SJSON: A Succinct Representation for JSON Documents

Abstract

The massive amounts of data processed in modern computational systems are becoming a problem of increasing importance. This data is commonly stored directly or indirectly through the use of data exchange languages, such as JSON (JavaScript Object Notation) and XML (eXtensible Markup Language), for human-readable platform-agnostic access.

This paper focuses on exploring a set of succinct representations for JSON documents, which we call SJSON, achieving both reduced RAM and disk usage while supporting efficient queries on the documents. The representations we propose are mainly based on the idea that JSON documents can be decomposed into structural part and raw data part. In our method, we emulate the structure of the JSON document as a rooted ordered tree and represent it using succinct data structures, as opposed to the usual pointer-based implementation. Furthermore, the remaining raw data is reorganized into arrays of attributes and values. This deconstruction between structure and data allows for a straightforward connection between a node in the succinct tree and its corresponding name-value pair, dispensing pointers altogether.

The proposed scheme is implemented as the SJSON library in C++, and evaluated with respect to a number of metrics, comparing its performance with popular alternative JSON parsers. Empirical results show that the library is able to represent JSON files succinctly while efficiently supporting traversal queries.

Keywords: JSON, succinct data structure, semi-structured document representation, heterogeneous array indexing.

1. Introduction

Every minute 300 hours of video data is uploaded to Youtube; 4,000 unique visitors access the Amazon website; App Store users download 31,000 applications; and Facebook users like over 4,000,000 posts [1]. Content creation grows at such an explosive rate that currently over 90% of the world's data has been generated in the last two years [2]. Tremendous amounts of information are produced every day, and the storage needed to save this is increasing significantly. Given the limitations of available storage, it is more and more imperative for data to be compressed as much as possible. On the other hand, modern systems are still expected to execute operations and data analysis efficiently, even on large amounts of data. This requires functional compression approaches to be adopted, thereby allowing data to be compactly stored while permitting the efficient execution of operations.

In the real world, data is rarely a large sequence of random numbers; real-world string data tend to exhibit foreseeable characteristics, structural regularities, and often subject to domain constraints. This predictability enables compression schemes to represent the same information in a space-efficient manner, while still permitting the original data to be retrieved in full [3].

Succinct Data Structures. In succinct data structures [4], we strive to solve algorithmic problems by designing data structures that use amounts of space close to the information-theoretic lower bound, while still supporting the operations efficiently. Succinct data structures for a wide range of fundamental problems have been designed in the past couple of decades. Examples of problems whose solutions with succinct data structures have been extensively studied and documented in the literature include, among others, trees [5, 6], range minimum queries [7], and text indexing [8].

Data Exchange Formats. A common method for storage and exchange of the data in modern systems is through data exchange languages. These languages tend to be formats designed to describe data in ways that permit it to be read, parsed, and understood by the most diverse set of languages and platforms, decoupling data from particularities of its processing environment.

Over the course of the years, several data exchange formats were suggested, including XML (eXtensible Markup Language) [9] and JSON (JavaScript Object Notation) [10]. Although XML was the prevalent format for over a decade, due to its inherent high verbosity and low readability, the use of XML has declined over time in favor of the more compact and better-structured JSON. Many modern systems such as CouchDB [11] and MongoDB [12] have reported using JSON or BSON (Binary JSON) [13] as their native format for data storage and in-memory representation; while web service APIs – Twitter [14], Facebook, Google [15] – commonly adopt JSON as their data interchange language for transferring information between servers and clients.

There is another scheme to represent JSON documents, known as BSON. The main objective of this representation is to efficiently support schema-less lightweight network communication through fast encoding and decoding. By adding extra information, the document is able to be traversed easily. Unfortunately, this representation does not exploit the repetition of elements to achieve better compression. In addition, BSON only handles integers as 32-bit or 64-bit values regardless of their actual size. Therefore, although one of its design goals is to be efficient in space, BSON is not an ideal option for compact storage.

Ottaviano and Grossi [16] proposed a scheme that supports random access to the JSON document stored on the disk, more efficiently, using a *semi-index*. The semi-index enables us to navigate the file by encoding the document structure succinctly. The semi-index is basically a bit vector in which bits are set for specific locations that separate the elements. This scheme is feasible since a different separator is employed for each possible type. At the cost of a space overhead for storing a succinct representation of the document tree structure, their semi-index allows for random access of specific values without having to load the JSON file entirely into the main memory. Furthermore, the semi-index includes pointers that indicate the position of the corresponding element in the JSON file stored on disk. In contrast to our work, their representation neither actually compresses the document, nor strives to represent the JSON content succinctly in memory, but rather offers a layer of indirection for accessing the underlying stored data. In this respect, the total amount of disk space required by their approach is strictly higher, though not by much, than the original document, as it additionally requires the storage of the semi-index.

Nevertheless, to the best of our knowledge, there are no other schemes in the literature specifically suggested for efficient compression or efficient in-memory representation of JSON documents [17]. Libraries like JSONC [18], written in Javascript, focus on the compression of documents transferred between clients and web service APIs by employing traditional text compression methods such as gzip (DEFLATE, a combination of LZ77 and Huffman encoding) [19]. It could be possible to achieve both lower verbosity and higher utility in JSON representations by applying the fundamentals of notable XML compression algorithms to JSON.

In this paper, we suggest a memory-efficient JSON representation and compression library named *SJSON*, engineered by leveraging ideas of succinct data structures. Our scheme saves memory in three aspects of JSON documents. First, we model the document structure as an ordinal tree and encode it through succinct tree representations [20, 6]. Second, redundancies in attributes are removed and the remaining unique strings are stored in a simple contiguous array, which can be compressed using succinct data structures for strings, including compressed suffix arrays. Lastly, values of the JSON document are encoded compactly and stored in a heterogeneous structure named as *bit string indexed array*. Users can store this set of representations, either on RAM or on the disk. For the RAM and disk representations we allow users to query general information of a JSON document, without retaining the original JSON document.

The rest of this paper is organized as follows. Preliminary information on succinct data structures – such as tree representations and string representations – and data exchange languages – such as XML and JSON – will be introduced in Section 2. In Section 3, we elaborate on the theoretical and technical details of our JSON representation and compression scheme. Experiments and empirical results along with their analyses are given in Section 4. Finally, conclusions and future work will be discussed in Section 5.

2. Preliminaries

In this section, we introduce preliminary work upon which this paper is based. We start with an overview of the XML and JSON data exchange languages, stating characteristics, common usages, and the syntax of the format (Section 2.1). Next, we give a brief introduction to succinct data structures (Section 2.2), and describe succinct representations for ordinal trees that we use subsequently (Section 2.3). Finally, we introduce well-known general-purpose compression schemes and string-optimized compression algorithms (Section 2.4).

