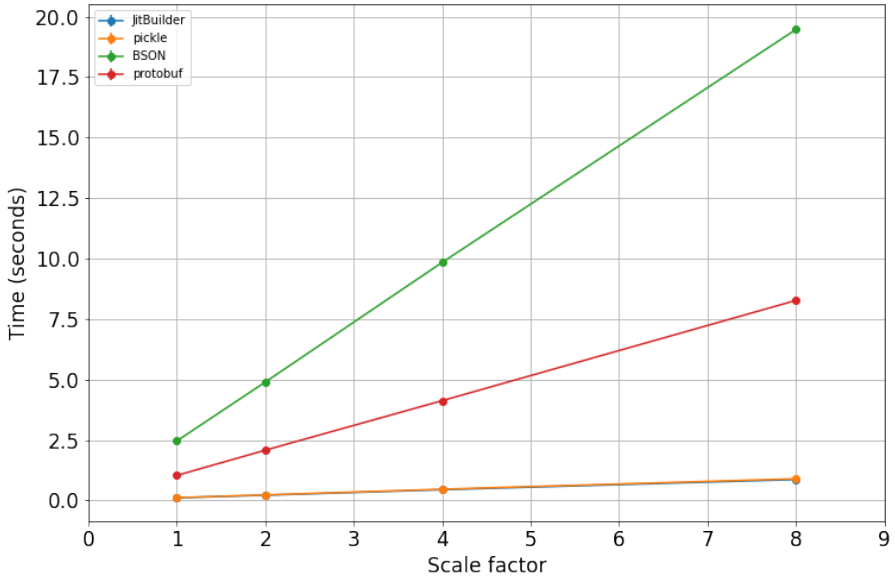

Figure 5.3: Execution times of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when deserializing data in the Python runtime

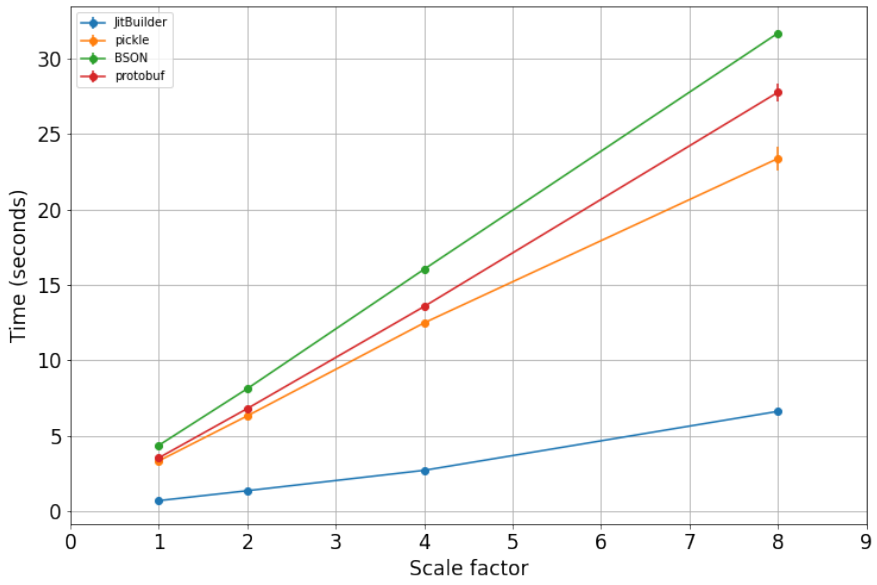

Figure 5.4: Execution times of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when serializing data in the JVM

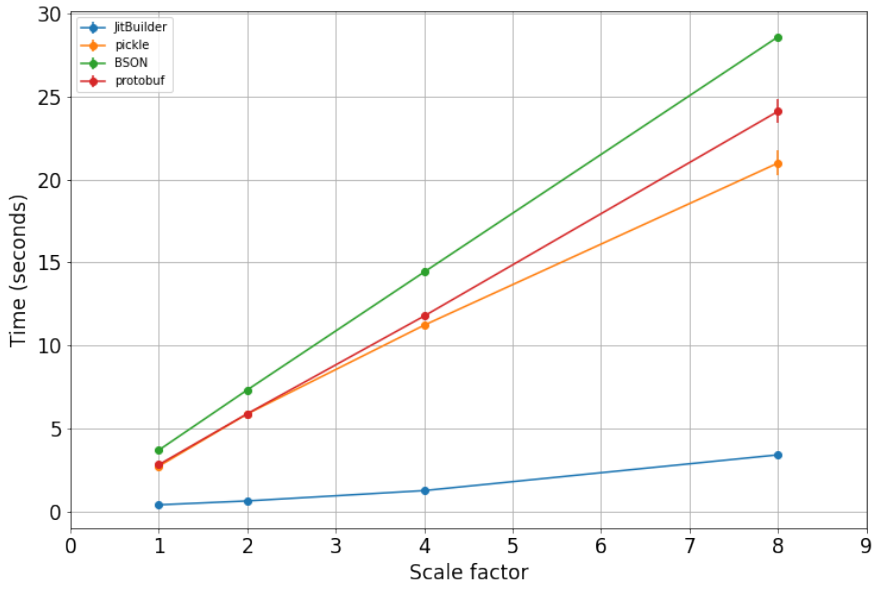

Figure 5.5: Execution times of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when deserializing data in the JVM

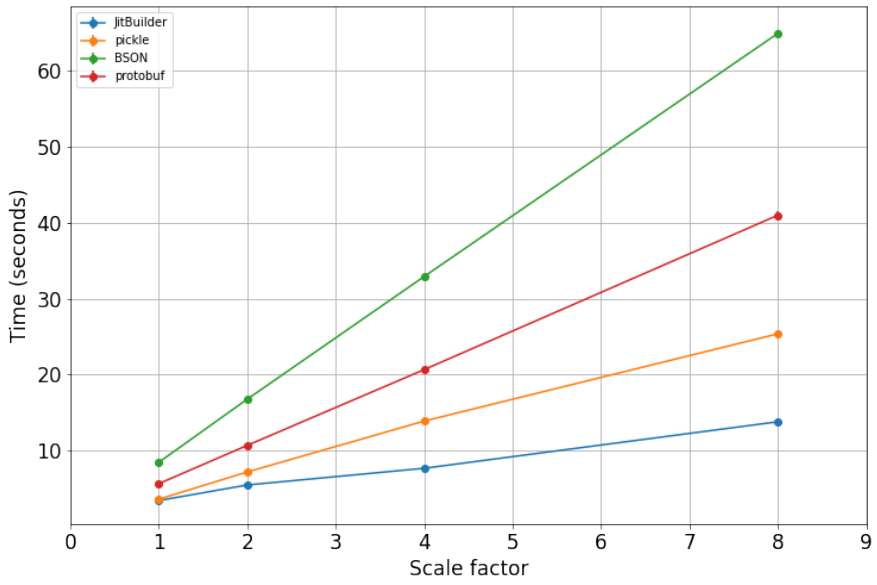

Figure 5.6: Execution times of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when the data is serialized in the Python runtime and deserialized in the JVM

