UNIVERSIDADE D
COIMBRA

Data Representation and Serialization Formats

Master's in Software Engineering
Integration Systems
2022/2023

Advisor's Names:
Filipe João Boavida Mendonça Machado de Araújo

Author's Names:
Sofia Santos Neves nº2019220082
Tatiana Silva Almeida nº2019219581

September 30, 2022

Contents

1. Introduction

The purpose of this assignment is to learn and compare the different data representation and serialization formats: XML, XML compressed with Gzip and Google Protocol Buffers.

At the end of this report, the reader is supposed to have a better understanding concerning serialization size and encoding and decoding speed of the different data representation technologies.

Given that we had the choice of doing either XML or JSON, we opted to do XML.

2. Conditions of the experiments

We decided to do the project using the Java programming language.

We also tried to use as much similar code as possible whilst using the different serialization formats for greater fairness in the discussion of the results. Moreover, all the trials we ran were done a total of thirty times (30x) with the intention of evaluating the mean and the standard deviation of the results so as to account for potential outliers.

2.1. Computer characteristics

The computer used to do the various experiments was an Asus Laptop with the following specifications:

- **CPU** Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99GHz
- **RAM** 8.00GB

2.2. Technologies and libraries used with their versions

The technologies/libraries used, as well as their respective versions, were:

- openJDK, version: 17.0.2
- protoc 21.6 win64, version: libprotoc 3.8.0
- [dependency] jakarta.xml.bind, version: 3.0.0
- [dependency] org.eclipse.persistence, version: 3.0.0
- [dependency] com.google.protobuf, version: 3.0.0
- [optional] IntelliJ IDEA 2022.1

2.3. Number of data structures used

For the experiments, we decided to use three different numbers of data structures:

- 110 structures (100 Professors and 10 Students);
- 1 100 structures (1 000 Professors and 100 Students);
- 11 000 structures (10 000 Professors and 1 000 Students).

2.4. Types of experiments

Whilst running the experiments, we observed that there is, in the beginning of it, a couple of outliers as demonstrated in the following graphic:

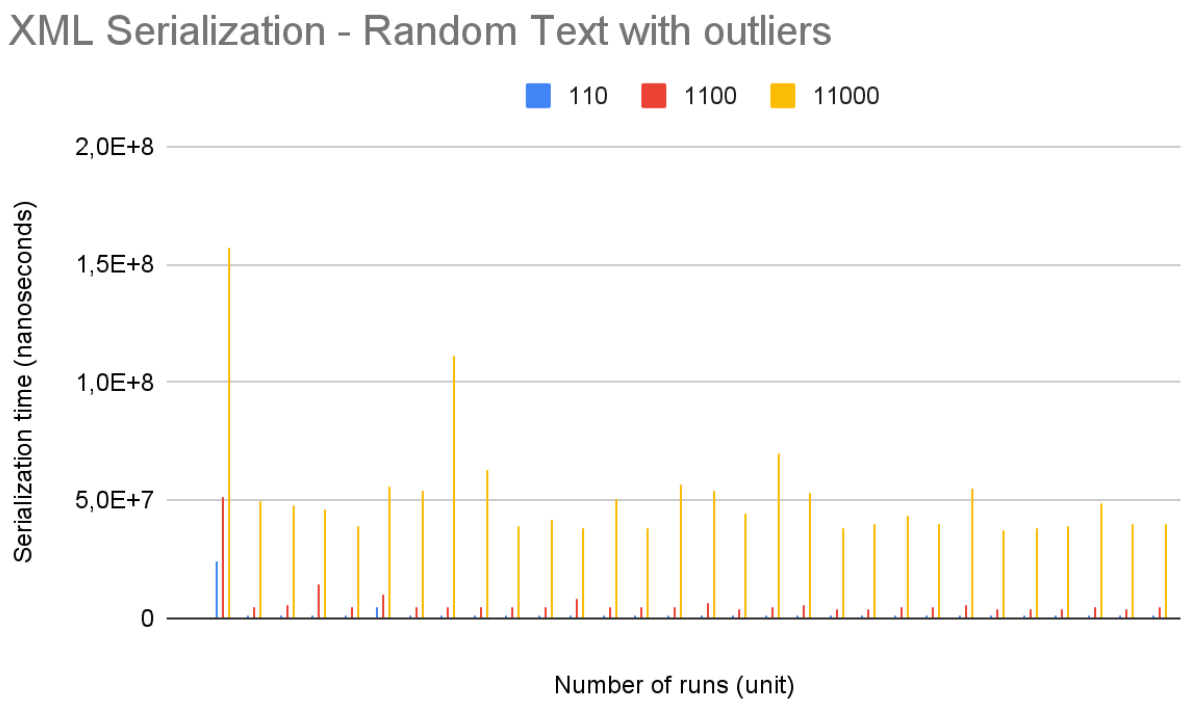

Figure 1 - Serialization time for 110, 1100, 11000 structures

From here, we can conclude that at least set 1 and 8 of the experiment are outliers. So, in order to eliminate this problem, we did not consider the first ten sets of the experiment, leaving us with only twenty from which we extracted the conclusions of this paper.

In addition to this, in order to understand the major differences between all of the formats, we decided to only evaluate two different input types - Random text and Same text:

2.4.1. Random Text

This input intends to represent the most natural form of data: random names, telephones, genders, birthdates, registration dates as well as addresses since in our day to day lives there is not, for most of it, many repetitions of words.

2.4.2. Same Text

As opposed to ‘random text’, ‘same text‘ is the exact opposite as it represents the unnatural side of things, repetitive words. So, in this input, we demonstrate how the different formats react to this by always using the same letter, with the same size, in all the names and addresses of the various classes.

3. How to Run

Assumptions: We are assuming that the reader has installed a Java IDE as recommended in point 2.2 above.

1. Download the .zip file
2. Extract the contents of the .zip file
3. Install openJDK and protoc 21.6 with the same versions on your device
4. Open the project on your Java IDE of choice and check if the dependencies listed above in point 2.2 are present in the pom.xml file
5. Build the App Class in the folder 'src/main/java'
6. Click on "Run 'App'"

4. Experiment results

4.1. XML

XML stands for eXtensible Markup Language and it was designed to store and transport data as well as to be self-descriptive. XML itself does not do anything, it is only information wrapped in tags.

In short, XML is more of a tool due to the fact that it keeps your design work organized into practical compartments, it is easy to understand due to its easy nature and saves the programmer time since data is already separated.

4.1.1. Data structures

There are three main structures used for the XML, namely the class Student, Professor and Holder (that corresponds to a School).

For the Student class the following attributes are being used:

- int id;
- String name;
- long telephone;
- String gender;
- String BirthDate;
- String RegistrationDate;
- String Address.

The Professor's class attributes are:

- int id;
- String name;
- long telephone;
- String BirthDate;
- String Address;
- ArrayList<Student> Students.

The Holder class attributes are:

- ArrayList<Professor> professors.

4.1.2. Code where time is measured

Marshall	Unmarshall

startTime = System.nanoTime();
 	
jaxbMarshaller.marshal(holdProfessors, file);
 	
totalTime = System.nanoTime() - startTime;
	startTime = System.nanoTime();
	
	Holder profs = (Holder) jaxbUnmarshaller.unmarshal(file);
	
	totalTime = System.nanoTime() - startTime;

4.1.3. Serialization size

Random Text	
Total number of structures (unit)	**Size (bytes)**
110	32 944
1 100	329 788
11 000	3 309 068

Same Text	
Total number of structures (unit)	**Size (bytes)**
110	32 940
1 100	329 660
11 000	3 309 067

Figure 2- Size file comparison between 'RandomText' and 'SameText' in XML.

