JSON Compression Algorithms

Alexandru Objelean
Moldova State University
alex.objelean@gmail.com

***Abstract* — JSON (Java Script Object Notation) [1] is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It can be used as a data interchange format, just like XML. When comparing JSON [1] to XML, it has several advantages over the last one. JSON [1] is really simple, it has a self-documenting format, it is much shorter because there is no data configuration overhead. That is why JSON is considered a fat-free alternative to XML.**

However, the purpose of this article is not to discuss the pros and cons of JSON [1] over XML. Though it is one of the most used data interchanged format, there is still room for improvement. For instance, JSON [1] uses excessively quotes and key names are very often repeated. This problem can be solved by JSON [1] compression algorithms. There are more than one available. Here you'll find an analysis of two JSON [1] compressors algorithms and a conclusion whether JSON [1] compression is useful and when it should be used.

I. INTRODUCTION

Though JSON [1] data format is much more efficient than XML, still it is inefficient exchange between a web server and a browser. For one, it converts everything to text. A second problem is its excessive use of quotes, which add two bytes to every string. Thirdly, it has no standard format for using a schema. When multiple objects are serialized in the same message, the key names for each property must be repeated, even though they are the same for each object.

JSON [1] used to have an advantage because it could be directly parsed by a javascript engine, but even that advantage is gone because of security and interoperability concerns. About the only thing JSON [1] going for it is that it is usually more compact than the alternative, XML, and it is well supported by many web programming languages.

Compression of JSON data is useful when large data structures must be transmitted from the web browser to the server [2]. In that direction, it is not possible to use gzip compression, because it is not possible for the browser to know in advance whether the server supports gzip. The browser must be conservative, because the server may have changed abilities between requests.

II. CJSON COMPRESSION ALGORITHM

CSJON[5] compress the JSON [1] with automatic type extraction. It tackles the most pressing problem: the need to constantly repeat key names over and over. Using this compression algorithm, the following JSON [1]:

```
[
    { // This is a point
        "x": 100,
        "y": 100
    },
    { // This is a rectangle
        "x": 100,
        "y": 100,
        "width": 200,
        "height": 150
    },
    {}, // an empty object
    ... // thousands more
  ]
```

You can notice that a lot of the space is taken up by repeating the key names "x", "y", "width", and "height". They only need to be stored once for each object type:

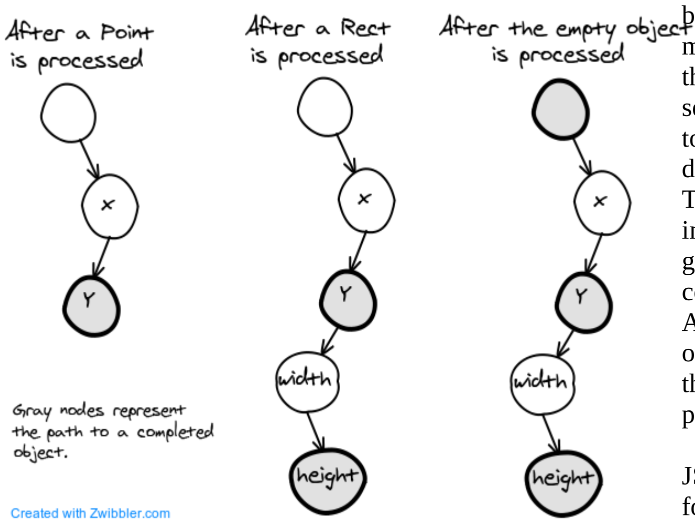
```
{
    "templates": [ ["x", "y"], ["x",
"y", "width", "height"] ],
    "values": [
        { "type": 1, "values": [ 100,
100 ] }, { "type": 2, "values": [100,
100, 200, 150 ] }, {} ]
  }
```

Each object in the original input is transformed. Instead of listing the keys, the "type" field refers to a list of keys in the schema array. But we are still repeating "x", and "y". The rectangle shared these properties with the point type, and there is no need to repeat them in the schema. The optimization can go even farther. Since we are trying to save space, we rename our properties, and stick in a format code so we can detect that compresed json is used. The compressed json can look like this:

```
{
    "f": "cjson",
    "t": [ [0, "x", "y"], [1, "width",
"height"] ],
    "v": [ { "": [1,  100, 100 ] }, {
"": [2, 100, 100, 200, 150 ] }, {} ]
  }
```

The above example shows how a small sample of json is transformed into a compressed version. The impact is even bigger when you are trying to compress a large json file, containing hundreds or even thousands of entries.