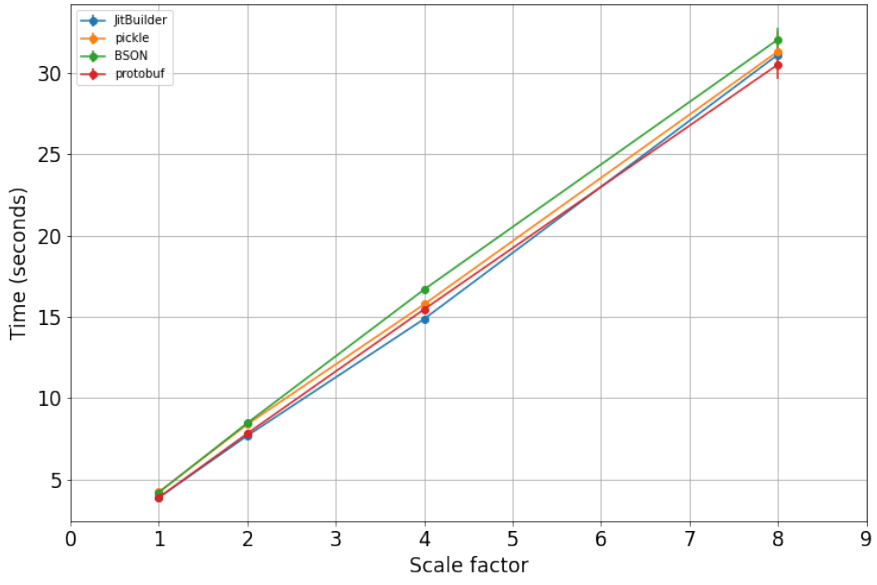

Figure 5.7: Execution times of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when the data is serialized in the JVM and deserialized in the Python runtime

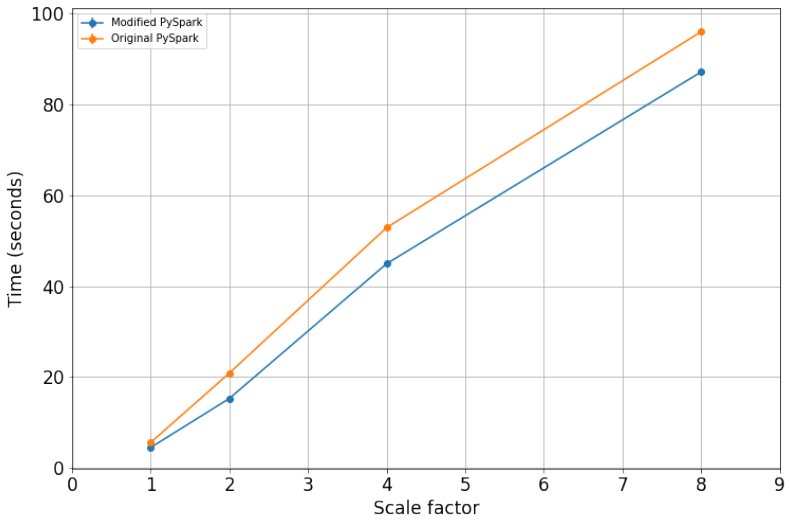

Figure 5.8: Execution times of modified PySpark and original PySpark on Part table, at scale factors 1, 2, 4, and 8 for sort application

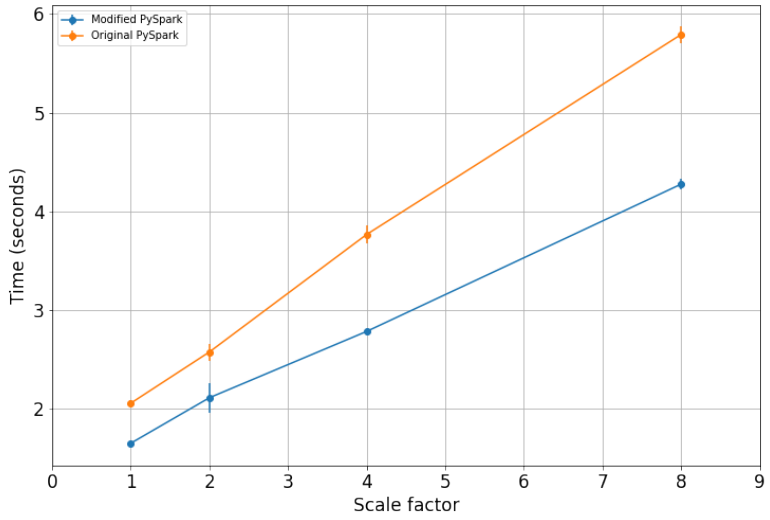

Figure 5.9: Execution times of modified PySpark and original PySpark on Part table, at scale factors 1, 2, 4, and 8 for look up application

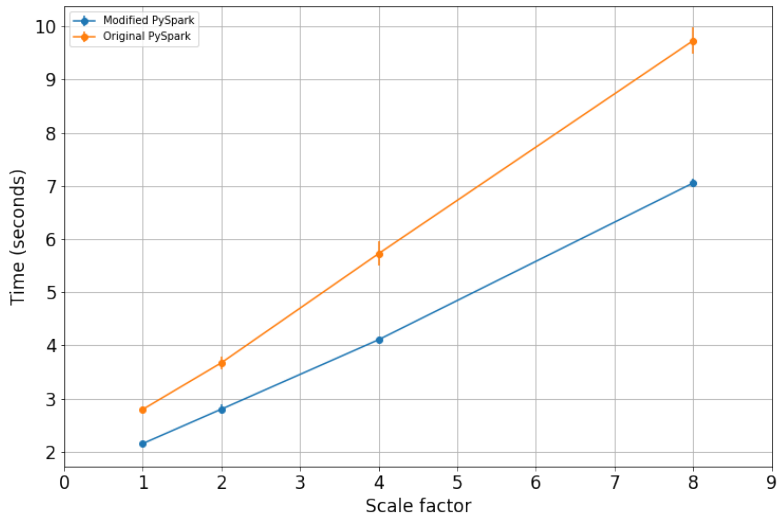

Figure 5.10: Execution times of modified PySpark and original PySpark on Part table, at scale factors 1, 2, 4, and 8 for filter application

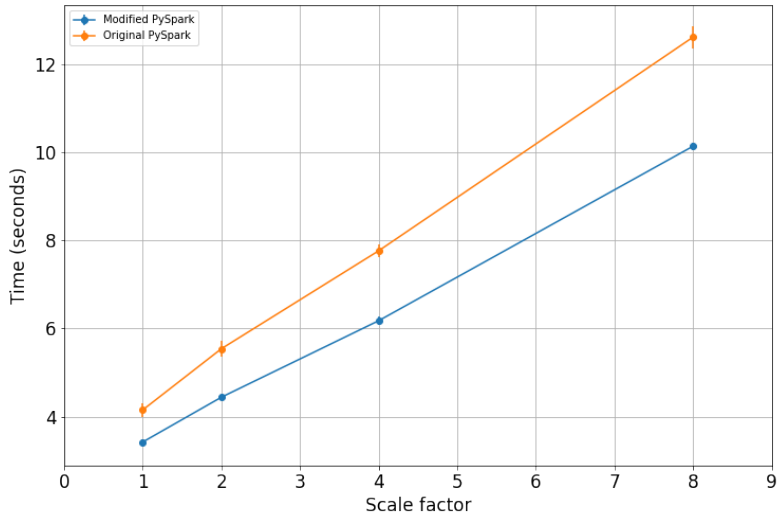

Figure 5.11: Execution times of modified PySpark and original PySpark on Part table, at scale factors 1, 2, 4, and 8 for combination application

Chapter 6

Results

This chapter breaks down the results collected in the experiments. It then analyzes the meanings of these values and plots, and explains the possible causes of the difference in performance gain between JitBuilder serializer and the others in Python and Java runtime.

