JSON Patch for turning a pull REST API into a push

Hanyang Cao[1], Jean-Rémy Falleri[1], Xavier Blanc[1], Li Zhang[2]

[1] University of Bordeaux, LaBRI, UMR 5800
F-33400, Talence, France
{cao.hanyang, falleri, xblanc}@labri.fr
[2] Beihang University, Beijing, China
lily@buaa.edu.cn

Abstract. REST APIs together with JSON are commonly used by modern web applications to export their services. Such an architecture however makes the services reachable in a pull mode which is not suitable for accessing data that periodically changes. Turning a service from a pull mode to a push mode is therefore frequently asked by web developers that want to access changing data and to get notified of performed changes. Converting a pull mode API into a push mode obviously requires to make periodical calls to the API but also to create a patch between each successive received versions of the data. The latter is the most difficult part and this is where existing solutions have some imperfections. To face this issue, we present a new JSON patch algorithm that is compliant with JSON Patch RFC, and that supports *move* and *copy* change operations. We implement our algorithm in a JavaScript library and evaluate its performance. Our evaluation done with real industrial data shows that our library creates small patches compared with other libraries, and creates them faster.

Keywords: REST, JSON, Diff, Patch, Web Application.

1 Introduction

Most of the web applications[3] provide an access to their services thanks to a REST API [9]. Their services are then directly reachable by HTTP requests, where the exchange of data is commonly done in JSON, the JavaScript Object Notation [7].

REST APIs have been however designed to be used in a pull mode request, which is inadequate for services that provide access to data that periodically change. For example, Twitter[4] provides a REST API with a service that returns a timeline of tweets. As any timeline changes quite frequently, the clients that use this API have actually to periodically call the service to refresh their views. Worst, if they just want to be aware of new tweets appearing in the timeline,

[3] https://www.publicapis.com/
[4] https://twitter.com

they also have to create the patch describing the differences between the data they previously received and the new one just returned by the request, which can be highly complex depending on the structure of the JSON documents contained in the response of the request.

In opposite to the pull mode, the push mode is more adequate for accessing changing data. Its principle is to send notification messages to the clients that have registered, and only when the data have changed. Further the messages contain the set of changes performed to the data rather than the new version of the data, letting the client react to them if needed.

Turning a service from a pull mode to a push mode is therefore frequently asked by web developers that want to access changing data and to get notified of performed changes. Some companies already supports this need. For example, our partner StreamData.io provides a proxy server that converts the pull mode API of an existing web application into push mode one.[5]

Converting a pull mode API into a push mode one obviously requires to make periodical calls to the API but also to create a patch between each successive received versions of the data. The latter is the most difficult part and this is where existing solutions have some imperfections. Indeed, creating a patch between two documents is a well-known very complex problem [15,4], which has not been studied yet for JSON documents. A JSON document is a labelled unordered tree that contains arrays (ordered sequences). Creating a patch between two JSON documents may therefore lead to a NP-hard problem depending both on the change operations that are considered (add, remove, move, copy), and on the quality of the created patch (in terms of size).

In this paper we propose a new patch algorithm that is tailored to JSON documents, and that drastically improves the conversion of pull mode APIs into push mode ones. Our algorithm returns a JSON Patch as specified by the JSON Patch RFC [3]. It therefore handles any changes that can be done on JSON documents, either on their basic properties or on their arrays, and supports simple changes (add, remove) as well as complex ones (move, copy), which allows clients to deeply understand changes that have been done.

We implemented our algorithm in JavaScript as it is the most commonly language used in web applications. We validate it by making a comparison with other JavaScript libraries that support the JSON patch RFC. This validation has been done by using real data provided by our partner StreamData.io.

As a main result, we provide:

- A new JSON patch algorithm that is fully complies with the JSON Patch RFC.
- A JavaScript implementation of our algorithm that performs better than the existing ones.

The structure of the paper is as follows. To start, the Section 2 gives a background about the JSON patch format as well as its computation, and further presents the existing approaches that support patch creation. The Section 3

[5] http://streamdata.io/

presents our algorithm (named JDR). The Section 4 then presents the evaluation of our JavaScript framework implementing our algorithm. The Section 5 finally presents our conclusion.

