INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING & TECHNOLOGY (IJCET)

ISSN 0976 – 6367(Print)
ISSN 0976 – 6375(Online)
Volume 5, Issue 12, December (2014), pp. 148-153
© IAEME: www.iaeme.com/IJCET.asp
Journal Impact Factor (2014): 8.5328 (Calculated by GISI)
www.jifactor.com

IJCET

©IAEME

SPACE EFFICIENT STRUCTURES FOR JSON DOCUMENTS

Rincy T. A.

(Research Scholar, Bharathiar University, Coimbatore).
Dept. of Computer Science, Prajyoti Niketan College, Pudukad, Kerala, India

Dr. Rajesh R

Professor, Dept. in Computer Applications,
Sree Narayana Gurukulam College of Engg., Kadayiruppu, Ernakulam, Kerala, India

ABSTRACT

With the rapid increase of JSON documents on the web, methods to index, store and retrieve these documents has become a very significant problem. Implementations that load JSON documents and give access to them, suffer from huge memory demands. The in-memory representation of JSON documents is larger than its file size. This is a problem for machines with limited memory such as mobile devices, where processing even moderately-sized JSON documents requires more memory than is available. . Both JSON data and the existing queries possess an inherent tree structure and thus fast child-parent lookup is a necessity to improve performance. We focus on in-memory representations of JSON documents for situations where space is limited and where rapid processing time is important. With this paper, we hope to spark a discussion on the application of succinct data structures that supports operations on the document tree at speed comparable with an in-memory deserialized object, thus bridging textual formats with binary formats.

Keywords: JSON Documents, Ordinal Trees, Semi Structured Data, Succinct Data Structures.

I. INTRODUCTION

Semi-structured data is increasingly occurring since the advent of the Internet where full-text documents and databases are not the only forms of data anymore and different applications need a medium for exchanging information. With the advent of large-scale distributed processing systems such as MapReduce and Hadoop[1] it has become very frequent to store large amounts of data in textual semi-structured formats such as JSON and XML, as opposed to the structured databases of classical data warehousing.

The use of JSON documents is attractive and abundant owing to its flexible data representation, simplicity and scalability. It is a representative of the emergence of so-called key-value store systems that give up many features of the relational model, including schema, structured querying and consistency guarantees.

A number of applications that involve indexing and processing textual or semi structured data now need to deal with increasingly large volumes of data. Typically, these applications do not have satisfactory external-memory solutions and so the data has to be held and processed in main memory; examples include the various applications of suffix trees and a number of XML processing tasks (including XQuery search, XSLT transformation)[2].

To execute a JSON query, the entire specified document is assumed to fit within the main memory. Very large documents may not fit in the memory available at run time to the query processor. The documents are stored on the disk under some form of persistent data structures. The focus of this paper is on the proposed in-memory storage format of semi structured documents - ordinal trees [3, 4], which are arbitrary rooted trees where the children of each node are ordered.

A significant problem in the applications that involve processing, is the space required to represent the tree structured object: in such applications, succinct [5, 6], or highly space-efficient, representations of trees are having increasing impact. We consider succinct, or space-efficient, representations of ordinal trees. Representations exist that take 2n+O(n) bits to represent a static n-node ordinal tree close to the information-theoretic minimum and support navigational operations in O(1) time on a RAM model[7]. In the context of JSON documents, we study succinct representations for ordinal trees.

As an example, we consider the following JSON document.

```
{
        "category":"node",
        "tests": [{"name":"o", "score":97},
                        {"name":"t", "score":93}
                ]
}
```

Paper Organization: In section 2, we review the tools and notations used in this paper. In section 3, the succinct representations of the JSON documents and the supporting operations are mentioned. We evaluate the different representations in section 4 and give some suggestions.

