A Performance Analysis of Native JSON Parsers in Java, Python, MS.NET Core, JavaScript, and PHP

Hardeep Kaur Dhalla
Department of Computing and New Media Technologies
University of Wisconsin-Stevens Point
Stevens Point, USA
hdhalla@uwsp.edu

***Abstract*—JavaScript Object Notation (JSON) has gained prominence over Extensible Markup Language (XML) due to its lightweight and flexible nature. This has also made it a first-class citizen when it comes to choose a data interchange format. All major programming platforms provide native libraries to parse JSON formatted data, but the performance of JSON parsers varies with their implementation. For further processing and analysis of semi-structured JSON data, it is first parsed which is expensive. In this research paper, performance analysis of JSON parsers in the native environment of 5 different programming languages has been done in terms of parsing speed and resource consumption. The experiment data is created using an algorithm written in Java that generates 10 different JSON files with an increasing number of key-value pairs at each level of JSON nesting. The native performance monitor on Windows 10 operating system is used to monitor and measure memory and CPU usage of the running processes. The output of data collector set is saved in the comma separated format for further analysis. The experimental results are discussed and shared at the end.**

***Index Terms*—JSON, Parsing, Parsers, performance analysis**

I. Introduction

The JavaScript Object Notation (JSON) is an emergent data format used by all major software solutions nowadays. Due to its lightweight and flexible nature, it has gained more importance as compared to Extensible Markup Language (XML) over the years. JSON's simplicity and less overhead than XML is paving the way for it to become a first choice to be used as data interchange format. It is also widely used to store and analyze large volume of data in big data applications and data warehouses. Many data analytical system such as Apache Spark have added native support to query JSON data [1]. Not only NoSQL databases, but all major non-relational databases such as Oracle, MS SQL server have also started to support it in their native environment [2]. The presence of JSON can already be seen in the smart grid systems [3], and mobile computing [4]. All business solutions have common need to extract the information from the raw data for the business users. With the ubiquity of distributed systems data needs to be transferred over the web by exploiting different means such as web applications and web services. The growing importance of Internet of Things (IoT) [5] have also emphasized the need of efficient data transfer. However, most of the time is spent to parse and process the data by the enterprises [1]. Though JSON falls under the JavaScript realm all major programming languages facilitate JSON parsing by providing their own implementation of JSON parsers [6]. In this research, comparative analysis of native JSON parsers in 5 different programming languages Java, Python, PHP, MS.NET Core, and JavaScript is done in terms of parsing speed, and resource consumption such as memory and CPU usage.

The rest of this paper is organized as follows: Section II presents the research objective, the related work is discussed in the Section III, Section IV explains Research methodology, experimental results are analyzed and discussed in the Section V, and Section VI provides the conclusion.

II. Research Objectives

This research will contribute to the understanding of how efficiently 5 different popular programming languages (Java, Python, MS.NET Core, PHP, and JavaScript) natively provide support for parsing the JSON files with fixed nesting depth and varying count of key-value pairs at each level of nesting in order to access the innermost key-value pair. The results of this research will help in choosing a right technology platform for parsing JSON in multifarious scenarios in the field of IoT, real time messaging and many other software solutions involving data exchange.

III. Related Work

Different serialization engines and programming platforms have been studied in the past to do the performance analysis using different data format such as XML, JSON and binary. The objective of the study [7] was to do the performance analysis of serialization using data formats in terms of native and third-party libraries on the .NET programming platform. It was observed that binary format provides better performance in terms of speed than other formats. Authors in [8] have done the performance comparison of object serialization in Java and .NET in terms of XML and binary data formats. In another study [9] different marshalling and unmarshalling solutions were analyzed in Java using different data format. 12 object serialization libraries were studied in [10] to do the performance comparison in terms of XML, JSON, and binary formats. In the research [4], authors have done the comparison of data serialization efficiency using XML and JSON data formats on the mobile platform.