2 JSON Patch background

2.1 JSON document and JSON patch

A JSON document is a very simple textual serialization of a JavaScript object. More precisely, it is a tree composed of three kinds of nodes (literal, array or object), where the root node cannot be a literal. A literal node can be either a boolean, a number or a string. An array node is a sequence of nodes. An object node has a set of child properties, each of them has a label[6] unique within the object, and a value that is a node. As an example, the Figure 1 presents two simple JSON documents that contain literals, objects and arrays.

```
{
  "isOk": true,
  "rm": "2",
  "val": 3,
  "mes1": {"who":"me", "exp":0},
  "res": [
    "v1",
    "v2",
    "v3",
    "v4",
    "v5"
  ],
  "inner" : {
    "elts" : ["a","b"] ,
    "sum" : "test is ok"
  }
}
```

```
{
  "rank": 6,
  "isOk": false,
  "va": 3,
  "mes1": {"who":"me", "exp":0},
  "mes2": {"who":"me", "exp":0},
  "res": [
      "v6",
      "v1",
      "m2",
      "v1",
      "v5",
      "v3"
  ],
  "inner" : {
    "in": {
      "elts" : ["a","b","c"]
    }
  },
  "sum" : "test is ok"
}
```

Fig. 1: A *source* (left) JSON document with several properties. A *target* (right) JSON document that has been transformed from the *source* JSON document.

The JSON Patch RFC is an ongoing standard that specifies how to encode a patch that can be performed on a JSON document to transform it into a new one[3]. The RFC specifies that a patch is a sequence of change operations. It then specifies the five following change operations (a sixth operation is defined to perform tests):

– Add: this operation is performed to add a new node into the JSON document. The new node can be added within an array or as a new property of an object.

[6] A string or a JavaScript name.

– Remove: this operation is performed to remove an existing node of the JSON document.
– Replace: this operation is performed to replace an existing node by another one.
– Move: this operation is performed to move an existing node elsewhere in the JSON document.
– Copy: this operation is performed to copy an existing node elsewhere in the JSON document.

The RFC specifies a standard way to encode a patch into a JSON document. More precisely a patch is an array of change operations where each change operation is encoded by a single object with properties specifying the kind of operation, the source and target nodes, and the new value if needed. For instance, the Figure 2 presents a patch that can be applied to *source* JSON document presented in the Figure 1, and that contains change operations (adding a new literal node *rank*, removing a node of the array *res*, etc.). We use that example in the following sections.

```
[
  { "op": "add",     "path": "/rank",  "value": 6 },
  { "op": "remove",  "path": "/rm"},
  { "op": "replace", "path": "/isOk",  "value": false},
  { "op": "move",    "path": "/va",    "from": "/val"},
  { "op": "copy",    "path": "/mes2",  "from": "/mes1"}
  { "op": "add",     "path": "/res/0", "value": "v6"},
  { "op": "replace", "path": "/res/2", "value": "m2"},
  { "op": "remove",  "path": "/res/4"},
  { "op": "copy",    "path": "/res/3", "from": "/result/1"},
  { "op": "move",    "path": "/res/5", "from": "/result/4"},
  { "op": "move",    "path": "/inner/in/elts",   "from": "/inner/elts"},
  { "op": "add",     "path": "/inner/in/elts/2", "value": "c"},
  { "op": "move",    "path": "/sum",   "from": "/inner/sum"}
]
```

Fig. 2: A RFC JSON Patch that, if applied to *source* JSON document of the Figure 1, would get the *target* JSON document.

Applying a patch to a JSON document is quite easy. It consists in applying all the editing operations of the patch in their defined order. Creating a patch that, given two versions of a JSON document, expresses how to transform the first version into the second one is however much more complex, especially when the goal is to create small patches and to create them as fast as possible.

2.2 Related works

JSON documents are mainly labelled unordered trees (object nodes and their properties), where some nodes are arrays, hence ordered. The theory states that when just the *add*, *remove* and *replace* operations are considered, the problem

of finding a minimal patch is $O(n^3)$ for ordered trees and NP-hard for unordered trees [16,2,14,10]. When the *move* operation is also considered, the problem is NP-hard for both kind of trees [2]. That is why several algorithms from the document engineering research field use practical heuristics. One of the most famous is the algorithm of Chawathe et al. [5] that computes patches (containing move actions) on trees representing LaTeX files. Several algorithms have also been designed specifically for XML documents [6,1]. One of them [13] is even capable of detecting copy operations.