II. PRELIMINARIES

2.1 JSON

Relational database systems are facing new competition from various NoSQL ("Not Only SQL") systems. These systems [MongoDB [8] and CouchDB [9] are appealing to Web 2.0 and mobile application programmers since they generally support JSON (JavaScript Object Notation) as their data model. This data model fits naturally into the type systems of many programming languages, avoiding the problem of object-relational impedance mismatch. JSON data is also highly-flexible and self-describing, and JSON document stores allow users to work with data immediately without defining a schema upfront. This "no-schema" nature eliminates much of the hassle of schema design, enables easy evolution of data formats, and facilitates quick integration of data from different sources. JSON is now a dominant standard for data exchange among web services (such as the public APIs for Twitter, Facebook, and many Google services), adding to the appeal of JSON support [10].

In light of these advantages, many popular Web applications (e.g. Craigslist, Foursquare, and bit.ly), content management systems (e.g. LexisNexis and Forbes), big media (e.g. BBC), and scientific applications (e.g. at the Large Hadron Collider) today are powered by document store NoSQL systems like MongoDB and CouchDB [10].

JSON or JavaScript Object Notation is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. Although originally derived from the JavaScript scripting language, JSON is a language-independent data format, and code for parsing and generating JSON data is readily available in a large variety of programming languages. JSON documents are particularly convenient when persistent data must be tightly integrated in a programming environment because objects can be instantiated from the JSON serialization with minimal programming effort [11].

The JSON data model consists of four primitive types, and two structured types. The four primitive types are:

- Unicode, Strings wrapped in quote characters.
- Numbers, which are double-precision IEEE floats.
- Boolean values, which are true or false.
- Null, to denote a null value.
- The two structured types are:
- Objects, which are collections of attributes. Each attribute is a key (String) and value (any type) pair.
- Arrays, which are ordered lists of values. Values in an array are not required to have the same type.

A value in an object or array can be either a primitive type or a structured type. Thus, JSON allows arbitrary nesting of arrays and objects. A document is also an object.
The following code shows an example of a JSON document.

```
{
        "category":"node",
        "tests":[ {"name":"o", "score":97},
                        {"name":"t", "score":93}
                ]
}
```

Here, data is self described, encoded independently from any application or system, and the representation supports simple but powerful constructs that allow to build arbitrarily complex structures. In this example, the JSON object has an attribute tests, which is an array consisting of two values (it is perfectly legal to mix value types in an array) and they represent objects. These objects define its own mapping of keys to values (recall that JSON can nest objects and arrays arbitrarily deep within each other).

2.2 JSON vs. XML

JSON is similar to XML in that both are hierarchical semi-structured data models. In fact, JSON is replacing XML in some applications due to its relative simplicity, compactness, and the ability to directly map JSON data to the native data types of popular programming languages (e.g. JavaScript). There is a rich body of research on supporting XML data using succinct structures of ordinal trees. We are inspired by this research, and we have adapted some techniques originally proposed for dealing with XML to the JSON data model.

2.3 SUCCINCT DATA STRUCTURES (SDS)

In computer science, a succinct data structure is a data structure which uses an amount of space that is "close" to the information-theoretic lower bound, but (unlike other compressed representations) still allows for efficient query operations. The concept was originally introduced by Jacobson [6] to encode bit vectors, (unlabeled) trees, and planar graphs. Unlike general lossless data compression algorithms, succinct data structures retain the ability to use them in-place, without decompressing them first. A related notion is that of a compressed data structure, in which the size of the data structure depends upon the particular data being represented.

Trees are one of the most fundamental data structures, needless to say. A classical representation of a tree with n nodes uses $O(n)$ pointers or words. Because each pointer must distinguish all the nodes, it requires log n bits in the worst case. Therefore the tree occupies $\Theta(n \log n)$ bits[5], which causes a space problem for manipulating large trees. Much research has been devoted to reducing the space to represent static trees and dynamic trees, achieving so-called succinct data structures for trees.