6.1 Serializer

The following subsections investigate the plots of the standalone JitBuilder serializer against other serializers in various scenarios. The two metrics used for the evaluation are the serialized data size and the execution time of each serializer.

6.1.1 Serialized data size

From the plot in Figure 5.1 showing the size of the serialized data generated by various serializers, it shows that Protocol Buffers has the best result, then JitBuilder serializer, then BSON and pickle. On this graph, there is no error bar showing standard deviation. This is because the size of serialized data size generated is constant for each serializer at that specific scale factor of that Part table.

The reason why pickle has the biggest serialized data size is because it does not require a schema. Therefore, the serialized data also needs to carry the data type of each element, as well as the structure to traverse through the data.

On the other hand, BSON also does not require a schema. However, like JSON, it requires the data to be in dictionary form, so it can access the correct field and cast it to the correct type. This means that the serialized data also needs to carry extra information, which are the keys of the dictionary.

The Protocol Buffers library is on the other side of the spectrum. It requires a strict schema that needs to be defined beforehand in a proto file. This file is then compiled to generate the classes that the Protocol Buffers would use. Because of this, the data is as compact as possible, without needing to contain data types or structure information.

The JitBuilder serializer is in the middle of this spectrum. This serializer assumes that the input data has the same format throughout. Therefore, the data type of each element in a row and the structure of a representative row is only sent once before sending a batch of data, hence reducing the serialized data size significantly. It actually works quite similarly to Protocol Buffers, but offers more flexibility, as no schema needs to be defined beforehand. In a sense, Protocol Buffers could be considered as the Ahead-of-Time version of the specialized serializer. However, JitBuilder serialized data takes up more space compared to Protocol Buffers because currently, the JitBuilder serializer only supports long and double. This results in many empty bytes, which results in using excessive space.

6.1.2 Execution time

The first thing to note is that the execution time of all the serializers in both the serialization and deserialization processes is approximately linearly proportional to the size of the data. With the plots in Figure 5.2 and Figure 5.3 showing the actual

values of the execution time, it is quite hard to see the degree of difference in terms of performance between the JitBuilder serializer and pickle in the Python runtime. However, on the Java side, there is a clear speedup with the JitBuilder serializer compared to all the other alternatives.

These results are very promising, especially compared to Protocol Buffers. The Protocol Buffers library already "knows" the data schema in advance, and the JitBuilder serializer does not, but the specialized serializer still manages to be faster.

On the Python side, since Protocol Buffers and BSON take much more time, the difference between pickle and JitBuilder serializers is less clear. Therefore, the execution times of these experiments are also plotted on a log scale to better visualize the difference. Figures 6.1 – 6.6 show the execution times (in log scale) of the JitBuilder serializer, pickle, BSON and Protocol Buffers on Part table, at scale factors 1, 2, 4, and 8 for various scenarios.

In the Python runtime, the specialized serializer notably outperforms BSON and Protocol Buffers. To be more specific, it is approximately 30× faster than BSON and 10× faster than Protocol Buffers. However, its performance is so close to pickle that the lines for the two serializers are inseparable in Figure 5.3. Even on log scale in Figure 6.2, JitBuilder still performs approximately the same as pickle.

From these graphs, it is visible that in terms of serialization and deserialization time in the Java runtime, the JitBuilder serializer is significantly faster than the other alternatives, with the smallest difference being 3× faster than pickle in serialization time, and the largest being 10× faster than BSON in deserialization time. Overall, this is good performance for a prototype.

The reason why the JitBuilder serializer significantly outperforms pickle in Java, but has quite competitive performance in Python is because pickle is a specialized serializer built for Python. This library was developed in 1995 [8], and has been constantly worked on to improve its performance for Python ever since. In fact, it is

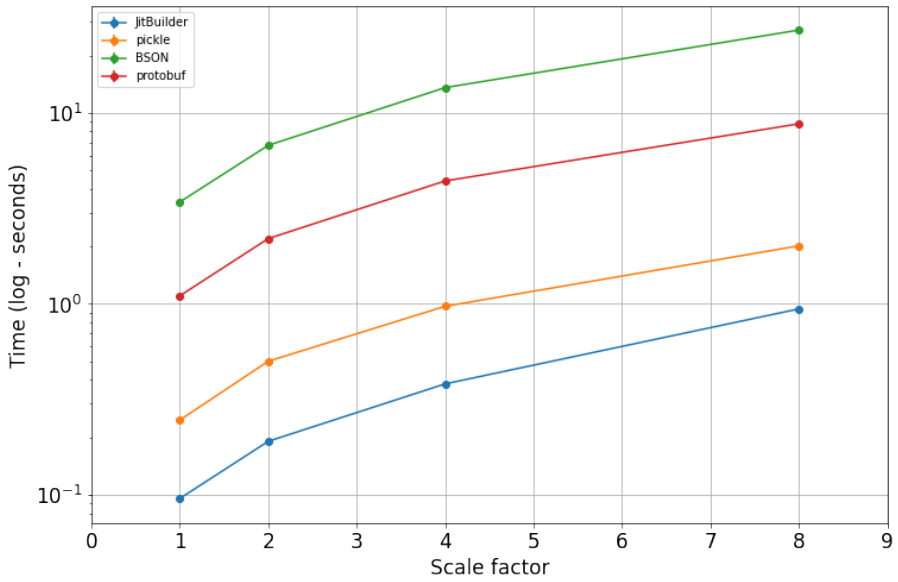

Figure 6.1: Execution times in log scale of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when serializing data in the Python runtime

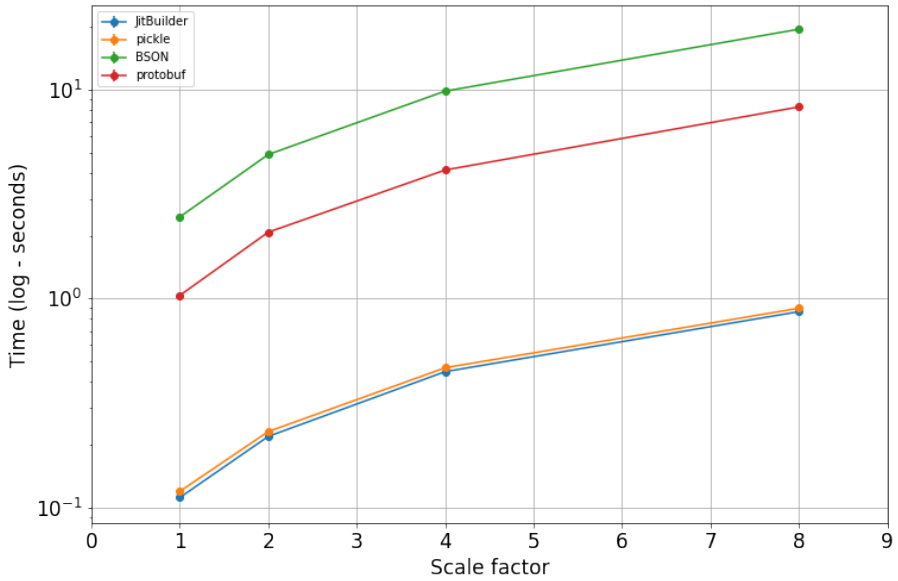

Figure 6.2: Execution times in log scale of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when deserializing data in the Python runtime