Several existing approaches support the creation of JSON Patches.[7] By analyzing all of them, it appears that they all take one or two of these simplifications to make the problem tractable (see Table 1):

- They choose not to support the *move* and *copy* operations that are yet specified in the RFC, and therefore provide non-optimal patches. As an example in the Figure 2, an optimal patch uses *move* operation to handle the property label renaming from *val* to *va*. Without such a *move* operation, the patch then uses a *remove* property *val* and a *add* property *va*. Moreover, an optimal patch uses a *copy* operation for the property $mes2$ and its value copied from $mes1$. The Table 1 shows that only one existing approach does support these operations.
- They choose not to support array node, or to support them poorly. In principle all the editing operations of the JSON RFC apply to array nodes as well as object nodes. A patch can then express changes done within an array. For instance in Figure 2, an optimal patch uses the *move* operation to put $v3$ to the end of the array. Moreover, it uses the *copy* operation for copying the existing node $v1$. Regarding the support of array, the Table 1 shows that half of the approaches do not support array at all, and consider them as simple node (with nothing inside). The other half simply considers that an array is a stack, and therefore supports change operation that can apply to a stack (*push* and *pop*).

The table 1 clearly shows that there is no approach that fully complies with the RFC in terms of change operation coverage. By compliance we mean that it can handle all editing operations that are defined by the RFC including the move and copy ones (the test one is not an editing operation). However there is no formal process that truly checks the RFC compliance. There is only JSON test[8] that just checks if the given patches can be applied. As we describe it in the next section, our algorithm goes beyond and does support all the changes operations both on objects and on arrays, which is fully complies with JSON Patch RFC. However as there is no silver bullet, as a simplification it considers that changes made to JSON documents always target complete sub-trees rather than internal nodes, and therefore create patches that reflect this kind of change.

[7] http://jsonpatch.com/
[8] https://github.com/json-patch/json-patch-tests

Table 1: Comparison of existing approaches.

		Libraries	
Category	**Scenario**	*move* and *copy*	*Arraynode*
JavaScript	`jiff`	No	Stack
	`Fast-JSON-Patch`	No	No
	`JSON8 Patch`	No	No
	`rfc6902`	No	Stack
Python	`python-json-patch`	No	Stack
PHP	`json-patch-php`	No	No
Java	`json-patch`	Yes	No

3 JDR: a JSON patch algorithm

By running several APIs we observed that changes performed to JSON documents commonly target a complete sub-tree, but never target several internal nodes. More precisely, a change either adds, removes, replaces, moves or copies a complete sub-trees but never changes the topology of a sub-tree by inserting, removing, or moving some nodes inside the sub-tree. The same is true for arrays, changes made to arrays always target one array but they never target two or more different arrays. These observations have then driven the design of our algorithm that aims to identify large sub-trees or arrays, which are targets of changes.

Based on this consideration, our algorithm inputs two versions of a JSON document (the *old* and the *new* versions) and proceeds the three following steps:

- First it builds a large common sub-tree that is shared between the *old* and the *new* versions. This sub-tree contains the root node of both the *old* and *new* versions, and all the object and literal nodes that both exist in the *old* and *new* versions, in the same locations, with the same labels (values can be different). The array nodes are considered in the following steps. The center part of the Figure 3 presents the common sub-tree for our example. Once the common sub-tree has been created, for each of its label leaf node, if the value is not the same in the *old* and *new* version, a *replace* operation with the value of the *new* version is put into the patch . With our example, the *isOk* node corresponds to such a case.
- Second, for each object or literal node of the *old* version that does not belong to the common sub-tree but whose direct parent belongs to it, put a remove operation in the patch and mark the node as a removed one, unless there is a marked added node with the same value. In that case, put a move operation in the patch and mark the node as a moved one. The left part of the Figure 3 presents these nodes. The *rm* node is a removed node. The *val* and *sum* nodes are moved nodes. Symmetrically, for each node of the *new* version that does not belong to the common sub-tree but whose direct parent belongs to

it, put an add operation in the patch and mark the node as an added one, unless there is a marked removed node or a node in the common sub-tree with the same value. In case of a removed node, put a move operation in the patch and mark the node as a moved one. In case of a node in the common sub-tree, put a copy operation in the patch and mark the node as a copied node. The right part of the Figure 3 presents these nodes for our example. The *rank* and *in* nodes are added nodes. The *val* and *sum* nodes are moved nodes. The *mes*2 node is a copied node.

– Third, for each array node in the *old* version whose direct parent belongs to the common sub-tree, check if there is an array node in the *new* version, child of the same parent and with the same label. If so compare the two arrays (see the following array algorithm). If not, put a remove operation in the patch. For each array node in the *new* version whose direct parent belongs to the common sub-tree, put an add operation in the patch. The Figure 3 presents these nodes. The *res* nodes are then compared. The *elts* node is removed.