From this data we can extract the following conclusion: there is no difference, or at least not a significant one, regarding the XML file size when using different experiments since both of them have attributes with the same size.

4.1.4. Serialization and deserialization speed

Serialization speed			
Random Text		**Same Text**	
Total number of structures (unit)	**Time (nanoseconds)**	**Total number of structures (unit)**	**Time (nanoseconds)**
110	1 337 725	110	1 272 930
1 100	6 383 970	1 100	6 244 390
11 000	56 406 875	11 000	72 410 675

Deserialization speed			
Random Text		**Same Text**	
Total number of structures (unit)	**Time (nanoseconds)**	**Total number of structures (unit)**	**Time (nanoseconds)**
110	8 221 665	110	8 309 440
1 100	16 954 045	1 100	19 416 600
11 000	89 466 165	11 000	92 856 765

Analyzing the serialization/deserialization time between the different types of experiments, the results of the three sets were similar. So, taking that into account, we are only going to look at the first set, from which we extracted the conclusions of this paper

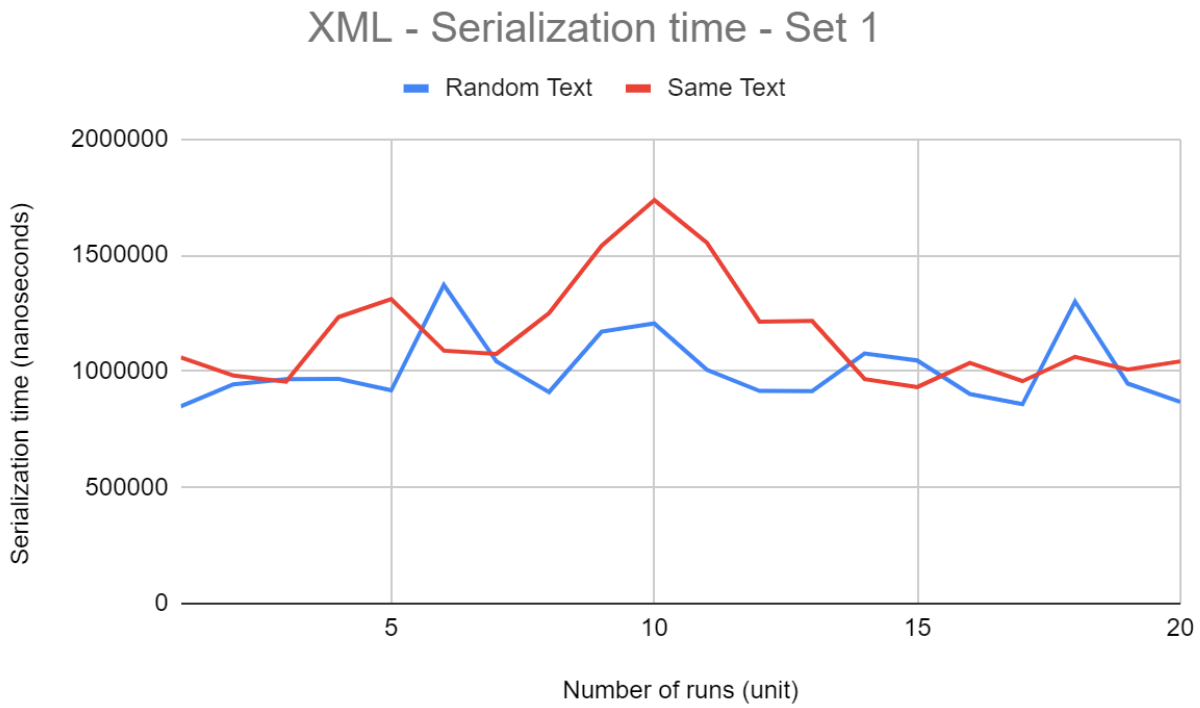

Figure 3 - Serialization time comparison between 'RandomText' and 'SameText' in XML – set 1.

When comparing the differences, one can infer that, maybe due to other processes in the computer, the serialization time for the 'Random Text' is, in most cases, smaller than 'Same Text'. In theory, this shouldn't be possible because the information that was serialized was the exact same size, that is, it contained the same number of structures and the names/addresses had exactly the same length. So, it should take the same time to serialize or always be a very close number.

	Standard Deviation - Serialization	
Total number of structures (unit)	**Random Text**	**Same Text**
110	336000	313813
1 100	1200777	1396286
11 000	7213233	56001391

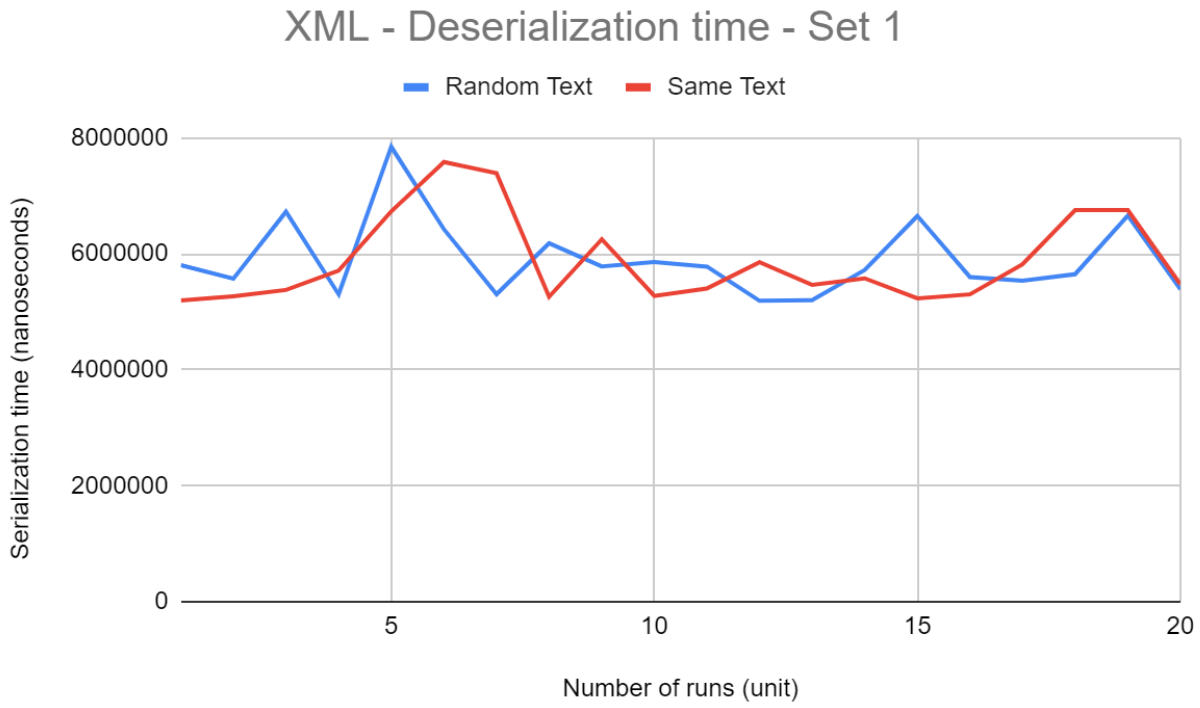

Figure 4 - Deserialization time comparison between 'RandomText' and 'SameText' in XML - set 1.

