Shredding JSON Data into Relational Environment

Dušan Petković
Technical University of Applied Sciences
Rosenheim, 83924
Germany

Ali Piriyae
Technical University of Applied Sciences
Rosenheim, 83024
Germany

ABSTRACT
The use of JSON (JavaScript Object Notation) as a data format has expanded significantly in the last couple of years. The reason for this expansion is that JSON can represent structured as well as semi-structured data in a simple way. Storing JSON documents into relational tables is an obvious next step, because that way features of RDBMSs, such as transaction processing and recovery mechanisms, can be used. In this paper we compare representatives of two groups of mapping techniques: the family of Argo algorithms and the family of XML-to-Relational storage algorithms, which can be used to store JSON documents, too. Our results show that the former outperforms the latter in relation to time efficiency, while the XML-to-Relational storage algorithm needs less disk memory to load JSON data.

General Terms
Relational database systems

Keywords
JSON, RDBMSs, Argo, STDM algorithm, space efficiency, time efficiency

1. INTRODUCTION
JSON (JavaScript Object Notation) is a simple data format used for data interchange. The structure of JSON content follows the syntax structure for JavaScript. The example below shows a JSON document called info that describes a single person, Fred, his affiliation, BBC, and his friends, Lili and Hank.

```
{ "info": { "who": "Fred", "where": "BBC", "friends": [{ "name": "Lili", "rank": 5 }, { "name": "Hank", "rank": 7 }] }
```

Generally, a JSON string contains either an array of values or an object, which is a list of name/value pairs. An array is surrounded by a pair of square brackets and comprises a list of values, separated by commas. An object is surrounded by a pair of curly brackets and contains a comma-separated group of name/value pairs. Each name/value pair has a field name, followed by a colon (:), followed by the value of the corresponding field. Additionally, each array can contain other objects or arrays. The same is true for objects.

The foundation of relational database systems (RDBMSs) is based upon the relational data model. This model requires that the structure of the data must be known before the same data is loaded and queried. Recently, there have been more and more data that do not fit into the relational model. For instance, there are a lot of semi-structured data e.g. data presented in a hierarchical form whose representation in the relational form is not easy.

The first data exchange format that suited well to present hierarchies of data has been XML. Soon, after its introduction, the XML data format has evolved as a standard data model for semi-structured data. One of properties of XML that is also valid for the relational model is the “schema first, data later” approach. This approach is guaranteed for XML data through the existence of XML Schema. In the course of time, SQL/XML [1] has been introduced to allow standardized approach to semi-structured data within relational database systems.

On the other hand, XML has its origins in the document processing, and is therefore a complex data format. This complexity causes that XML file sizes are usually very large and difficult to read. Hence, XML is not the best data format for representation of semi-structured data.

In the meantime, another data format called JSON has become popular, because it can represent data in a simple way and is easy to read. Native support of JSON in a language such as JavaScript led to its increasing acceptance for representation of semi-structured data. Today, many applications that would benefit of using JSON documents will also need to access data stored into relational tables. For this reason, it is of extreme value to store and manipulate JSON data inside relational database systems, as well as to provide the way to present JSON documents as relational data and vice versa.

In contrast to relational and XML data, JSON supports the “data first, schema later” paradigm, meaning that each JSON document has flexible schema with variety of attributes. In other words, there is no standardized schema language for JSON data.

In this paper we discuss two different techniques for storing JSON data into a relational environment. We implement algorithms for each storage technique and worked them out. We create two different PostgreSQL databases based on the implemented mapping techniques and compare their performance in relation to the time needed to store a complex JSON document as well as the size of storage needed to load the document into relational table(s).

1.1 Related Work
There are several papers whose goal is to discuss mapping of JSON documents into relational environment. In one of the first papers of this art, Agraval et al [2] discuss how data, which constantly evolve and are sparsely populated can be stored and queried. (These two properties of data, constantly evolving and sparsely populated, are the main properties of JSON documents.)

Chaussier et al [3] implemented an automated mapping layer called Argo, for storing and querying JSON documents into a relational environment. One of Argo’s advantage is the ability to represent nested objects in a schema. The main concept is to store data in one table with object ID, keys, and three
columns for JSON standard data types (string, number and Boolean), or in three tables with separate types. The Argo system was implemented in two relational databases (PostgreSQL and MySQL) and compared with MongoDB using the NoBench benchmark specifically designed for this purpose.

Another system, called Sinew [4], which is a combination of a storage system and a mapping layer, stores JSON documents inside real and virtual columns of a relational table and adds a layer above the database system that automatically provides a dynamic relational view to the user, against which the standardized SQL queries can be applied.