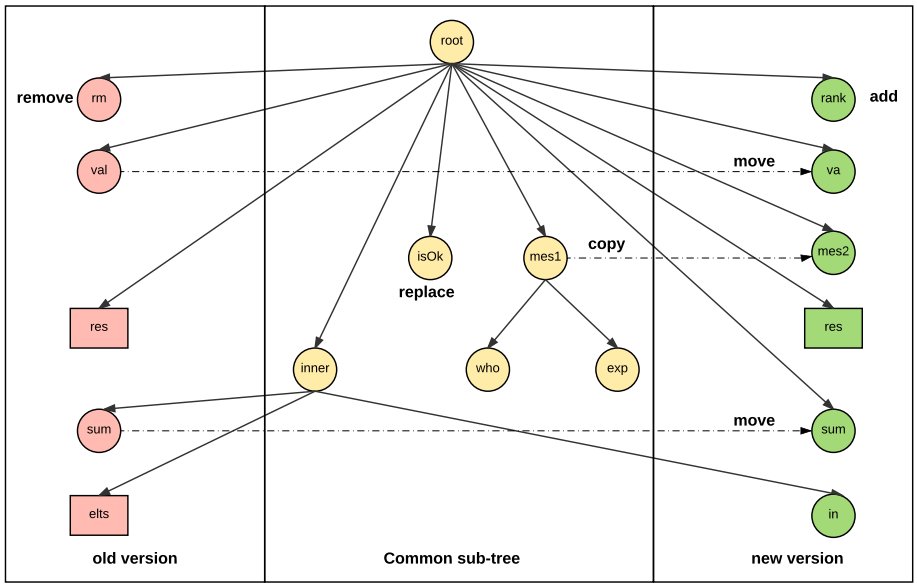


Fig. 3: The two versions of our example as a tree with object and label node presented with circles and array nodes with square. The central part represents the common sub-tree. The left part presents nodes direct children of the common tree and that belong to the *old* version. The right part presents nodes direct children of the common tree and that belong to the *new* version.

As just described, our algorithm only creates a patch for two versions of a same array if and only if the array is in the exact same location in the two versions of the JSON document. Further as the change operations defined by the RFC can only target cells one by one (it is not possible to remove or move several cells with one operation), there is then no need to compute a LCS (Longest Common Subsequence [11]). Comparing pairs of cells is therefore sufficient for creating a patch.

The creation of the patch is then done by comparing the cells of the array with the intent to identify the common ones, the ones that have been removed and the ones that have been added. More precisely, our algorithm first sorts the cells of the two versions of the array by computing a similarity hash [9] of their value. Secondly, thanks to the similarity hash order, it iterates through the cells in the two versions of the array and creates a temporary array patch by applying the following rules. If an *old* cell has a corresponding *new* cell (with the same value), a move operation is put into a temporary patch. If an *old* cell has no corresponding *new* cell, a remove operation is put into the patch. If a *new* cell has no corresponding *old* cell, an add operation is put into the patch. Thirdly, it transforms the temporary array patch into a final patch by taking care of the indexes of the changed cells because the execution of a change operation may have an impact on the indexes of the following ones. This transforming index method is inspired by the classical Operational Transformation (OT) technology, which aims to solve concurrency control of collaborative editing in distributed systems [8,12]. To that extent, it sorts the operations of the temporary patch according to the indexes of the changed cells and to the type of change ($move < remove < add$), iterates through them and recompute the indexes. Further, if a *move* operation moves a cell to the same operation (the target index is equal to the source index), it is removed from the patch. This step is not so complex and this why we do not explain it in details. The whole pseudo code of our algorithm JDR is available on GitHub[10].

The Figure 4 finally presents the patch created by our approach. The main difference with an optimal patch is that nodes that are not direct children of the common sub-tree are not target of any change. With our example, the sub-tree with the node *in* as a root is therefore fully created by the patch, and its child node *elts* is created from scratch whereas it should have been moved.

4 Efficiency evaluation

Our patch algorithm has been developed in JavaScript and is available as an Open Source library.[11] We present in this section its efficiency evaluation in comparison with all other existing JavaScript libraries that support the JSON Patch RFC (see Table 1).