As for the deserialization, there is no major difference between the different experiments made. Which, considering the previous explanation, is an expected result.

	Standard Deviation - Deserialization	
Total number of structures (unit)	**Random Text**	**Same Text**
110	1958458	1782759
1 100	3398997	9401717
11 000	8280563	24387392

4.2. XML compressed with Gzip

If you want to understand more about XML please refer back to point 4.1. where the concept is explained in detail. In this section, we are going to focus on what Gzip is.

Gzip is a lossless compression algorithm based on the deflate algorithm. Deflate is a combination of LZ77 and Huffman encoding. So, the main purpose is to decrease the file size to a point where reversion is possible if needed.

4.2.1. Data structures

Since the data structures used in XML compressed with Gzip are the same as the XML, refer back to point 4.1.1. if you want to see them.

4.2.2. Code where time is measured

Marshall	Unmarshall

startTime = System.nanoTime();
 	
jaxbMarshaller.marshal(holdProfessors, file);
totalTime = System.nanoTime() - startTime;
…
startTime = System.nanoTime();
while ((len = fis.read(buffer)) != -1) {
  gzipOS.write(buffer, 0, len);
}
 	
totalTime += (System.nanoTime() - startTime);
	startTime = System.nanoTime();
	
	while ((len = gis.read(buffer)) != -1) {
	  fos.write(buffer, 0, len);
	}
	Holder profs = (Holder)
	jaxbUnmarshaller.unmarshal(file);
	
	totalTime = System.nanoTime() - startTime;

4.2.3. Serialization size

Random Text	
Total number of structures (unit)	**Size (bytes)**
110	5 139
1 100	45 382
11 000	446 686

Same Text	
Total number of structures (unit)	**Size (bytes)**
110	3 120
1 100	26 911
11 000	264 584

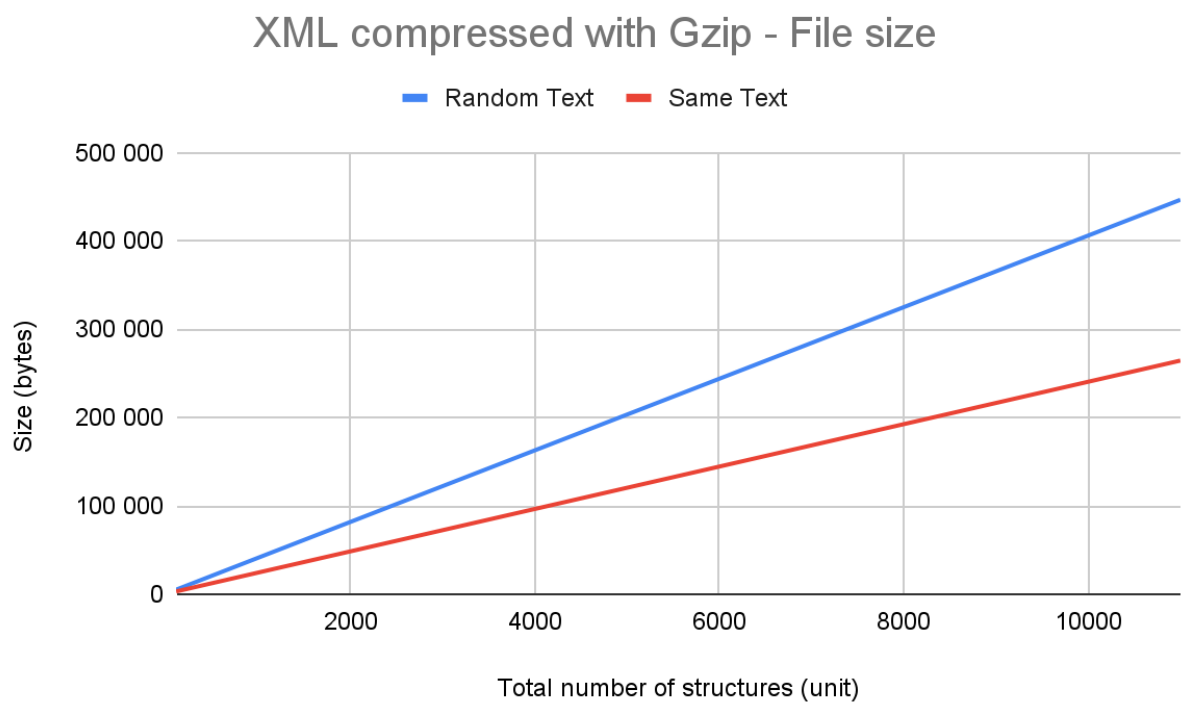

Figure 5 - File size comparison between 'RandomText' and 'SameText' in XML compressed with Gzip.

It is clear that since the XML is compressed with Gzip, a lossless compression algorithm, the results correspond to the expected value meaning that it is normal for the deflate codec to compress more the 'Same text' input due to the fact that it repeats the same letter/word a lot more than 'Random Text'.

4.2.4. Serialization and deserialization speed

Serialization speed			
Random Text		**Same Text**	
Total number of structures (unit)	**Time (nanoseconds)**	**Total number of structures (unit)**	**Time (nanoseconds)**
110	2 077 705	110	3 057 820
1 100	14 536 310	1 100	13 915 135
11 000	151 933 245	11 000	129 692 750

Deserialization speed			
Random Text		**Same Text**	
Total number of structures (unit)	**Time (nanoseconds)**	**Total number of structures (unit)**	**Time (nanoseconds)**
110	9 358 620	110	10 527 095
1 100	20 885 780	1 100	21 301 215
11 000	127 737 200	11 000	116 299 770

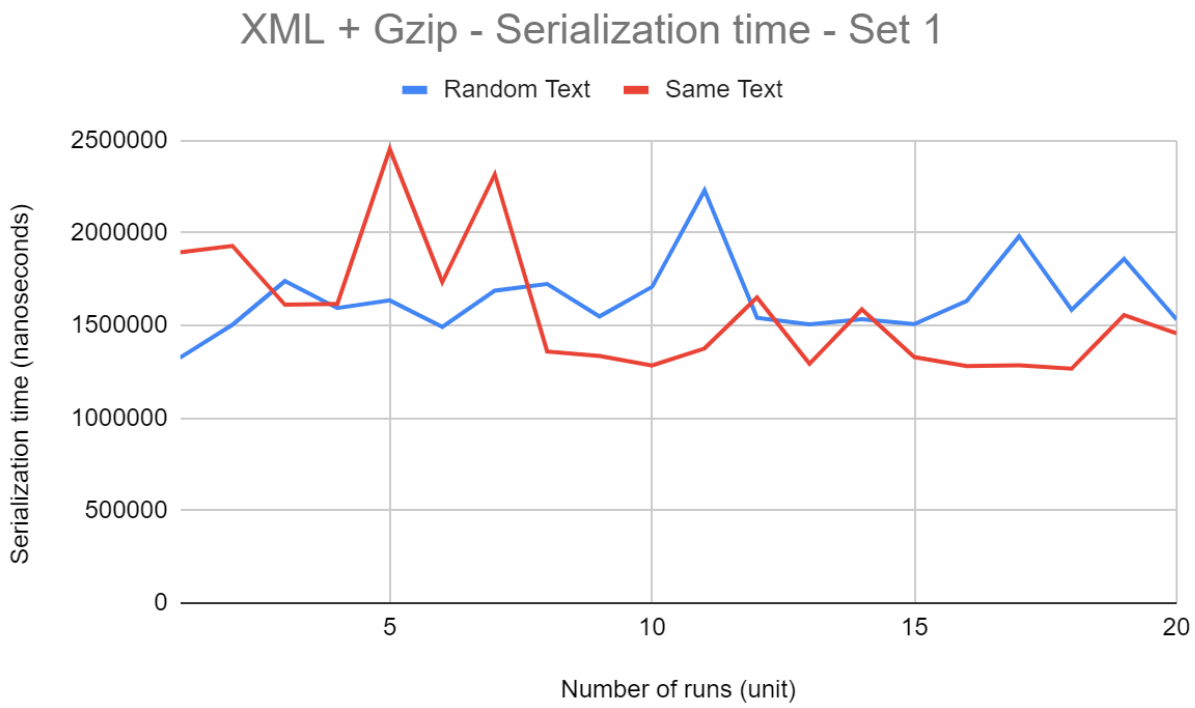

Figure 6 - Serialization time comparison between 'RandomText' and 'SameText' in XML compressed with Gzip - set 1.