2.1. Data Exchange Formats

Data exchange (also called data interchange) is the process of taking data structured under a source schema and transforming it into data structured under a target schema, in a systematic way such that the target data is an accurate representation of the source data [21]. A data exchange language, or format, is a language that is domain-independent and can be used for diverse types of data. Its semantic expression, capabilities, and qualities are largely determined in comparison with the capabilities of natural languages. A common characteristic of such formats is that they can be parsed and correctly interpreted independently of programming language, running environment, and platform.

The following sections discuss two specific data interchange languages – XML and JSON – in more detail.

2.1.1. XML (eXtensible Markup Language)

XML (eXtensible Markup Language) is a text-based data exchange language derived from SGML (Standard Generalized Markup Language), being a simpler alternative that is both human-readable and machine-readable [9]. While its original goal is to meet the challenges of large-scale electronic publishing, XML also plays a major role in the data exchange and storage of a wide variety of systems and web services. This position was attained primarily due to its higher level of application-independence compared to other data interchange formats of its time.

An XML document is a string of Unicode characters. These are then divided into markups and contents. Markups are strings that begin with a less-than sign (<) and end with a greater-than sign (>). A component in the document is called an element, which has a start tag in the front and an end tag in the end. Inside the start tag, name-value pairs called attributes can exist.

Figure 1 depicts an example XML document.

```xml
<Books>
    <Book ISBN="055321419">
        <title>Sherlock Holmes: Complete Novels</title>
        <author>Sir Arthur Conan Doyle</author>
    </Book>
    <Book ISBN="0743273567">
        <title>The Great Gatsby</title>
        <author>F. Scott Fitzgerald</author>
    </Book>
    <Book ISBN="0684826976">
        <title>Undaunted Courage</title>
        <author>Stephen E. Ambrose</author>
    </Book>
    <Book ISBN="0743203178">
        <title>Nothing Like It In the World</title>
        <author>Stephen E. Ambrose</author>
    </Book>
</Books>
```

Figure 1: Example XML document [22].

One of the advantages of the XML format is that it is possible to check the validity of a document with respect to a given schema. More than one XML documents can be associated with a single DTD (Document Type Definition). DTD is a schema language that contains a set of markup declarations defining elements and attributes, which in turn can be tested against XML documents for validity. Based on the schema, the structure of an XML document could be modeled into a tree of components, namely elements, attributes, and textual data. Due to this characteristic, XML is also known as a semi-structured data interchange format.

Furthermore, there are two major query languages for working with XML endorsed by the W3C. XPath is a query language for selecting nodes from an XML document using location paths that resemble tree navigation [23]. XQuery is a more functional language that is designed to query and transform collections of data in XML documents [24].

Although advantages exist, the XML data exchange format is utterly verbose, cluttering the resulting file with nonessential structural information and metadata, which hinders the efficiency of the language in terms of usability, readability, and compactness. In an attempt to mitigate this size overhead, several algorithms implementing XML compression have been suggested in the literature, including XMill [25], TREECHOP [26], XQueC [27] and XBZipIndex [28]. These compressors are commonly classified with respect to whether or not the resulting compressed file support queries as it is; and whether structure and data content are stored alongside or separately. An XML compression method is called non-queryable if it necessitates the entire XML document to be decompressed before querying can take place. Homomorphic compressors encode both the structure and content of a document in a single container, while permutation-based compressors try to improve the compression ratio by differentiating the structural and content sections of a document.

For parsing XML documents, some libraries such as pugixml [29] provide DOM (Document Object Model)-like interface to manipulate the original document, though for pugixml it only supports documents that could be fit in main memory and it does not reduce the size of the processed representation. The SiXDOM library suggested by Delpratt et al. [30] utilizes succinct data structures while constructing the DOM structure in RAM, which supports some navigational queries. However, this library does not consider storing components other than the tree structure.

2.1.2. JSON (JavaScript Object Notation)

JSON is an open standard document format that uses human-readable structured text to represent data objects [10]. Designed as an alternative to XML, JSON is originally based on a subset of the JavaScript programming language. Even though its name includes a specific language, JSON is a programming language-independent format widely used nowadays to exchange data on the web and to represent structured information.

The JSON interchange format is designed around two types of entities – objects and arrays. An *object* is an unordered list of name-value pairs, i.e., an associative array. Names are strings, and values are one of the possible JSON value types. Objects are wrapped around curly brackets ("{" and "}"), and successive name-value pairs are separated with commas ("`,`"). Though the specification states that pairs inside an object are, in fact, unordered, JSON parsers commonly assume some implicit ordering. There could be pairs with identical names inside an object, and in our representation we explicitly allow it. An *array* is an ordered list of values. It is enclosed in square brackets ("`[`" and "`]`") and subsequent values are separated by comma. Values in an array do not have associated names.

Core types of JSON values, beside the two s discussed above, include *number* (integer and real), *string*, *boolean* and *null*. Figure 2 shows an example of a small but complete JSON document, and Figure 3 illustrates a basic automata that generates documents in JSON document format.

Similar to XPath for XML, there are some standards for JSON querying. Google devised a functional query language Jaql [31] which is based on a flexible data model inspired by JSON, supporting manipulation of arrays and user-provided functions. A statement consists of a source, a sink, and pairs of an operator and a parameter. Some of the operators this language supports are `FILTER`, `GROUP` and `JOIN`. JSONiq [32] is another functional query language which can also process unstructured documents. This language has two different

```
{
    "id": 35420,
    "name": "Toaster",
    "tags": ["Kitchen", "Appliances"],
    "price": 32.99,
    "on_sale": true,
    "stock": {
        "warehouse": 300,
        "store": 20
    }
}
```

Figure 2: Example JSON document.

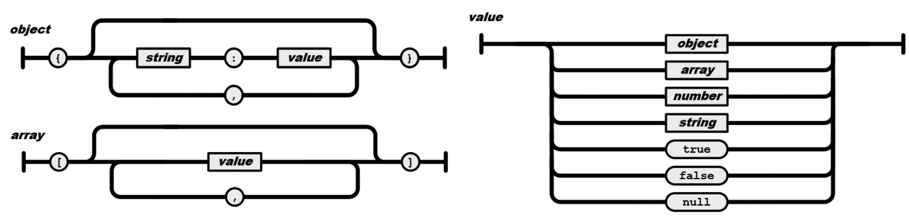


Figure 3: JSON automata.

behaviors: its independent syntax and XQuery-like grammar. Since JSONiq is highly influenced by XQuery, its data model and list of supported queries resemble those in XQuery.

As opposed to XML, there are no well-known compression schemes to both encode and query JSON documents. Additionally, existing JSON libraries, such as JSONC [18], naively apply well-known generic text compression methods (e.g, gzip [3]) in a JSON document, so that the entire document needs to be decompressed for querying. Another approach commonly used when transferring JSON files is stripping unnecessary whitespaces in a process called JSON minification. This, however, does not constitute an actual compression technique. BSON [13] derived from MongoDB aims high processing speed by encoding documents in binary format and including additional information for traversals.