[9] https://github.com/darkskyapp/string-hash

[10] https://github.com/caohanyang/json_diff_rfc6902/blob/master/Algorithm.pdf

[11] https://github.com/caohanyang/json_diff_rfc6902

```
[
  { "op": "add",     "path": "/rank",  "value": 6 },
  { "op": "remove",  "path": "/rm"},
  { "op": "replace", "path": "/isOk",  "value": false},
  { "op": "move",    "path": "/va",    "from": "/val"},
  { "op": "copy",    "path": "/mes2",  "from": "/mes1"}
  { "op": "add",     "path": "/res/0", "value": "v5"},
  { "op": "replace", "path": "/res/2", "value": "m2"},
  { "op": "remove",  "path": "/res/4"},
  { "op": "copy",    "path": "/res/3", "from": "/result/1"},
  { "op": "move",    "path": "/res/5", "from": "/result/4"},
  { "op": "remove",  "path": "/inner/in/elts"},
  { "op": "add",     "path": "/inner/in", "value": {"elts":["a", "b", "c"]}},
  { "op": "move",    "path": "/sum",   "from": "/inner/sum"}
]
```

Fig. 4: A RFC JSON Patch generated by our approach that, if applied to *source* JSON document of the Figure 1, would get the *target* JSON document.

Our evaluation consists in asking all the libraries to create JSON patches. We then compare them according to two quantitative factors: the time required to create the patch, and the size of the patch. Our claim is that a library is considered to be efficient if the patches it creates are small and if it creates them quickly.

Our evaluation is fully automated. It inputs a given REST service that provides access to a changing data, and periodically calls it 61 times to get 61 different versions of the changing data. Then, for each of the 60 consecutive versions it asks to all the existing libraries to generate the corresponding patch, and compares the time they take as well as the size of their returned patch. We repeat the generation of the patch 100 times to get an average value for both time and size. Our evaluation then returns 60 average values for both time and size for each library and for any given REST service. The evaluation has been executed on a desktop computer Intel Core i7-4770 CPU @3.40GHz8, 16GB of RAM, and Ubuntu 14.04.2 LTS x86 64.

The choice of the called REST service has obviously an impact on the results obtained by our evaluation. We therefore choose to include into our dataset only real services provided by well-known web applications. Further, as the existing libraries mainly differ by their support of changes (see Table 1), we choose to include into our dataset three kinds of services: the one where changes are only made to objects' properties, the one where changes are only made to arrays, and the one where changes are made to both. Our industrial partner Streamdata.io then provides us one service for each such kinds. Our dataset, available on GitHub[12], includes the *Xignite* GetRealTimeRate, *Stackoverflow* Answers and *Twitter* Timeline services.

[12] https://github.com/caohanyang/json_diff_rfc6902/tree/master/dataset

The Xignite GetRealTimeRate[13] service provides real-time currencies in the global financial market. The service returns a JSON document that contains one node object for each of the selected currencies (i.e. EURUSD, USDGBP). Changes between two successive versions are then only made to the properties of these objects. It should be note that a period of 15 seconds has been advised by our industrial partner between two consecutive versions.

The Stackoverflow Answers[14] service provides a list of Stackoverflow's answers . The service returns a JSON document that contains an array with the latest 20 answers. Changes between two successive versions are then only made to the array (new answers are added, last ones are deleted).

The Twitter Timeline[15] service provides the home timeline of a specific account with up-to-date Tweets. The service returns a collection of the most recent 20 Tweets of the authenticated user. Changes between two successive versions can be made to the array or to the objects themselves when tweets' properties change.

The Figure 5 shows an extract of successive versions that have been obtained by calling the services of our dataset. It clearly shows that changes performed to the data can be done either on objects' properties with the Xignite service, or on the array with Stackoverflow, or on both with Twitter.

The Figure 6, 7, 8 then present the results of our evaluation for each service. Each figure presents one figure for the time and one figure for the size where the black dots present the 60 average values, and the bold red dot presents the median of these average values.

For the Xignite service, Fast-JSON-Patch, JDR and rfc6902 always generate small patches while jiff doesn't (see Figure 6a). Curiously JSON8 chooses to simply replace the whole JSON document. Regarding time, Fast-JSON-Patch is the fastest followed by our library but the difference is no more than 0.5 milliseconds (see Figure 6b). rfc6902 takes much more time than the others.

For the Stackoverflow service it is interesting to see that Fast-JSON-Patch performs bad in term of size as it generates large patches (see Figure 7a). JSON8 is again quite bad as it generated also large patches. JDR, jiff, rfc6902 behave quite well regarding size as they always yield small patches. Regarding time all the libraries behave quite well but rfc6902, which is slower (see Figure 7b).