When we look for the serialization time for XML compressed with Gzip, we have to take in consideration the previous explanation about the serialization time for XML. Adding Gzip, should, in theory, make the serialization in 'SameText' slightly faster because of the codec used to compress the information. The codec uses techniques that are faster when we are dealing with the same type of characters.

If we take a close look at the figure 6, we can see that, in most of the cases, the serialization time for 'SameText' is inferior to the time for serializing 'RandomText', as expected.

	Standard Deviation - Serialization	
Total number of structures (unit)	**Random Text**	**Same Text**
110	521519	5476317
1 100	2420843	4575210
11 000	33396980	22559936

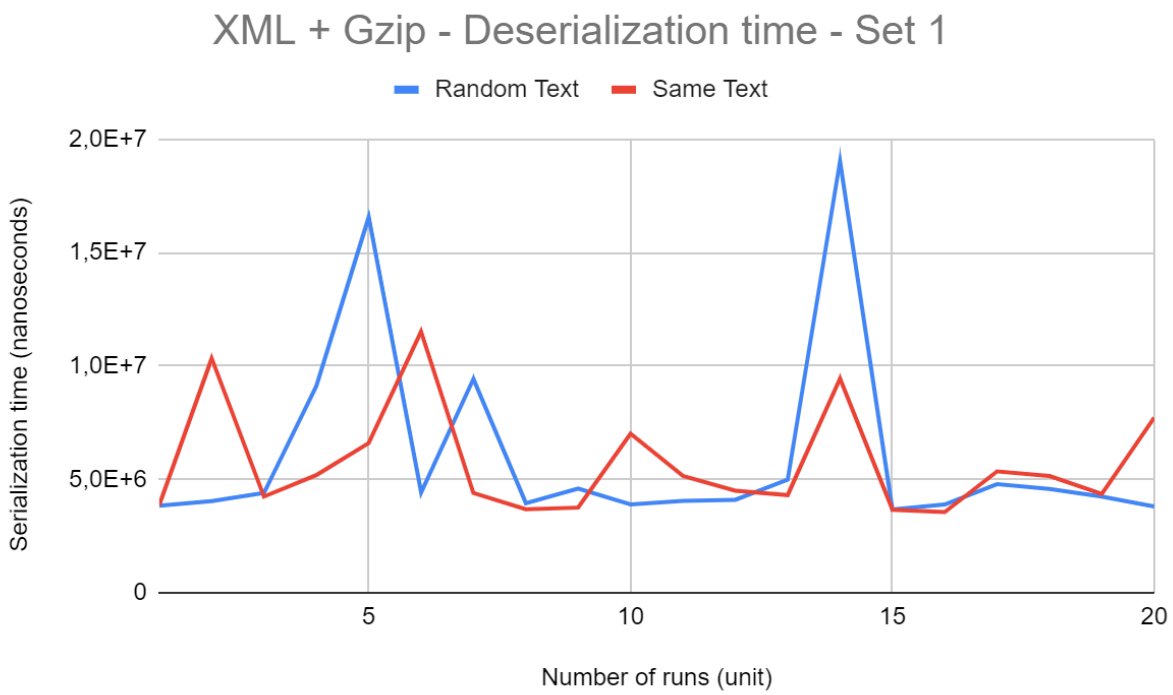

Figure 7 - Deserialization time comparison between 'RandomText' and 'SameText' in XML compressed with Gzip - set 1.

The deserialization should behave exactly the same as the serialization, where the time for 'SameText' is less than the time for 'RandomText'. As you can see in the figure 7, the deserialization time for both types are not consistent. This may have to do with the fact that the computer is running different processes during different runs of the experiment. However, in most of the cases, the theory holds up and 'SameText' has a short deserialization time.

	Standard Deviation - Deserialization	
Total number of structures (unit)	**Random Text**	**Same Text**
110	2346929	10126006
1 100	4242035	9440165
11 000	21799492	15396500

4.3. Google Protocol Buffers

According to Google, “Protocol buffers are Google’s language-neutral, platform-neutral, extensible mechanism for serializing structured data - think XML, but smaller, faster, and simpler.”[1].

In practice, what happens is that the user defines how he wants the data to be structured and then he can use special generated source code to read and write the data to and from a variety of places. It also enables users to use a wide range of languages besides Java, for example.

4.3.1. Data structures

For the definition of the protocol format, we decided to do it in this way:

Student class:

```
Message Student{
        int32 id = 1;
        string name = 2;
        int32 telephone = 3;
        string gender = 4;
        string birthdate = 5;
        string registrationDate = 6;
        string address = 7;
}
```

Professor class:

```
Message Professor{
        int32 id =1;
        string name = 2;
        int32 telephone = 3;
        string birthDate = 4;
        String address = 5;
        repeated Student students = 6;
}
```

School class (equivalent to the Holder class in point 4.1.1):

```
Message School{
        Repeated Professor professors = 1;
}
```

Once the compilation of this .proto file was done, we were able to obtain very equivalent data structures to the ones used in XML and XML compressed with Gzip (both shown in point 4.1.1.). However, there was one tangible difference that was the fact that both Student and Professor’s class attribute ‘telephone’ was of the type *int* instead of *long*.

4.3.2. Code where time is measured

For Google Protocol Buffers we decided to measure the time during two stages: the building/unbuilding phase and the serialization/deserialization phase (writing to the file).

	Marshall

Building / unbuilding phase	for(int i = 0; i < numberOfProfessors; ++i){
	…
	for(int j = 0; j < numberOfStudents; ++j){
	…
	startTime = System.nanoTime();
	Student s = student.build();
	totalTime += (System.nanoTime() - startTime);
	…
	}
	startTime = System.nanoTime();
	Professor p = professor.build();
	totalTime += (System.nanoTime() - startTime);
	…
	}
	startTime = System.nanoTime();
	School s = school.build();
	totalTime += (System.nanoTime() - startTime);

Serialization / deserialization	long start = System.nanoTime();
	schoolBuilder.writeTo(output);
	long writer = System.nanoTime() - start;

	Unmarshall

Building / unbuilding phase	long startTime = System.nanoTime();
	Holder newSchool = new Holder();
	List<generated.protobuf.classes.Professor> professors = school.getProfessorsList();
	ArrayList<Professor> newProfessors = new ArrayList<>();
	for (int i = 0; i < numberOfProfessors; ++i) {
	List<generated.protobuf.classes.Student> students = professors.get(i).getStudentsList();
	ArrayList<Student> newStudents = new ArrayList<>();
	
	for (int j = 0; j < numberOfStudents / numberOfProfessors; ++j) {
	Student s = new Student(students.get(j).getId(),...,students.get(j).getAddress());
	newStudents.add(s);
	}
	Professor professor = new Professor(professors.get(i).getId(),,..., newStudents);
	newProfessors.add(professor);
	}
	newSchool.setProfessors(newProfessors);
	return System.nanoTime() - startTime;