A succinct data structure stores objects using space close to the information-theoretic lower bound, while simultaneously supporting a number of primitive operations on the objects in constant time. The information theoretic lower bound for storing an object from a universe with cardinality L is $\log L$ bits because in the worst case this number of bits is necessary to distinguish any two objects. The size of the corresponding succinct data structure is typically $(1 + o(1)) \log L$ bits[4].

The information-theoretic lower bound to store an ordinal tree with n nodes is $2n - \Theta(\log n)$ bits. We assume that the computation model is the word RAM with word length $\Theta(\log n)$ in which arithmetic and logical operations on $\Theta(\log n)$-bit integers and $\Theta(\log n)$-bit memory accesses can be done in constant time. Under this model, there exist many succinct representations of ordinal trees achieving $2n+o(n)$ bits of space[4].

Basically there exist many types of such tree representations like the balanced parentheses sequence(BP)[12], the level-order unary degree sequence (LOUDS)[13], the depth-first unary degree sequence (DFUDS) and the partition representation.

III. SDSS FOR JSON DOCUMENTS

In this paper, we carefully implement the theoretical proposals we consider most promising in practice, and compare them for JSON trees and typical tree traversals. If the space issue is a primary concern, we have to renounce to pointer-based tree representation and resort the notion of succinct data structures. Succinct tree representations can be broadly classified into the following tracks:

BP: The balanced parentheses representation of a tree, first advocated as a succinct data structure by Jacobson, and later achieving constant times, is built from a depth-first preorder traversal of the tree, writing an opening parenthesis when arriving to a node for the first time, and a closing parenthesis when going up (after traversing the subtree of the node)[6]. In this way, we get a sequence of 2n balanced parentheses. Each node is represented by a matching pair parentheses ’(’

and ')', and we identify the node with its opening parenthesis. The subtree of x contains those nodes (parentheses) enclosed between its representing opening and closing parentheses. The core operations on this sequence are sufficient to implement most of the functionality [14].

The JSON documents are stored on the disk under some form of persistent data structures. Here, we focus on the in-memory storage format of the semi-structured documents - ordinal trees.

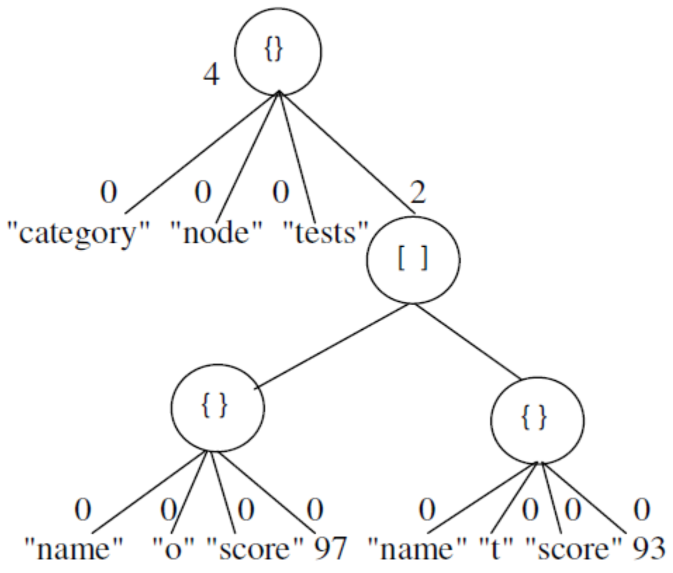

Fig. 1

The ordinal tree representation (with the degree of each node) of the above example is given in Figure 1.

The Balanced Parenthesis representation of this ordinal tree is (()()()((()()()())(()()()()))). Here, we have 15 nodes and let it be n, then the BP bit vector can be represented with 2n bits (where '(' is 1 and ')' is 0) . And one can easily reconstruct the tree from this sequence. The BP representation supports many operations like parent, ith child, rank, degree, select, LCA, preorder, postorder etc..