Nevertheless, although this may reduce size of a JSON document, no actual compression is performed during the conversion. Thus, this representation also does not compose a compression scheme.

The most popular arguments in favor of XML are around the benefits of its interoperability and openness. However, none of these are inherent to XML itself. JSON offers the same qualities while improving in a number of aspects, especially with respect to conciseness, human-readability and ease of parsing and processing by a machine. JSON represents data as collections of arrays and records, which is what data actually is. XML represents data based on elements, attributes, content text, entities, and other metadata. Furthermore, XML is document-oriented, while JSON is data-oriented. Data-oriented formats can be more easily mapped to object-oriented systems.

2.2. Succinct Data Structures

Succinct data structures as a field started with Jacobson's 1988 Doctoral Dissertation [4]. In this variation of data structures, we design algorithmic solutions that use an amount of space close to the information-theoretic lower bound of the problem in hand, while still allowing for the execution of efficient operations. One can think of succinct data structures as an extension of data compression, in which space is close to the information-theoretic lower bound and queries are efficient.

One of the fundamental structures employed when devising a new succinct data structure solution is the *bit string*. A bit string is a string over the alphabet $\Sigma = \{0, 1\}$. A bit string by itself has limited use. Although its compactness provides us a framework upon which information can be concisely represented, few useful operations can be efficiently performed on raw bit strings. To enhance its usability, bit strings can be extended in terms of functionality with auxiliary structures for `rank` and `select` operations.

Given a string S of length n over the alphabet Σ, the `rank` and `select` operations are defined as follows:

- $rank_\alpha(S, i)$: the number of occurrences of α in the first i positions of S, for any $\alpha \in \Sigma$.

- $select_\alpha(S, i)$: the position of the i-th α in S, for any $\alpha \in \Sigma$.

In the case that S is a bit string and, thus, $\Sigma = \{0, 1\}$, we have the operations $rank_0$, $rank_1$, $select_0$ and $select_1$. For instance, if $S = 110101$, then $rank_0(S, 4) = 2$ and $select_1(S, 2) = 1$. Extensive research has been conducted on succinct implementations of rank and select structures over bit strings [33, 34, 4, 5]. One can support both operations in $O(1)$ time while using $o(n)$ additional bits of space.

On the same vein, a second auxiliary structure built around bit strings is the *balanced parentheses*. This data structure conceptually interprets set bits (i.e., 1s) and unset bits (i.e., 0s) of a bit string as open and close parentheses, respectively. When this resulting sequence of opening and closing parentheses is balanced, it is considered as a balanced parentheses structure.

Just as it was the case with `rank` and `select`, the core operations in balanced parentheses over an n length bit string can be performed in constant time with additional $o(n)$ bits [5]. For convenience we can define `rank` and `select` operations over balanced parentheses bit strings as $rank_{open}(S, i) \equiv rank_1(S, i)$, $rank_{close}(S, i) \equiv rank_0(S, i)$, $select_{open}(S, i) \equiv select_1(S, i)$, and $select_{close}(S, i) \equiv select_0(S, i)$.

2.3. Succinct Ordinal Tree Representations

Succinct data structures are fundamentally based on representing elements of a given set in a compact form, in such a way that operations on its domain can still be executed efficiently [4]. In general, succinct data structures aim for representing instances of a set using space as close as possible to the information-theoretic lower bound, while still supporting operations efficiently.

We outline two space-efficient ordinal tree representations – Balanced Parentheses (BP) and Depth-First Unary Degree Sequence (DFUDS). Both achieve the optimal space of $2n$ bits for representing ordinal trees (since there are

Tree Operation	Description
$pre\_rank(x)$	preorder rank of node x
$pre\_select(p)$	the node with preorder p
$isleaf(x)$	whether node x is a leaf
$ancestor(x, y)$	whether node x is an ancestor of y
$depth(x)$	depth of node x
$parent(x)$	parent of node x
$first\_child(x)$	first child of node x
$next\_sibling(x)$	next sibling of node x
$subtree\_size(x)$	number of nodes in the subtree of node x
$degree(x)$	number of children of node x
$child(x, i)$	i-th child of node x
$child\_rank(x)$	number of siblings to the left of node x

Table 1: Operations on ordinal trees [35].

$C_n = \frac{1}{n+1}\binom{2n}{n}$ ordinal trees on n nodes, we need at least $2n - O(\log n)$ bits to encode an arbitrary ordinal tree on n nodes), and are able to perform a number of tree operations efficiently with the aid of `rank` and `select`, and balanced parentheses auxiliary structures in total $2n + o(n)$ bits of space. A summary of some of the most significant tree operations is given in Table 1 [35].

The BP tree representation is first proposed by Jacobson [20] and later improved by Munro and Raman [5]. In this method, a balanced parentheses bit sequence is constructed from a depth-first traversal of the tree, by writing an opening parenthesis when first arriving at a node, and a closing parenthesis after visiting all of its children, namely all nodes in its subtree. In this way every node has exactly two parentheses associated with it: an open parenthesis “`(`” and a close parenthesis “`)`”. Thus, this encoding represents a tree with a bit string composed of $2n$ balanced parentheses. This representation uses space that is within lower-order terms of the information-theoretic lower bound (of $2n - O(\log n)$ bits) for encoding trees.

To support operations in this tree representation we then need to make use of the auxiliary structures equipped with `rank`, `select`, and balanced parentheses, discussed in Section 2.2. Notice that in this encoding nodes of a subtree are stored contiguously in the designated bit string. Therefore, the subtree size can be computed by simply taking half the distance between the opening and closing parentheses that correspond to a node.

From the core operations provided by the `rank`, `select`, and balanced parentheses structures we can derive several tree operations efficiently. In fact, it is known that all of the core tree navigational operations presented in Table 1 can be performed in $O(1)$ time utilizing this encoding.

The DFUDS tree representation [6, 36] is an alternate approach to LOUDS (Level-Order Unary Degree Sequence) [20] and BP. To combine the virtues of these two representations, DFUDS writes a unary degree sequence of each node in a depth-first traversal of the tree. That is, whenever we arrive at a node during a depth-first traversal, we append d open parentheses and one closing parenthesis, where d is the number of children of the node being visited. A node is represented by the position of its first open parenthesis.

With the addition of one artificial open parenthesis prepended at the beginning of the bit string, the resulting encoding is also a balanced parentheses bit sequence. As a result, each node has exactly two bits corresponding to it. A 1 bit (open parenthesis) is written in the bit string when visiting its parent, and one 0 bit (close parenthesis) is written in the bit string when visiting the node itself. This generates a $2n$ length bit string, which is again within lower-order terms of the information-theoretic lower bound for representing a tree on n nodes.