Authors in [11] have revealed that Xerces Java parser performed better than MSXML and the proposed xParser. The aim of this study was to benchmark the performance of XML parsers in Java and .NET. Another study [12] showed DOM API takes more time and consumes more memory to parse XML data as compared to SAX API. Java platform was chosen to do the performance comparison of parsing efficiency of both XML parser implementations with varying file size of XML files. In [13], [1] authors have provided a faster implementation of JSON parser in the data analytical software solutions. To the best of my knowledge there has not been any study conducted in the past to do the performance analysis of the native JSON parsers with 10 different JSON files of increasing size with varying key-value pair count on 5 popular programming platforms (Java, Python, PHP, JavaScript, and MS.NET Core) in terms of parsing speed and resource consumption such as memory and processor time.

IV. Research Methodology

This section explains the performance criteria, algorithm used for generating hypothetical JSON data along with details of the setup for the experiment.

A. Performance Criteria

In this research, the parsing efficiency of JSON parsers is evaluated in terms of parsing speed in milliseconds (ms) and consumption of resources such as CPU, and memory. Using Windows 10 native performance monitor application, two performance counters Working-Set private and %Processor Time, are monitored and measured. The Working-Set private performance counter is used to measure memory allocated to the process in bytes. Process (%Processor Time) performance counter is used to measure CPU usage. For all 5 parsers, user defined data collector sets are created, and output is saved in a comma separated format.

B. Experiment data

The experiment data generation was automated using a Java console application in order to be able to adjust the key-value pair count at each level of the nesting and the number of elements in an array placed at the end of the innermost level as shown in the Fig. 1.

Considering, the two main factors influencing the complexity of a JSON string and in turn adding the challenge to parse it are the JSON nesting depth and the number of key-value pairs at each level. Hence, a JSON schema and a algorithm is designed for this study to produce JSON strings of varying key-value pairs at each level of nesting and fixed nesting depth, in turn affecting the length of the JSON path to be accessed by the parser. 10 JSON files were generated by varying key-value pair count i.e. n at each level with an increment of 10 from 10 to 100. For the purpose of this experiment, JSON nesting depth was fixed to 20 (i.e. m=20). The program accepts the user inputs to produce a JSON file with a depth fixed to 20, with each level having 'n' number of key-value pairs. Each key contains a random string as its value except the n^{th} key at each level contains the JSON object representing the next level. The n^{th} key of the innermost level contains a random string as a depth terminator and $(n+1)^{th}$ element contains an array of JSON objects. Each JSON object in the array has a depth of 20 levels but contains n/2 key-value pairs at each level.

This experiment will be conducted by parsing the JSON files with fixed nesting depth and varying key-count in order to access the innermost key-value pair of the last element of the array. Hence, the depth of the JSON path being accessed turns out to be 40.

