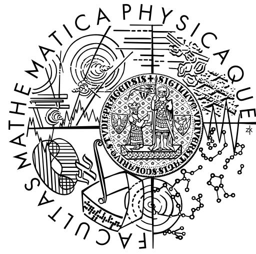


FACULTY OF MATHEMATICS AND PHYSICS
Charles University

BACHELOR THESIS

Matěj Míček

Automatically generating code for fast data serialisation in Python

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Petr Hnětynka, Ph.D.
Study programme: Informatics (B1801)
Study branch: General Informatics (1801R008)

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

I would like to express gratitude to doc. RNDr. Petr Hnětynka, Ph.D. for being supportive during the makings of my thesis. I deeply appreciate his patience and support.

Title: Automatically generating code for fast data serialisation in Python

Author: Matěj Míček

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Petr Hnětynka, Ph.D., Department of Distributed and Dependable Systems

Abstract: Data serialization is not often a bottleneck for modern applications. However, with the rising popularity of Python and constantly increasing requirements for speed, a faster Avro serialization library was necessary. We developed a solution for Avro serialization in Python that uses automatically generated code and speeds the process of serialization 1.5 to 3.5 times compared to the fastest available alternative. The solution is available in the form of a Python library under the name Cerializer.

Keywords: serialization Python Avro Apache fast

Contents

1 Introduction **3**

2 Problem Analysis **4**
2.1 Definitions 4
2.1.1 Serialisation 4
2.1.2 Schema 4
2.1.3 Microservice architecture 5
2.1.4 Cython 6
2.2 Target user description 7
2.3 Typical use case 7
2.4 Serialisation formats and Apache Avro 8
2.4.1 CSV 8
2.4.2 JSON 8
2.4.3 Apache Avro 8
2.5 Existing solutions for Python 9
2.5.1 Avro for Python 10
2.5.2 Fastavro 10
2.5.3 Key takeaways 10
2.6 Schemaless writer 10
2.7 Code/Schema Distribution 11
2.7.1 Source Distribution 12
2.7.2 Wheels 12
2.7.3 Local compilation 12
2.8 Goals revisited 13

3 Cerializer **14**
3.1 Proposed speed-ups 14
3.2 Kafka Confluent Platform 14
3.3 Solution architecture 14
3.4 Object design 15
3.5 Cerializer Schemata 16
3.5.1 Schema fetching 16
3.5.2 Storage of compiled code 17
3.5.3 Cycle detection 17
3.6 Cerializer Daemon 18
3.7 Code Generator 18
3.7.1 Jinja2 templates 19
3.8 Compilation 20
3.8.1 read.pxd, write.pxd and prepare.pyx 20
3.9 Cerializer 22

4 Evaluation **23**
4.1 Code usage examples 23
4.2 Schema Generator 24
4.3 Benchmarks 25

4.3.1 Timeit.timeit module 26
4.3.2 Benchmark procedure 26
4.3.3 Custom schemata benchmark 27
4.3.4 Generated schemata benchmark 27
4.4 Tests 27
4.4.1 Cerializer schemata 29
4.4.2 Cerializer 30
4.5 Limitations 30
4.6 Related work 31

5 Conclusion **32**
5.1 Future work 32

Bibliography **33**

A Appendix **35**
A.1 Cerializer.zip 35

1. Introduction

Data serialization is often an overlooked topic. It is the process of transforming a programming object into a series of bytes. Serialization is a necessary step before sending any message to another system. The reverse process, deserialization, is then carried out on the receiving end. These processes happen most of the time without the user even noticing; nearly all current communication libraries have a built-in serialization framework.

However, as distributed systems gain popularity, the need for sending messages and data between computer rises. In most systems, there is not enough demand for speed that serialization would cause any performance issues. However, there are niche markets such as quantitative finance, where even the slightest delays can cause companies to lose money. In these sectors, there is an emerging demand for fast serialization systems.

Python, with its cutting edge data science libraries, is one of the most popular programming languages in the World [9] and in finance in particular. Still, at the same time, it lacks behind many of its competitors like Java in terms of serialization speed [17].

In this work, we propose a fast serialization library for Python based on automatic code generation. We named our solution Cerializer. This name comes from the combination of the word serialization and Cython, the programming language.

In Chapter 2, we analyze serialization in general and discuss the currently available formats and corresponding implementations for Python. Chapter 3 is dedicated to our implementation (Cerializer) of a Python library that automatically generates code and uses for fast serialization. In Chapter 4, we evaluate the library in terms of speed and compare it to the currently available solutions in Python. Chapter 5 then concludes this work and proposes future work and potential additions to our solution.

2. Problem Analysis

In this chapter, we define the typical users of Cerializer, describe their specific needs and analyze all the significant components and prerequisites of serialization in Python. We discuss the major serialization formats, currently available solutions, and possible ways of updating a distributed system with new schemata. We also provide the reader with all the necessary definitions.

We also assume at least intermediate knowledge of Python and general programming concepts like algorithms and data structures in this entire text.

2.1 Definitions

2.1.1 Serialisation

Serialization is the process of transforming a programming object or data structure into an array of bytes. It is an essential part of storing or sending data. The reverse process is called deserialization. How data is serialized is called a *serialization standard or format*. It is a set of rules defining how to convert basic and complex data types into bytes. There are many serialization standards such as JSON [7], CSV [15] or Avro [1]. We discuss all of them them in this chapter.

2.1.2 Schema

A schema is a set of restrictions imposed on data. These restrictions can include, for example, mandatory fields, set types or default values. In an imaginary use case, we could define a schema called 'student' that would restrict all the data travelling over our network to have a field called 'age' with a value of type int and a 'name' field with a value of type string. We can see an example of a schema and corresponding data in Listing 2.1 and Listing 2.2.

Note that the example schema is parsed into a form of a Python dictionary. Typically this schema would be defined in JSON, YAML or nearly any other competent format.

Listing 2.1: Schema example

```
schema = {
    'doc': 'schema␣to␣describe␣a␣student',
    'fields': [
        {
            'doc': 'name␣of␣the␣student',
            'name': 'name',
            'type': 'string'
        },
        {
            'doc': 'age␣of␣the␣student',
            'name': 'age',
            'type': 'int'
        },
        {
            'doc': 'study␣average',
            'name': 'average',
            'type': 'float'
        }
    ],
    'name': 'student_schema',
    'namespace': 'school',
    'type': 'record'
}
```

Listing 2.2: Data example

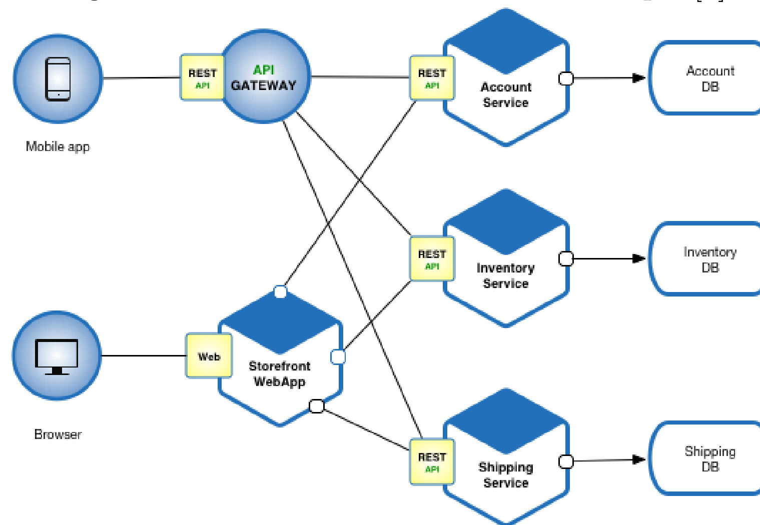
```
data = {
    'age': 23,
    'name': 'Matej␣Micek',
    'average': 1.0
}
```

2.1.3 Microservice architecture

Microservice architecture [8] is a software architecture where the application is seen as a set of many smaller services communicating with each other. Lately, this architecture is gaining in popularity since it is easier to maintain, test and scale.