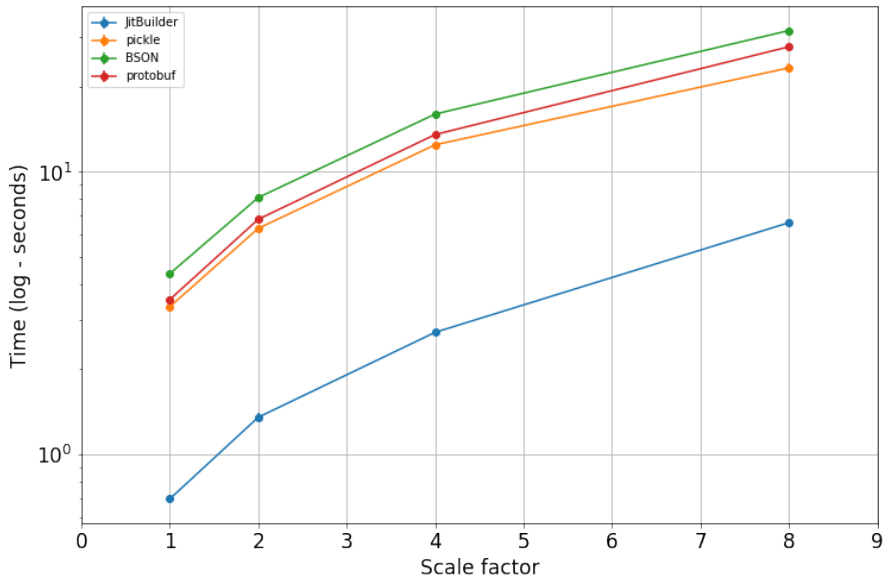

Figure 6.3: Execution times in log scale of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when serializing data in the JVM

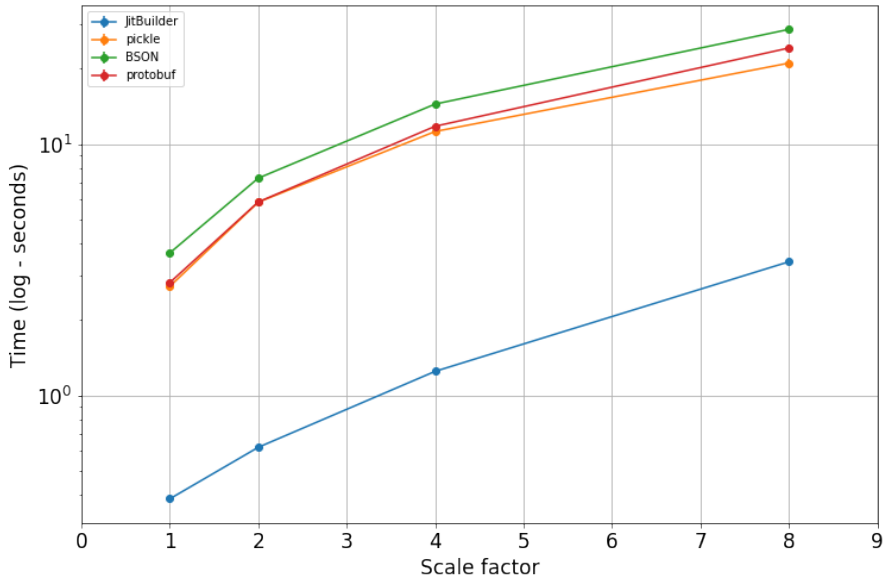

Figure 6.4: Execution times in log scale of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when deserializing data in the JVM

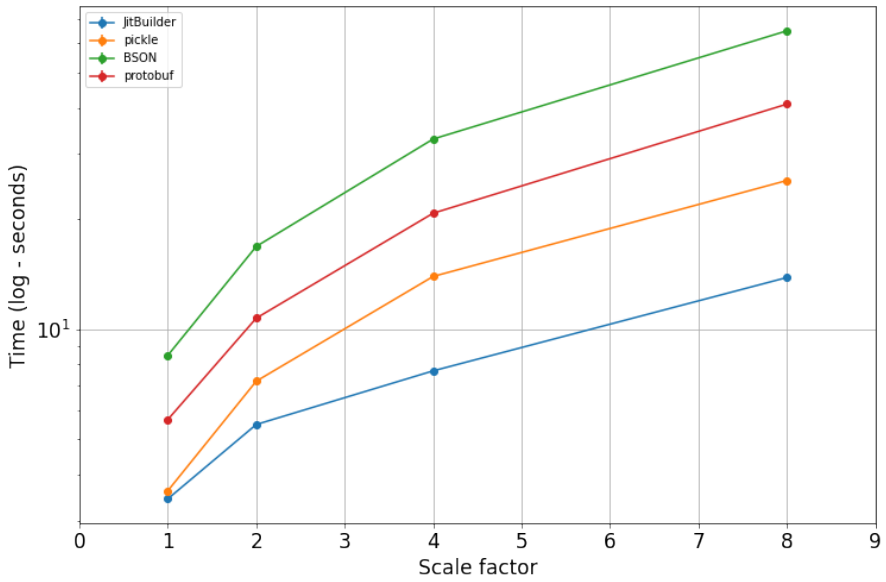

Figure 6.5: Execution times in log scale of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when the data is serialized in the Python runtime and deserialized in the JVM

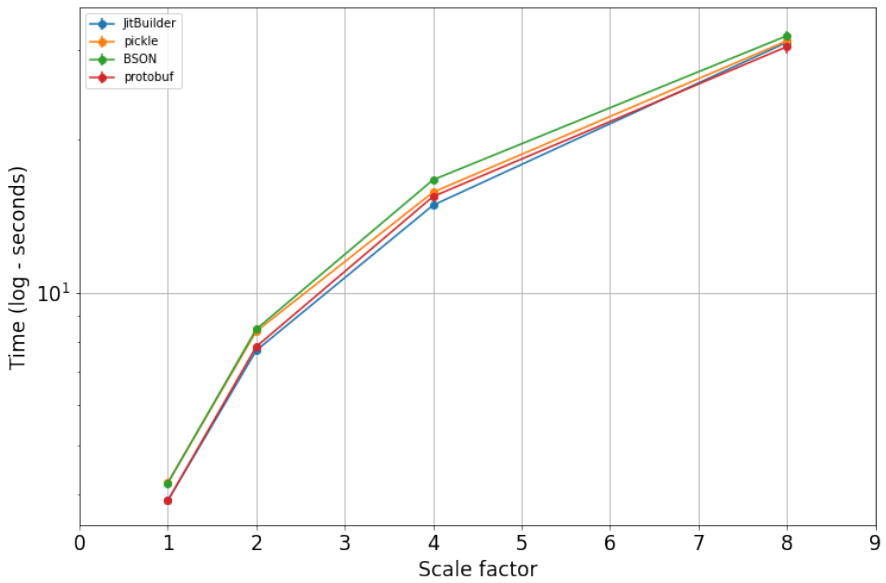

Figure 6.6: Execution times in log scale of JitBuilder serializer, pickle, BSON and protobuf on Part table, at scale factors 1, 2, 4, and 8 when the data is serialized in the JVM and deserialized in the Python runtime

the de facto standard serializer for applications in Python because of its effectiveness and inclusiveness. However, the Java API for pickle is only recently developed, therefore it has not been developed to the same level yet.