C. Experiment Setup

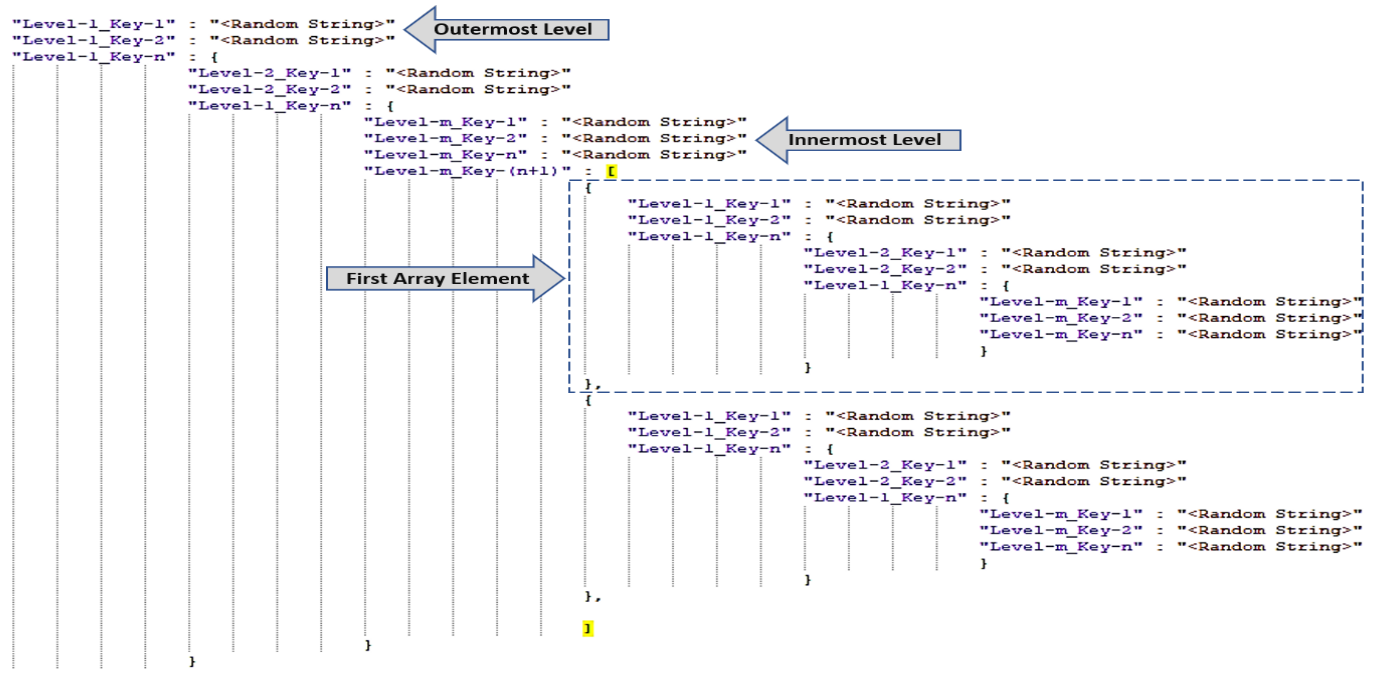
In this experiment, 10 different JSON files were generated using algorithm described in the previous section with increasing number of JSON key-count. The details of generated JSON data in terms of file size in KB and JSON key-count are shown in the Table I. The parsing speed in ms was measured for each parser while accessing the innermost key-value pair of the last element of the array in order to maximize the effort required by the JSON parser. This experiment was run on a machine with the following specifications: Windows 10 Home OS, Intel core i5-7200U processor, 2.50 GHz processor, and 16 GB RAM.

TABLE I
JSON Key Count and File Size

File No.	JSON Key Count	File size in KB
1	10	54
2	20	115
3	30	177
4	40	239
5	50	301
6	60	364
7	70	426
8	80	488
9	90	550
10	100	612

Oracle's JSON-P [14] library is an implementation of the Java Application Programming Interface (API) for JSON Processing [15]. This library provides the capabilities to work with both: The Object Model API and the Streaming API. To enable the frequent random access of parsed JSON data, JSON class of the JSON-P library was used as the entry point to create a Document Object Model (DOM) in the memory using Object Model API approach. Using Eclipse Integrated environment (IDE) 2019-12, a Java 8 console application was created with required dependencies javax.json-1.1.4.jar, and javax.json-api-1.1.4.jar added to the build path. Max heap size was set to 50MB in the run configurations of the application.