For the Twitter service, the Figure 8a clearly shows that JDR always yields small patches in all situation. In some cases, Fast-JSON-Patch totally fails (see some black dots with high patch sizes). rfc6902 succeeds almost all the times but is sometimes not that fast. Regarding time, the Figure 8b shows that JSON8 is definitively the fastest, then Fast-JSON-Patch. JDR and Jiff performs almost within the same time. Finally rfc6902 is slow.

[13] http://globalcurrencies.xignite.com/xGlobalCurrencies.json/GetRealTimeRate?Symbol=EURUSD,USDGBP,EURJPY,CHFDKK&_token=[YOUR_TOKEN]

[14] https://api.stackexchange.com/2.2/answers?order=desc&sort=activity&site=stackoverflow

[15] https://api.twitter.com/1.1/statuses/home_timeline.json

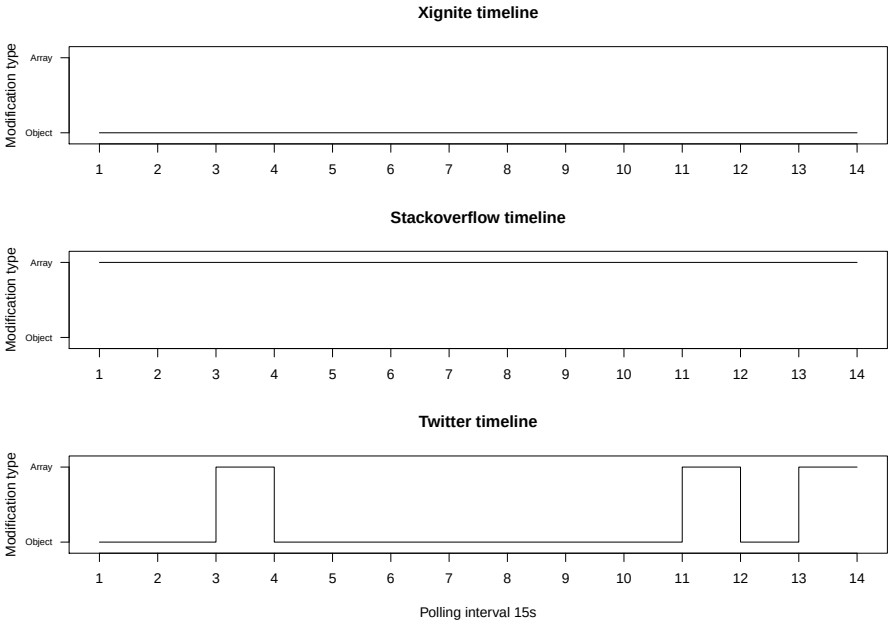


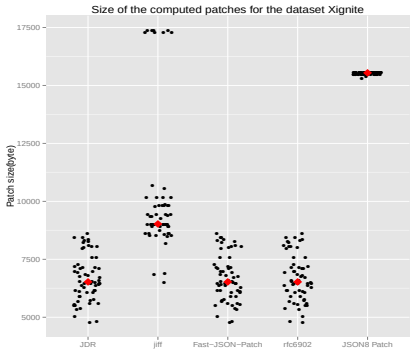
Fig. 5: Timeline modification type analysis for Xignite (top), Stackoverflow (middle) and Twitter (bottom), which represent *object server*, *array server* and *shift server* respectively.

The Figure 6, 7, 8 are consistent with the analyse we provided in the section 2, and clearly show the advantages and drawbacks of the existing libraries, depending on the support they provide to object or array. We then decided to combine the size and time factors considering that a patch has to be sent into the internet after it has been created (with a bandwidth of 10 Mbit/s) (See Tables 2, 3 and 4). The Table 2 shows that Fast-JSON-Patch is the best when the changes are only performed to the objects' properties but our library JDR is very close. Then, the Table 3 shows that our library JDR performs the best when the changes are only performed to arrays. Finally, the Table 4 shows that our library JDR performs the best when the changes are performed to both objects' properties and arrays.

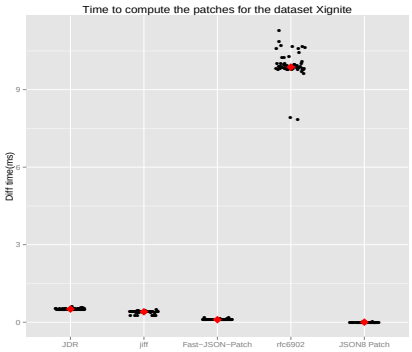
In conclusion, based on industrial real data, our JDR outperforms existing libraries regarding the size of created patches and the time needed to create them.