Finally, the most important scenarios are the ones where the data is transferred across the language boundary. These experiments most closely represent how the serializers would be used in PySpark. In the scenario when Python does the serialization, sends the serialized data across the socket, and Java deserializes it, the JitBuilder serializer generally outperforms the other serializers, as shown in Figure 5.6 and Figure 6.5.

One interesting trend to note is that as the scale factor of the dataset increases, the performance gap between the JitBuilder serializer and the other options also increases. This makes sense, as the most expensive part in the specialized serializer pipeline is the JIT initialization and the compilation and binding of the specialized code specified according to the data schema. As the data size grows bigger, assuming that this dataset has a consistent schema, the compilation step is only performed once. All the subsequent operations can therefore fully utilize the speed-up of the specialization done by JitBuilder.

However, on the other pipeline where Java serializes the data and sends it over the socket, and Python deserializes it, the performance of these serializers are quite close. There are a few factors to be considered here. First, with regard to just the serialization in the JVM, JitBuilder serializer is approximately three to five times faster than the other options. Another factor that could have an impact here is the serialized data size, as the bigger it is, the more time it takes to send over the socket. Nevertheless, the specialized serializer produces only slightly larger serialized data than Protocol Buffers, but they are still smaller than the rest. When deserializing on the Python side, all of these serializers have approximately the same performance.

With this information, the JitBuilder serializer should still be faster than the alternatives. The issue here lies in the data transferring process. While investigating

this phenomenon, it turns out that the Python output stream occasionally has some issues. From time to time, for unknown reasons, from the Python library socket, the output buffer takes in data from the next buffer. This makes the following data corrupted. However, in the Python library for socket communication, there is no easy way to clear the output buffer after every use.

There are two options to deal with this data corruption issue, either setting a timeout between each send, or redoing the serialization of this specific row and re-sending the data when this happens. Both of these approaches are not ideal, as they add overhead to the performance of the serializer. For this experiment, the approach selected is re-serializing and re-sending data. The data needs to be reserialized because this error also corrupts the new buffer. The program knows that the data is corrupted because when it tries to deserialize the buffer, it cannot extract the data and parse it to the correct data type. This method is chosen because this error only happens every once in a while, therefore the overhead is smaller compared to adding a timeout after every send.

This result does not correctly reflect the performance of the JitBuilder serializer, and it also does not properly represent possible usage in real-world applications. However, it is difficult to measure the exact time taken to perform these corrections.

6.2 PySpark integration

The graphs in Figures 5.8 - 5.11 show a clear improvement of the PySpark version with JitBuilder serializer. In fact, on average, it is approximately 20% faster than the original PySpark. This speed-up is consistent with the previous results where the standalone specialized serializer outperforms pickle in most of the cases. However, this is a smaller improvement than the expected values from the microbenchmarks. This is because the serialization and deserialization is only part of the computation

in PySpark. There are other factors in the specialized serializer such as figuring out the schema of new data, or re-compiling the serialization and deserialization code.

With regard to the scaling factor of the input data, the plots are not quite linear. There are multiple causes that could lead to this effect. First, as the scale factor increases, so does the number of partitions that the input data is split into for parallel processing over multiple threads. Even though the number of partitions is proportional, the effort of keeping track and mapping dependencies between them across stages is not. In addition, more data means more logging and monitoring effort. This might not be completely dependent on the scaling factors either.

Overall, the results indicate that the JitBuilder integrated PySpark performs better than the original PySpark. Furthermore, it can scale up to at least 200 MB, which is the scale factor 8 of the Part table, and possibly even more.

Chapter 7

Conclusions

This thesis focuses on alleviating the bottleneck in PySpark, which lies in the data marshalling process across the language boundary between Java and Python. There are a few options considered, but the final decision is to implement a specialized serializer according to input data using JIT technology.

With the help of OMR and JitBuilder, an efficient serializer is built in C++. This JitBuilder serializer takes in a data schema, and then builds and compiles the code for the serializer and deserializer dynamically according to that schema, and finally assigns it to a normal native function.

From there, the wrappers for the specialized serializer for Python and Java are created. On the Python side, some modifications are made to the serializer code so that it can directly access and modify Python objects to enhance its performance. In addition, ctypes, a library to bind Python to C++, is utilized to easily cast Python objects to C++ objects and vice versa. On the Java side, the wrapper is implemented using JNI so that Java can directly pass Java objects into native calls.

The specialized serializer is then evaluated by serializing and deserializing the Part table, which is a part of the TPC-H dataset. The data is generated at different scale levels, and the serializers' performance is tested in various scenarios. The results

indicate that this JitBuilder serializer has promising results, as it has competitive performance, and often outperforms other serializers.

With this serializer, the first prototype of PySpark with JitBuilder is implemented. This process is very challenging due to the complexity of the Spark source code, which has over 100,000 lines of code. Through experiments, the modified PySpark outperforms the original PySpark. This helps with alleviating the bottleneck in this Spark API, and reducing the gap between PySpark and other frameworks to handle big data.

7.1 Contributions

This thesis makes two main contributions, with the first one being the specialized serializer implemented with JIT technology, specifically with OMR JitBuilder. The bare bones version is implemented in C++. To be able to integrate this serializer into PySpark, the wrappers for Python and Java are also implemented. This is done using ctypes, and a header file belonging to the Python developer package, to allow C++ to directly access and modify Python objects for the Python wrapper. For the JVM side, JNI is used to the same effect.

Through experiments, the results show that this serializer generates compact serialized data and has competitive performance in terms of execution time when evaluating against other serializers, especially to pickle, which is the standard serializer that PySpark is using.

From this serializer, a basic prototype is implemented to integrate this JitBuilder serializer into PySpark. This is done by replacing certain data serialization and deserialization calls in the Spark source code. The results of this integration are shown to be promising.

7.2 Lessons learned

There are many valuable lessons learned through this project, with the first one being gaining an in-depth knowledge of the PySpark inner architecture. From this, we have a better understanding of how PySpark communicates and transfers data between processes, and especially across the language boundary between Python and Java.

Furthermore, we learned a great deal about serializers and different techniques used to represent data type and schema, and the serialized value. Some of this knowledge is then applied to build the specialized JitBuilder serializer.

To be able to implement this serializer, we also learned about JIT technology, about why it is important and how it is used to speed up applications. From that, we get to understand the basics of the OMR framework, and gain a thorough understanding of JitBuilder.

There are other smaller discoveries made during the project. We explored different techniques to bind Python and Java to C++ using libraries such as ctypes and JNI. We also gained very insightful experience on how to easily distribute the Python package by uploading it to TestPyPI.