Serialization / deserialization	startTime = System.nanoTime();
	School school = School.parseFrom(f);
	totalTime = System.nanoTime() - startTime;

4.3.3. Serialization size

Random Text	
Total number of structures (unit)	**Size (bytes)**
110	8 165
1 100	83 043
11 000	834 812

Same Text	
Total number of structures (unit)	**Size (bytes)**
110	8 297
1 100	83 205
11 000	834 948

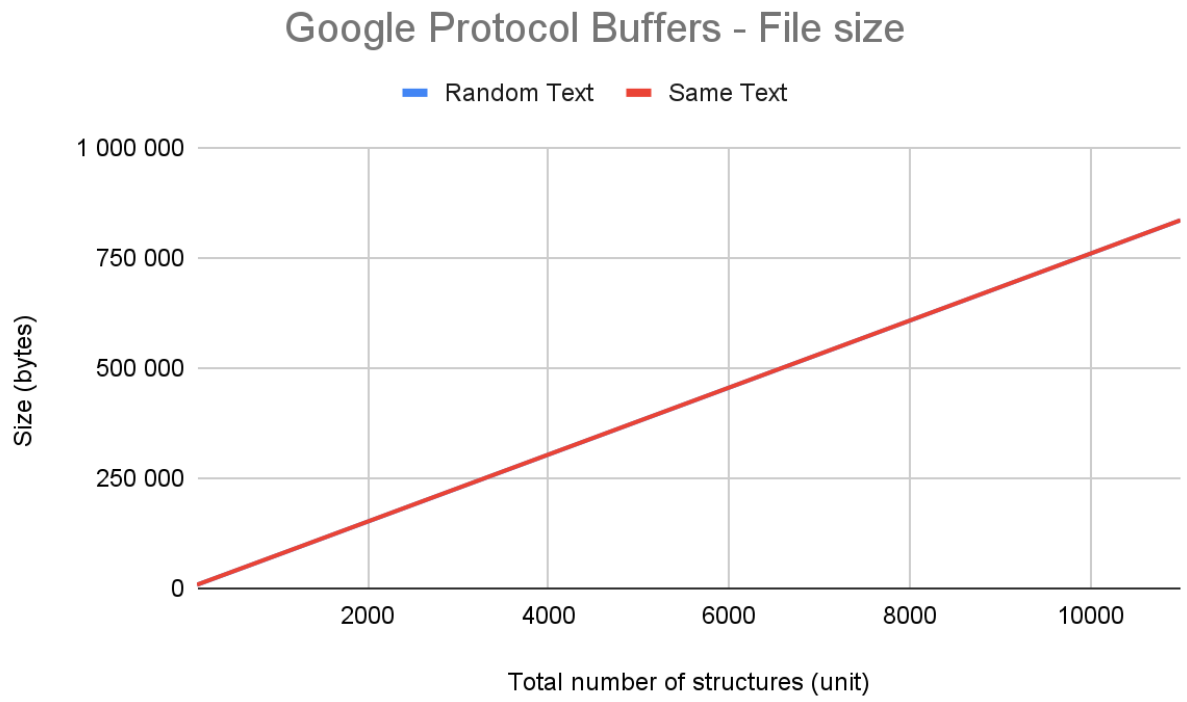

Figure 8 - File size comparison between 'RandomText' and 'SameText' in Google Protocol Buffers

Given that, there is no compression algorithm in Google Protocol Buffers, then it makes sense that the file size stayed the same, or very close, regardless of the experiment conducted.

4.3.4. Serialization and deserialization speed

For the purpose of the experiment, we decided to separate the building/unbuilding time from the serialization/deserialization, as explained in point 4.3.2..This allows us to have a better understanding of what takes up the most time during the overall process.

In the case of serialization, one can notice that the building phase is super quick in comparison to the serialization time of the data.

Serialization speed			
Random Text			
Total number of structures (unit)	**Building phase time (nanoseconds)**	**Serialization time (nanoseconds)**	**Total time (nanoseconds)**
110	15 625	242 095	257 720
1 100	92 290	985 980	1 078 270
11 000	1 404 300	7 757 620	9 161 920

Serialization speed			
Same Text			
Total number of structures (unit)	**Building phase time (nanoseconds)**	**Serialization time (nanoseconds)**	**Total time (nanoseconds)**
110	16 340	232 165	248 505
1 100	97 515	1 021 190	1 118 705
11 000	887 235	7 618 290	8 505 525

There is no difference regarding time when doing both experiments.

Like XML, since Google Protocol Buffers have no type of compression, the serialization time for both experiments should stay the same or close. And that is, in fact, what we can see if we look at the figure 8.

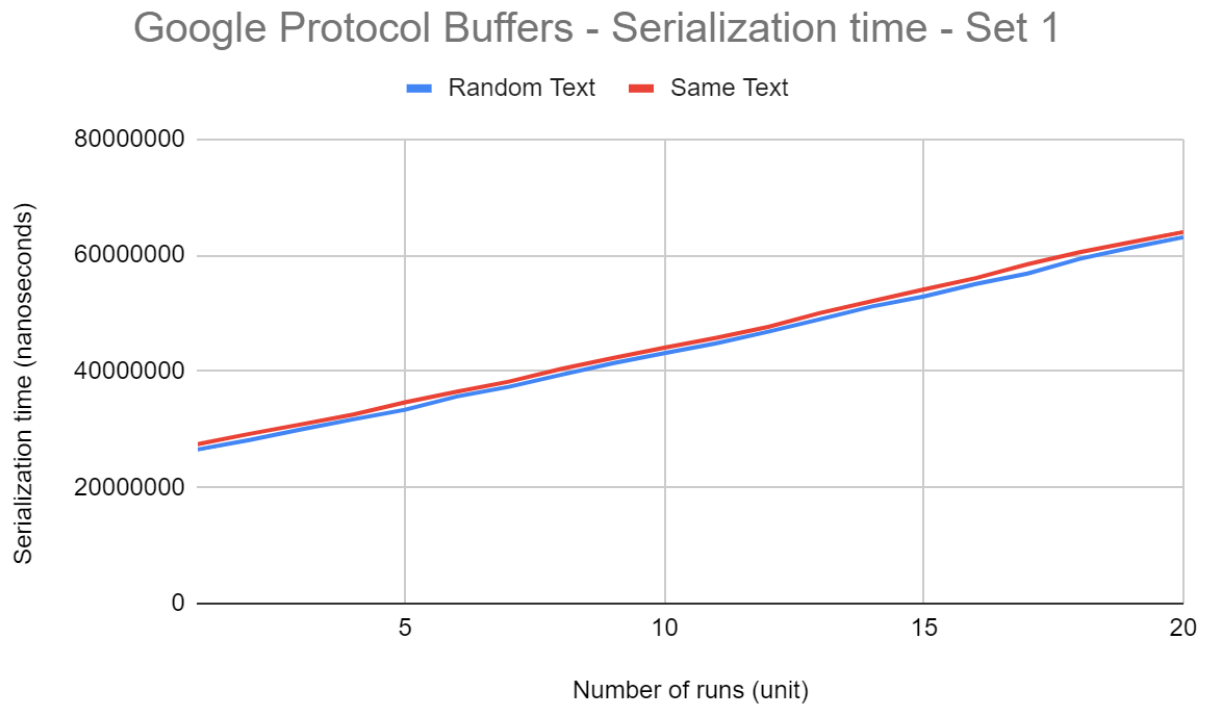

Figure 9 - Serialization time comparison between 'RandomText' and 'SameText' in Google Protocol Buffers - set 1

	Standard Deviation - Serialization	
Total number of structures (unit)	**Random Text**	**Same Text**
110	123568	77080
1 100	281701	321864
11 000	2588549	1177779

On the contrary, in deserialization, we can observe bigger times in the building phase - conversion of protocol buffer objects to Java objects. But, ultimately, the total time of deserialization is very close to that of serialization.