The hard part is finding the objects which share sets of keys. It sounds a lot like the Set Cover problem, and if so, an optimal solution is NP-complete. Instead, we will approximate the solution using a tree structure. While we are building the value array, when we encounter an object, we add all of its keys to the tree in the order that we encounter them.


After a Point is processed
After a Rect is processed
After the empty object is processed
x
Y
width
height
Gray nodes represent the path to a completed object.
Created with Zwibbler.com

At the end of the process, the nodes of the tree can be traversed and templates created. Nodes which represent the end of a key list (shown in gray) must have entry in the key list. Although not illustrated here, nodes with multiple children are also points where the the child object types inherit from a common parent, so they also get an entry.

The astute reader will realize that the final schema depends on the order that we inserted the keys into the tree. For example, if, when we encountered the rectangle, we inserted the keys "width" and "height" before "x", and "y", the algorithm would not find any common entries.

It is possible to gain more efficient packing by using a greedy algorithm. In the greedy algorithm, before we begin, an initial pass through all the objects would be made to build a list of unique object types. Then when it comes time to insert keys into the tree, they are first sorted so that the ones which occur in the most unique types are inserted first. However, this method adds a lot of extra processing and I feel the gains would not be worthwhile.

III. HPack Compression Algorithm

The most common practice to serve documents, such XML, plain text or JSON, is the gz or deflate compressed output. Unfortunately, even if every browser has a built in zlib module to decompress strings on requests completed, JavaScript cannot use this feature to compress/decompress same strings. This limit is more “mono-directional” because thanks to JSON [1] and gzipped outputs we can send to the client huge amount of data without compromising both bandwidth and response time.
A big limit is to manipulate received collection, and send back in “one shot” a consistent amount of data (the received collection itself, why not). Thanks to JSON.hpack [3] we can send from client to server up to 70% less characters than a normal JSON [1] post request.

The result is a faster interaction in both ways and, even if JavaScript or the server will spend few milliseconds to pack or unpack long collections, the total elapsed time between the sent action and the response, plus the total bandwidth used to both send and receive (think about mobile connections as well) will be less than ever. For these reason, the most important thing is to have many server-side implementations as possible in order to be able to unpack collections sent via client or to understand that data and to manipulate it on the server without problems. The unpack operation is indeed truly fast and simple to implement as well. To send back data we can still use gzipped/deflated strings, especially because these compressor algorithms are both fast and bandwidth savers.
As summary, without gzip the generated JSON.hpack [3] output could fall down from 70Kb to 26Kb while via gzip the difference will not be that consistent (repeated JSON property names are well compressed).

JSON.hpack [3] is a lossless, cross language, performances focused, data set compressor. It is able to reduce up to 70% number of characters used to represent a generic homogeneous collection.

The HPack [3] compression algorithm is based on the idea that JSON data format contains a lot of redundant property names.

Using HPack algorithm [3], the following sample of JSON:

```
{
  "id" : 1,
  "sex" : "Female",
  "age" : 38,
  "classOfWorker" : "Private",
  "maritalStatus" : "Married-civilian
spouse present",
  "education" : "1st 2nd 3rd or 4th
grade",
  "race" : "White"
}
```

Can be transformed into:

```
["id","sex","age","classOfWorker","mari
talStatus","education","race"],[1,"Fema
le",38,"Private","Married-civilian
spouse present","1st 2nd 3rd or 4th
grade","White"]
```

The HPack [3] algorithm provides several levels of compression (from 0 to 4). Each level introduces an additional feature, by improving the compressing efficiency. The level 0 compression performs the most basic compression by removing keys (property names) from the structure creating a header on index 0 with each property name. Next levels make it possible to reduce even more the size of the JSON by assuming that there are duplicated entries.

IV. ANALYSIS

The purpose of this analysis is to compare each of the described JSON compressor algorithms. For this purpose we will use 5 files with JSON content having different dimensions, varying from 50K to 1MB. Each JSON file will be served to a browser using a servlet container (tomcat) with the following transformations:

- Unmodified JSON - no change on the server side

Minimized JSON - remove whitespaces and new lines (most basic js optimization)

- Compressed JSON using CJSON algorithm
- Compressed JSON using HPack algorithm
- Gzipped JSON - no change on the server side
- Gzipped and minimized JSON
- Gzipped and compressed using CJSON algorithm
- Gzipped and compressed using HPack algorithm

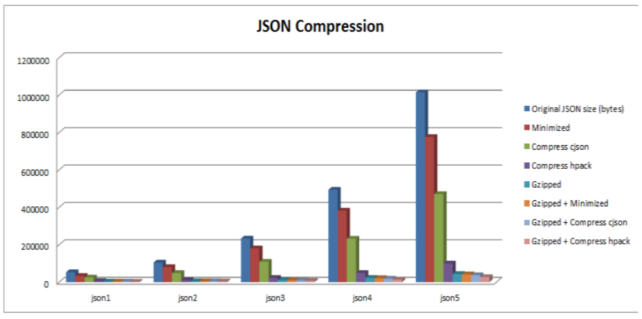
This table contains the results of the benchmark. Each row of the table contains one of the earlier mentioned transformation. The table has 5 columns, one for each JSON file we process.