DFUDS: Depth-first unary degree sequence. It is built by the same traversal, except that, upon arriving at each node, we append *i* opening and one closing parentheses, being *i* the number of children of the node. The node is represented by the position where its *i* parentheses start. The resulting sequence turns out to be balanced and the same core operations on parentheses are used to support the same functionality of BP in a different way (except for depth, which requires extra structures) [4] plus others, most notably child, in constant time.

For the above ordinal tree, first write the degree sequence in depth-first order.

4 0 0 0 2 4 0 0 0 0 4 0 0 0 0

In unary,

11110000110111100000111100000

It takes only 2n-1 bits and the representation of a subtree is together. It supports many core operations like BP.

LOUDS: Level-ordered unary degree sequence. Nodes are processed and represented as in DFUDS, but they are traversed level-wise, instead of in depth-first order [6, 15]. It turns out that just rank and select are sufficient to support a few key operations such as parent and child in constant time, yet most other operations are not supported.

For the above ordinal tree, first write the degree sequence in level-wise order.

4 0 0 0 2 4 4 0 0 0 0 0 0 0 0

But this requires n lg n bits. To reduce the number of bits, write them in unary,

11110000110111101111000000000

It takes only 2n-1 bits.

To support more operations, add a dummy root as in Figure 2 so that each node take the corresponding one. Here, the level-order representations of each node is given and it supports the operations parent (number of zeros upto the kth 1), i-th child and degree(using rank and select).

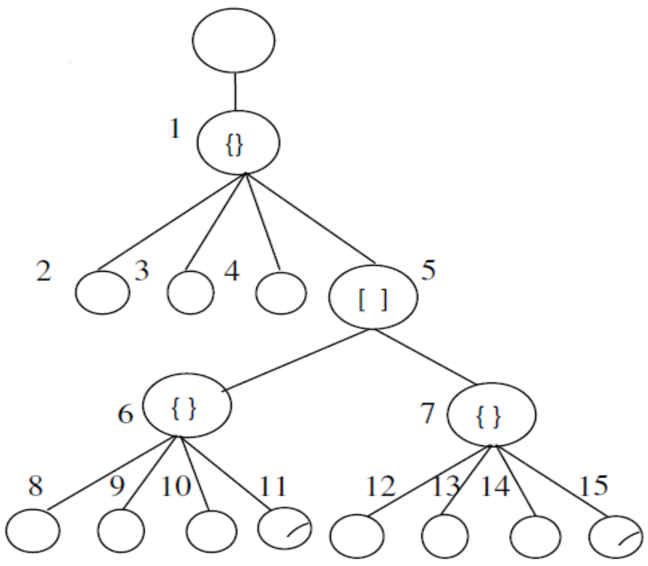

Fig. 2:

So the bit vector is

```
1 0 1 1 1 1 0 0 0 0 1 1 0 1 1  1   1   1   1   1   1
1    2 3 4 5             6 7    8 9 10 11 12 13 14 15
```

Here, node k corresponds to the kth 1 in the bit sequence. And it takes 2n+O(n) bits.

LOUDS, had received comparatively little attention, is an extremely simple and efficient alternative when only the simpler operations are needed and it lacks support for some sophisticated operations.

IV. EVALUATION

Table 1 summarizes the operations that are supported by the succinct data structures of JSON documents. LOUDS is easy to implement and is fast in practice. Some operations like child() performs exceptionally good because of the locality of reference. But many operations like subtree size, level ancestor and LCA is not supported in O(1) time. At the same time BP an DFUDS supports more operations like subtree size, LCA, level ancestors etc. In the BP and DFUDS, the representation of a subtree is together. As far as the structural pattern of the JSON documents are concerned, these two forms have significant advantage over the LOUDS representation.