Tree operations can then be supported with an additional $o(n)$ bits of space through auxiliary structures, as discussed in Section 2.2. As it was the case with the BP representation, nodes of a subtree are stored contiguously in the bit string generated through the DFUDS representation. Thus, the subtree size can be computed by simply taking half the distance between the opening and closing parentheses that correspond to a node.

Tree Operation	BP	DFUDS
$pre\_rank(x)$	$rank_{open}(x)$	$rank_{close}(x-1)+1$
$pre\_select(p)$	$select_{open}(p)$	$select_{close}(p-1)+1$
$isleaf(x)$	$S[x+1] =')'$	$S[x] =')'$
$ancestor(x,y)$	$x \leq y \leq findc(x)$	$x \leq y \leq findc(enclose(x))$
$depth(x)$	$excess(x)$	–
$parent(x)$	$enclose(x)$	$prev_{close}(findo(x-1))+1$
$first\_child(x)$	$x+1$	$child(x,1)$
$next\_sibling(x)$	$findc(x)+1$	$findc(findo(x-1)-1)+1$
$subtree\_size(x)$	$(findc(x)-x+1)/2$	$(findc(enclose(x))-x)/2+1$
$degree(x)$	–	$next_{close}(x)-x$
$child(x,i)$	–	$findc(next_{close}(x)-i)+1$
$child\_rank(x)$	–	$next_{close}(y)-y; y=findo(x-1)$

Table 2: Operation details of tree operations in BP and DFUDS representations [35]. A dash is used to indicate operations that require additional auxiliary structures.

From the core operations on parantheses, we can derive several tree operations efficiently. In fact, all the tree operations presented in Table 1 can be performed in $O(1)$ time, using this set of succinct ordinal tree representations along with auxiliary data structures. Arroyuelo et al. [35] provide emulation of navigational queries to preliminary operations supported in auxiliary data structures. Table 2 summarizes those operations. $findc$ and $findo$ operations find the position of matching close and open parenthesis of a parenthesis, respectively. $excess$ operation finds the difference between the number of open and closing parenthesis before a position. $enclose$ operation in an open parenthesis returns the position of the open parenthesis corresponding to the closest matching parenthesis pair enclosing the input open parenthesis. For the DFUDS representation, $prev_{close}(x) \equiv select_{close}(rank_{close}(x))$ and $next_{close}(x) \equiv select_{close}(rank_{close}(x)+1)$.

Figure 4 shows an example ordinal tree, along with its BP and DFUDS

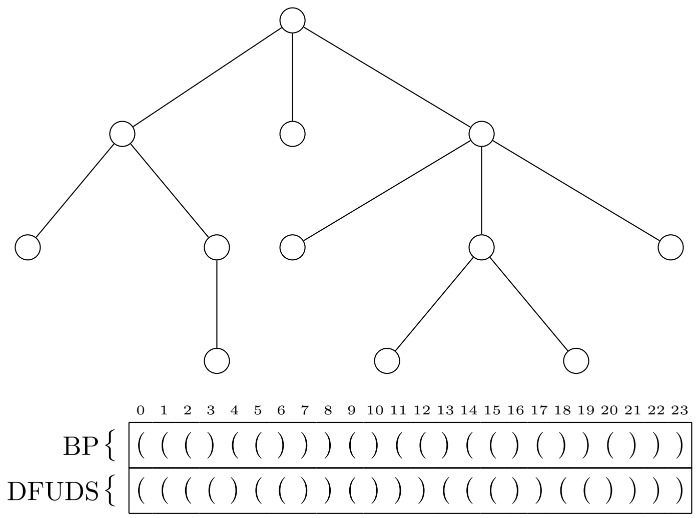


Figure 4: An ordinal tree with the succinct representations.

represented tree structure.

2.4. String Compression Schemes

General-purpose lossless compressors such as LZ77 [3], LZ78 [37], and LZW [38] perform dictionary-based encoding to support compression. These algorithms substitute contiguous length of text into the location of entry inside the dictionary. Although dictionary-based compression algorithms may provide high compression ratio, if one needs to randomly access and manipulate the encoded content, the whole sequence of it should be decompressed again, which can be a significant computation overhead.

To alleviate this handicap, a plethora of algorithms have been suggested. In the early ages, indexable data structures such as suffix trees [39] and suffix arrays [40] are widely used to deal with string compression. Suffix trees are compressed tries containing all the suffixes of the given text as their keys and positions in the text as their values. The given text is terminated with a special character $, which is considered the lexicographically smallest. Construction of a suffix tree takes $O(n)$ time, where n is the length of the given text. Each constructed suffix tree occupies $O(n)$ space. When a suffix tree is constructed, string search and finding the longest substring queries can be done in either $\Theta(m)$ or $\Theta(n)$ time, where m is the length of a substring.

Suffix arrays are sorted arrays of all suffixes of a given string. When the length of a string is n, construction time and space usage are both $O(n)$, similar to suffix trees. Note that suffix arrays can be constructed by performing a depth-first traversal of the relevant suffix tree. Locating every occurrence of a substring pattern in the string using suffix arrays takes $O(m \log n)$ time, where m is the length of the pattern. Constructing compressed suffix arrays [41] also takes $O(n)$ time, and when compressed space usage becomes $O(nH_k(T)) + o(n)$. The operation to query a pattern in the compressed array takes $O(m)$ or $O(m+\log n)$ time.

Ferragina and Manzini proposed an alternative text indexing scheme known as the FM-index [42]. This index relies on BWT (Burrows-Wheeler Transform) [43]. BWT is a reversible transformation for strings for the preparation of efficient compression. While the transformation itself does not reduce the size of the string, it is able to convert the string to runs of repeated characters, feasible to be compressed using run-length encoding schemes. FM-index supports counting and locating operations in $O(p)$ and $O(p + occ \log^{\epsilon} u)$ time, respectively, where p is length of a pattern, u is length of a text, occ is number of the pattern occurrence and $0 < \epsilon < 1$ is an arbitrary parameter.

Wavelet trees [44] and wavelet matrices [45] are also used as text indexing schemes. Wavelet tree is a succinct data structure for strings, and it supports `access` as well as `rank` and `select` operations for an alphabet in $O(\log \sigma)$ time, where σ is size of the alphabet. A string S occupies $nH_0(S) + o(|S| \log \sigma)$ bits. Later version of the FM-index [46] also utilizes wavelet trees to reduce space for large alphabets.

3. *SJSON*: Succinct Representations of JSON Documents

This section further explores details of the JSON representations and related data structures implemented in our library SJSON. The goal of this section is to suggest a compact representation for JSON documents exploiting bit strings and succinct ordinal tree data structures.

3.1. Bit String Indexed Array

In contiguous homogeneous arrays all entries are of a single type, and hence have the same fixed size in bytes. In such arrays, indexing is easily computed from the array's starting position and the length of each array element. In heterogeneous arrays, however, auxiliary structures are required for efficient indexing, as the size of entries is variable. A common technique used in modern programming languages to provide the illusion of heterogeneous arrays is to use a homogeneous array of pointers to elements. Each element pointed to may be of a different type, but the array is still homogeneous. This approach has the downside of requiring additional pointers, which, in modern 64-bit computers, correspond to 8 extra bytes per array entry.

We propose an indexing scheme based on bit strings along with the `select` auxiliary structure, which we denote as *bit string indexed array*. Consider a contiguous heterogeneous array A with n elements and a total size of m bytes. We generate a bit string S of length m bits such that the i-th bit is set if and only if the i-th byte of A corresponds to the beginning of one of its elements. Notice the bit string S as generated above has exactly n set bits, and occupies

<table>
<tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr>
<tr><td>S {</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>A {</td><td colspan="2">2189</td><td colspan="4">3.141592</td><td>T</td><td colspan="4">322000</td></tr>
</table>

Figure 5: A bit string index built on top of the example heterogeneous array $A = \{2189, 3.141592, \text{true}, 322000\}$. Notice how each and every set bit in S corresponds to the starting position of an element in A.

a total $m/8$ bytes. Now the problem of indexing the i-th entry of a bit string indexed array is reduced to a call to $select_1(S, i)$, that is, the position of the i-th 1 in S.

Figure 5 depicts how the bit string indexed array is structured on a sample heterogeneous array. In this example *booleans* occupy 1 byte, *numbers* take either 1, 2, 4 or 8 bytes based on their capacities.

The overall space overhead incurred by this scheme is m bits for the bit string S and extra $o(m)$ bits for the auxiliary `select` structure. More precisely the bit string index requires $m + o(m)$ extra bits in addition to the input array.

If more compression is needed, one can encode the index using *sdarray* [47]. Since we can assume the bit vector is sparse (i.e., number of set bits are extremely smaller than unset bits), *sdarray* structure efficiently encodes the index in $n(2 + \log \frac{m}{n})$ bits, while supporting `select` queries in $O(1)$ time.

3.2. Main SJSON Representation

One of the main points in which SJSON improves memory usage compared to other libraries lies in the fact that we devise a compact variable-length encoding for JSON values. In order to store a series of variable length encodings, we design a memory-efficient heterogeneous array discussed in Section 3.1. This array is in turn used to compose the underlying data structures used in SJSON.

In order to represent a given JSON document in a memory-efficient manner, we deconstruct the document tree structure portion from its content data. The

JSON Type	Encoding	Size (bytes)
null	{type}	1
object	{type}	1
array	{type}	1
boolean	{type}	1
string	{type, index}	9+
number	{type, value}	$2, 3, 5, 9$

Table 3: Encodings of JSON types and respective sizes.

two subdivisions are in turn separately encoded. This, in turn, allows us to leverage the characteristics and patterns particular to each specific data type, to achieve better memory usage.

We model the document structure using ordinal trees and implement encodings through the DFUDS succinct tree representation discussed in Section 2.3. Improvements in memory usage here derive from the fact that traditional JSON libraries represent the tree structure through pointer-based implementations, incurring overhead of about 8 bytes per pointer in the JSON document in 64-bit systems. Succinct trees allow us to reduce this overhead to $2 + o(1)$ bits in our scheme. A preliminary version of the paper [48] used the BP representation to represent the document tree, however this representation lacks efficient support for `child` and `degree` operations needed for querying the document, as denoted in Table 2. Therefore, we use the DFUDS ordinal tree representation to store the document tree in the library.

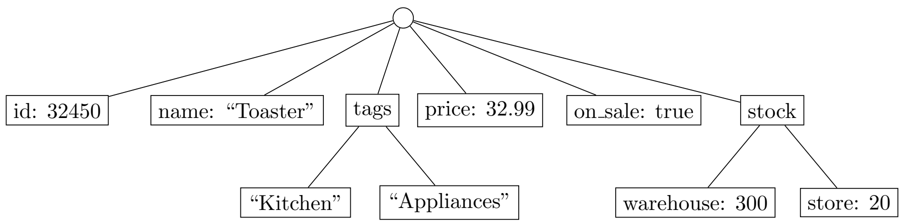
Given that the structure is dissociated from the document content, the raw data that remains is a series of names and values. These two components are also represented separately. According to the JSON specification, names are exclusively strings which are repeatable among objects. Thus, names container may constitute lots of redundant entries. It is common for JSON documents with millions of nodes to have no more than a few dozen unique names. In our scheme, we strip redundancies in names and store the unique strings in a contiguous memory array. Values that have a name associated should encode with itself the index of its corresponding name.

Finally, a value can be any of the JSON types outlined in Section 2.1.2, and may or may not have a name associated with it. We encode a specific JSON value according to its type as described in Table 3. All encodings start with a byte identifying its JSON type, and whether the value has a name associated with it. If a JSON value has an attribute associated, its encoding includes an extra 8-byte index that identifies the corresponding entry in the attributes array. String values require special treatment as their lengths are highly variable. We store all strings in an array, and the JSON encoding only stores the index of its corresponding entry in an array `stringValues`. Number type encodings occupy 9 bytes for 64-bit numbers and 5 bytes for 32-bit numbers. Smaller values may use 16-bit numbers instead, for total encoding sizes of 3 bytes. Decimal numbers also follow this notation, depending on their precision.

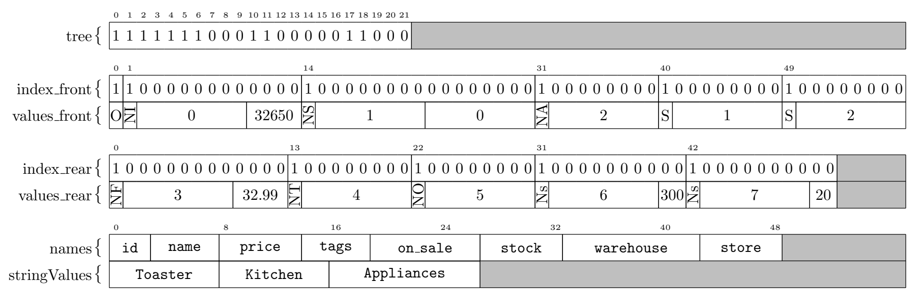
Initial version of the SJSON library [48] maintains strings in the `stringValue` array as-is. This allows easy extraction of relevant value in a pair. Nevertheless, since maintaining the original array does not actually involve compacting storage, we give an option to apply additional compression to this array. When space compaction needs to be considered in high priority, we provide apply additional compression utilizing concepts of compressed suffix arrays discussed in Section 2.4.