7.3 Future work

Even though the results are promising, there are still many areas that can be improved. The first important work is to perform more experiments with more complex tasks to measure the full impact of this serializer on PySpark. Furthermore, an interesting scenario to evaluate the modified PySpark version is to run it in cluster mode to test its performance. Currently, all the experiments are run locally on a single machine. However, in real-world applications, PySpark is usually run across a cluster of nodes to efficiently handle big data. This is an important aspect that is

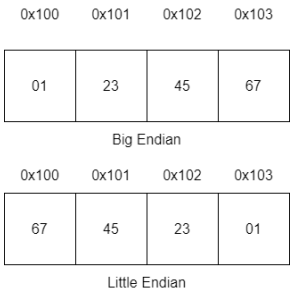

Figure 7.1: Memory representation of integer 1234567 in the big-endian and little-endian byte order

worth further testing.

The next improvement targets generality. The JitBuilder serializer can also be extended to include more types. For now, it can handle long, double, string, list and tuple. However, there are a few other common types missing, such as int, float, and dictionary. These can be easily added to make the serializer more general.

Currently, the JitBuilder serializer assumes that the data is serialized in the little-endian byte order. This means that the least significant byte of the data is stored first. The difference between big-endianness and little-endianness can be seen in Figure 7.1, which shows how the integer value 1234567 is represented in both byte orders.

However, various processors can have either big- or little-endian byte order. When the JitBuilder serializer sends the data over the network to a machine of big-endianness, this could result in corrupted data [2]. Therefore, to make it more platform-independent, the serializer either needs to force a certain byte-order on the data, or has the ability to detect the endianness of the coming data and process it accordingly.

Besides improving the generalizability of the serializer, there is also work to be done on reducing the amount of space that the serialized data occupy. For now, all the numeric values are reserved with eight bytes. However, in many cases, the number does not take up all of the space, resulting in many empty bytes, making the serialized data bigger than necessary. The approach that Protocol Buffers takes

can be studied further to reduce the size of the serialized data even more. This would not only reduce memory waste, but also improve the serializer performance in PySpark. This is because PySpark constantly sends data between processes, so the smaller the size of the serialized data is, the faster the communication, hence better performance.

Another possible future work is a bit more complex. Currently, JNI is used to enable Java to communicate with C++. However, making JNI calls is expensive. For now, there is no off-the-shelf solution to solve this with HotSpot JVM on JDK 8. One solution is to use another JVM that is written in C and C++, so that the calls between Java and the native code could be done at a cheaper cost. An example of such a JVM is OpenJ9. This would alleviate the overhead in making JNI calls, and therefore improve the performance of the JitBuilder serializer significantly.

The next possible improvement is ambitious. The JitBuilder serializer currently serializes the data into a byte array, and the source language, either Python or Java, needs to deserialize it into a format that it can understand. However, it is possible to go a level lower. At the language compilation level, the data can actually be converted into a format that the other source language would understand directly without the need to deserialize it and put it in its own format. This would bring the serializer as close to zero-copy operation as possible, and remove the need for serialization all together. This approach would have a great impact on PySpark, so that the only overhead left in data marshalling between nodes is the time taken to send data.

All the previous options aim to improve the performance of the proposed solution. However, there is another metric that can be optimized, which is the ease of use. Currently, the source code of PySpark is modified, so the user has to download the source code, manually insert in the modification, and build the entire project from source. To alleviate this issue, on the Python side, a script can be created to

automatically import the new serializer and use it to replace the pickle functions to serialize and deserialize data. For Java, there is an easier way. It is possible to use ASM, which is a library that allows direct modification of Java classes in Java byte-code form [25]. With this, the pre-built version of PySpark can be changed easily, without requiring too much user effort to modify and rebuild the entire project from scratch.

Bibliography

[1] *2017 The State of Data Science & Machine Learning*, April 2017.

[2] Harsha Adiga, *Writing endian-independent code in C*, (2007).

[3] Dayton J. Allen, *High Performance Python Through Workload Acceleration with OMR JitBuilder*, Master's thesis, University of New Brunswick, 2020.

[4] Irmen de Jong, *Pickle 1.1*, https://mvnrepository.com/artifact/net.razorvine/pickle/1.1, 2020.

[5] C. Dünner, T. Parnell, K. Atasu, M. Sifalakis, and H. Pozidis, *Understanding and Optimizing the Performance of Distributed Machine Learning Applications on Apache Spark*, 2017 IEEE International Conference on Big Data (Big Data), Dec 2017, pp. 331–338.

[6] Eclipse Foundation, *Eclipse OMR*, https://github.com/eclipse/omr, 2021.

[7] ———, *OMR*, https://www.eclipse.org/omr/, 2021.

[8] Python Software Foundation, *cpython - pickle.py*, https://github.com/python/cpython/blob/main/Lib/pickle.py, 2021.

[9] ———, *ctypes — A foreign function library for Python*, https://docs.python.org/3/library/ctypes.html, 2021.

[10] ———, *pickle — Python object serialization*, https://docs.python.org/3/library/pickle.html, 2021.

[11] ———, *Python/C API Reference Manual*, https://docs.python.org/3/c-api/index.html, 2021.

[12] The Apache Software Foundation, *Apache Maven Project*, https://maven.apache.org/what-is-maven.html, 2021.

[13] Google, *Protocol Buffers*, https://developers.google.com/protocol-buffers/, 2021.

[14] IntelliPaat, *What is Apache Spark?*, https://intellipaat.com/blog/what-is-apache-spark/, 2019.

[15] Martin Kleppmann, *Chapter 4. Encoding and Evolution*, https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html, 2017.

[16] Dawid Kurzyniec and Vaidy Sunderam, *Efficient Cooperation between Java and Native Codes – JNI Performance Benchmark*, The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications, 2001.

[17] Daryl Maier and Xiaoli Liang, *Supercharge a Language Runtime!*, Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering (USA), CASCON '17, IBM Corp., 2017, p. 314.

[18] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu, *Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation*, Proceedings of the 27th ACM Symposium on Operating Systems Principles (New York, NY, USA), SOSP '19, ACM, 2019, pp. 538–553.

[19] Oracle, *JNI Functions*, https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html, 2020.

[20] Josh Rosen, *PySpark Internals*, https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals, 2016.

[21] L. Salucci, D. Bonetta, and W. Binder, *Efficient Embedding of Dynamic Languages in Big-Data Analytics*, 2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW), June 2016, pp. 19–24.

[22] Mark Stoodley, *Explore supporting OMR JitBuilder API from OpenJ9 JIT*, https://github.com/eclipse-openj9/openj9/issues/5836, 2019.

[23] TPC, *TPC-H*, http://www.tpc.org/tpch/, 2021.

[24] Linh-Nga Tran, *Evaluation of Language Interoperability Issues in PySpark*, R&D report, 2020.

[25] Nga Tran, *Proof of Concept: Replacing Spark Serializer using Static Analysis and Java Bytecode Manipulation*, Tech. report, University of New Brunswick, 2020.

[26] Mindaugas Vinkelis, *Bitsery*, https://github.com/fraillt/bitsery, 2020.