TABLE I. RESULTS

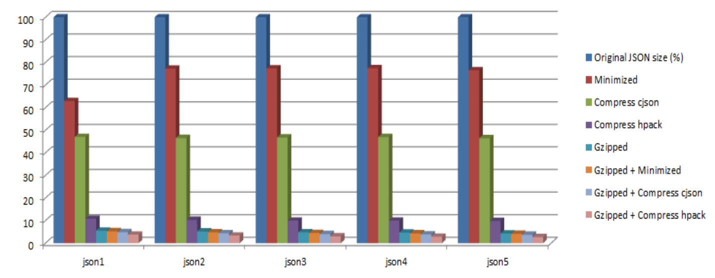
	JSON1	JSON2	JSON3	JSON4	JSON5
Original JSON size (bytes)	52966	104370	233012	493589	1014099
Minimized	33322	80657	180319	382396	776135
Compress CJson	24899	48605	108983	231760	471230
Compress HPack	5727	10781	23162	49099	99575
Gzipped	2929	5374	11224	23167	43550
Gzipped and minimized	2775	5035	10411	21319	42083
Gzipped and compressed with CJson	2568	4605	9397	19055	37597
Gzipped and compressed with HPack	1982	3493	6981	13998	27358

The following two graphics are the representations of the data included in the above table. The first graphic groups results for each processed JSON using all types of transformations using Y axis for absolute size of JSON file in bytes. The second graphic is similar, but uses the Y axis for relative size in percentage.

Transformation Results in size (bytes)



Transformation Results in relative size (percentage)



The relative size of transformation graphic is useful to see if the size of the json to compress affects the efficiency of compression or minimization. You can notice the following:

- the minimization is much more efficient for smaller files. (~60%)
- for large and very large json files, the minimization has constant efficiency (~75%)
- compressors algorithms has the same efficiency for any size of json file
- CJson compressing algorithm is less efficient (~45%) than hpack algorithm (~8%)
- CJson compressing algorithm is slower than hpack algorithm
- Gzipped content has almost the same size as the compressed content
- Combining compression with gzip or minimization with gzip, doesn't improve significantly efficiency (only about 1-2%)

V. CONCLUSION

Both JSON compression algorithms are supported by the web resource optimizer for java (wro4j) [6] framework by the following processors: CJsonProcessor & JsonHPackProcessor. Both of them provide the following methods: pack & unpack. The underlying implementation uses Rhino engine to run the javascript code on the serverside.

JSON Compression algorithms considerably reduce JSON file size. There a several compression algorithms. We have covered two of them: CJson [4] and HPack [3]. HPack seems to be much more efficient than CJson and also significantly faster. When two entities exchange JSON and the source compress it before it reach the target, the client (target) have to apply the inverse operation of compression (unpacking), otherwise the JSON cannot be used. This introduces a small overhead which must be taken into account when deciding if JSON compression should be used or not.

When gzipping of content is allowed, it has a better efficiency than any other compression algorithm. In conclusion, it doesn't worth to compress a JSON on the server if the client accepts the gzipped content. The compression on the server-side does make sense when the client doesn't know how to work with gzipped content and it is important to keep the traffic volume as low as possible (due to cost and time).

Another use-case for JSON compression algorithm is sending a large JSON content from client to server (which is sent ungzipped). In this case, it is important to unpack the JSON content on the server before consuming it.

REFERENCES

[1] JSON RFC 4627 documentation: http://www.ietf.org/rfc/rfc4627.txt?number=4627

[2] G. Michael Connolly; Mehmet Akin; Ankur Goyal; Robin Howlett; Matthew Perrins, Building Dynamic Ajax Applications Using WebSphere Feature Pack for Web 2.0 Pp. 579

[3] HPack algorithm implementation: https://github.com/WebReflection/json.hpack

[4] CJson algorithm implementation: http://stevehanov.ca/blog/cjson.js

[5] CJson algorithm description: http://stevehanov.ca/blog/index.php?id=104

[6] Web Resource Optimization for java framework: http://code.google.com/p/wro4j/