Deserialization speed			
Random Text			
Total number of structures (unit)	**Building phase time (nanoseconds)**	**Deserialization time (nanoseconds)**	**Total time (nanoseconds)**
110	152 615	37 665	190 280
1 100	571 260	356 810	928 070
11 000	4 903 575	4 063 970	8 967 545

Deserialization speed			
Same Text			
Total number of structures (unit)	**Building phase time (nanoseconds)**	**Deserialization time (nanoseconds)**	**Total time (nanoseconds)**
110	111 880	39 580	151 460
1 100	549 220	340 615	889 835
11 000	4 544 480	3 230 155	7 774 635

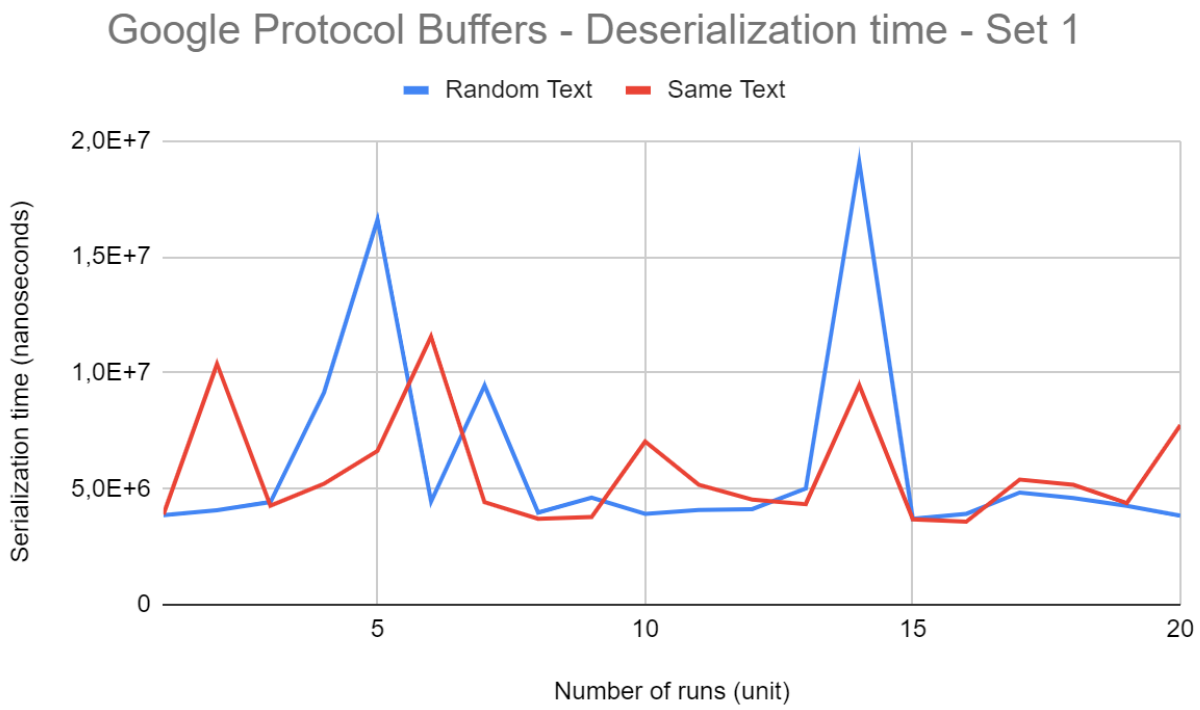

Figure 10 - Deserialization time comparison between 'RandomText' and 'SameText' in Google Protocol Buffers - set 1

Like serialization time, the behavior in deserialization should be the same. In this case, we can see more inconsistencies. Like explained in previous sections, this may have to do with the fact that the computer is running different processes during different runs of the experiment.

	Standard Deviation - Deserialization	
Total number of structures (unit)	**Random Text**	**Same Text**
110	99604	26532
1 100	279197	156947
11 000	2310724	1600017

5. Comparison and Discussion of results

In the next graphics, the data used for plotting was the mean time of each set - 110, 1100 and 11000 for XML, XML compressed with Gzip and Google Protocol Buffers.

We made sure to have the results in the same graphic so that the reader can better visualize the results and their meaning.

5.1. Serialization

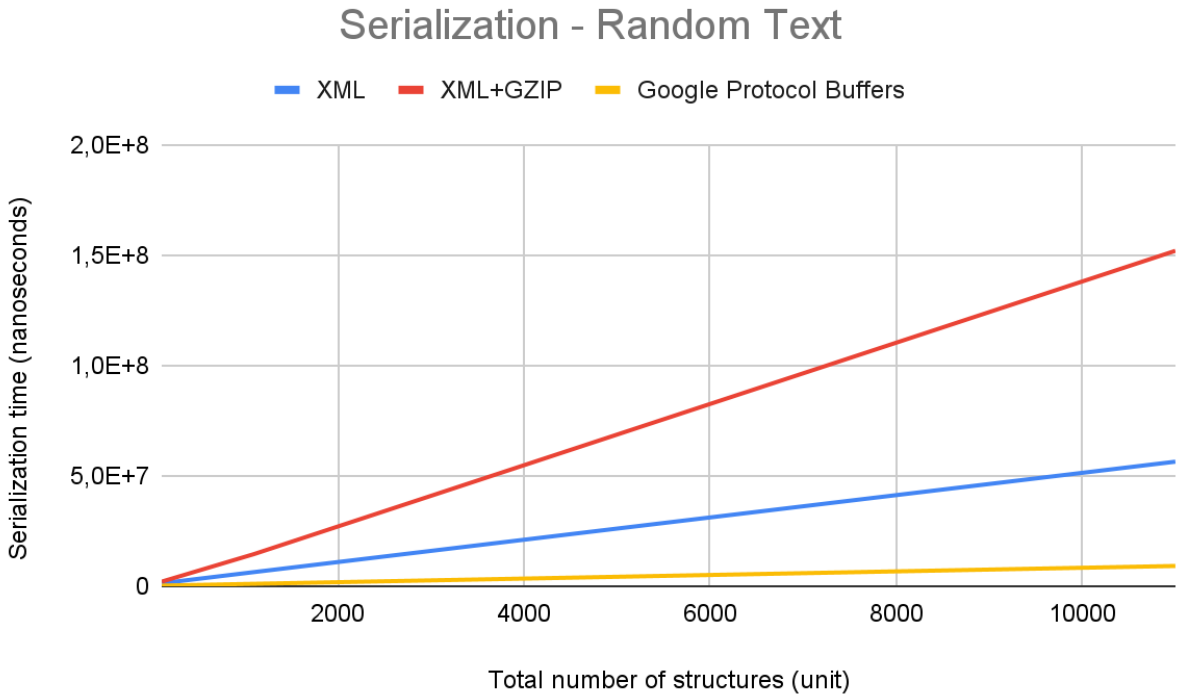

Figure 11 - Serialization time for XML vs XML compressed With Gzip vs Google Protocol Buffers in 'RandomText'.