Table 1: Operations supported by the succinct data structures. In the static case all operations are performed in constant time

Representation	Supporting Operations
LOUDS	Parent, first child, sibling, ith child, child rank, degree, next node in the level, level order rank, select
DFUDS	Parent, first child, sibling, ith child, child rank, subtree size, degree, depth, LCA, level ancestor, pre order rank, select, post order rank, select, DFUDS order rank, select, leaf operations
PARENTHESIS	Parent, first child, sibling, ith child, child rank, sub tree size, degree, depth, LCA, next node in the level, level ancestor, pre order rank, select, post order rank, select, leaf operations, height.

We can see an interesting property that enables the DFUDS and BP bit patterns to be used as an index to the JSON trees. Suppose, in the above example, we want to find the names of tests. While traversing through these bit patterns, after the occurrence of a sequence of one or more ones, if there are n number of zeros, where n is even, then you can omit the tree nodes represented by the first n-1 zeros and continue the traversing with the tree node represented by the last zero and its siblings. Here we are considering the structural property of the JSON documents (key-value pairs). This property is very useful, if the memory is limited. Here, the number of nodes that is loaded into memory can be reduced. Even though it needs slightly more space, as far as the JSON documents are concerned, DFUDS and BP representations are better than the LOUDS representation.

V. CONCLUSION

The results of our paper are mainly theoretical. We have performed a study and an evaluation of the implementation of succinct trees for JSON documents. We focused on the most succinct schemes, which uses 2n+O(n) bits. It provides implementations of the most promising techniques to represent succinct trees and the recommendations on which structures to use depending on the application. We observe that the performance of static succinct tree implementations, which have logarithmic worst-case per-operation time complexity, particularly those based on the DFUDS and BP tree is very useful for the operations that could be performed on JSON objects. We obtain good performance, since the JSON objects are a collection of Key- Value pairs. However, further work is required to expand the functionality, to better understand the effects of succinct data structures on traversal performance and to investigate implementations of O(1)-time dynamic succinct trees.

REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Joannou, S. and Raman, R.2012. Dynamizing succinct tree representations. InProc. 11th International Symposium on Experimental Algorithms (SEA)\. 224, 235.
[3] R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1–10, 2004.
[4] J. Jansson, K. Sadakane, and W.-K. Sung. Ultrasuccinct representation of ordered trees. In Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (*SODA)*, pages 575–584, 2007.
[5] Munro and V. Raman. Succinct representation of balanced parentheses and static trees. SIAM Journal on Computing, 31(3): 762, 776, 2001.
[6] Jacobson. Succinct Static Data Structures. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1989.
[7] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. Rao. Representing trees of higher degree. Algorithmica, 43(4):275–292, 2005.
[8] CouchDB. http://couchdb.apache.org/.
[9] MongoDB. http://www.mongodb.org/.
[10] Craig Chasseur, Yinan Li and Jignesh M. Patel. Enabling JSON Document Stores in Relational Systems, In Proceedings of the 16th International Workshop on the Web and Databases 2013, WebDB 2013, New York, USA, 2013.
[11] JSON specification. http://json.org/.
[12] R. F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. Theoretical Computer Science, 368(3): 231, 246, 2006.
[13] O. Delpratt, N. Rahman, and R. Raman. Engineering the LOUDS succinct tree representation. In Proc. 5th Workshop on Efficient and Experimental Algorithms (WEA), pages 134–145. LNCS 4007, 2006.
[14] H.-I. Lu and C.-C. Yeh. Balanced parentheses strike back. ACM Transactions on Algorithms, 4(3), 2008.
[15] A. Farzan, R. Raman, and S. S. Rao. Universal succinct representations of trees? In Proc. 36th International Colloquium on Automata, Languages and Programming, pages 451, 462. Springer, 2009.
[16] S.K. Muthusundar and Dr. Paul Rodrigues, “Automation Testing of Web Application Based on the Navigation using JSON”, International journal of Computer Engineering & Technology (IJCET), Volume 3, Issue 1, 2012, pp. 191 - 197, ISSN Print: 0976 – 6367, ISSN Online: 0976 – 6375.