[27] _____, *CPP serializers benchmark*, https://github.com/fraillt/cpp_serializers_benchmark, 2020.

[28] A. Watson, D. S. V. Babu, and S. Ray, *Sanzu: A Data Science Benchmark*, 2017 IEEE International Conference on Big Data (Big Data), Dec 2017, pp. 263–272.

Appendix A

Appendix - Collected data

	Serializer / Scale factor	JitBuilder serializer	pickle	BSON	protobuf
Size (bytes)	1	35,087,576	50,156,251	44,087,576	24,071,066
	2	70,173,841	100,442,516	88,173,841	48,157,331
	4	140,349,983	201,018,658	176,349,983	96,333,473
	8	280,686,850	402,155,525	352,686,850	192,670,340

Table A.1: Size of generated serialized data of different serializers at scale factors 1, 2, 4, and 8

Scale factor	Serializer	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	JitBuilder serializer	0.109911	0.116287	0.111312	0.109867	0.111573	0.111790	0.002633
	pickle	0.117110	0.118141	0.116832	0.120925	0.125270	0.119655	0.003531
	BSON	2.465743	2.514178	2.473709	2.433657	2.419196	2.461296	0.037103
	protobuf	1.020951	0.999243	1.056278	1.056550	1.049632	1.036531	0.025473
2	JitBuilder serializer	0.216235	0.223358	0.214802	0.218393	0.223198	0.219197	0.003939
	pickle	0.228254	0.232468	0.230096	0.236013	0.229382	0.231243	0.003081
	BSON	4.882132	4.993225	4.894574	4.826213	4.878750	4.894979	0.060847
	protobuf	2.088086	2.109730	2.063184	2.049155	2.090415	2.080114	0.023937
4	JitBuilder serializer	0.444336	0.436714	0.475569	0.442014	0.438854	0.447498	0.015961
	pickle	0.461030	0.465020	0.468345	0.466934	0.474235	0.467113	0.004839
	BSON	9.889698	9.864152	9.860230	9.806699	9.789786	9.842113	0.042043
	protobuf	4.192748	4.215107	4.051237	4.110832	4.061767	4.126338	0.074734
8	JitBuilder serializer	0.876957	0.885134	0.851020	0.862110	0.857942	0.866633	0.014041
	pickle	0.897518	0.903127	0.889026	0.900646	0.908745	0.899812	0.007299
	BSON	19.442429	19.432393	19.380733	19.540699	19.554334	19.470117	0.074588
	protobuf	8.210484	8.104239	8.564788	8.206433	8.265934	8.270376	0.174625

Table A.2: Execution time, average and standard deviation of different serializers when deserializing data in the Python runtime at scale factors 1, 2, 4, and 8

Scale factor	Serializer	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	JitBuilder serializer	0.689169	0.696890	0.698037	0.689186	0.701283	0.694913	0.005478
	pickle	3.264392	3.379069	3.370455	3.278769	3.359072	3.330352	0.054355
	BSON	4.532534	4.300542	4.354742	4.202577	4.414387	4.360957	0.123558
	protobuf	3.496192	3.535715	3.480250	3.642705	3.501728	3.531318	0.065467
2	JitBuilder serializer	1.391592	1.377236	1.357621	1.363603	1.269970	1.352004	0.047696
	pickle	6.331387	6.317852	6.259341	6.278238	6.366409	6.310645	0.042644
	BSON	8.045350	8.052734	8.156596	8.214784	8.102879	8.114468	0.071695
	protobuf	6.720172	6.927542	6.888261	6.727950	6.711794	6.795144	0.104023
4	JitBuilder serializer	2.662582	2.686005	2.734852	2.712069	2.720513	2.703204	0.028834
	pickle	12.460113	12.575495	12.489484	12.470602	12.387102	12.476559	0.067572
	BSON	15.815326	16.263109	15.776746	16.004004	16.291857	16.030208	0.241768
	protobuf	13.372350	13.693118	13.465675	13.474853	13.799368	13.561073	0.177744
8	JitBuilder serializer	6.498481	6.645570	6.565413	6.672212	6.676262	6.611588	0.077360
	pickle	23.579259	24.162471	24.009156	22.867845	22.229905	23.369727	0.811217
	BSON	31.434370	31.543150	31.914938	31.594015	31.906020	31.678498	0.219502
	protobuf	27.156012	28.000402	27.268218	28.675499	27.690081	27.758042	0.613892

Table A.3: Execution time, average and standard deviation of different serializers when serializing data in JVM runtime at scale factors 1, 2, 4, and 8

Scale factor	Serializer	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	JitBuilder serializer	0.387215	0.387701	0.390883	0.384126	0.384698	0.386925	0.003111
	pickle	2.650776	2.809352	2.707735	2.651324	2.735503	2.710938	0.065696
	BSON	3.840873	3.639817	3.662997	3.547070	3.729484	3.684048	0.075451
	protobuf	2.791567	2.808816	2.775210	2.900607	2.758751	2.806990	0.063364
2	JitBuilder serializer	0.615581	0.629386	0.618870	0.618189	0.619771	0.620360	0.005261
	pickle	5.945606	5.854111	5.808833	5.846511	5.872988	5.865610	0.026927
	BSON	7.243331	7.237349	7.281947	7.436507	7.354241	7.310675	0.087178
	protobuf	5.815283	5.988584	5.931456	5.833552	5.811587	5.876093	0.083198
4	JitBuilder serializer	1.235115	1.245278	1.246656	1.260947	1.234807	1.244560	0.010743
	pickle	11.238246	11.289803	11.235254	11.136523	11.162345	11.212434	0.069789
	BSON	14.213167	14.687000	14.244173	14.324520	14.613467	14.416465	0.215874
	protobuf	11.573450	11.898129	11.575792	11.792682	11.979789	11.763969	0.174870
8	JitBuilder serializer	3.390041	3.370880	3.430998	3.441857	3.353687	3.397493	0.043604
	pickle	21.191239	21.480111	21.650691	20.518581	20.083413	20.984807	0.754531
	BSON	28.391410	28.463957	28.660821	28.625555	28.792715	28.586892	0.135257
	protobuf	23.496183	24.487800	23.574075	25.152152	23.786113	24.099265	0.717052

Table A.4: Execution time, average and standard deviation of different serializers when deserializing data in JVM runtime at scale factors 1, 2, 4, and 8

Scale factor	Serializer	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	JitBuilder serializer	3.419358	3.441771	3.468143	3.425958	3.459166	3.442879	0.020883
	pickle	3.617791	3.705427	3.641105	3.569251	3.578167	3.622348	0.054877
	BSON	8.459963	8.497402	8.433455	8.495819	8.478127	8.472953	0.026815
	protobuf	5.676335	5.635319	5.618345	5.666926	5.721488	5.663683	0.039913
2	JitBuilder serializer	5.507815	5.470833	5.496600	5.508134	5.470833	5.490843	0.018848
	pickle	7.194842	7.202891	7.241511	7.177988	7.209901	7.205427	0.023417
	BSON	16.780601	16.794766	16.762545	16.767933	16.802636	16.781696	0.017078
	protobuf	10.713589	10.689821	10.696420	10.670593	10.733377	10.700760	0.023859
4	JitBuilder serializer	7.660416	7.679282	7.719991	7.657172	7.722096	7.687791	0.031516
	pickle	13.887821	13.929045	14.008782	13.821037	13.860508	13.901439	0.071781
	BSON	32.907640	32.881187	32.924958	32.909022	32.998274	32.924216	0.044278
	protobuf	20.614462	20.661354	20.695070	20.656122	20.731329	20.671667	0.043954
8	JitBuilder serializer	14.041273	13.686689	14.067810	13.669220	13.585585	13.810115	0.226572
	pickle	25.234264	25.374971	25.217288	25.642975	25.458747	25.385649	0.175312
	BSON	64.466079	64.406404	65.364118	65.082576	65.313925	64.926620	0.460558
	protobuf	41.273443	40.971828	41.650027	40.569832	40.507275	40.994481	0.481085