We can find an example of this architecture in Figure 2.1. There, we can see a clear separation of responsibilities to multiple smaller services. These services then communicate through REST API. For each message sent, serialization and deserialization need to be performed. Note that the number of connections and hence messages grows rapidly with new nodes being added.

Figure 2.1: Microservice architecture example [8]



2.1.4 Cython

Cython [4] is a programming language and also a compiler. It aims to provide a way of writing Python-like code and allow for C-like static type definitions. Cython then generates C extensions for Python. This way, the code can get marginal speed ups.

Listing 2.3 shows an example of a function defined in Cython aiming to compute the first n primes.

Listing 2.3: Cython code example [4]

```
def primes(int nb_primes):
    cdef int n, i, len_p
    cdef int p[1000]
    if nb_primes > 1000:
        nb_primes = 1000

    len_p = 0  # The current number of elements in p.
    n = 2
    while len_p < nb_primes:
        # Is n prime?
        for i in p[:len_p]:
            if n % i == 0:
                break

        # If no break occurred, we have a prime.
        else:
            p[len_p] = n
            len_p += 1
        n += 1

    # Let's return the result in a python list:
    result_as_list  = [prime for prime in p[:len_p]]
```

```
    return result_as_list
```

According to Cython's website [4], this code runs 13 times faster than a pure Python alternative of the same algorithm.

2.2 Target user description

At the beginning of the problem analysis, we have to state that this project's target group is very narrow. We are aiming at a user who requires its codebase to be written in Python but at the same time puts significant emphasis on runtime speed. One of the sectors where these users could typically be found is quantitative trading [20]. Quantitative trading is a discipline where traders utilize quantitative methods to consider or even execute trades. Quantitative traders use Python because of its advanced data analytics libraries but care about overall speed since every split second of delay may cause significant losses.

Our target user also sends and receives many messages through multiple channels, both from within and outside his/her company. Therefore, the final serialized payload's size and subsequent time spent on data transfer play a considerable role in his serialization system choice.

The transmitted data is also not arbitrary but organised according to a given schema. These schemata are known ahead of time and respected by all the participants.

The user is also willing to pay the cost of any startup overhead or an overhead when a new schema is released. Overhead time alone is not a significant issue for our user, but no processes can block the rest of the system from serializing data.

2.3 Typical use case

A typical use case scenario would be bypassing the automatic serialization when sending or receiving data through RabbitMQ [5] or other messaging systems. Most popular libraries offer either to send the data directly in binary format or to pass in a Python object, which is internally serialized typically to JSON. By bypassing the internal serialization and replacing it with a more time-efficient system, the user can gain priceless milliseconds. This approach can be particularly impactful when used internally within a company that builds its software following the Microservice Architecture.

Another common scenario could be a client connected to a stock exchange data stream. The client receives packages in binary Avro format, a format widely used within the stock exchange industry, and needs to deserialize them. The client knows which schema is the data in since there is only one schema per communication channel. The client could use the standard Avro Python library or the independent FastAvro, which is roughly ten times faster. However, our target user is even more demanding and needs an even faster solution.

We need to note that these two use cases do not contradict each other; they would probably be co-occurring in practice. We also need to specify that a schema distribution system that would update the entire distributed system with new schemata and or code is necessary for the system's proper functioning. Since

these procedures are so closely bound to serialization, we discuss them as a part of the project itself.

2.4 Serialisation formats and Apache Avro

2.4.1 CSV

One of the most common serialization formats is CSV (Comma-separated values). It is a row-oriented format that prints the data into a file line by line, separating the columns by commas. The advantage of this is that the raw data is human readable and transferable between office tools such as Excel. This serialization format may be popular but does not allow nested data or any specific schema restrictions. Also, ambiguities are common in CSV, caused mainly by the data itself containing commas.

Below we can see our example data serialized into CSV. CSV is designed to make it ideal for writing more rows of the same data into one file. If we needed to add more students into the "database" we could add more rows.

Listing 2.4: CSV data example

```
name,age,average
Matej Micek,23,1.0
```

2.4.2 JSON

We also need to mention JSON (JavaScript Object Notation). This format also produces human-readable string output, but unlike CSV, it allows nesting. However, enforcing schemata or any proper typing is impractical when using JSON. Even considering the typing issue, the most significant disadvantage of JSON remains hidden for most users. Since JSON serializes data as key-value pairs, it is space consuming. The key can sometimes be multiple times longer than the value part itself. Storage is typically not a problem, but considering the magnitude of time-saving we want to achieve, even a slight difference in size combined with data transfer speeds is significant.

Below we can see an example of serialized JSON data. This data resembles the structure of a Python dictionary.

Listing 2.5: JSON data example

```
{
    "name": "Matej Micek",
    "age": 23,
    "average": 1.0
}
```

2.4.3 Apache Avro

The last option we consider is Apache Avro (later on referred to only as Avro). Avro is similar to JSON in terms of capabilities and structure, but it uses binary format instead of strings and has built-in schema support. The schema support

not only allows for enforcing typing but also assures backwards and or forward compatibility. Also, when writing multiple data records, Avro writes their corresponding schema only once and then writes the values only. This can save an enormous amount of space compared to JSON.

Furthermore, since we are trying to limit down the amount of data transferred, we can use a specific part of the Avro format that allows for writing the data only, without the schema at the beginning. We talk about this more in section Schemaless vs regular writer. This saves even more space, but both sides of the communication must know the schema in advance. However, this is easy to ensure since we deal with stable data streams such as messages from a stock exchange whose schemata do not change often. If so, all participants are notified ahead of time.

In Listing 2.6, we can see the example data serialized using Avro schemaless writer. The format of the data is binary, so it is mostly not human readable.

Listing 2.6: Avro data example

```
b'\x16Matej Micek.\x00\x00\x80?'
```

For illustration, refer to the Table 2.1, which compares the serialized size of messages in JSON and Avro schemaless. We used several custom schemata, which we use further on for testing and are available in the software attachment.

Table 2.1: JSON vs Avro size comparison [Bytes]

schema_name	JSON_size	Avro_size	SON_size/Avro_size
array_int_str	779	534	1.46
enum	14	1	14.00
str	20	5	4.00
union	10	3	3.33
reference	290	55	5.27
int	7	4	1.75
array_str	260	157	1.66
nested	386	214	1.80
map_str	182	114	1.60
map_int_null	141	76	1.86
double	20	8	2.50
long	17	3	5.67
array_bool	637	99	6.43
array_int	723	287	2.52

Considering the inability to store nested data in CSV and the spatial demand together with the lack of typing in JSON led us to settle with Avro as our serialization standard of choice.

2.5 Existing solutions for Python

In this chapter we go over two most popular and practical implementations of the Avro standard for Python. We evaluate their practicality and most importantly speed, since it is a critical measure for our user.

2.5.1 Avro for Python

The Avro library for Python [16] is developed and maintained by Apache themselves. It is a standard library that is easy to use and, in most cases, the solution. However, since it is written in pure Python, it is much slower than alternatives written in other languages such as Java.

It uses a recursive approach to serialization, which is possible due to the specification of schema definitions. To represent schemata, Avro uses Python classes. As we discuss in the section below, there is much room for speed improvement.

2.5.2 Fastavro

Fastavro [17] is an independent library developed and maintained mostly by Miki Tebeka [18]. The sole purpose of this project was to speed up Avro serialization in Python and level the field with other languages. This was successful since, according to the project's website, the benchmarks of Fastavro scored on the same level as Java and roughly 10x better than the standard Avro library mentioned above.

The speedups were achieved by migrating the code to Cython and using the generated C extensions. The interface mainly was untouched, and the library behaves nearly identically to Avro.

2.5.3 Key takeaways

Both of the current solutions for Python are similar in most of the approaches. The significant speedups come primarily from the migration to Cython. Therefore, since we aim to speed the process up, we need to consider parts of the process that we could skip. One of these processes we could skip is schema analysis, but we need to know the schemata ahead of time for this. Or, we could perform the schema analysis ahead of time and distribute a custom made code for each schema.

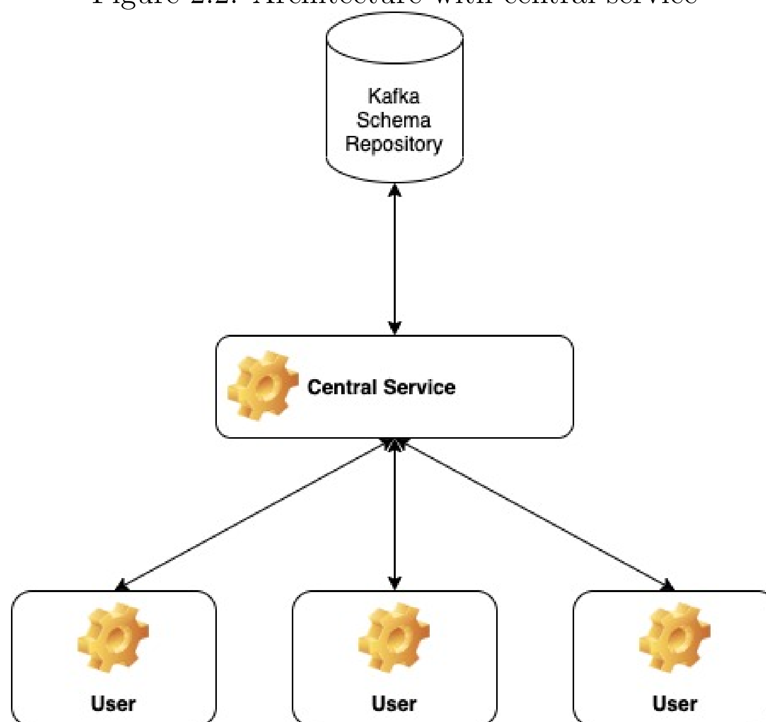
Both of these options would bring speedups to the system, and therefore, we need to consider a schema or code distribution infrastructure which would update the entire system.

2.6 Schemaless writer

As we stated in section 2.4.3, there are two possible ways of writing data in Avro format. The first one is the classical writer, which first writes the schema of the data and then all the data records themselves. The presence of the schema has a considerable advantage of not having to communicate the schemata ahead of time. It also means that anyone with an Avro reader can receive Avro encoded data and read it straight away. This allows for seamless data transition.

However, when we start to put more and more emphasis on the speed of things, transferring the schema along with data every time starts to be more and more costly. The schema has an asymptotically more significant impact with decreasing data records transferred per file or message. Moreover, if speed is one of our concerns, we are more likely to send very frequent messages containing

Figure 2.2: Architecture with central service



only several records. This scenario could occur when receiving stock price changes from our broker or sending an intention to buy. We would not want to send bulk messages in these cases.

Now is when the second option comes into play. The schemaless writer writes the data records without the preceding schema. It causes the data to be practically illegible, but if the sender and receiver agree on the schema beforehand, they can save significant data transfer volumes and hence time. Knowing all this, we settle on using the schemaless writer and reader. The regular reader and writer are not discussed in this work anymore.

As we already mentioned, the schemaless writer requires a schema distribution system or a compiled code distribution system. We could either distribute the schemata and later use them to read and write schemaless data or render and compile a custom code for each schema and distribute it. We discuss the distribution problem in the following section.

2.7 Code/Schema Distribution

As we concluded in the previous section, we need to update the entire distributed system with up-to-date schemata or code. Also, we need to note that there will be multiple machines running serializing operations using the same set of schemata in our system. We can imagine multiple instances of messaging agents of data readers.

2.7.1 Source Distribution

We evaluated the possibility of having a central service that would run permanently and check for new schemata. We can see an example of such architecture in Figure 2.2. New schemata could come from an update in the central schema repository, or users could manually add them. Then, if new schemata emerged, it would generate the corresponding Cython code and distribute the code for compilation to the destination (user) machines. This approach is referred to as source distribution. However, this approach would require the final user to compile the code himself.

Since this solution would not provide any significant benefit and would also make the system's architecture more complicated by adding another element, we decided to reject this option.

2.7.2 Wheels

The second approach, commonly referred to as Wheels [11] is similar in the sense of the central authority. However, with Wheels, the code not only gets generated but also compiled by the central service. This puts away the burden of compilation from the final user. However, since compilation is platform-specific, we would need to generate wheels for all the platforms in our system. This issue is typically addressed by publishing several wheels for the most popular platforms and a source distribution. Then, if a user decides to download some code, usually a package, they can check whether there is a wheel available for their platform and if not, the standard distribution is chosen and compiled on their machine.

If we decided to use the wheels system, we would need to address the abovementioned issue of missing wheels. This way, our Cerializer system at the final machines would have to be ready for both scenarios. This brings another element of complication to the system. However, the biggest argument against the wheels system is again the necessity for the central service.

2.7.3 Local compilation

The last option we consider is a system centred around compilation at the user's machine. The idea is that a user connects to a remote schema repository. Then, all the schemata can be processed (code generation and compilation) on the user's machine. This way, it is computationally harder for the user, but there is also less complication in the system. Also, each user can independently update their schema repositories. We need to note that, unlike source distribution, we are distributing the schemata themselves and not code. Not distributing code means a safer environment since the system is not vulnerable to outside attacks. Furthermore, there is no central service (except for Kafka), therefore each user has to have their local schemata saved on their machines.

To compile the code on the user's machine, we need to deal with the issue of blocking. This problem occurs when there is a new schema released, the user needs to compile the code, and at the same time, the user needs to receive and send messages. In this scenario, the schema compilation would block the flow of messages, which could cause severe problems in the system.

To address this, we propose to run the compilation on a different thread. By doing this, we can compile the code when the user itself is waiting for any IO operations or idle.

After considering all the benefits and drawbacks of the methods mentioned above, we decided to settle with local compilation. We simplify the entire system and make it more stable and durable against downtimes of the central service.

2.8 Goals revisited

To conclude the problem analysis, we specify our detailed goals. First, we need to speed-up Avro serialization in Python. We do so by writing a Python library that will serialize data in the Avro format. We aim at providing better performance than all the currently available solutions.

Second, we need to propose a schema distribution system that will allow the user to use the faster schemaless writer. This system will be capable of updating schemata across the entire distributed network.

3. Cerializer

In this chapter we go over suggested speed-ups, the decision why we will only support the schemaless writer, and the object design and brief overview of the functionality of our proposed solution: Cerializer.

3.1 Proposed speed-ups

Since Fastavro's code itself is already compiled, we need to come up with other ways of speeding things up. The first way is to support only the schemaless writer, which saves time and space.

The second way is to speed up the process by removing the schema analysis part of serialization and decreasing CPU usage. The schema is analyzed each time a value needs to be written. During the analysis, all the necessary information such as type or logical type is acquired. However, since we know the schema ahead of time, we can skip this part and save time. To do this, we propose a solution based on generating custom code for each schema. By doing this, we analyze the schema only once when the schema is released.

Schema analysis is time-consuming since, in both Avro a and Fastavro, there is a code block wrapped in the Try-Except statement for each key in the schema. Based on this discussion and measurement [10], it causes a slight slowdown even if they never enter the Except block. Also, there is an enormous number of if and elif statements. These are necessary for the schema parser to behave differently for each record type. However, if we know the schema ahead of time, we can rule out most of them.

We have also analyzed the code of Fastavro and found that it is not using any compiler directives supported by Cython. Many of these directives may speed up the resulting code. We explore all these options in this work.

3.2 Kafka Confluent Platform

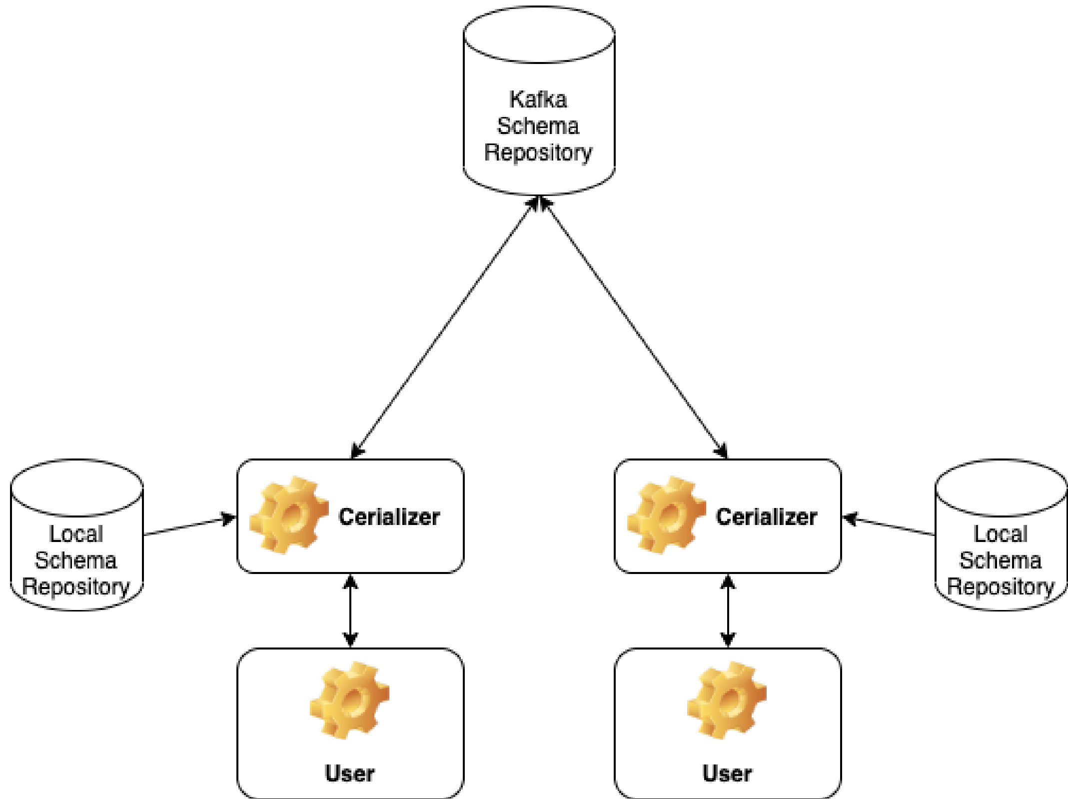
Kafka Confluent Platform is a full-scale event streaming platform that enables you to easily access, store, and manage data as continuous, real-time streams [2]. We use the Confluent Platform as a repository for our schemata. We can access the schemata by connecting to a running Confluent Platform and then looping over every messaging topic and retrieving the corresponding schema.

3.3 Solution architecture

In Figure 3.1, we can see the architecture of Cerializer. Each user runs an instance of Cerializer; these instances provide the users with the ability to serialize and deserialize data.

There are two ways of supplying schemata to Cerializer. First, the user can provide connection details to a running Kafka Confluent Platform. Cerializer then automatically loads all the schemata and spawns a daemon that periodically checks the Confluent Platform for any added topics and schemata.

Figure 3.1: Solution architecture



Second, the user can provide a local schema repository. The local schema repository does not contradict the Confluent Platform. There are use-cases when these two are used in synergy. For example, the user could decide to store public schemata in the Confluent Platform and private schemata in the local repository.

3.4 Object design

Cerializer is designed as a Python library. The main interface is based around the Cerializer and Cerializer Schemata classes. Cerializer class is used to serialize and deserialize data, and the Cerializer Schemata are used to store schemata and compile code.

Altogether, the project is divided into four classes, each one having a specific set of responsibilities. All the responsibilities are talked about below in the corresponding sections.

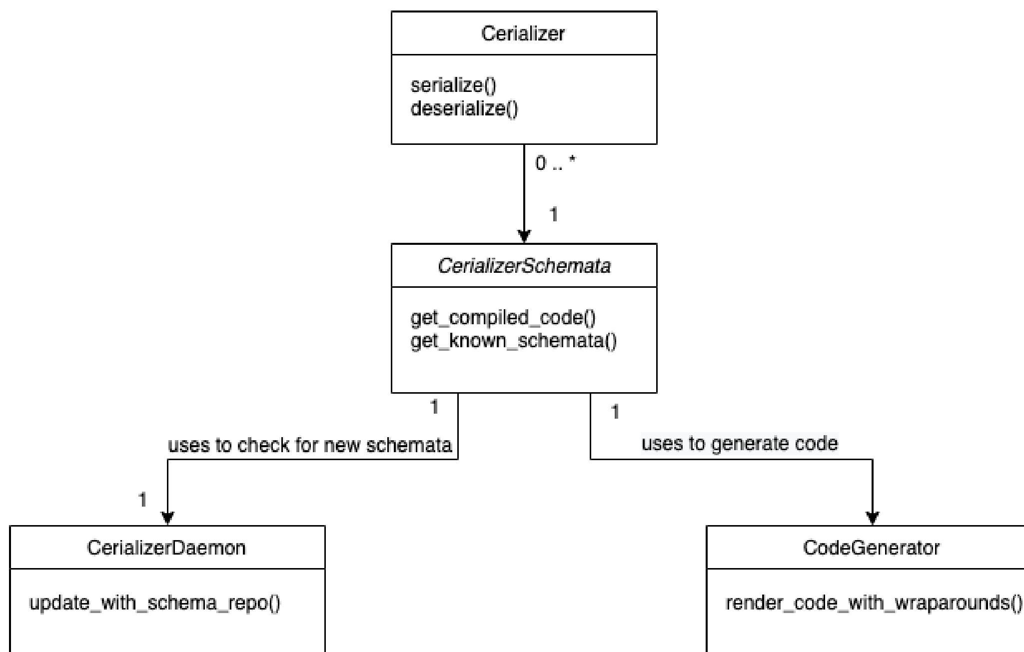
As far as the object design is concerned, please refer to Figure 3.2.

There are multiple instances of Cerializer, each dedicated to a specific schema. The number of these instances does not have to be limited since there are no computational expenses for initializing an instance.

There is also a minimal number of instances of Cerializer Schemata. We decided to store already compiled code in Schemata and hence avoid multiple compilations of the same code in each instance of Cerializer.

A Code Generator is required for each schema separately, and therefore, the Cerializer Schemata instance connects multiple Code Generator instances.

Figure 3.2: Object Design



Each Cerializer Schemata instance spawns one instance of Cerializer Daemon to periodically check for new schemata in the schema database (Kafka).

3.5 Cerializer Schemata

Cerializer Schemata is the umbrella class for this entire project. The class is designed in such a way that the number of its instances should be minimized. It stores the schema database along with the compiled code. The decision of why we store even the compiled code and not only the schemata themselves is further discussed in the section 3.5.2.

We also need to note that it is essential to store all the known schemata in one Cerializer Schemata instance since the Avro standard allows for references between schemata.

3.5.1 Schema fetching

Cerializer schemata is designed in such a way that it can fetch schemata from both Kafka schema repository and or load them from a list of tuples supplied. The reason for this decision was to make it easier for users who use Kafka and also for users who keep their schemata in the adequate package repositories or a combination of the two. The reason for not supporting only the list of tuples and not letting the user first download all the Kafka schemata by themselves was to have the possibility of checking the repository for any new schemata. Otherwise, the class would have to be reinitialized every time there is a new schema released.

The checking occurs in Cerializer Daemon, which runs in a separate thread for each Cerializer Schemata instance. One can read more about the daemon in

the section 3.6 section.

3.5.2 Storage of compiled code

As stated before, this decision of compiling the code in schemata was not straightforward. Schemata storage is computationally lightweight; it is only about storing several YAML files or Python dictionaries. However, in practical use, one wants to serialize according to a single schema in multiple parts of the code. Since serialization needs a compiled code, the code would have to be recompiled each time. This led us to move the responsibility of code compilation to CerializerSchemata. This way, it is only done once and then each Cerializer instance can request the corresponding code.

The schema code compilation happens during the class initialization and then, if any new schemata occur, in Cerializer Daemon. The compilation itself is carried out in a slightly modified method from Cython. More about this in the section section 3.8.

3.5.3 Cycle detection

As the Avro standard allows for schema references, we needed to deal with cycles in our schema database. Schema referencing means that instead of defining a particular part of a schema, we define that this subsection of our new schema is the same as some other schema in our database. To show this on a simple example, we can imagine a user schema that contains name and age. Then, if we wanted to define a trade schema that contains two users, the seller and the buyer, we would reference the schema user twice instead of defining the fields for age and name again.

These schema references do not cause any significant problems during the schema analysis on run time, such as in Fastavro. When the algorithm encounters a reference to another schema, it only recurses deeper. However, if we want to generate code for such a schema, let us imagine a tree schema, where the left son can be a value or a tree again, we would end up in an infinite cycle. The algorithm would generate the serialization code for the parent and then the duplicate copy of this code in both of the branches corresponding to the child nodes.

To avoid this, we needed to detect cycle starting schema identifiers. The cycle detection responsibility was moved to Cerializer Schemata since it is the only class with access to the entire schema repository and can comprehend the general structure and bonds between schemata.

When cycles are to be detected, a DFS algorithm is run that goes through the entire database of schemata and stores schema identifiers responsible for cycles. Then, when other parts of the Cerializer system need to know whether the node is cycle starting, they can request such information from the instance. For example, the set of cycle starting nodes is then used during the code generating process in Code Generator.

The decision to keep these methods responsible for cycle detection in the Schemata instance and not the Code Generator was made because we saw the cycles as an inherent property of the schema database, not only each schema themselves.

3.6 Cerializer Daemon

Cerializer Daemon is a class responsible for fetching schemata from the Kafka schema repository. It runs on a second thread and periodically checks the repository for new schemata. If new schemata are found, they are first added to the Cerializer Schemata instance by a callback method. This does not mean these schemata can be used to create Cerializer instances; they are added only to avoid any missing references. An issue could occur if we added two schemata at once to our schema repository, and one of them would be referencing the other. If the daemon started the code compilation straight away, an error would be thrown caused by the missing reference and the referenced schema not yet being added to the instance.

After the schemata are added to the Cerializer Schemata instance, the daemon starts to generate and compile the corresponding code. This process can be lengthy, depending on the size and complexity of the schemata. We have observed that some schemata can take up to a minute or two to compile on a regular laptop. This would be faster on a conventional server; however, even several seconds of delay could cause unnecessary problems to the rest of the system. Fortunately, since the daemon runs on a second thread, the serialization and deserialization can continue without any delays. After the code is compiled, it is added to the schemata instance by another callback. Now the compiled code can be requested to construct a new Cerializer instance.

3.7 Code Generator

The Code Generator is the essential class of the project. It handles the code generation for both serialization and deserialization. The way this class is designed, there needs to be a new instance created for each schema we want to generate code for. This is mainly because the generator keeps a lot of internal state variables and other generators, such as the name generators talked about later. It would be possible to reset all the state variables of the generator after each code generating run, but it seemed counter-intuitive.

In Listing 3.1 we can see the corresponding code for our example student schema. The two essential methods are serialize and deserialize on lines 13 and 22, respectively.

On lines 9-12, we can see Cython compiler directives, which were proposed as a potential speed up in section 3.1. These directives are described in section 3.8.

All the imported modules except for Cython itself are discussed in subsection 3.8.1.

Listing 3.1: Generated code example

```
1 #cython: language_level=3
2 cimport write
3 cimport read
4 cimport prepare
5 import cython
6
7
```

```
8
9 @cython.boundscheck(False)
10 @cython.wraparound(False)
11 def __invoke():
12
13
14     def serialize(data, output):
15         cdef bytearray buffer = bytearray()
16         write.write_string(buffer, data['name'])
17         write.write_int(buffer, data['age'])
18         write.write_float(buffer, data['average'])
19         output.write(buffer)
20
21
22     def deserialize(fo):
23         cdef dict data = {}
24         data['name'] = read.read_string(fo)
25         data['age'] = read.read_int(fo)
26         data['average'] = read.read_float(fo)
27         return data
28
29
30     return locals()
```

3.7.1 Jinja2 templates

To generate code efficiently, Code Generator utilizes multiple templates written in jinja [6]. Jinja2 is a templating language for Python which can be used to generate code or markups.

We can see an example of such template in Listing 3.2. This very template ensures the general structure of the inside of the ___*invoke* function in Listing 3.1. Jinja2 not only allows for global environment variables such as buffer_name but also for calling functions directly such as in Listing 3.3.

Cerializer uses these templates to generate code for complex data types and also for the general structure of the resulting code. All the templates can be found in the templates directory of the project.

Listing 3.2: Jinja2 template example

```
1
2 def serialize({{ location }}, output):
3     cdef bytearray {{ buffer_name }} = bytearray()
4 {{ serialization_code|indent(4, True) }}
5     output.write({{ buffer_name }})
6
7
8
9 def deserialize(fo):
```

```
10 {{ deserialization_code|indent(4, True) }}
11     return {{ location }}
12
13
14 {{ necessary_defs }}
```

Listing 3.3: Jinja2 function call example

```
1 {{ generate_serialization_code(values, val_name) }}
```

3.8 Compilation

The compilation itself happens using the Cython inline function. This function accepts a string representing the code and returns the output of the __invoke function. The output has a form of a dictionary representing the local symbol table.

In Listing 3.1 on lines 9-12 we can also see Cython compiler directives. These directives could potentially speed up the code since they are removing certain safety features.

Both compiler directives are related to array indexing. Cython boundscheck turns off checking for out of bounds access to array items. Following the nature of the auto-generated code, we can assume that such behaviour will never happen since any indexing into an array is done based on the array's length. Cython wraparound then turns off Python's negative indexing. We can see an example of negative indexing in Listing 3.4. Negative indexing can be very useful in data science, but since our code is not utilizing this feature, we can turn it off and save time.

Listing 3.4: Negative indexing code example

```
1
2 my_array = [0, 1, 2, 3]
3 last_element = my_array[-1]
4 second_to_last_element = my_array[-2]
5
6 print(last_element)
7 >>> 3
8
9 print(second_to_last_element)
10 >>> 2
```

3.8.1 read.pxd, write.pxd and prepare.pyx

As we could see in Listing 3.1 on lines 16-18 and 24-26, basic types are serialized using methods or functions from the read and write modules. These files were mostly taken from Fastavro. However, since there were many unnecessary

arguments passed into the functions, we deleted these. Fastavro decided to have these arguments to keeping integrity between all these functions (all of them had the same arguments). This way, there could be only a simple generic mapping between a data type and the corresponding function. However, since we generate a custom code, we could account for that and keep the methods more straightforward.

There is also a "prepare.pyx" module that deals with the preparation of logical types. This means turning a logical type such as DateTime into a primary type such as int. In practice, this could be done by converting the DateTime object into a UNIX timestamp or converting the time into miliseconds, as we can see in Listing 3.5. This primary type is then serialized accordingly. We can see this behaviour in Listing 3.6 on line 16 and then the reverse process in line 27.

Listing 3.5: Prepare code example

```
1  cdef inline prepare_time_micros(object data):
2          if isinstance(data, datetime.time):
3                  return int(
4                  data.hour * MCS_PER_HOUR
5                  + data.minute * MCS_PER_MINUTE
6                  + data.second * MCS_PER_SECOND
7                  + data.microsecond
8                  )
9          else:
10                 return data
```

Listing 3.6: Logical type code example

```
1  #cython: language_level=3
2  cimport write
3  cimport read
4  cimport prepare
5  import cython
6
7
8
9  @cython.boundscheck(False)
10 @cython.wraparound(False)
11 def __invoke():
12
13
14     def serialize(data, output):
15         cdef bytearray buffer = bytearray()
16         write.write_long(
17             buffer,
18             prepare.prepare_time_micros(
19                 data['date']
20             )
```

```
21          )
22          output.write(buffer)
23
24
25      def deserialize(fo):
26          cdef dict data = {}
27          data['date'] = prepare.read_time_micros(
28              read.read_long(fo)
29          )
30          return data
31
32
33      return locals()
```

3.9 Cerializer

Having distributed all the responsibilities to the above mentioned classes, the sole purpose of Cerializer is to serve as a wrapper for the serialize and deserialize methods.

When a new Cerializer instance is created, it fetches compiled code from the schema database supplied. The code contains two methods: serialize and deserialize. The Cerializer instance saves these two methods. When a user then wants to serialize and deserialize data, they call the cerializer_instance's serialize (or deserialize) methods. These two methods wrap around the compiled methods in order to keep the same interface as Fastavro. This decision was made to remain consistent with the current solutions and make it easier for potential new users to switch.

4. Evaluation

In this chapter, we evaluate the Cerializer project. We show a practical use case of Cerializer, describe our schema generator, and discuss the benchmark results. We also discuss the procedure of the benchmarks and the module chosen to time the process.

4.1 Code usage examples

To use Cerializer, we first have to initialize an instance of CerializerSchemata. As we can see in the example below, we have done so by providing the constructor with a list of tuples containing our local schemata and a Kafka schema repository URL.

To obtain a Kafka repository and subsequently the corresponding URL, we first start a local Confluent platform. We do so by following this quick start tutorial [3]. Then, if we decided to add a topic and restrain it by a schema, Cerializer Schemata would automatically download such schema and compile the corresponding code.

To load all our local schemata, we have created a simple function that returns a list of tuples in the form of (schema identifier, schema). Note that both the schema and data variables on lines 11 and 23 refer to the data in Listing 2.1 and Listing 2.2. Typically we would iterate over multiple directories of schemata and produce a much longer list. However, we have limited our database to just one local schema for the simplicity of this example.

Then, we only need to initialize an instance of Cerialzier. To do so, we call the constructor and provide it with the schemata instance together with the schema name and schema namespace. Note that we must structure the schema identifier as schema_namespace.schema_name. The initialization is done on line 18 and can be repeated for multiple schemata without any overhead. The instantaneous initialization was made possible by compiling the code in Cerializer Schemata and not in the Cerialzier instances.

To serialize data into binary format, all we have to do is call the serialize function on our cerializer instance. Note that since the cerializer instance is dedicated solely to the student schema, no other data types can be serialized. To reverse the process of serialization, we can call the deserialization function. The data is then returned in the form of a Python object.

Listing 4.1: Code usage example

```
1 from cerializer.schemata import CerializerSchemata
2 from cerializer.cerializer import Cerializer
3
4
5
6 KAFKA_URL = '...'
7
8 def get_schemata_from_local_repository():
```

```
9      # iterates through all your schemata and returns
10     # a list of (schema_identifier, schema) tuples
11     return ['school.student', schema]
12
13 cerializer_schemata = CerializerSchemata(
14     get_schemata_from_local_repository(),
15     KAFKA_URL
16 )
17
18 cerializer_instance = Cerializer(
19     cerializer_schemata,
20     'school',
21     'student'
22     )
23
24 serialized_data = cerializer_instance.serialize(data)
25 print(serialized_data)
26
27 deserialized_data = cerializer_instance.deserialize(
28     serialized_data
29     )
30 print(deserialized_data)
```

Listing 4.2: Code output

```
1 b'\x16Matej␣Micek.\x00\x00\x80?'
2
3 {
4     "name": "Matej␣Micek",
5     "age": 23,
6     "average": 1.0
7 }
```

4.2 Schema Generator

We designed a simple schema generator to evaluate Cerializer on other schemata than those mentioned in Table 2.1. This generator can produce schema and data YAML files stored the same way as the schemata in the test directory of Cerializer. The same storing logic allows for simple integration into Cerializer testing and benchmarks.

Due to the complexity of schema generation, we need to note that we limited the generated types to strings, integers, booleans, maps, arrays, and unions. Further, we refer to the former three as primitive types and the latter three as complex. These names copy the terminology of the Avro specification [1].

We also limited the depth of the data to 1, meaning it never generates an array of arrays or a map of maps. An example of such a schema is the huge_schema in our test set.

The generator takes three arguments in total:

- string length to limit the number of characters in field names and string values
- the number of primitive fields
- the number of complex fields

Based on these three variables, it produces a schema and the corresponding data example. We can see an example of such output in Listing 4.3 and Listing 4.4.

Listing 4.3: Generated schema example

```
name: Schema_2_2
namespace: generated
type: record
fields:

- name: mLqNiFoCDvF
  type: string

- name: pRQynKwc
  type: string

- name: AKXjFSGOEBV
  type: [boolean, string]

- name: drulMMYyrcPUFDo
  type: [int, string]
```

Listing 4.4: Generated data example

```
mLqNiFoCDvF: wIQAstsJoaIOPddrmFHe

pRQynKwc: BkSoFGnzxxqwM

AKXjFSGOEBV: EQCAJoUN

drulMMYyrcPUFDo: 8846617
```

4.3 Benchmarks

This section argues why we chose to use the timeit module and discuss the benchmark results produced by comparing Cerializer to its two main competitors: Fastavro and Avro for Python.

4.3.1 Timeit.timeit module

The timeit.timeit is a standard Python module containing tools for measuring the execution time of code snippets [14]. Unlike measuring the duration of a code execution by wrapping it by time statements as in Listing 4.5, the Timeit module has multiple features to prevent any potential influences on the measurement [13].

First, it provides the user with a simple interface for running the tests multiple times. We can see the parameter number in our example snippet in Listing 4.6. There, we used the number one just for comparison to the time module. Running the tests multiple times lowers the influence of other tasks on the system and any OS processes.

Second, the Timeit module turns the garbage collection off during the tests.

And last but not least, it picks the most accurate timer for your operating system. The default timer for the Timeit module is the time.perf_counter [12].

Listing 4.5: Timing using the Time module

```
1  import time
2
3
4  def do_something():
5          for _ in range(100):
6                  i = list(range(10**5))
7                  sum = 0
8                  for item in i:
9                          sum += item
10
11 start = time.time()
12
13 do_something()
14
15 print(time.time() - start)
16 >>>0.695213794708252
```

Listing 4.6: Timing using the Timeit module

```
1  import timeit
2
3
4  print(timeit.timeit(do_something, number = 1))
5  >>>0.614848332999999
```

4.3.2 Benchmark procedure

As speed is technically the only measure of success of our project, we needed to perform benchmarks to compare it to its competitors. To do so, we have created a benchmarking system. The system is based on the timeit module mentioned in the previous section.

When a schema is benchmarked, we measure the time to serialize the data for three systems:

- Cerializer,
- Fastavro,
- and Avro for Python.

To measure the time as objectively as possible, we measure the time of one but one thousand iterations. Each iteration consists of serializing the data into binary format in memory and deserializing the data back into a Python object. Data preparation and code compilation happen in setup and are thus not measured.

Furthermore, we split the total one thousand iterations into two equally sized parts and serialize them separately. We then mix the serialization order and serialize the data in a zig-zag (Cerializer, Fastavro, Avro, Fastavro, Avro, Cerializer). To measure the time, we use the timeit module.

We add the two measurements and normalize them to the interval [0, 1] across the three systems to acquire the final benchmark report. The normalization is necessary since we want to include an average performance in our output. The average performance across all the benchmarked schemata is then acquired by averaging all the normalized time measurements.

4.3.3 Custom schemata benchmark

We evaluated Cerializer on the set of schemata mentioned already in Table 2.1. These schemata were designed manually, mostly to test all the primitive datatypes and also combinations with complex datatypes. We can see the results of the benchmark in Table 4.1.

In the last two columns of Table 4.1, we can see the speedup ratio calculated for Fastavro and Avro. We can see the tendency of better Cerializer performance with maps and nested maps (nested schema).

4.3.4 Generated schemata benchmark

We also used our Schema Generator to evaluate Cerializer's performance more generally. We generated schemata, each containing between 1 and 30 primitive and complex types. We have benchmarked using the same procedure as before. We need to note that Avro for Python was omitted in this benchmark since it was more than ten times slower.

Having collected the results, we plotted two graphs. The X-axis represents the corresponding number of fields for either complex or primitive type. The Y-axis shows the proportional difference between the time to serialize by Fastavro and Cerializer.

In Figure 4.1 we can see that the performance of Cerializer with primitive types is oscillating around 1.6 times faster mark. As seen in Figure 4.2, the performance is even better with complex types, converging towards 1.9.

4.4 Tests

To evaluate the speed and the correctness of the Cerializer, we have created a set of tests. These tests aim mainly at the proper functioning of Cerializer schemata

Table 4.1: Cerializer Benchmark

schema	Ceri. [s]	Fast. [s]	Avro [s]	A/C	F/C
cerializer.array__int__str	0.0336	0.0971	1.00	29.78	2.89
cerializer.enum	0.0581	0.0895	1.00	17.20	1.54
cerializer.str	0.0846	0.1177	1.00	11.82	1.39
cerializer.time__micros	0.0778	0.1537	1.00	12.85	1.97
cerializer.union	0.0479	0.0846	1.00	20.87	1.76
cerializer.reference	0.0523	0.1032	1.00	19.13	1.97
cerializer.fixed	0.0665	0.0964	1.00	15.04	1.45
cerializer.int	0.1225	0.1947	1.00	8.16	1.59
cerializer.fixed__decimal	0.2597	0.3377	1.00	3.85	1.30
cerializer.array__str	0.0891	0.1184	1.00	11.23	1.33
cerializer.decimal	0.1698	0.2796	1.00	5.89	1.65
cerializer.nested	0.0460	0.1206	1.00	21.72	2.62
cerializer.map__str	0.1076	0.1392	1.00	9.29	1.29
cerializer.map__int__null	0.0698	0.0955	1.00	14.32	1.37
cerializer.double	0.0738	0.1041	1.00	13.55	1.41
cerializer.bytes	0.0629	0.0944	1.00	15.90	1.50
cerializer.long	0.0692	0.1076	1.00	14.46	1.56
cerializer.date__int	0.0917	0.3163	1.00	10.91	3.45
cerializer.array__bool	0.0321	0.0686	1.00	31.18	2.14
cerializer.boolean	0.1507	0.3007	1.00	6.64	2.00
cerializer.array__int	0.0405	0.0550	1.00	24.67	1.36
Average	0.0860	0.1464	1.00	11.63	1.70

Figure 4.1: Performance graph primitive types

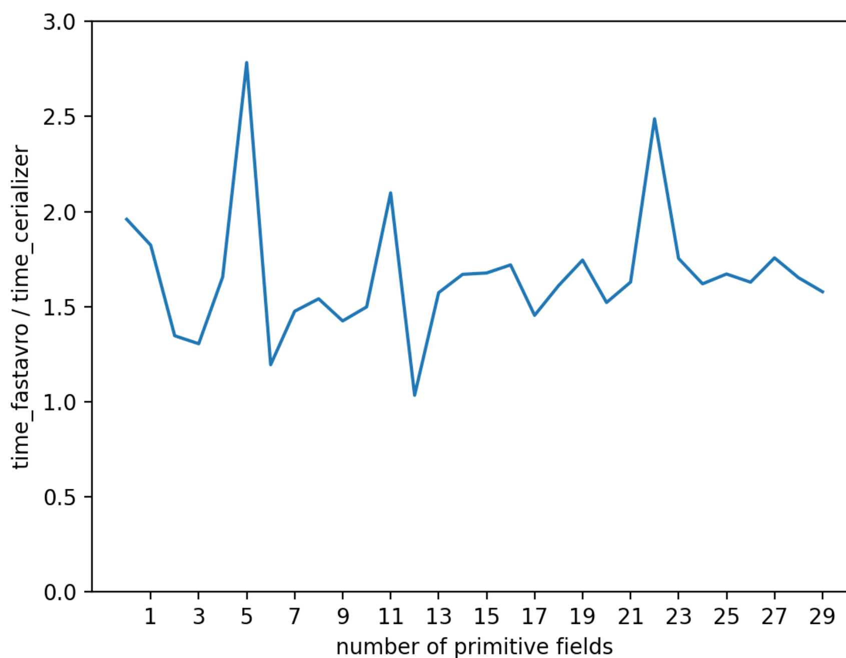
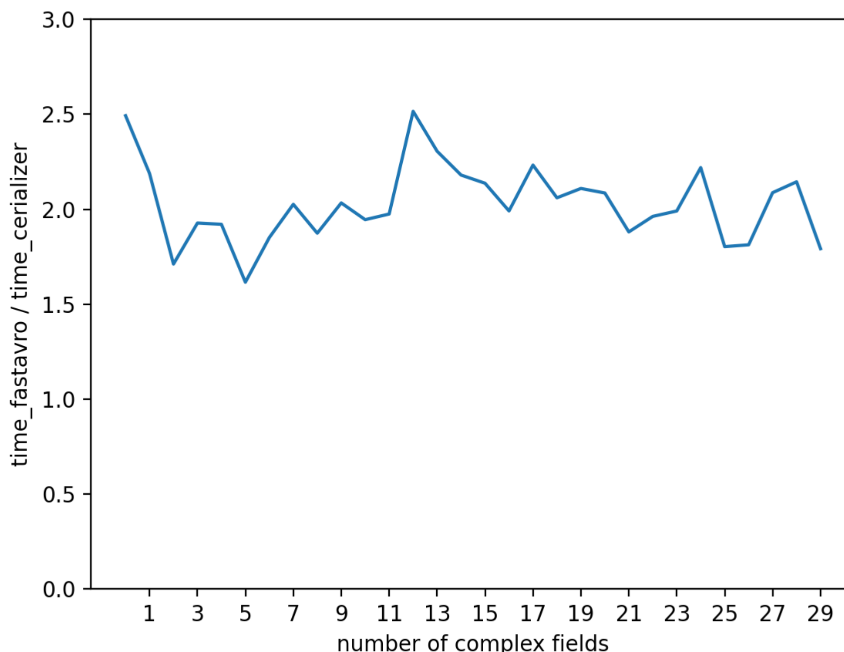


Figure 4.2: Performance graph complex types



and Cerializer itself. Cerializer daemon is mainly tested with Cerialzier schemata since its main functionality relies on the already tested callbacks. The tests for Cerializer are mainly testing the Code generator since Cerializer itself is only a wrapper around the generated code.

4.4.1 Cerializer schemata

When testing Cerializer schemata, we first tested whether the schemata added were accepted and noted by the instance. Then, we checked whether all the schemata that needed to be compiled got compiled. In Listing 4.7 we can see two tests aiming at precisely this problem. The first test asserts that the schema was recognized when a schema was added to an existing instance. However, it also asserts no code generated (this would happen in the Cerializer daemon, and the code would be added by another callback later on). The second tests assert that there are both code and schema recognized in the instance. This behaviour is different because the schema was added during the initialization, meaning the code should be compiled in schemata.