The list of value encodings is then stored in a bit string indexed array as outlined in Section 3.1 without further memory overhead. Each JSON value is indexed by order of discovery in the depth-first traversal step performed when creating the succinct tree representation, as noted in Section 2.3. That is, the i-th node in the succinct tree corresponds to the i-th element in the values array. This characteristic provides us with a lightweight and straightforward correspondence between tree nodes and associated data, based merely on array indexes.

Figures 6 and 8 depict example document tree of a JSON file (top and Figure 7, respectively) and contain an illustration of the data structure generated by SJSON to represent that document (bottom). `tree` and `index` are bit strings, while `values`, `names`, and `stringValues` are regular arrays. Entries in the `values` array start with a byte representing type of an element, where O, NA, NC, Ns, NI, NF, NS, and NT stand for *Object*, *Named Array*, *Named Char*, *Named Short*, *Named Integer*, *Named Float*, *Named String*, and *Named True*, respectively. Notice how named types include an additional 8-byte index to its associated entry in the attributes array. Note that the `stringValues` array is not converted to the compressed suffix array in these figures.



(a) Document tree structure corresponding to the sample JSON shown in Figure 2.



(b) In-memory representation of the sample JSON from Figure 2 as encoded by SJSON.

Figure 6: In-memory representation of the sample JSON from Figure 2 as encoded by SJSON.

3.3. Supporting Queries

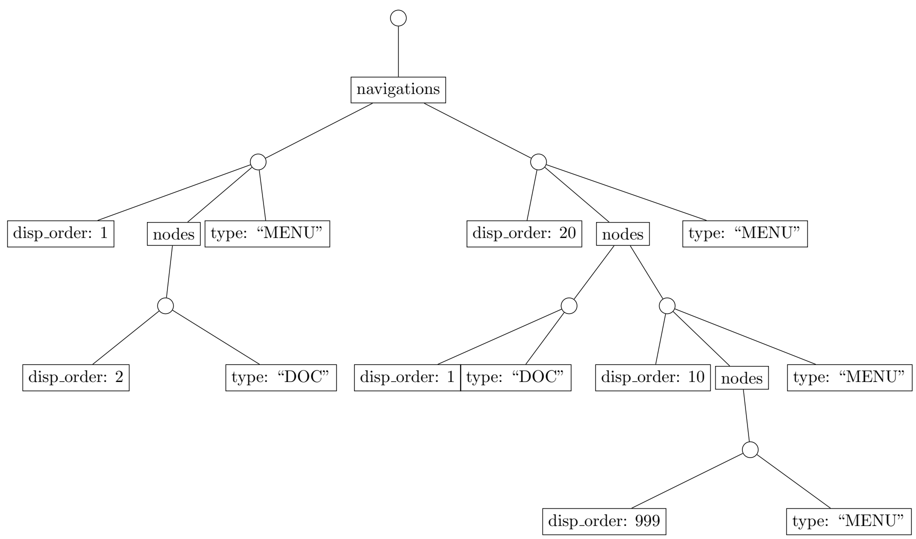
From the two query languages dealt in Section 2.1.2, it could be understood that it is indispensable for JSON libraries to support efficient traversals of the DOM tree as well as retrieval of the relevant name-value pairs. By utilizing the succinct tree structure as well as bit indexed arrays, our space-efficient representation suits those two core query objectives.

We support the following query operations in the SJSON library.

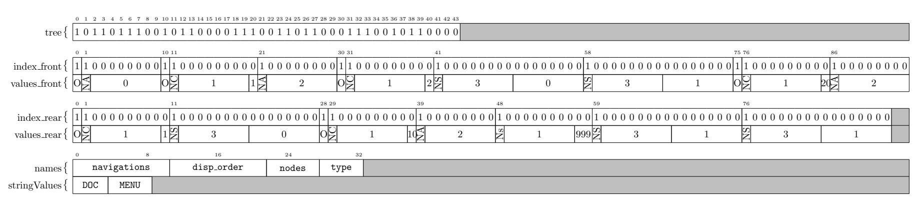
- `listObjectNames(o)`: Given an JSON object o, return its list of names.
- `getObjectValue(o,n)`: Given an JSON object o, return value of an entity with name n. If multiple entities with identical names exist, return a list of values.
- `countArrayElements(a)`: Given an JSON array a, return its size.
- `getArrayValue(a,i)`: Given an JSON array a, return value of its i-th element.

```
{
    "navigations": [{
        "disp_order": 1,
        "nodes": [{
                "disp_order": 2,
                "type": "DOC"
        }],
        "type": "MENU"
    }, {
        "disp_order": 20,
        "nodes": [{
            "disp_order": 1,
            "type": "DOC"
            }, {
            "disp_order": 10,
            "nodes": [{
                "disp_order": 999,
                "type": "DOC"
            }],
            "type": "MENU"
        }],
        "type": "MENU"
    },
]}
```

Figure 7: Example nested JSON document.



(a) Document tree structure corresponding to the sample JSON shown in Figure 7.



(b) In-memory representation of the sample JSON from Figure 7 as encoded by SJSON.

Figure 8: In-memory representation of the JSON document from Figure 7 as encoded by SJSON.

For example, on the document in Figure 6, the answers to some of the queries are shown below:

- `listObjectNames(0)`: id, name, tags, price, on_sale, stock
- `getObjectValue(8,‘‘store’’)`: 20
- `countArrayElements(3)`: 2
- `getArrayValue(3, 2)`: “Appliances”

These queries first perform navigation of the succinct tree, using `child` or `parent` operations. Once the relevant node is located, the queries extract the desired information by either inquiring the bit indexed arrays or calling additional `degree` tree operation. Locating the exact place for answering queries in the bit indexed array takes theoretically constant time, by running `select` operation in the `index` array. Additionally, as mentioned in Section 2.3, the DFUDS representation supports all of the aforementioned operations in theoretically constant time. Therefore, a combination of those libraries enables efficient JSON query processing.

Note that if users choose to compress the string values in a compressed suffix array, then extracting dedicated characters for answering queries takes $O(v)$ time instead of constant, where v is the length of the value.

4. Experimental Results

The library has been implemented in the C++ programming language and compiled with g++ 10.1.0. The environment in which the tests were executed features an Intel Core i7-6700K 4.20GHz CPU, 64GB DDR4 RAM, and 512GB NVMe drive. The machine runs Linux kernel version 5.8. RAM usage readings are done with `valgrind`, and elapsed time values for construction and querying are measured with the C++ STL library `chrono`.

The library borrows core concepts of the popular JSON processing library RapidJSON [49] while parsing a JSON document. We make use of the SDSL [50] library to aid our implementation with bit strings and auxiliary structures, employing the `rank` structure as proposed by Vigna [51], `select` structure by Clark [33], balanced parentheses structure by Navarro and Sadakane [52] and compressed suffix arrays structure. Balanced parentheses structure is mainly used to query the succinct tree stored in bit strings.

We evaluate our scheme against three popular JSON libraries – JsonCpp [53], JSON for Modern C++ [54], and RapidJSON by measuring RAM usage and elapsed time during construction. If the libraries support querying functionalities, we compare their relevant performance to our SJSON library. With respect to compression, we compare our scheme against the original file size, blank-eliminated JSON file using JSONC [18], and gzip-applied [3] result. For evaluating querying performance, we also consider semi-indexing suggested by Ottaviano and Grossi [16].

The source code for SJSON is available at `https://github.com/wombatkik5/sjson` for reproduction.

4.1. Datasets

The experiments were performed on a collection of datasets of both synthetic and real world corpora. We generate synthetic datasets of single possible types in JSON (`array`, `bool`, `double`, `int`, `null`, `object` and `string`) with number of nodes from 2,000,000 to 100,000,000. With these datasets we intend to illustrate the performance behavior of the libraries on different value types. More details of the real world corpora are described in Table 4.

- `Twitter`: A list of 20,000 tweets and metadata collected in 2015.
- `SNLI` [55]: The Stanford Natural Language Inference corpus is a collection of human-written English sentences coupled with semantic metadata.
- `Citylots` [56]: This dataset is a JSON converted document of the CityLots spatial data layer, a representation of the City and County of San Francisco's Subdivision parcels.

Corpus	Nodes	Size (MB)
`Twitter`	3,249,499	90
`SNLI`	6,757,124	465
`Citylots`	13,805,883	181
`DBLP`	64,714,826	1,741
`150JS-evaluation`	420,358,521	4,910
`150JS-training`	878,277,103	10,247

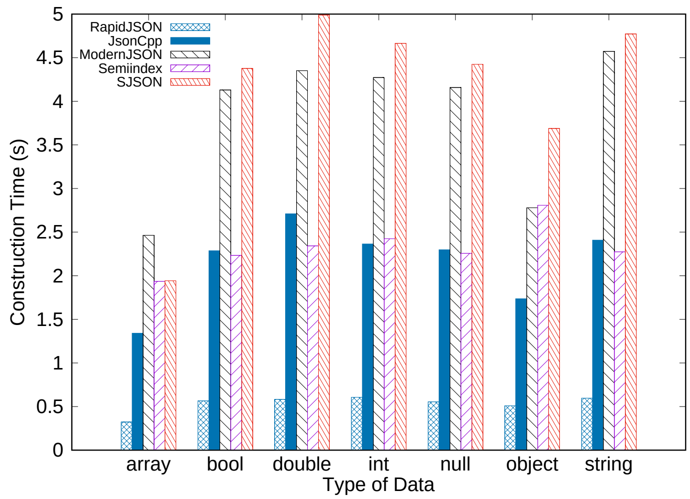
Table 4: Overview of the real world datasets used in our experiments.

- `DBLP` [57]: This dataset offers bibliography entries recorded in DBLP [58] until October 2014.
- `150JS` [59]: For 150,000 JavaScript files, their corresponding parsed AST (Abstract Syntax Tree)s are collected as two JSON documents: training (100,000) and evaluation (50,000).

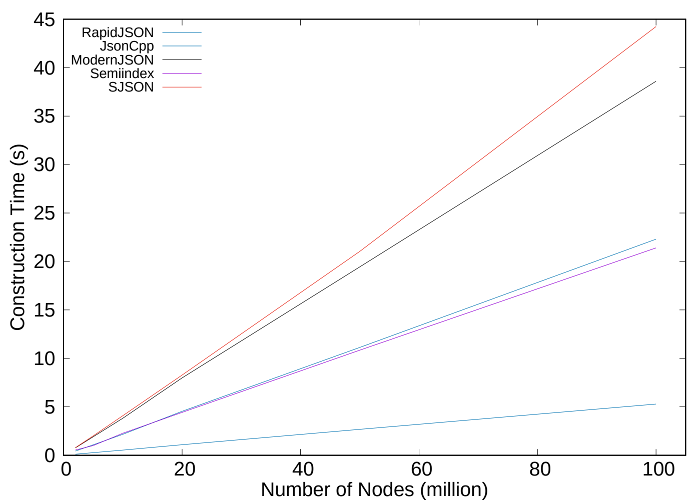
4.2. Construction Time

Figures 9 and 10 represent construction time of the libraries mentioned above. For all libraries, construction time includes reading a JSON file from disk and constructing auxiliary data structures based on that file in RAM.

By examining the tendency of construction time on synthetic documents described in Figure 9b, it is clear that the time is proportional to the number of nodes. As our JSON parsing scheme is derived from that of RapidJSON, we can observe that data structure construction takes most of the construction time. Establishing tree structure is concurrently done while traversing the document. Therefore, it is evident that preparing a succinct data structure for balanced parentheses needs to be further optimized for construction. Although construction time is an important factor to process a JSON document, once the library supporting serialization (other than the original JSON document) parses the document it need not reconstruct the whole representation. As mentioned later in Section 4.4, SJSON supports serialization and deserialization of the repre-



(a) $n = 10,000,000$.



(b) Varying n.

Figure 9: Construction time of SJSON compared to different libraries, for synthetic data.

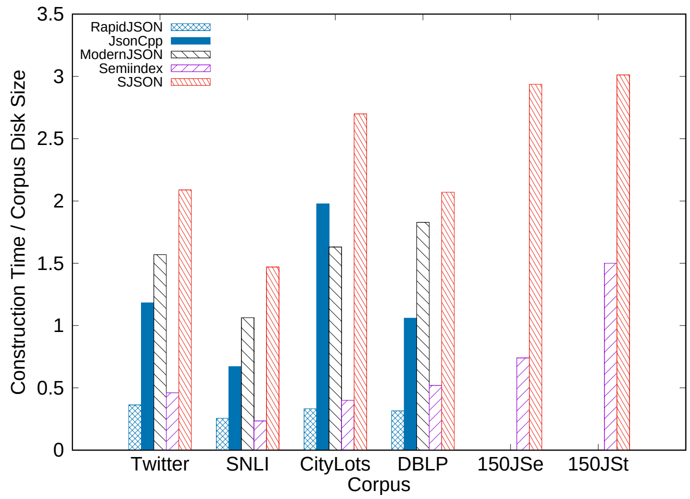
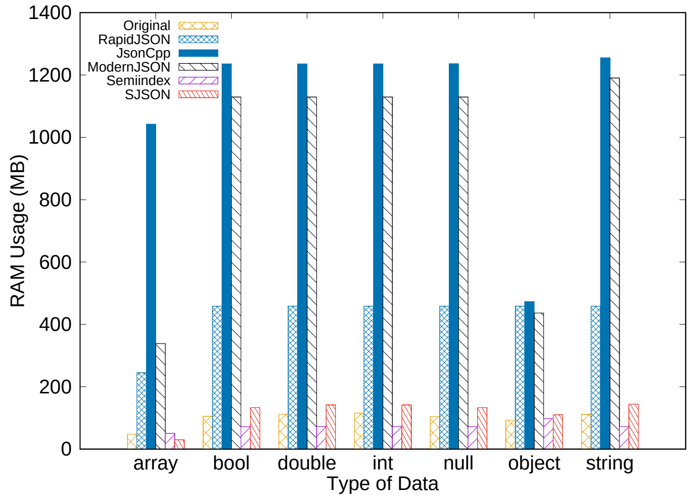