Table A.5: Execution time, average and standard deviation of different serializers when the data is serialized in the the Python runtime and deserialized in the JVM at scale factors 1, 2, 4, and 8

Scale factor	Serializer	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	JitBuilder serializer	3.905916	3.908503	3.882559	3.890553	3.904523	3.898411	0.011276
	pickle	4.242996	4.214585	4.279985	4.213162	4.201470	4.230439	0.031629
	BSON	4.169376	4.204004	4.267332	4.244611	4.199840	4.217033	0.038817
	protobuf	3.896565	3.907637	3.902557	3.864906	3.919329	3.898199	0.020406
2	JitBuilder serializer	7.684773	7.705616	7.678130	7.632465	7.775736	7.695344	0.052274
	pickle	8.367288	8.353254	8.304551	8.497691	8.444501	8.393457	0.076934
	BSON	8.437648	8.536422	8.422966	8.445202	8.529854	8.474418	0.054246
	protobuf	7.884859	7.772407	7.824829	7.807086	7.863591	7.830554	0.044736
4	JitBuilder serializer	14.774396	14.865970	14.868450	14.964479	14.860175	14.866694	0.067318
	pickle	15.829803	15.635968	15.870379	15.765758	15.738350	15.768052	0.090302
	BSON	16.657134	16.664314	16.772885	16.606277	16.693750	16.678872	0.061267
	protobuf	15.479243	15.425272	15.403236	15.386882	15.621283	15.463183	0.095008
8	JitBuilder serializer	31.358531	31.094646	31.246960	30.988962	30.820858	31.101991	0.211361
	pickle	31.570953	32.024453	30.542608	32.722745	29.654347	31.303021	1.214930
	BSON	31.267699	32.779658	31.306887	32.683896	32.134101	32.034448	0.725310
	protobuf	30.759221	31.090640	29.998693	29.240957	31.408685	30.499639	0.877248

Table A.6: Execution time, average and standard deviation of different serializers when the data is serialized in the JVM and deserialized in the Python runtime at scale factors 1, 2, 4, and 8

Scale factor	Spark version	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	Modified PySpark	4.601470	4.599685	4.513887	4.506348	4.407931	4.525864	0.079997
	Original PySpark	5.727768	5.564066	5.601804	5.565468	5.671338	5.626089	0.071585
2	Modified PySpark	15.323133	15.273663	15.364305	15.368862	15.193212	15.304635	0.073147
	Original PySpark	20.402409	21.079236	20.580646	20.965417	21.408085	20.887159	0.401035
4	Modified PySpark	44.954655	44.741867	45.044842	44.814462	45.230659	44.957297	0.193139
	Original PySpark	52.067100	52.223355	53.016653	53.314903	53.845385	52.893479	0.746804
8	Modified PySpark	87.513989	87.808733	86.434783	86.856702	87.087863	87.140414	0.540226
	Original PySpark	95.666808	95.776634	95.308047	96.565416	96.617655	95.986912	0.578802

Table A.7: Execution time, average and standard deviation of modified PySpark and original PySpark for the task sort at scale factors 1, 2, 4, and 8

Scale factor	Spark version	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	Modified PySpark	1.627707	1.669256	1.651469	1.662003	1.649490	1.651985	0.015768
	Original PySpark	2.054169	2.031228	2.057124	2.042375	2.101427	2.057265	0.026737
2	Modified PySpark	2.044170	1.984542	2.107660	2.368862	2.057909	2.112629	0.149806
	Original PySpark	2.725840	2.538627	2.528150	2.514500	2.572841	2.575992	0.086502
4	Modified PySpark	2.758775	2.805928	2.754192	2.785035	2.816894	2.784165	0.027788
	Original PySpark	3.683845	3.687467	3.883712	3.826547	3.747299	3.765774	0.087733
8	Modified PySpark	4.328946	4.273309	4.267662	4.315071	4.190564	4.275110	0.054088
	Original PySpark	5.703572	5.862172	5.694949	5.817758	5.871241	5.789938	0.085270

Table A.8: Execution time, average and standard deviation of modified PySpark and original PySpark for the task look up at scale factors 1, 2, 4, and 8

Scale factor	Spark version	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	Modified PySpark	2.124266	2.203622	2.131313	2.195713	2.114249	2.153833	0.042370
	Original PySpark	2.825241	2.723968	2.882226	2.768661	2.784924	2.797004	0.059882
2	Modified PySpark	2.696830	2.711251	2.874202	2.862896	2.860251	2.801086	0.088891
	Original PySpark	3.746934	3.508156	3.744681	3.777821	3.591615	3.673841	0.117670
4	Modified PySpark	4.173912	4.116834	4.037792	4.122559	4.087902	4.107800	0.049918
	Original PySpark	5.629779	5.967358	5.387187	5.820051	5.830580	5.726991	0.224769
8	Modified PySpark	7.021746	7.087662	6.966223	7.157096	7.042719	7.055089	0.071813
	Original PySpark	10.030915	9.580428	9.668452	9.957584	9.429403	9.733356	0.254365

Table A.9: Execution time, average and standard deviation of modified PySpark and original PySpark for the task filter at scale factors 1, 2, 4, and 8

Scale factor	Spark version	Trial (seconds)					Average (seconds)	Standard deviation
		1	2	3	4	5		
1	Modified PySpark	3.427358	3.331240	3.403866	3.505636	3.447799	3.423180	0.063735
	Original PySpark	4.375694	4.047940	4.007453	4.247752	4.082412	4.152250	0.154735
2	Modified PySpark	4.366934	4.457023	4.403976	4.463030	4.511682	4.440529	0.056110
	Original PySpark	5.415233	5.831030	5.584218	5.337802	5.541597	5.541976	0.189134
4	Modified PySpark	6.091387	6.071655	6.226135	6.170923	6.332672	6.178554	0.106184
	Original PySpark	7.917316	7.839484	7.627546	7.608400	7.841443	7.766838	0.139635
8	Modified PySpark	10.180062	10.037590	10.156028	10.098068	10.248717	10.144093	0.080380
	Original PySpark	12.849376	12.265671	12.824926	12.701356	12.470971	12.622460	0.249477

Table A.10: Execution time, average and standard deviation of modified PySpark and original PySpark for the combination task at scale factors 1, 2, 4, and 8

Vita

Candidate's full name: Linh-Nga Tran
University attended (with dates and degrees obtained): Rhine-Waal University of Applied Sciences (2018 - B.Sc. Electronics)
Publications: N/A
Conference Presentations: