Alvarium SDK Performances Report

BATISTA Maxime

May 27, 2024

Abstract

This report compares some benchmark results of the current Java and Rust SDK versions. The Java is first much slower than its Rust counterpart, and the profiling analysis will show that this is due to 1) the annotations are being signed individually, which causes both Rust and Java performances to be impacted by the amount of annotations. 2) some issues are specific to the Java SDK, such as bad use of the JSON serialization library, the signing process, and unique identifier generation. In the third part we raise several optimisation ideas, and we'll see that they all have a significant performance improvement. The SDK version with all optimisations being applied even beats the Rust version very quickly.

Contents

1 Introduction **1**

2 Definitions **2**

3 Current state **2**
- 3.1 Benchmark results . . . 2
  - 3.1.1 Impact of annotations count . . . 2
  - 3.1.2 Impact of annotated data size . . . 3
- 3.2 First observations . . . 4
- 3.3 Profiling results . . . 4

4 What can be optimized **6**
- 4.1 Non-breaking changes . . . 6
  - 4.1.1 Annotation serialization . . . 6
  - 4.1.2 Hash data once per annotation . . . 7
  - 4.1.3 Cache ULID Generator . . . 8
- 4.2 Annotation generation . . . 10
- 4.3 Signing process . . . 13
  - 4.3.1 Signing proposal : Identity Strings . . . 14
  - 4.3.2 Results . . . 15
- 4.4 Overall comparison . . . 16

5 Conclusion **17**

1 Introduction

This report will do some benchmark tests and profiling of the Java Software Development Kit from commit version 7e53f04.

We will analyze the Profiling results of the current implementation of the Java SDK, and will provide some optimisation ideas to help drastically reduce the Alvarium's overhead.

2 Definitions

SDK call: The expression `SDK call` is used to designate the different kind of data annotation operations we can do using Alvarium. `SDK call` designates either the `create`, `mutate`, `transit` or `publish` actions

3 Current state

3.1 Benchmark results

The benchmark is done on an intel i5-6500.

The produced annotations always comes from the same annotator being replicated : the `Source` annotator.

This annotator has been selected for its insignificant CPU consumption (just checks if the computer has a hostname), which is useful to only measure the Alvarium's overhead, regardless of the annotators CPU consumption.

3.1.1 Impact of annotations count

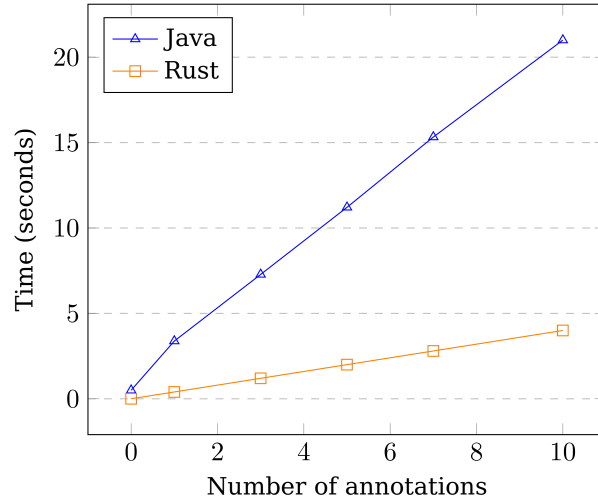
Measure the impact of the amount of annotators when annotating 20000 bytes, 5000 times

Java

annotations count	min	avg	max
0	0.33 s	0.511 s	1.022 s
1	3.08 s	3.38 s	4.681 s
3	6.997 s	7.287 s	8.581 s
5	10.682 s	11.217 s	13.691 s
7	15.028 s	15.325 s	15.8 s
10	19.9 s	20.99 s	22.868 s

Rust

annotations count	min	avg	max
0	1.8751 ms	2.1038 ms	2.5666 ms
1	414.12 ms	416.47 ms	419.13 ms
3	1.2216 s	1.2345 s	1.2503 s
5	2.0533 s	2.0740 s	2.0950 s
7	2.8562 s	2.8649 s	2.8734 s
10	4.0410 s	4.0642 s	4.0907 s



3.1.2 Impact of annotated data size

Measure the impact of the size of the data annotated by 5 annotators, 5000 times.

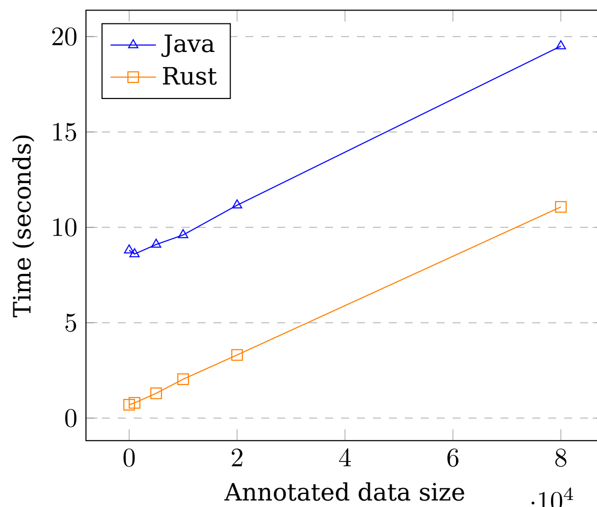
Java

data size (bytes)	min	avg	max
0	8.544 s	8.837 s	9.705 s
1000	8.447 s	8.67 s	8.93 s
5000	8.88 s	9.193 s	9.409 s
10000	9.426 s	9.693 s	9.993 s
20000	10.592 s	11.16 s	11.559 s
80000	18.508 s	19.455 s	21.489 s

Rust

data size (bytes)	min	avg	max
0	720.28 ms	760.02 ms	811.40 ms
1000	824.60 ms	825.44 ms	826.25 ms
5000	1.3717 s	1.3884 s	1.4110 s
10000	2.0358 s	2.0414 s	2.0463 s
20000	3.2777 s	3.3196 s	3.3600 s
80000	10.853 s	11.071 s	11.372 s

Benchmark results of Java compared to Rust, lower is better

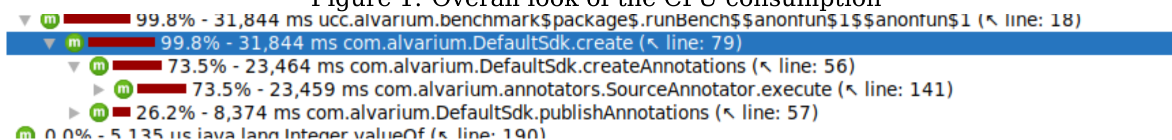


3.2 First observations

Without any surprise, the Java SDK is much slower than its Rust counterpart, but the number of annotations per sdk call seems to affect more the java sdk's performances, as suggested by the curve steepness of the first java plot.

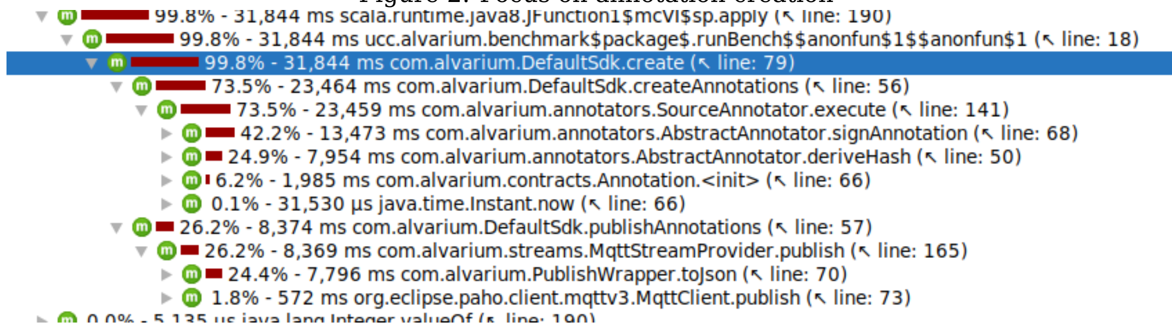
3.3 Profiling results

Figure 1: Overall look of the CPU consumption



We can see that a SDK call is composed of two main operations : Create the annotations, and publish the annotations. Taking 73.5% and 26.2% of the whole CPU time, respectively.

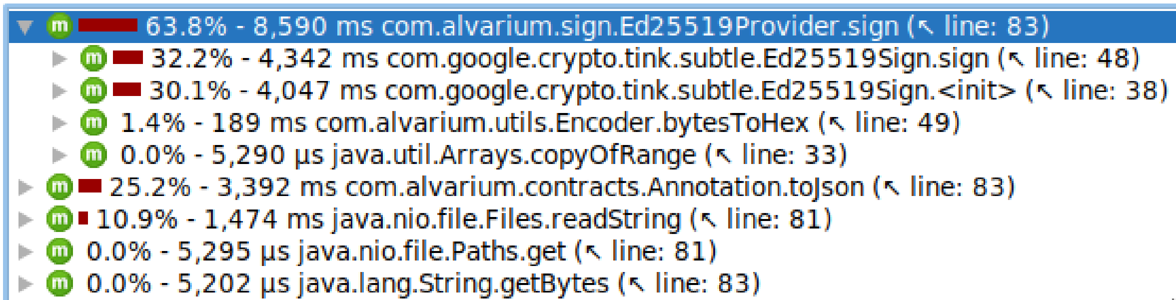
Figure 2: Focus on annotation creation



The main overhead of annotation creation seems to be due to the cost of signing the annotations, and the hashing of the data.

We can see that the data is hashed for every annotators (the hashing process occurs inside the `SourceAnnotator.execute` function), which is not necessary as all the annotators will handle the same data.

Figure 3: Focus on annotation signature



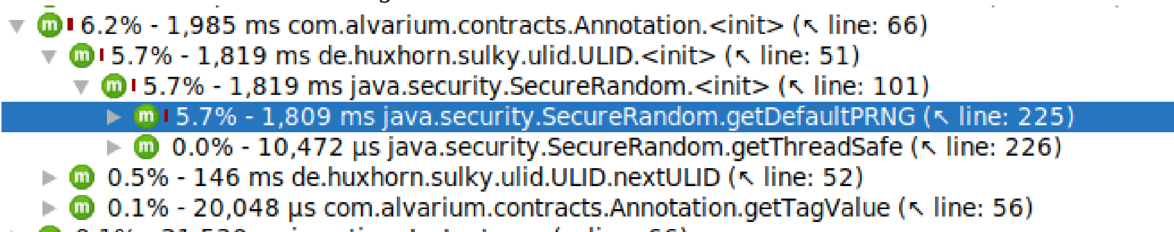
NOTE: percentages here are relative to the `AbstractAnnotator.signAnnotation` method and not the whole benchmark test as it was previously.

The signing is the most consuming operation, taking 42.2% of the whole benchmarking test.

However, we can see that inside of the signing operation, the main costs are divided in four operations:

- `10.9%` Goes to retrieving the private key from the file system.
- `25.2%` Goes to converting the annotation to JSON, as the signature is done on the produced JSON. The produced JSON is then lost and not used to be written to the annotations stream.
- `30.1%` Goes to initializing the signer to the freshly retrieved key.
- `32.2%` Goes to actually signing the annotation data.

Figure 4: Focus on annotation creation

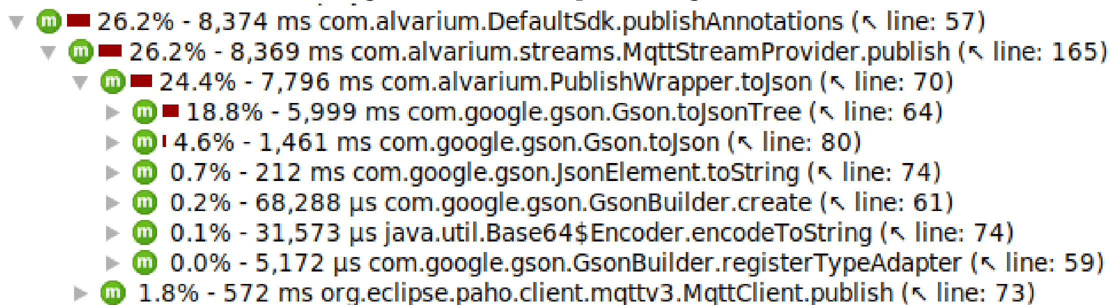


Even creating annotations seems to have a significant cpu cost.

This cost is due to the instantiation of a `ULID`, which initializes a `SecureRandom`.

Initializing a `SecureRandom` is quite CPU consuming as we can see.

Figure 5: Focus on publishing annotations



Publishing annotations is really expensive in Alvarium : 24.4% of the whole CPU is used just to serialize the data.

The entire overhead seems to be due to the Gson library.

4 What can be optimized

4.1 Non-breaking changes

4.1.1 Annotation serialization

As seen by profiling results, the JSON serialization seems to be quite expensive, and the annotations gets converted to JSON to be signed, so there is two JSON serialization operations done per annotation.

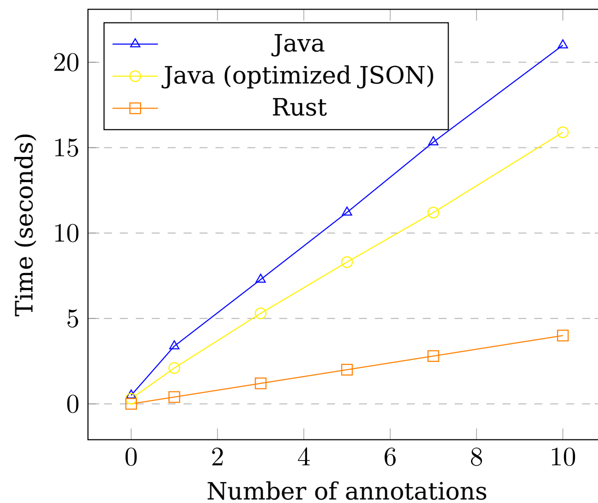
Also, the annotations JSON that is sent to the Alvarium stream, is first converted to base64 string and then be put inside another json object. Why not inline the two JSON Objects ?

The second reason of the serialization being so expensive, is that the library used (GSON) isn't efficient at all when it encounters unknown structures, because it has to use the reflection library, which is very inefficient. Defining GSON's `TypeAdapters` for all the serializable Alvarium objects did reduce the JSON serializing process to being nearly unsignificant.

Here are the results of the original Java SDK with optimized JSON serialization.

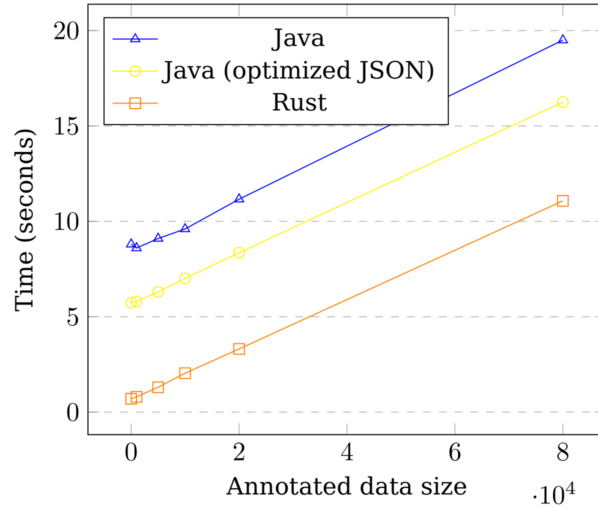
	Java		
annotations count	min	avg	max
0	0.177 s	0.301 s	0.755 s
1	2.098 s	2.196 s	2.43 s
3	5.256 s	5.311 s	5.43 s
5	8.229 s	8.309 s	8.369 s
7	11.165 s	11.244 s	11.388 s
10	15.53 s	15.919 s	17.781 s

Benchmark results of Java (event set) compared to Rust (not modified), lower is better



	Java		
data size (bytes)	min	avg	max
0	5.689 s	5.735 s	5.755 s
1000	5.752 s	5.857 s	5.922 s
5000	6.312 s	6.379 s	6.425 s
10000	6.935 s	7.005 s	7.135 s
20000	8.257 s	8.356 s	8.479 s
80000	16.082 s	16.249 s	16.645

Benchmark results of Java (event) compared to Rust (not modified), lower is better



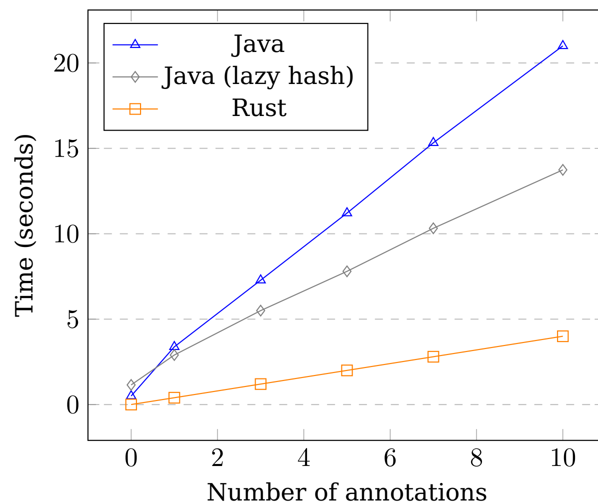
4.1.2 Hash data once per annotation

Hashing functions are pure, meaning that for the same data input, you'll always get the same output. The profiling results shows that the function `AbstractAnnotator.deriveHash` have a significant cost, hashing can be quite expensive, this is why we want to avoid redundant hashing processes, and hash the data only once per SDK call.

Currently, the data is hashed inside the annotators (which is useless), you can see below the performances improvements Alvarium can benefit from hashing the data only once for all annotations.

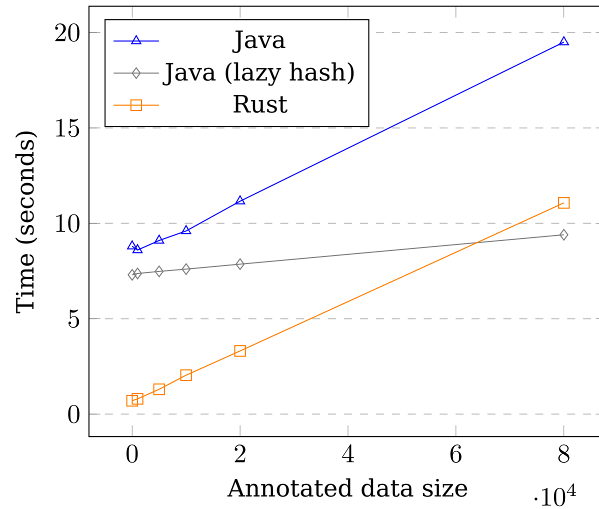
	Java		
annotations count	min	avg	max
0	0.9 s	1.15 s	1.8 s
1	2.7 s	2.9 s	3.9 s
3	5.4 s	5.5 s	5.6 s
5	7.77 s	7.8 s	7.9 s
7	10.2 s	10.32 s	10.76 s
10	13.7 s	13.74 s	13.794 s

Benchmark results of Java (event set) compared to Rust (not modified), lower is better



data size (bytes)	Java min	avg	max
0	7.3 s	7.3 s	7.4 s
1000	7.32 s	7.37 s	7.44 s
5000	7.45 s	7.48 s	7.52 s
10000	7.59 s	7.6 s	7.6 s
20000	7.8 s	7.86 s	7.94 s
80000	9.3 s	9.4 s	9.42

Benchmark results of Java (event) compared to Rust (not modified), lower is better



4.1.3 Cache ULID Generator

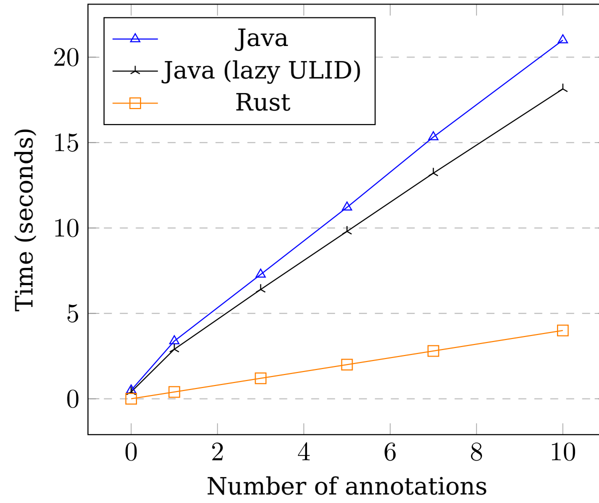
Profiling results have shown that the instantiation of a ULID class takes 5.7% of total CPU Time.

This is due to the fact that the ULID class does initializes a `SecureRandom` class from `java.security`.

We only instantiate this class to generate one ULID, let's just instantiate one generator, put it in a static field of `com.alvarium.contracts.Annotatation` and then call the method 'nextULID' as we currently do :

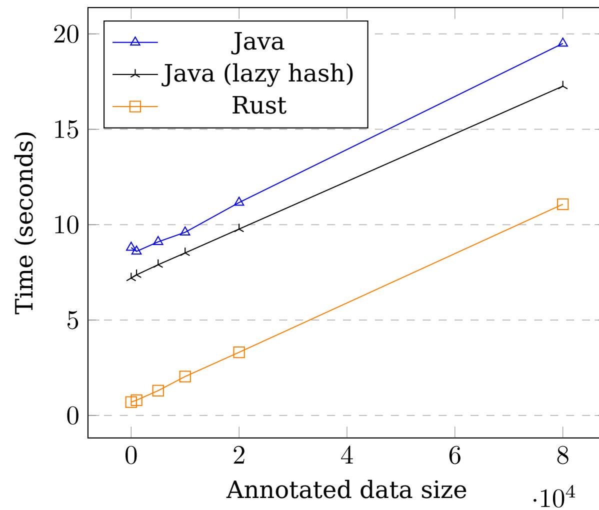
annotations count	Java min	avg	max
0	0.2 s	0.4 s	1.16 s
1	2.7 s	2.9 s	4.1 s
3	6.4 s	6.4 s	6.5 s
5	9.7 s	9.8 s	10.11 s
7	13.11 s	13.21 s	13.67 s
10	18.07 s	18.14 s	18.33 s

Benchmark results of Java (event set) compared to Rust (not modified), lower is better



	Java		
data size (bytes)	min	avg	max
0	7.2 s	7.2 s	7.2 s
1000	7.3 s	7.37 s	7.42 s
5000	7.86 s	7.9 s	7.98 s
10000	8.48 s	8.52 s	8.56 s
20000	9.7 s	9.77 s	9.85 s
80000	17.18 s	17.26 s	17.35

Benchmark results of Java (event) compared to Rust (not modified), lower is better



4.2 Annotation generation

Alvarium currently produces the following JSON output:

Java SDK JSON Action, pretty-printed


```json
{
    "action": "create",
    "messageType": "com.alvarium.contracts.AnnotationList",
    "content": {
      "items": [
        {
          "id": "01HXY45Q6DPT9XZQ0XCV7YSVVZ",
          "key": "E2E8736387CB5F774BD8A2849C7EC131338671CFE057B7A7DFB9C2AE2F6CCF64",
          "hash": "sha256",
          "host": "2aad2bf8b134",
          "tag": "",
          "layer": "app",
          "kind": "src",
          "signature": "0B554EFC1DBAAC5BDB391DD4C0F1892BD912D0C89C7BB3049A6AF76C988233F84AA70776FF9A10F83B3FA1EE0E5403CFAA7D6A398238394293F0FCB8BA656F09",
          "isSatisfied": true,
          "timestamp": "2024-05-15T12:38:27.789668031Z"
        },
        {
          "id": "01HXY45Q6EQ3Y25N88B8QWV629",
          "key": "E2E8736387CB5F774BD8A2849C7EC131338671CFE057B7A7DFB9C2AE2F6CCF64",
          "hash": "sha256",
          "host": "2aad2bf8b134",
          "tag": "",
          "layer": "app",
          "kind": "src",
          "signature": "C91FED38FC453114D91C42A1D5268FB5B4829A403A65771DA1EB770D5036FA4F5997A100A79626F0CA116BD6284BB3EA6732B9980323369F7A46DBA426603802",
          "isSatisfied": true,
          "timestamp": "2024-05-15T12:38:27.790431705Z"
        }
      ]
    }
}
```


This JSON is the result of a `SDK call` with 2 annotations.

We can see that there is a lot of data redundancy, but some work on previous papers has already be done to reduce the size of annotations, so let's not try to discuss how we can reduce the size of the produced JSONs.

What we can consider though, is that the annotations are always set into an object that represents the kind of SDK call that produced the annotations.

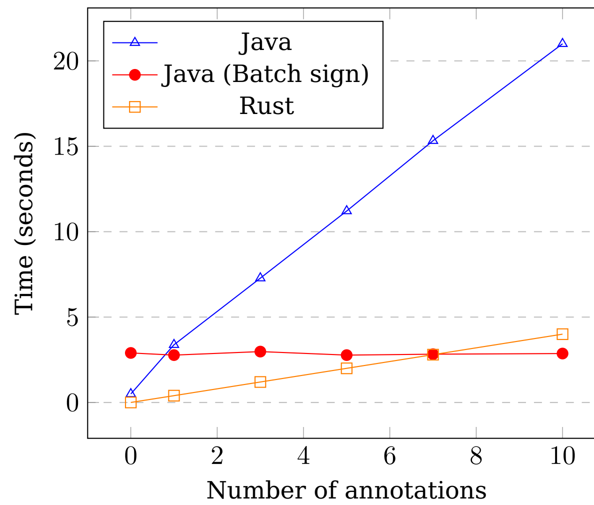
The profiling of annotations generation shows us that signing an annotation is very expensive, Alvarium's main drawback is that annotations are signed individually. As the annotations are all set inside an object that represent one SDK call, we should just sign the end object.

Also, as found in the profiling results, the data gets hashed for each annotations, so the hash should just be made once, ahead of time, and then given to the annotators.

Here is the benchmark results with the annotations being all signed at once, and the data being hashed only once per SDK call, compared to the previous benchmark results

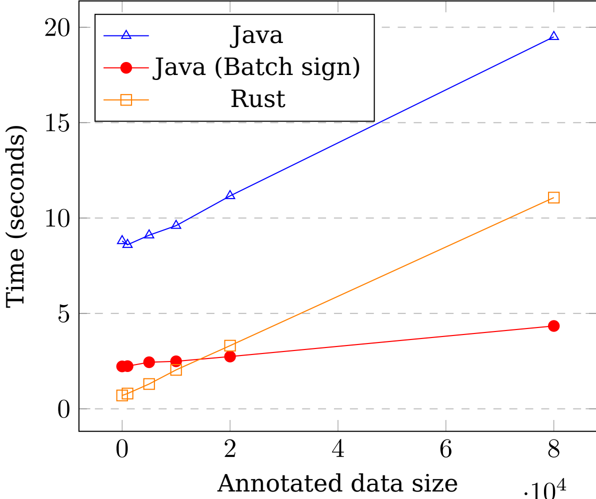
	Java		
annotations count	min	avg	max
0	2.639 s	2.901 s	4.551 s
1	2.558 s	2.774 s	3.004 s
3	2.675 s	2.98 s	3.493 s
5	2.646 s	2.774 s	2.942 s
7	2.674 s	2.828 s	3.022 s
10	2.76 s	2.866 s	3.026 s

Benchmark results of Java (optimized signature/hash) compared to original Java/Rust, lower is better



	Java		
data size (bytes)	min	avg	max
0	1.925 s	2.223 s	2.585 s
1000	2.06 s	2.236 s	2.728 s
5000	2.174 s	2.441 s	3.336 s
10000	2.282 s	2.49 s	2.698 s
20000	2.626 s	2.739 s	2.917 s
80000	4.277 s	4.342 s	4.535 s

Benchmark results of Java (optimized signature) compared to original Java/Rust, lower is better



We can see that with this simple trick, the amount of annotation does no longer affect the alvarium's overhead, and the data size seems to be less important.

4.3 Signing process

We just gain some performance boost by not signing each annotations individually, but we can also optimize the signing process.

Remember from the profiling results, we saw that only 32% of the signing process was to actually sign the data. The remaining CPU time is used to retrieve the private key from the file system then initialize the signer, and by converting the annotation to JSON (as the signature is based on the annotation's JSON).

What we could do is to cache the signer so the private key does not have to be retrieved and initialized, and we could get rid of the JSON serialization process by only signing the annotation's content, this will also reduce the length of the data to be signed.

Currently, the annotations are signed by using their JSON representation.

First, the current implementation has a serious drawback as the signature is directly stored inside the signed JSON, so to verify the signature JSON, you have to recreate a JSON with the signature set to `null` :

```
75  protected String signAnnotation(KeyInfo keyInfo, Annotation annotation)
76        throws AnnotatorException {
77    SignProviderFactory signFactory = new SignProviderFactory();
78
79    try {
80      SignProvider provider = signFactory.getProvider(keyInfo.getType());
81      Path keyPath = Paths.get(keyInfo.getPath());
82      String key = Files.readString(keyPath, StandardCharsets.US_ASCII);
83
84      // Will produce the annotations json as shown in the section "4.1 Annotation
          generation", excepted that "signature" is set to ‘null‘
85      String json = annotation.toJson().getBytes();
86
87      return provider.sign(Encoder.hexToBytes(key), json);
88    } catch (SignException e) {
89      throw new AnnotatorException("cannot sign annotation.", e);
90    } catch (IOException e) {
91      throw new AnnotatorException("cannot read key.", e);
92    }
93  }
```

Then, after signing the JSON with no signature set, the signature is put into the same annotation :

```
1    String annotationSignature = super.signAnnotation(
2        this.signature.getPrivateKey(),
3        annotation
4    );
5    // signature set here
6    annotation.setSignature(annotationSignature);
7    return annotation;
```

Which means that the annotations are in a way already tempered ! And to verify the signature, you have to know that you have to verify it on a JSON with the `"signature"` value set to null.

What can be done is to move the annotation's signature outside of its JSON representation, like so :

```
{
    "signature": "0B554EFC1DBAAC5BDB391DD4C0F1892BD912D0C89C7BB3049A6AF76C988233F84AA70776FF9A10F83B3FA1EE0E5403CFAA7D6A398238394293F0FCB8BA656F09",
    "content": {
        "id": "01HXY45Q6DPT9XZQ0XCV7YSVVZ",
        "key": "E2E8736387CB5F774BD8A2849C7EC131338671CFE057B7A7DFB9C2AE2F6CCF64",
        "hash": "sha256",
        "host": "2aad2bf8b134",
        "tag": "",
        "layer": "app",
        "kind": "src",
        "isSatisfied": true,
        "timestamp": "2024-05-15T12:38:27.789668031Z"
    }
}
```

This way, the JSON content can be reused, and it does not have to be modified to be verified.

4.3.1 Signing proposal : Identity Strings

There still an important technical drawback to signing a JSON object : At some point you have to retrieve the JSON in order to verify the signature.

If you already store the annotations in JSON with no transformation then signing the stored JSON is straightforward. But if the annotations need to be stored in binary, to be compressed, or in SQL table, roughly speaking, in any other format than the one produced by Alvarium, you'll have to serialize it back to the same JSON produced by Alvarium.

There are two performances issues of this: First, this will cost you the serializing to JSON, and second, JSON is not a very compact format, so if you have thousands of annotations to handle, you'll lose performances over also signing and verifying the JSON's embellishments (field names and other JSON syntax characters).

To gain performances over those two issues, we could simply create a string containing all the annotation fields' values appended in a defined order :

Compacted JSON (258 characters)

```
{"content":{"id":"01HXY45Q6DPT9XZQ0XCV7YSVVZ","key":"E2E8736387CB5F774BD8A2849C7EC131338671CFE057B7A7DFB9C2AE2F6CCF64","hash":"sha256","host":"2aad2bf8b134","tag":"","layer":"app","kind":"src","isSatisfied":true,"timestamp":"2024-05-15T12:38:27.789668031Z"}}
```

A string containing only the annotation's values in a specified order (148 characters, 74% shorter than using compact JSON)

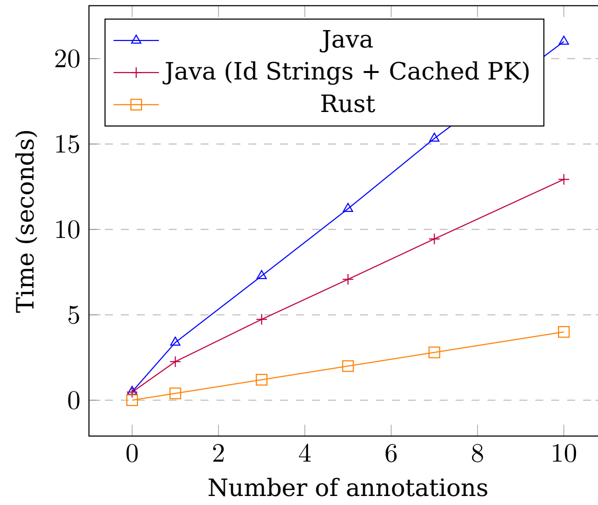
```
01HXY45Q6DPT9XZQ0XCV7YSVVZE2E8736387CB5F774BD8A2849C7EC131338671CFE057B7A7DFB9C2AE2F6CCF64sha2562aad2bf8b134appsrctrue2024-05-15T12:38:27.789668031Z

// the order can be defined like so
<id><key><hash><host><tag><layer><kind><isSatisfied><timestamp>
```

4.3.2 Results

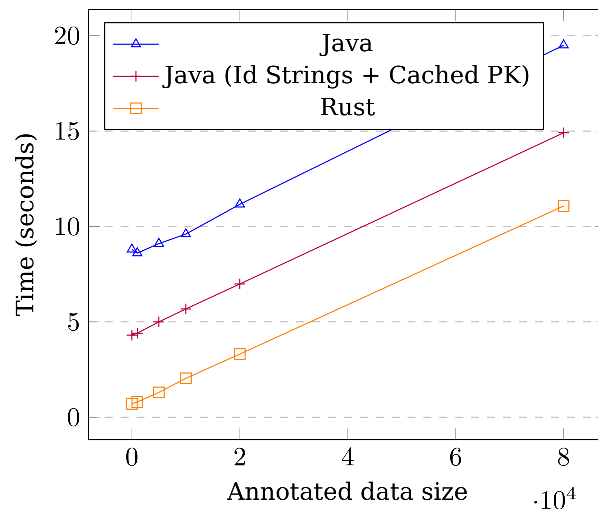
annotations count	Java min	Java avg	Java max
0	0.303 s	0.475 s	1.143 s
1	2.098 s	2.26 s	3.117 s
3	4.688 s	4.735 s	4.783 s
5	6.921 s	7.078 s	7.296 s
7	9.375 s	9.443 s	9.615 s
10	12.829 s	12.93 s	13.112 s

Benchmark results of Java (optimized signature) compared to original Java/Rust, lower is better



data size (bytes)	Java min	Java avg	Java max
0	4.247 s	4.304 s	4.387 s
1000	4.343 s	4.405 s	4.473 s
5000	4.933 s	4.997 s	5.127 s
10000	5.569 s	5.675 s	5.862 s
20000	6.92 s	6.985 s	7.027 s
80000	14.818 s	14.909 s	15.081 s

Benchmark results of Java (optimized signature) compared to original Java/Rust, lower is better