Microsoft has replaced Newtonsoft.json with System.Text.Json [16] library as the native implementation in MS.NET Core 3.0 to parse JSON data. This new high performing library provides capabilities to serialize objects to JSON and deserialize JSON to objects besides supporting utf-8 encoding natively. Using Visual Studio (2019), a



```
"Level-1_Key-1" : "<Random String>"
"Level-1_Key-2" : "<Random String>"
"Level-1_Key-n" : {
                "Level-2_Key-1" : "<Random String>"
                "Level-2_Key-2" : "<Random String>"
                "Level-1_Key-n" : {
                                "Level-m_Key-1" : "<Random String>"
                                "Level-m_Key-2" : "<Random String>"
                                "Level-m_Key-n" : "<Random String>"
                                "Level-m_Key-(n+1)" : [
                                           {
                                              "Level-1_Key-1" : "<Random String>"
                                              "Level-1_Key-2" : "<Random String>"
                                              "Level-1_Key-n" : {
                                                        "Level-2_Key-1" : "<Random String>"
                                                        "Level-2_Key-2" : "<Random String>"
                                                        "Level-1_Key-n" : {
                                                                  "Level-m_Key-1" : "<Random String>"
                                                                  "Level-m_Key-2" : "<Random String>"
                                                                  "Level-m_Key-n" : "<Random String>"
                                                                  }
                                                        }
                                           },
                                           {
                                              "Level-1_Key-1" : "<Random String>"
                                              "Level-1_Key-2" : "<Random String>"
                                              "Level-1_Key-n" : {
                                                        "Level-2_Key-1" : "<Random String>"
                                                        "Level-2_Key-2" : "<Random String>"
                                                        "Level-1_Key-n" : {
                                                                  "Level-m_Key-1" : "<Random String>"
                                                                  "Level-m_Key-2" : "<Random String>"
                                                                  "Level-m_Key-n" : "<Random String>"
                                                                  }
                                                        }
                                           },

                                           ]
                                }
                }
```

Fig. 1. JSON SCHEMA

console application in MS.NET Core 3.1 was created using C# and System.Text.Json namespace was added to exploit the methods of JsonDocument class to create a DOM in the memory.

Python, an interpreted high-level programming language, also support JSON parsing natively [17]. Python 3.8.3 console application was created in PyCharm IDE 2020.1 and have used native json module's loads method to parse JSON data.

Node.js 12.18.2 [18] was installed on the local machine and was used to run the JavaScript code outside the browser. It is an open-source framework and powered by Chrome's V8 JavaScript engine. Furthermore, it provides high performing JavaScript run time environment. JSON.parse() method was used to parse the JSON in the JavaScript code.

The Hypertext Preprocessor (PHP) [19] is a general-purpose server-side scripting language widely used in the web applications development. It provides json_decode function to parse JSON data. PHP script was written using a Notepad++ editor. To run the PHP script on the Apache Server, Apache Xampp 7.4.7 / PHP 7.4.7 was installed on the local machine. For the purpose of this experiment, max_execution_time and max_input_time was set to 12000 seconds in the php.ini file. After starting the server, Firefox browser was used as a client to request the server to run the PHP script.

The details of the experiment design are shown in Fig. 2. The JSON formatted data is read from the file into a string and parsed 500 times as a warm-up. The current timestamp is recorded in the start of each test run which includes parsing the JSON 50 times(m=50), followed by recording the duration of each run. To increase the accuracy and the reliability of results, 5000 of such test runs(n=5000) were performed and the average runtime was noted. This whole experiment is repeated for all 5 Java, JavaScript, PHP, MS.NET Core, and Python parsers with 10 different JSON files of increasing size with varying key-value pair count.

V. Experimental Results and Discussion

The experimental results of parsing speed, memory consumption, and CPU consumption are shown in Fig. 3, Fig. 4, and Fig. 5 respectively. For all file sizes as shown in Fig. 3 , a positive correlation has been observed between the file size and the parsing speed for all 5 parsers. For 54KB and 115KB files, JavaScript has performed better than other 4 parsers. Java parser got a little edge over JavaScript when the file size was incremented to 177KB and 239KB. Nonetheless, for file size range of 301KB-612KB, it has been observed that JavaScript has parsed JSON data more efficiently than its other 4 counterparts. Both PHP and Python parsers have performed comparably in terms of parsing efficiency for most of the file sizes and worse than other 3 parsers.