In [5], the authors discuss natively storing JSON data in a commercial database and querying them using extensions of the standard SQL. Their next paper [6] discusses how the functional and performance gap between SQL and NoSQL can be closed using indices. The latter also introduces OSON – a new query-friendly JSON format. The ideas from both papers have been implemented in the Oracle Database.

The well-known Entity Attribute Value (EAV) data model has been used in [7] to discuss how the support for JSON documents in two open-source RDBMSs and two commercial ones looks like. The results of this paper suggest that the use of JSON data simplifies queries and reduce their execution time in relation to the other approaches.

In [8], the authors compare two mapping algorithms, namely Single-Table Data Mapping (STDM) and Multi-Table Data Mapping (MTDM), which are inspired from the XML-to-Relational techniques called the Edge and Binary approach [9], respectively. The STDM algorithm maps all the JSON elements and their parent-child relationships into a big single relational table. The MTDM algorithm horizontally partitions the one big table into multiple tables based on the types or names of unique JSON elements. The paper demonstrates that STDM shows a better performance in data shredding and loading time, while the MTDM algorithm outperforms STDM in space efficiency.

In the meantime, the SQL standardization committee published a proposal called SQL/JSON [10]. This proposal specifies, among other things, that JSON data should be stored into relational tables as character or binary strings. In other words, the analog to the XML native data type that is specified in the SQL/XML standard, does not exist in the SQL/JSON standard. The advantages and deficiencies of such an approach is given in [11]. The implementation of JSON in enterprise database systems has been discussed in [12].

2. MAPPING TECHNIQUES

Although JSON documents can be theoretically stored in object-oriented databases (similarly the storage of XML documents as described in [2]), they are usually mapped to relational tables and stored in RDBMSs. This section discusses the properties of the two algorithms, which will be compared in this paper in relation to their time and space efficiency.

2.1 Argo Mapping Layer

The Entity Attribute Value (EAV) data model is often used as a target for systems that apply shredding on semi-structured data to store them in a relational DBMS. Therefore, this data model is applicable to JSON documents, too. Under this model, each object is transformed into groups of key/value pairs, with the object ID added to each such pair. That way, a series of triples (object ID, key, and value) is built.

The direct relationship between the EAV data model and such a triple can be established, when the object ID of the triple is viewed as an entity, and the key as an attribute. It is therefore straightforward to store semi-structured data in a relational database using the EAV model. By adding a mapping layer on top of a relational DBMS, we can translate queries over specific attributes into queries over the underlying table(s).

In [3], a family of the mapping layers called Argo that is based upon the EAV model, is introduced to store JSON documents into relational tables. Argo comes in two flavors: Argo/1 and Argo/3. The former was designed as a simple storage system, which, in spite of its simplicity, gives a comprehensive solution to store JSON data in relation environment. This system uses a single relational table with three columns to store a unique object ID, a key and a value. Rows are stored in the corresponding table only when data exists for a given key.

In Argo/1, JSON complex data types, such as objects and arrays, are flattened. In other words, the keys of an object are appended to the parent key. In the same way, each array element is handled by appending the element’s position in the array to the key.

The Argo/3 mapping technique generates three relational tables (one for each primitive data type) to store a single JSON collection. Each table contains the standard objID and keystr columns, and an additional column whose data type corresponds to the primitive data type (text for alphanumerical values, numeric for number, and boolean for Boolean).

For our experiments, we choose Argo/3 as a representative of the Argo family, because Argo/3 generally offers performance superior to Argo/1. When Argo/3 is run on MySQL, it outperforms even MongoDB in relation to time efficiency.

2.2 Single-Table Data Mapping (STDM)

In relation to the algorithms used in [8], we choose the STDM algorithm for our experiments for two reasons: First, the STDM algorithm outperforms the MTDM algorithm in relation to time efficiency. Even more important for the choice of the STDM algorithm is that the MTDM algorithm has a substantial drawback in relation to data modification: Each time a new JSON data are processed, the MTDM algorithm creates a new relational schema.

The STDM algorithm relies on the JSON tree structure derived from a JSON document. While all nodes in the tree represent JSON elements, all edges represent parent-child relationships between the nodes. The individual array elements are represented by separate nodes and differentiated by each other with indexes denoted by brackets. All array elements are connected to their container element as their parent. The nodes in the JSON tree are numbered in depth-first traversal order to capture and preserve the order of an arbitrary JSON element including array elements. The unique ID numbers are indicated as superscripts of the node labels. Preserving the order also utilizes the reconstruction process, which is more or less straightforward.
Table 1a. The Argo/3 output (table with valstr)