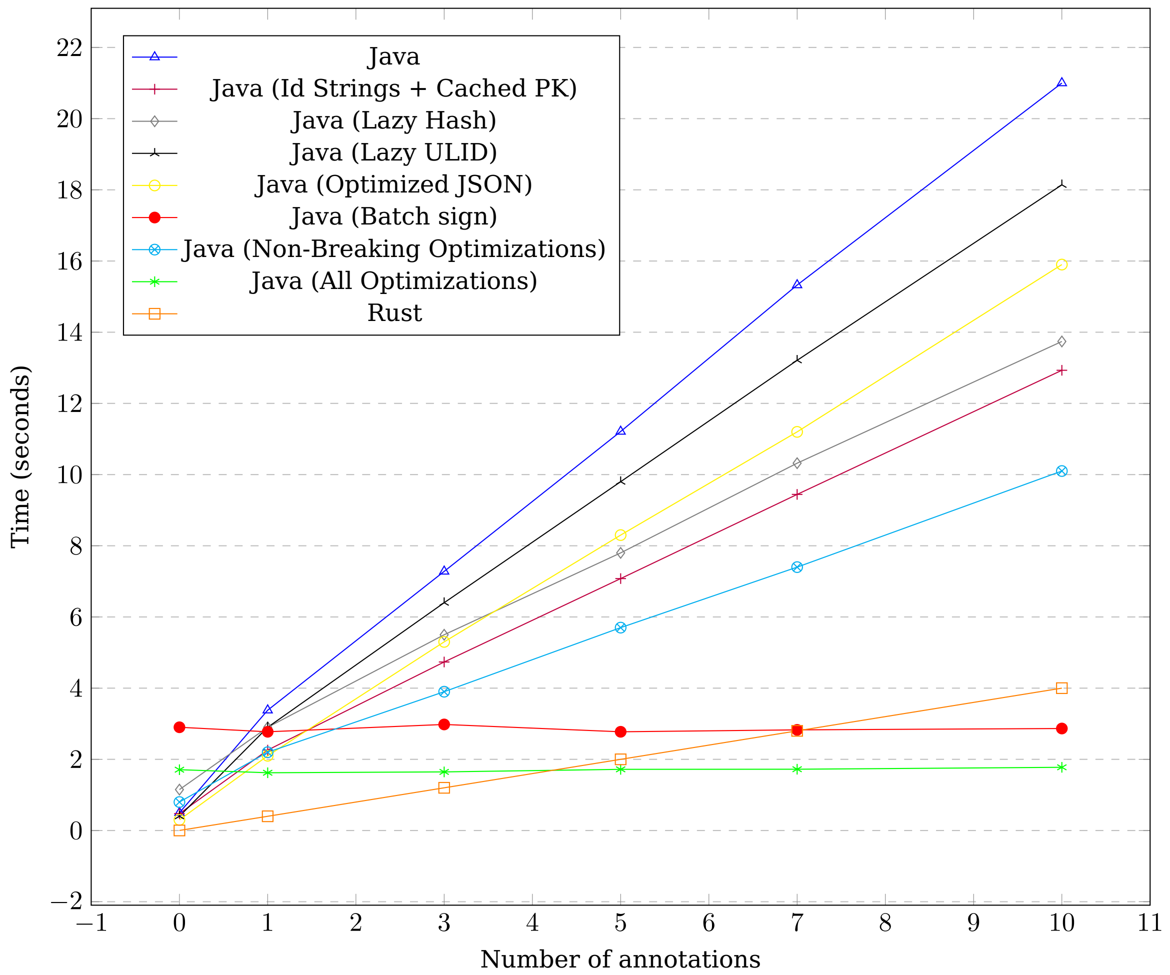


4.4 Overall comparison

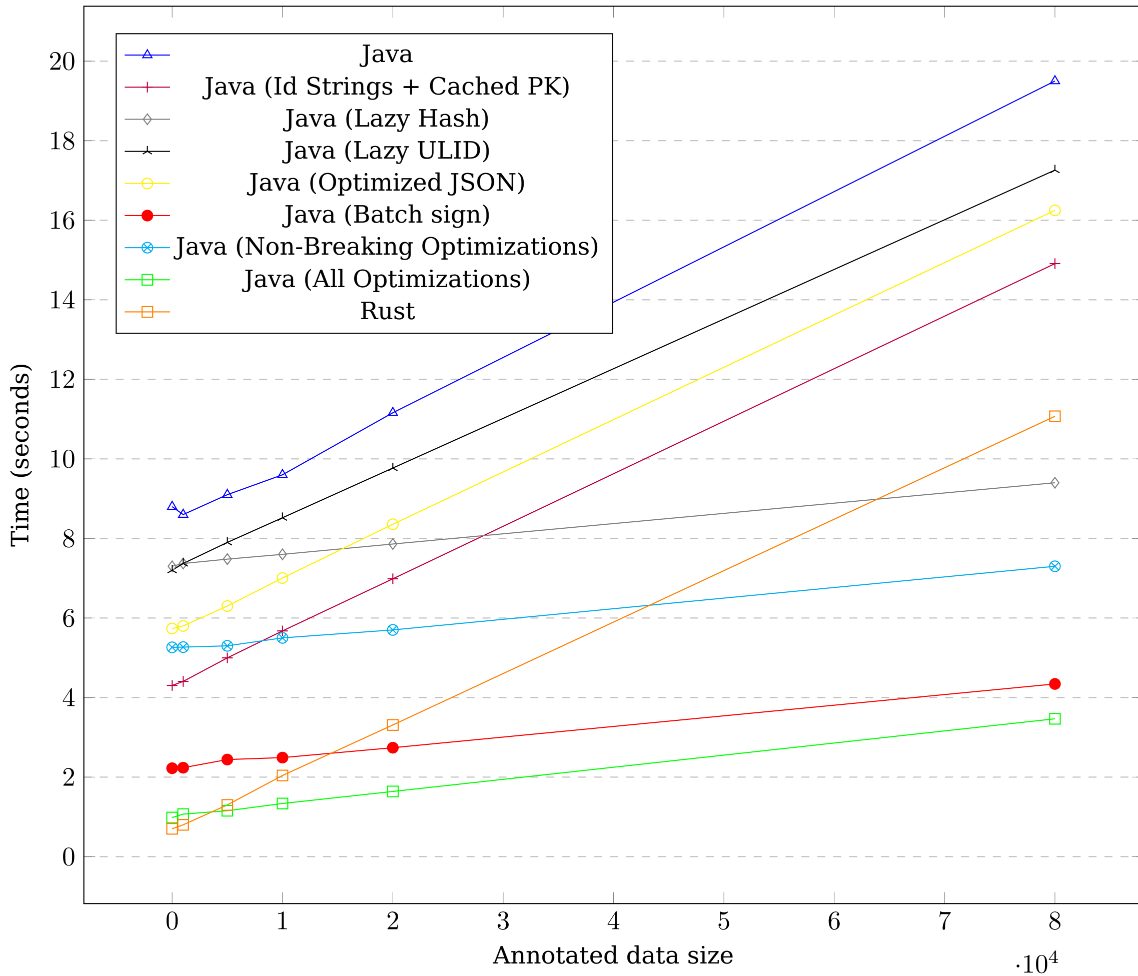
Here you can see all individual optimisations compared to the original Java/Rust versions, and to two other versions containing all non breaking changes (cyan), and all changes (green).

	data size (bytes)	min	avg	max
	0	1.502 s	1.709 s	2.701 s
	1	1.527 s	1.623 s	1.75 s
Java	3	1.564 s	1.647 s	1.789 s
	5	1.629 s	1.719 s	2.088 s
	7	1.631 s	1.722 s	2.079 s
	10	1.7 s	1.776 s	1.884 s

Benchmark results of Java (optimized signature) compared to original Java/Rust, lower is better



	data size (bytes)	min	avg	max
	0	0.916 s	0.983 s	1.087 s
	1000	0.96 s	1.069 s	1.481 s
Java	5000	1.098 s	1.154 s	1.199 s
	10000	1.264 s	1.336 s	1.408 s
	20000	1.595 s	1.639 s	1.686 s
	80000	3.43 s	3.466 s	3.502 s



You can see that starting at a certain data size and annotation count the version with all optimisations applied competes with the current Rust version and even gets quickly faster.

5 Conclusion

Current version of Alvarium has some serious performances issues.

There are some issues specific to the Java Versions such as the JSON serialization, the signing process and the annotations unique identifier generation.

Fixing those issues greatly decreased the Alvarium's overhead, but the main drawback, that is on all the SDK languages because it is a conceptual issue (regarding performances), is that all annotations gets signed individually. This is useless as the annotations are all sent in a single collection per SDK call event (e.g. create, mutate etc), and signing the event set once rather than each individual annotation entry brings the complexity of Alvarium from being linear to constant (it is no more affected by the number of annotators).

A good performance improvement would be to sign the annotation using Identity Strings as discussed in Signing proposal : Identity Strings. This is not a breaking change as such, and offer much more flexibility as well as performances when creating / signing the annotation, and when verifying them.