In the case of MS.Net Core, when the file size falls in the range 54KB-550KB it has performed comparably with Java, however, a sharp spike has been noticed when the file size is increased to 612 KB. It is revealed from the results that MS.NET Core's parsing time is consistently between the other two sets of parsers; one set is Java and JavaScript, and the other one is PHP and Python. For the smallest file size 54KB, the minimum parsing time is taken by JavaScript 10.7906 ms whereas PHP has parsed the same data in 30.03030254 ms. When the file size was incremented to 612KB which is the biggest file in the data set, the minimum time requirement

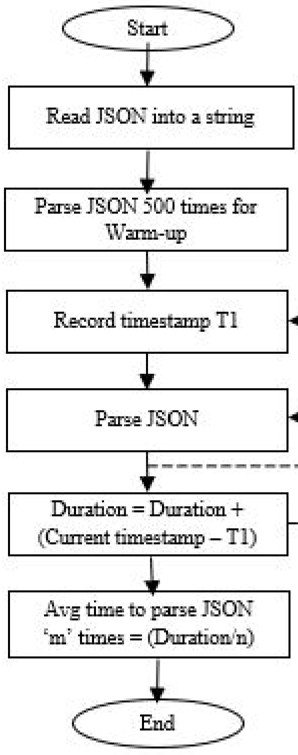

Start
Read JSON into a string
Parse JSON 500 times for Warm-up
Record timestamp T1
n
Parse JSON
m
Duration = Duration + (Current timestamp – T1)
Avg time to parse JSON 'm' times = (Duration/n)
End

Fig. 2. Experiment design flow

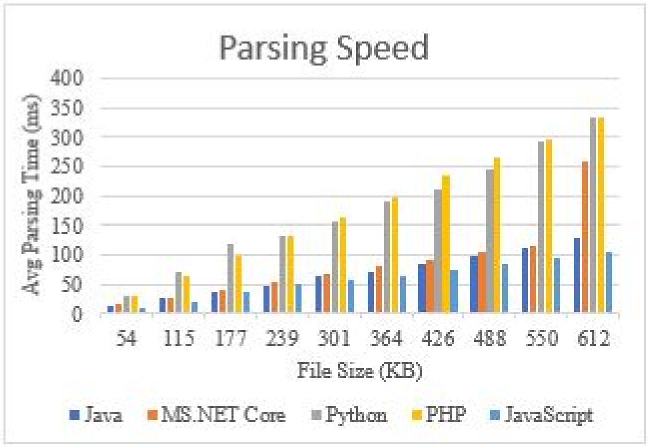


Fig. 3. Parsing Speed Comparison

is 105.1322105 ms by JavaScript whereas Python has taken 333.6603588 ms.

The results shown in the Fig. 4 reveals that Python has consumed the least amount of memory as compared to its peers. The usage of CPU resources by all the parsers is comparable as shown in the Fig. 5.

VI. Conclusion

From the results, it can be concluded that in this experiment JavaScript parsed the JSON string more efficiently than its peers. Though, for all file sizes and varying count of key-value pairs at each level of nesting, Java has performed comparably to JavaScript besides consuming less resources such as memory and CPU. The PHP and Python parsers have always taken more time to parse JSON formatted data as compared to other 3 parsers regardless of the file size and count of key-value pairs at each level of nesting. In future work, this experiment can be extended to more programming languages and more test scenarios can be added with real-world JSON data.