In the test set, we also emphasize cycle detection and schema loading. These two methods are necessary for the system not to end up in infinite cycles. We can find all the tests in the tests directory of the project.

Listing 4.7: Example of tests in Cerializer schemata

```
1 def test_schemata_add():
2         schemata = CerializerSchemata([])
3         schemata.add_schema(SCHEMA_IDENTIFIER, SCHEMA)
4         assert SCHEMA_IDENTIFIER in schemata
```

```
5           known_schemata = schemata.get_known_schemata()
6           # we do not yet have code for the schema
7           assert SCHEMA_IDENTIFIER not in known_schemata
8
9
10  def test_schemata_init():
11          schemata = CerializerSchemata(
12              [
13                  (SCHEMA_IDENTIFIER, SCHEMA)
14              ]
15          )
16          assert SCHEMA_IDENTIFIER in schemata
17          known_schemata = schemata.get_known_schemata()
18          # we already have code for the schema
19          assert SCHEMA_IDENTIFIER in known_schemata
```

4.4.2 Cerializer

When testing Cerializer, we compared the output of both of its functions with the corresponding counterpart of Fastavro. We provided the two systems with the same schema and data and asserted that the serialized and later the deserialized data are the same.

As the test set, we used a union of the hand-crafted custom schemata and all the generated schemata.

To parametrize the test, we used the pytest parametrize decorators. Using these, we were able to write tests as if we were testing only one schema and then let pytest iterate over the entire database. We also utilized the pytest fixtures, which saved us from compiling the entire schema database every time a schema is tested.

4.5 Limitations

In this section, we go over Cerializer's limitations. First, we need to state that Cerializer supports only the schemaless writer. We explained the decision not to support the regular writer in this work. However, the absence of regular writer incentivises the user to think twice about the necessity of transferring schemata together with their data.

Second, Cerializer does not support schemata with unions of two or more complex types of the same kind. An example of such schema is a union of an array of type integer or string and the second array of type string. The reason for not supporting such schema is that the computational complexity of verifying which array type to choose is proportional to the array's length (to distinguish between these two arrays, we would have to check the types of all the elements). We have also found that such behaviour is usually not necessary or easily avoidable.

Last but not least, Cerializer is limited to receiving only data in known formats. It restricts Cerializer from reading any arbitrary Avro data without knowing the schema. Cerializer also does not support reading data written in a different

schema than the one the data was encoded with. Avro supports this feature, but it adds time-consuming checks and verifications. Users seeking fast serialization should avoid this and use an Avro migrator in times of absolute necessity.

4.6 Related work

As of the time of writing, there is not a lot of related work available. The only related work found is the one of RTB House company based in Poland [19]. They developed a custom solution to Avro serialization in Java. Even though the Java implementation of Avro is comparable to Fastavro in terms of speed, they achieved 3-5 fold speedups in their production schemata. They achieved this by generating custom code for each schema and by avoiding the schema analysis part.

The benchmarks done on generated schemata showed nearly the same results as Cerializer: 1.5 fold speedup for flat schemata and 1.8 fold speedup for deeper schemata.

This solution is now adopted and maintained by LinkedIn.

5. Conclusion

The aim of this work was to speed up serialization in Python. To achieve this goal, we have analyzed the problem of serialization, chosen a fast serialization format (Avro) and made a decision to support only Avro schemaless writer. We also identified the need for a schema distribution system.

We have achieved the goal by developing Cerializer, a serialization solution based on automatic code generation. Cerializer is capable of generating a custom Cython code for an Avro schema. This way, it saves time on analyzing the schema every time a data record needs to be written. This approach has never been experimented with in Python.

Cerializer uses Cython methods to compile the code and store it for future use. The compilation is carried out in a way that does not block any running processes in the system.

We have also developed a solution for updating an entire distributed systems with up-to-date schemata. We did so by equipping each Cerializer instance with a daemon that checks the user's Kafka Confluent Platform for new schemata.

We have evaluated our solution by benchmarking it against the two main Python Avro serialization libraries: Avro for Python and Fastavro. To benchmark the solution, we have used a set of custom schemata and also a schema generator. The results have shown a 1.6 - 3.5x speed-ups compared to the faster of the two.

5.1 Future work

We see a promising future in Cerializer, and we think that implementing even the rest of the Avro standard would potentially attract more audience. These parts are especially the regular writer that writes not only the data but also the schema. Adding this feature would mean only a tiny adjustment, but we need to note that it contradicts in the very essence the speed requirements.

In the future, Cerializer could also receive support for more schema repositories than Kafka. The Cerializer daemon could be generalized, and the method for fetching the new schemata could be provided by the user.

Bibliography

[1] Apache avro specification. https://avro.apache.org/docs/current/spec.html. Accessed: 10.5.2021.

[2] Confluent platform documentation. https://docs.confluent.io/platform/current/platform.html.

[3] Confluent platform quick start (local). https://docs.confluent.io/5.5.0/quickstart/ce-quickstart.html. Accessed: 10.5.2021.

[4] Cython documentation. https://cython.readthedocs.io/en/latest/src/tutorial/cython_tutorial.html. Accessed: 10.5.2021.

[5] Documentation: Table of contents — rabbitmq. https://www.rabbitmq.com/documentation.html. Accessed: 10.5.2021.

[6] Jinja documentation. https://jinja.palletsprojects.com/en/2.11.x/. Accessed: 10.5.2021.

[7] Json documentation. https://www.json.org/json-en.html. Accessed: 10.5.2021.

[8] Microservice architecture. http://microservices.io/patterns/microservices.html. Accessed: 10.5.2021.

[9] Pypl popularity of programming language index. https://pypl.github.io/PYPL.html, abstractNote=PYPL popularity of programming language.

[10] Python exceptions costs statistics. https://stackoverflow.com/questions/2522005/cost-of-exception-handlers-in-python. Accessed: 10.5.2021.

[11] Python wheels. https://pythonwheels.com/. Accessed: 10.5.2021.

[12] Time documentation. https://docs.python.org/3/library/time.html#time.perf_counter. Accessed: 10.5.2021.

[13] Timeit advantages. https://stackoverflow.com/questions/17579357/time-time-vs-timeit-timeit. Accessed: 10.5.2021.

[14] Timeit module documentation. https://docs.python.org/3/library/timeit.html. Accessed: 10.5.2021.

[15] What is csv? http://super-csv.github.io/super-csv/csv_specification.html. Accessed: 10.5.2021.

[16] Apache avro github repository. https://github.com/apache/avro, Apr 2021. Accessed: 10.5.2021.

[17] Fastavro github repository. https://github.com/fastavro/fastavro, Apr 2021. Accessed: 10.5.2021.

[18] Miki tebeka github page. https://github.com/tebeka, Apr 2021. Accessed: 10.5.2021.

[19] P. Jaczewski. Our approach to fast avro serialization and deserialization in jvm. https://techblog.rtbhouse.com/2017/04/18/fast-avro/, Apr 2017. Accessed: 10.5.2021.

[20] R. Sharma. Quantitative trading definition. https://www.investopedia.com/terms/q/quantitative-trading.asp. Accessed: 10.5.2021.

A. Appendix

A.1 Cerializer.zip

The contents of the zip file are organized as follows:

- **Cerializer**
  - requirements, setup.py and shell scripts
  - README.md for usage examples and user documentation
- **Cerializer/cerializer**
  - main project files
  - /tests contains test schemata and test scripts
- **Cerializer/cerializer_demo**
  - demo scripts
- **Cerializer/cerializer_docs**
  - Sphinx technical documentation
  - html documentation available in _build/html/index.html