<table>
<thead>
<tr>
<th>objid</th>
<th>keystr</th>
<th>Valstr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>City</td>
<td>AGAWAM</td>
</tr>
<tr>
<td>1</td>
<td>State</td>
<td>MA</td>
</tr>
<tr>
<td>1</td>
<td>_id</td>
<td>01001</td>
</tr>
<tr>
<td>2</td>
<td>City</td>
<td>CUSHMAN</td>
</tr>
<tr>
<td>2</td>
<td>State</td>
<td>MA</td>
</tr>
<tr>
<td>2</td>
<td>_id</td>
<td>01002</td>
</tr>
</tbody>
</table>

Table 1b. The Argo/3 output (table with valnum)

<table>
<thead>
<tr>
<th>objid</th>
<th>keystr</th>
<th>valnum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pop</td>
<td>15338</td>
</tr>
<tr>
<td>1</td>
<td>loc[0]</td>
<td>-72,622739</td>
</tr>
<tr>
<td>1</td>
<td>loc[1]</td>
<td>42,070206</td>
</tr>
<tr>
<td>2</td>
<td>Pop</td>
<td>36963</td>
</tr>
<tr>
<td>2</td>
<td>loc[0]</td>
<td>-72,51565</td>
</tr>
<tr>
<td>2</td>
<td>loc[1]</td>
<td>42,377017</td>
</tr>
</tbody>
</table>

2.3 The STDM Algorithm vs. Argo/3

The main difference between Argo/3 and STDM is that the former stores the corresponding relational data in three tables, depending on the primitive data type. On the other hand, the STDM algorithm stores all JSON data into one table. The generated table contains a separate column for each data type. Additionally, the STDM algorithm, in contrast to Argo/3, does not differentiate between the String and Boolean data types.

The STDM output is the result of the mapping process with the STDM algorithm. The id column has the unique values for all elements of a single JSON document, starting with 0. The parent_id column has the non-zero value if the corresponding child element is an array element, because all array elements are connected to their container element as their parent. The name and text columns contain the name and the value of the name/value pair, respectively. (If the element is not name/value pair, the null value is assigned to both columns.) The last column, numeric, contains the corresponding values, if the element is of the numeric type. (If the element is not numeric, the null value is assigned to this column.)

3. EXPERIMENTAL STUDY

We implemented the STDM algorithm using Python Version 2.7. (For the Argo/3 algorithm, we used the Python implementation of this algorithm, which can be downloaded from the personal web site of the author at University of Wisconsin.) We ran the implemented algorithms on a MS Windows 10 computer with Intel Core i5-6200U and CPU with 2.40 GHz clock speed, as well as with 8GB RAM.

To test time and space efficiency of the two mapping...
techniques, we use the JSON documents containing the US zip codes, which has been used by Bahta and Atay [8] for benchmarking. The zip code data set contains ca. 30,000 US zip codes [13]. The structure of the JSON sample data includes the name of the city ("city"), its population ("pop") and location ("loc"), as well as the name of the state to which the city belongs ("state").

After that, the JSON data set has been used with the Argo/3 and the STDM algorithms to generate relational tables in two different PostgreSQL databases using PostgreSQL Version 12. We then compared performance for loading tables as well as the space necessary for these tables. To get more accurate results, we repeated the measurement of time and space efficiency several times, each time increasing the volume of the zip code data.

The chart given in Figure 1 shows the load time comparison for our sample data set for the Argo/3 and STDM algorithms. The chart given in Figure 2 shows the space comparison for the same algorithms.

As can be seen from Figure 1, the Argo mapping technique outperforms the STDM algorithm in relation to the time needed to store a complex JSON document into a relational environment. This is expectable, because, according to [3], the Argo system can achieve an order of magnitude improvement in performance over existing relational database JSON extensions as well as MongoDB.

On the other hand, Figure 2 shows that STDM needs less memory storage for the created relational table(s) than Argo/3. This can be explained with the number of tables needed to store JSON data: The STDM technique uses one monolithic table, while Argo/3 needs altogether three tables for the same process.

4. SUMMARY AND FUTURE WORK
In this paper, we implemented algorithms for two storage techniques: the Argo algorithm and XML-to-Relational storage algorithm, which can be generally applied to all hierarchical data structures. Our experimental result show that the Argo/3 mapping technique outperforms the STDM algorithm in relation to the time needed to store a complex JSON document into a relational environment, while STDM needs less memory storage for the created relational table(s).

In our future work, we intend to examine mapping of JSON documents into environments, other than relational one.
Particularly, the object-relational (OR) environment is interesting. Therefore, we intend to examine the benefits of storing JSON data in an object-relational environment. Two OO features: abstract data types (ADT) and arrays allows a native storage of JSON objects and arrays, respectively. Our goal will be to examine advantages of use of the OO features mentioned for storing JSON data in relation to using pure relational environment.

5. REFERENCES