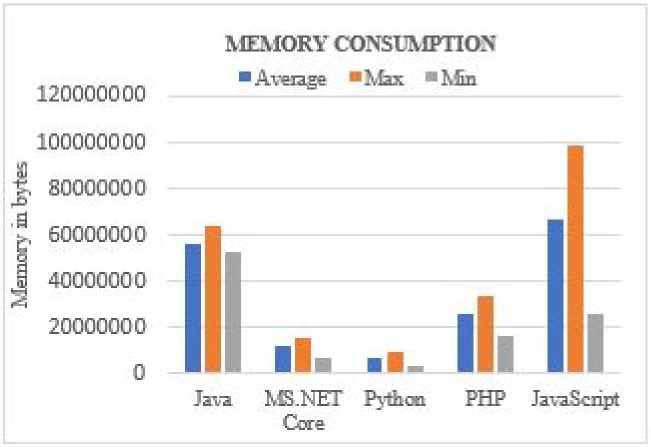


Fig. 4. Memory Consumption Comparison

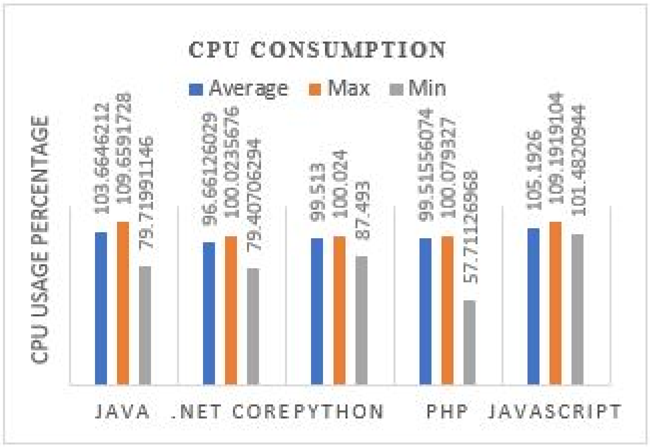


Fig. 5. CPU Consumption Comparison

References

[1] X. Shi, Y. Zhang, H. Huang, Z. Hu, H. Jin, H. Shen, Y. Zhou, B. He, R. Li, and K. Zhou, "Maxson: Reduce duplicate parsing overhead on raw data," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1621–1632.

[2] I. Hrubaru, G. Talabă, and M. Fotache, "A basic testbed for json data processing in sql data servers," in *Proceedings of the 20th International Conference on Computer Systems and Technologies*, ser. CompSysTech '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 278–283. [Online]. Available: https://doi.org/10.1145/3345252.3345285

[3] B. Petersen, H. Bindner, S. You, and B. Poulsen, "Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things," in *2017 Computing Conference*, 2017, pp. 1339–1346.

[4] A. Sumaray and S. K. Makki, “A comparison of data serialization formats for optimal efficiency on a mobile platform,” in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ser. ICUIMC ’12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2184751.2184810

[5] P. Wehner, C. Piberger, and D. Göhringer, “Using json to manage communication between services in the internet of things,” in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, 2014, pp. 1–4.

[6] Introducing json. https://www.json.org/json-en.html.

[7] A. Nagy and B. Kovari, “Analyzing .net serialization components,” in *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2016, pp. 425–430.

[8] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic, “Object serialization analysis and comparison in java and .net,” *SIGPLAN Not.*, vol. 38, no. 8, p. 44–54, Aug. 2003. [Online]. Available: https://doi.org/10.1145/944579.944589

[9] T. Aihkisalo and T. Paaso, “A performance comparison of web service object marshalling and unmarshalling solutions,” in *2011 IEEE World Congress on Services*, 2011, pp. 122–129.

[10] K. Maeda, “Performance evaluation of object serialization libraries in xml, json and binary formats,” in *2012 Second International Conference on Digital Information and Communication Technology and it’s Applications (DICTAP)*, 2012, pp. 177–182.

[11] D. Wu, K. T. Chau, J. Wang, and C. Pan, “A comparative study on performance of xml parser apis (dom and sax) in parsing efficiency,” in *Proceedings of the 3rd International Conference on Cryptography, Security and Privacy*, ser. ICCSP ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 88–92. [Online]. Available: https://doi.org/10.1145/3309074.3309124

[12] S. C. Haw and G. S. V. R. K. Rao, “A comparative study and benchmarking on xml parsers,” in *The 9th International Conference on Advanced Communication Technology*, vol. 1, 2007, pp. 321–325.

[13] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, “Mison: A fast json parser for data analytics,” *Proc. VLDB Endow.*, vol. 10, no. 10, p. 1118–1129, Jun. 2017. [Online]. Available: https://doi.org/10.14778/3115404.3115416

[14] Oracle. Json processing (json-p) - home. https://javaee.github.io/jsonp/.

[15] J. Kotamraju. Java api for json processing: An introduction to json. https://www.oracle.com/technical-resources/articles/java/json.html.

[16] Microsoft. Serialize and deserialize json using c# - .net. https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-text-json-overview.

[17] json — json encoder and decoder. https://docs.python.org/3/library/json.html.

[18] Node.js. https://nodejs.org/en/.

[19] Php. https://www.php.net/.