5 Conclusion

REST APIs together with JSON are commonly used by modern web applications to export their services. Such an architecture however makes the services

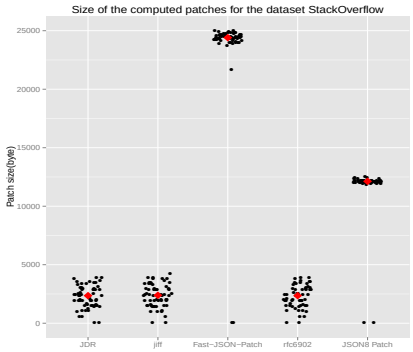


(a) Patch size

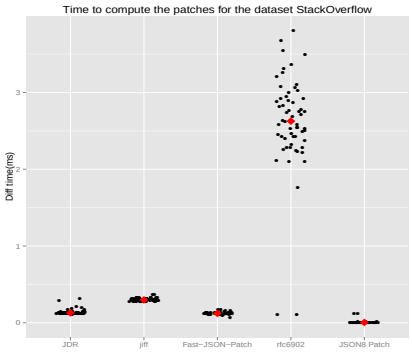


(b) Diff time

Fig. 6: Results for the Xignite dataset

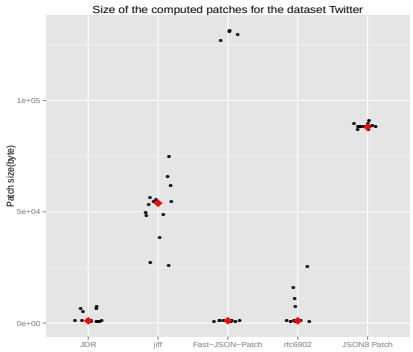


(a) Patch size

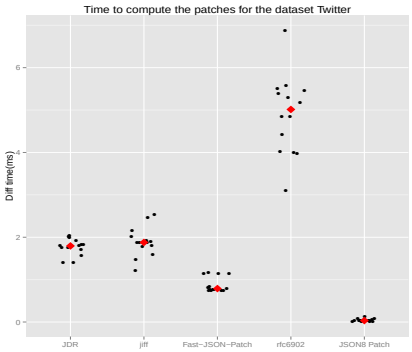


(b) Diff time

Fig. 7: Results for the StackOverflow dataset



(a) Patch size



(b) Diff time

Fig. 8: Results for the Twitter dataset

Table 2: Xignite performance of the 5 existing JavaScript libraries.

	Xignite		
Library	Patch-Size	Diff-time	**Total-time**
FastJSONPatch	100% (6683Bytes)	100% (0.103ms)	100% (5.45ms)
JDR	100%	502%	108%
jiff	152%	385%	157%
JSON8 Patch	232%	2%	228%
rfc6902	100%	2531%	281%

Table 3: Stackoverflow performance of the 5 existing JavaScript libraries.

	Stackoverflow		
Library	Patch-Size	Diff-time	**Total-time**
JDR	100% (2257Bytes)	100% (0.123ms)	100% (1.94ms)
jiff	104%	216%	112%
rfc6902	100.3%	1880%	228%
JSON8 Patch	232%	6%	484%
FastJSONPatch	1045%	88%	995%

reachable in a pull mode which is not suitable for accessing data that periodically changes (such as Twitter timeline, realtime currency, etc.). The push mode is on the contrary more adequate for accessing changing data, but very few web applications, if any, support it. Our partner StreamData.IO therefore provides a proxy server solution for turning a pull REST API into a push one. The proxy server makes periodical requests to the API and then generates patches that express the changes made to the new received versions of the data. Generating a patch for JSON document is obviously the difficult part and existing approaches handle it poorly. In this paper we then provide a new JSON patch algorithm towards this issue, with the objective to fully support the JSON patch RFC and to provide efficiency gain in comparison to existing libraries.

Our study first shows that the existing approaches are not optimal and that they take drastic simplifications. More precisely, we show that existing approaches do not support the *move* and *copy* change operations (except Java JSON-patch), and that few of them fully support changes performed to array.