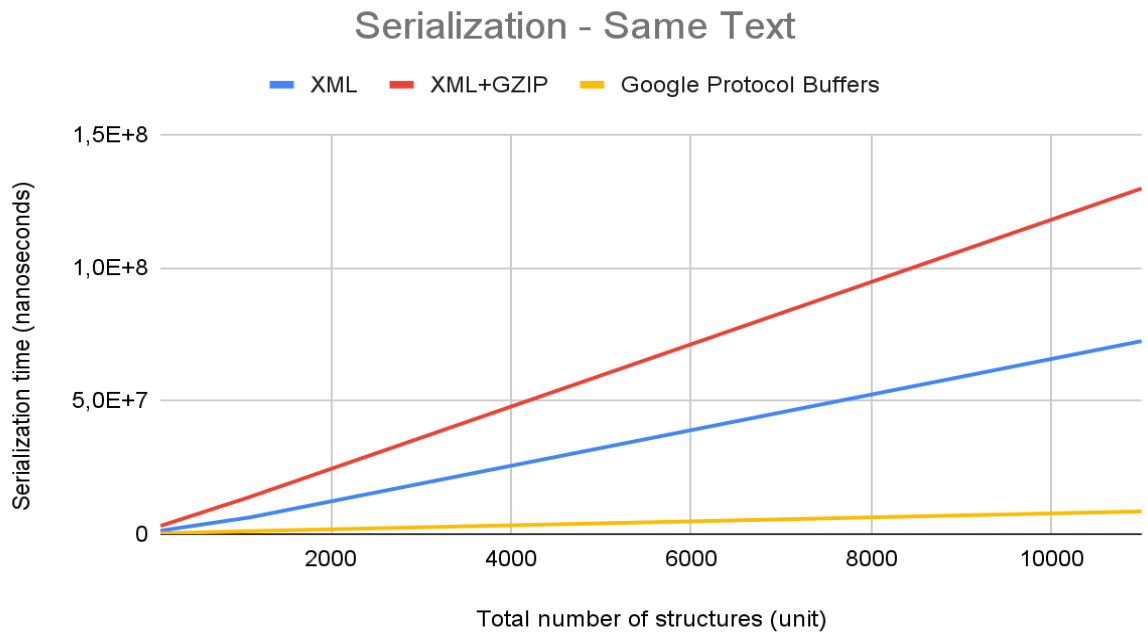

Figure 12 - Serialization time for XML vs XML compressed With Gzip vs Google Protocol Buffers in 'SameText'.

It is apparent that the XML compressed with Gzip takes more time to be serialized than XML. This behavior is normal given that it takes the same amount of time as serializing XML with the addition of time of the Gzip compression.

Since Protocol Buffers are binary-based serializers, then, during the serialization time they do not need to be worried (like XML) with data types and other class keywords for this matter giving them a time advantage in comparison to XML.

5.2. Deserialization

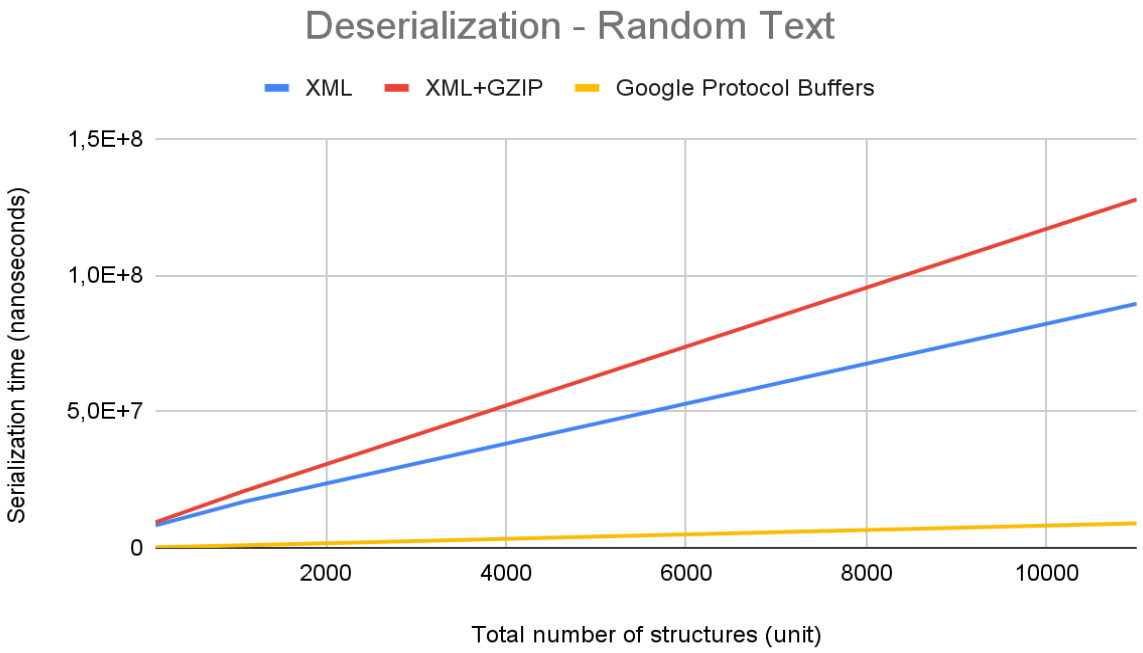

Figure 13 - Deserialization time for XML vs XML compressed With Gzip vs Google Protocol Buffers in 'RandomText'.