We then propose our JSON patch algorithm that is compliant witht the JSON Patch RFC and that further supports all of the 5 change operations. Indeed, our algorithm succeeds to support *move* and *copy* operations for object nodes and for arrays. Its limitation is that it only considers changes that are performed on a whole sub-tree, and does not consider changes that modifies the structure of a sub-tree. Further, it only considers change to array that are localized in the same place in the two versions of a JSON document. Those limitations have however been driven by our observations performed on existing REST API, which showed that such changes almost never happen. Our approach

Table 4: Twitter performance of the 5 existing JavaScript libraries.

	Twitter		
Library	Patch-Size	Diff-time	**Total-time**
JDR	100% (2475Bytes)	100% (1.77ms)	100% (3.75ms)
rfc6902	198%	276%	235%
FastJSONPatch	1525%	50%	827%
jiff	2064%	107%	1140%
JSON8 Patch	3575%	3%	1887%

only handles the transformation of pull mode services into push mode but not their updates. The future work is to study the subsequent API updates that may involve structural changes, which aims to better understand how far APIs are updated.

We evaluate the efficiency of the JavaScript implementation of our algorithm against existing JavaScript libraries that support the JSON Patch RFC. The evaluation has been done by requesting real web applications with data suggested by our industrial partner. It clearly shows that our library outperforms the other libraries. It creates small patch quite fast, and can handle different situations (where the changes target objects' properties or arrays).

As a main conclusion, we provide an efficient algorithm to create a path between two versions of a JSON document. The patch created by our approach is fully complies with RFC. Even it is not optimal, it however expresses all change operations such as the *move* and *copy* ones, and the ones that target arrays. Our work is the most essential part for turning a pull REST API into a push one, which is frequently requested by the web developers to get notified of data changes. As an example we provide a prototype framework that can be used to convert a pull service into a push one (see the online demo[16]).

References

1. Al-Ekram, R., Adma, A., Baysal, O.: diffX: An Algorithm to detect Changes in Multi Version XML Documents. In: In processing of the CASCON '05. pp. 1–11 (2005)
2. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. 337(1-3), 217–239 (2005)
3. Bryan, P., Nottingham, M.: JavaScript Object Notation (JSON) Patch. Tech. rep., RFC 6902, April (2013), http://www.hjp.at/doc/rfc/rfc6902.html
4. Buttler, D.: A short survey of document structure similarity algorithms. In: International conference on internet computing. pp. 3–9 (2004)
5. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change Detection in Hierarchically Structured Information. In: Jagadish, H.V., Mumick, I.S. (eds.) Proceedings of the 1996 ACM SIGMOD International Conference on Management

[16] http://diff-and-patch.pubstorm.site/

of Data, Montreal, Quebec, Canada, June 4-6, 1996. pp. 493–504. ACM Press (1996)
6. Cobena, G., Abiteboul, S., Marian, A.: Detecting Changes in XML Documents. In: Agrawal, R., Dittrich, K.R. (eds.) Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002. pp. 41–52. IEEE Computer Society (2002)
7. Crockford, D.: RFC4627: JavaScript Object Notation (2006)
8. Ellis, C.A., Gibbs, S.J.: Concurrency control in groupware systems. In: Acm Sigmod Record. vol. 18, pp. 399–407. ACM (1989)
9. Fielding, R.T., Taylor, R.N.: Principled design of the modern Web architecture. ACM Transactions on Internet Technology (TOIT) 2(2), 115–150 (2002), http://dl.acm.org/citation.cfm?id=514185
10. Higuchi, S., Kan, T., Yamamoto, Y., Hirata, K.: An A* algorithm for computing edit distance between rooted labeled unordered trees. In: New Frontiers in Artificial Intelligence, pp. 186–196. Springer (2012)
11. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. Journal of the ACM (JACM) 24(4), 664–675 (1977)
12. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Communications of the ACM 21(7), 558–565 (1978)
13. Lindholm, T., Kangasharju, J., Tarkoma, S.: Fast and Simple XML Tree Differencing by Sequence Alignment. In: Proceedings of the 2006 ACM Symposium on Document Engineering. pp. 75–84. DocEng '06, ACM, New York, NY, USA (2006), http://doi.acm.org/10.1145/1166160.1166183
14. Pawlik, M., Augsten, N.: RTED: A Robust Algorithm for the Tree Edit Distance. PVLDB 5(4), 334–345 (2011)
15. Zhang, K., Shasha, D.: Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. SIAM J. Comput. 18(6), 1245–1262 (Dec 1989), http://dx.doi.org/10.1137/0218082
16. Zhang, K., Statman, R., Shasha, D.: On the editing distance between unordered labeled trees. Information processing letters 42(3), 133–139 (1992)