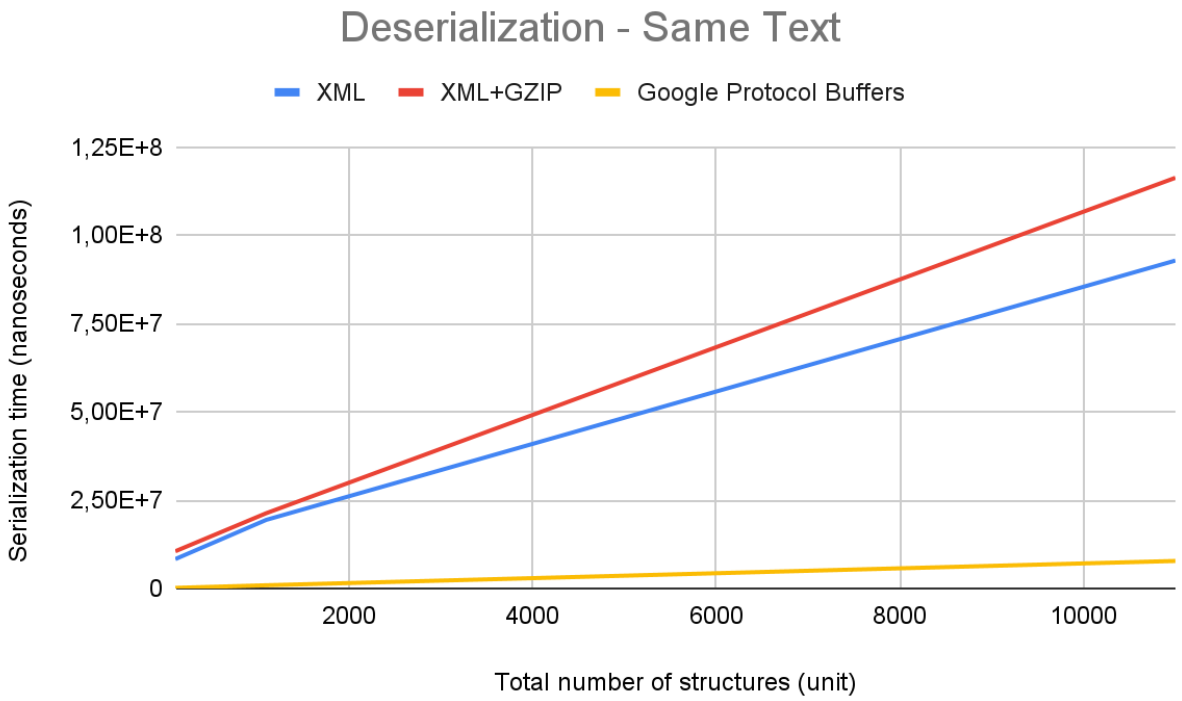

Figure 14 - Deserialization time for XML vs XML compressed With Gzip vs Google Protocol Buffers in 'SameText'.

If we take a close look at the graphic, we can see that, just like in serialization, the Google Protocol Buffers has the smallest time for deserialization. XML and XML compressed with Gzip are considerably slower, being the second one the worst of them. Because, as explained earlier, XML compressed with Gzip, has exactly the same steps to serialize/deserialize as XML, but adding the encoders/decoders time for Gzip.

5.3. File Size

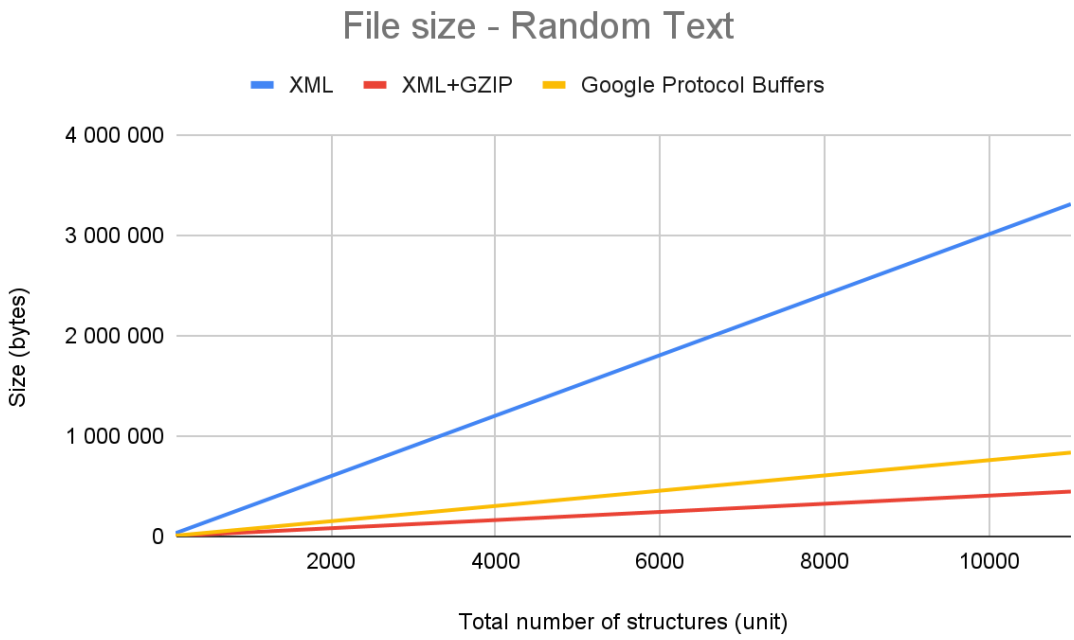

Figure 15 - File size for XML vs XML compressed With Gzip vs Google Protocol Buffers in 'RandomText'.

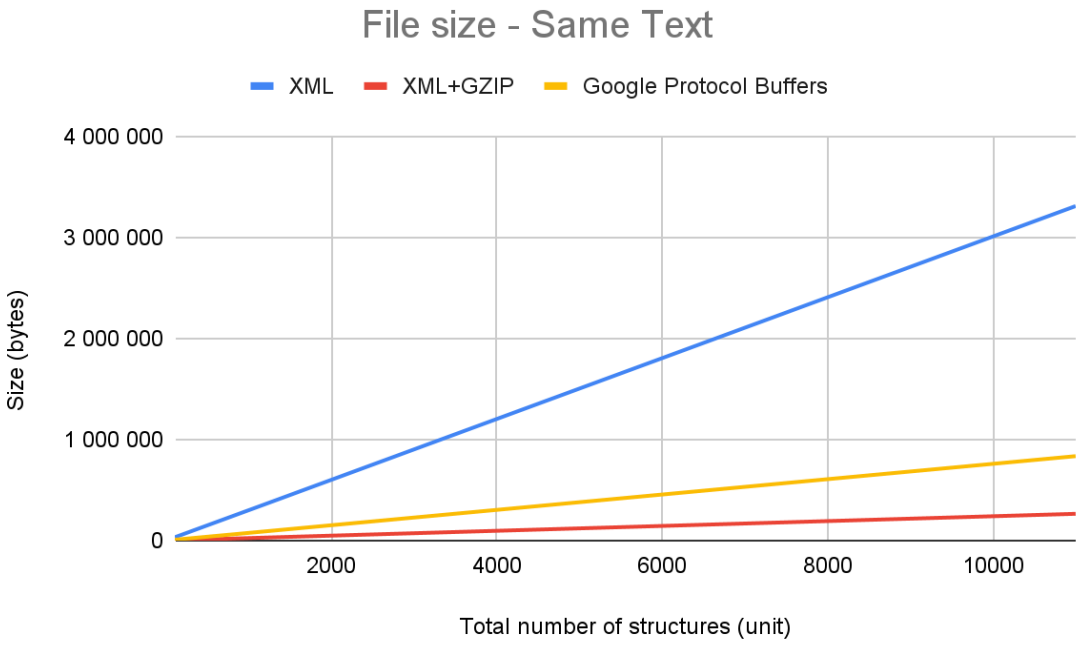

Figura 16 - File size for XML vs XML compressed With Gzip vs Google Protocol Buffers in 'SameText'.

When we look at the figure 13 we can see that XML compressed with Gzip is the best of all in terms of compressing files and storing them with the smallest size possible. However, Google Protocol Buffers are also a really good option if we compare him only with XML. This is due to the fact that Google Protocol Buffers is a binary-based serializer as opposed to XML which is text-based serializer. What this means is that XML is more human readable with an added cost of it being more verbose driven and, therefore, it allocates more space for this.

6. Conclusion

In conclusion, if you want a human readable file in which the data after serialization can be easily understood and modeled as text, you should be using XML besides it having a high serialization time as well as a bigger file size.

But if you do not have any constraints regarding the encoding type, and storage proves to be more important to you than the time component, then, on one hand, it is better to use the XML compressed with Gzip. On the other hand, you should use the Google Protocol Buffers if you want a faster serialization format.

Regarding the size of the file, since Google Protocol Buffers are slightly larger than XML compressed with Gzip this should not pose a problem due to the fact that it has a bigger ratio between file size vs serialization/deserialization time.

7. References

[1] https://developers.google.com/protocol-buffers
[2] https://mkyong.com/java/jaxb-hello-world-example/