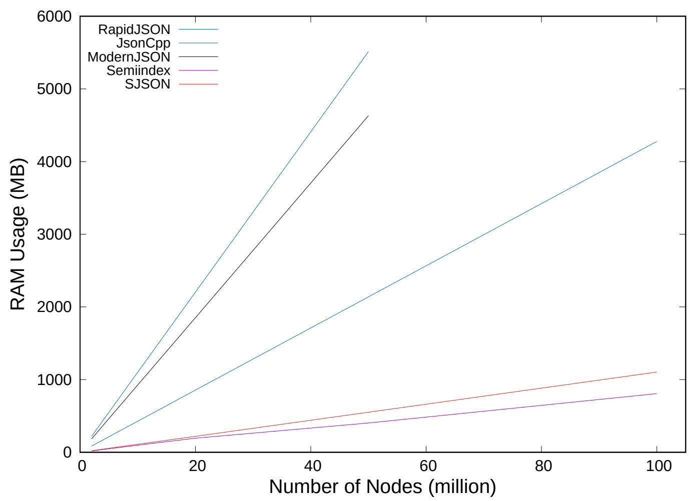
Figure 10: Relative construction time (with respect to corpus disk size) of SJSON compared to different libraries, for real world corpora.

sentation which significantly takes lesser time than construction shown above. Therefore, we consider that serializing the encoded representation to disk will mitigate demerits of slower construction time.

The experimental environment could not handle `150JS` corpora using third-party parsers because of insufficient RAM. Also, all the external libraries except RapidJSON could not run on the `DBLP` corpus, whereas our library is able to handle those documents as well. Since this is one of the merits of processing big data, we claim that our library has a strong point, suitable to handle larger JSON documents, even when the amount of RAM available is small. It is worth mentioning that semi-index could also handle those documents since this library mainly focuses on constructing the tree structure while retaining the original document on disk, not actually constructing the parsed representation.



(a) $n = 10,000,000$.



(b) Varying n.

Figure 11: Memory usage of SJSON compared to different libraries, for synthetic data.

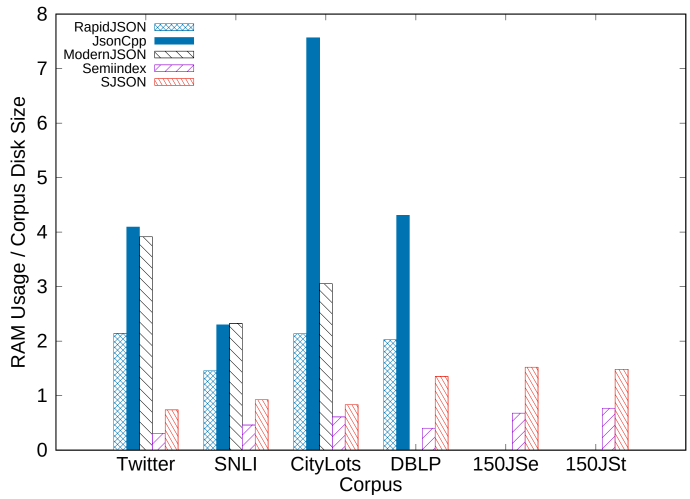


Figure 12: Relative memory usage of SJSON (with respect to corpus disk size) compared to different libraries, for real world corpora.

4.3. RAM Usage during Construction

Figures 11 and 12 show the main memory usage of SJSON compared to JsonCpp, JSON for Modern C++ and RapidJSON. For visual comparison purposes, Figure 11 also includes the original file size on disk, while Figure 12 shows the relative ratio of the RAM usage compared to the original disk size.

It is evident from the experiments that RapidJSON performs best among the third-party libraries evaluated, and JsonCpp is the worst. Our library, mostly represents the input datasets in strictly less RAM than RapidJSON by up to 91% on synthetic data and 66% on real-world corpora, while outperforming JsonCpp by up to 98% and 84% on synthetic and real-world corpora, respectively. For corpora with pairs containing large string values, our library representation uses more space than RapidJSON, when compression is not applied.

Our scheme offers a significant improvement in memory efficiency by encoding JSON values according to its type and data in a compact manner, using total memory proportional to the amount of information contained in a JSON file. On the other hand, common JSON libraries use fixed-length representations for all JSON values, leading to memory usage proportional to the total number of nodes. RapidJSON, for example, allocates 48 bytes for most values, regardless of type. Array entries are the exception, taking 24 bytes of memory. This explains why RapidJSON uses the same amount of memory for most synthetic datasets, except for array. JSON for Modern C++ and JsonCpp show similar behavior. Similar to construction time, RAM usage between the two succinct tree representations does not differ, reflecting the identical theoretic bound.

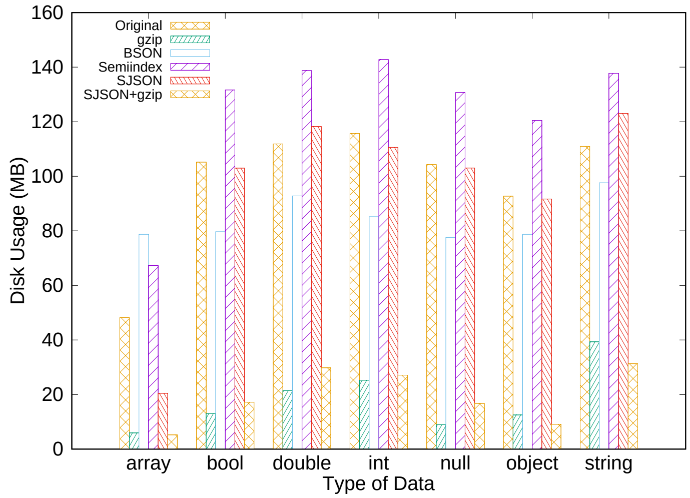
Compared to the conference version of this work [48], the new representation uses about 30% more RAM in some of the synthetic corpora. This is we allocate 8 bytes instead of 4 for recording the IDs of the names and string values, to support representing JSON documents with more than 2^{32} different possible strings. Nevertheless, by maintaining the string values efficiently in memory, representations of most of the corpora with strings use less RAM than the conference version.

As mentioned in the previous section, all other libraries except ours could not process larger corpora.

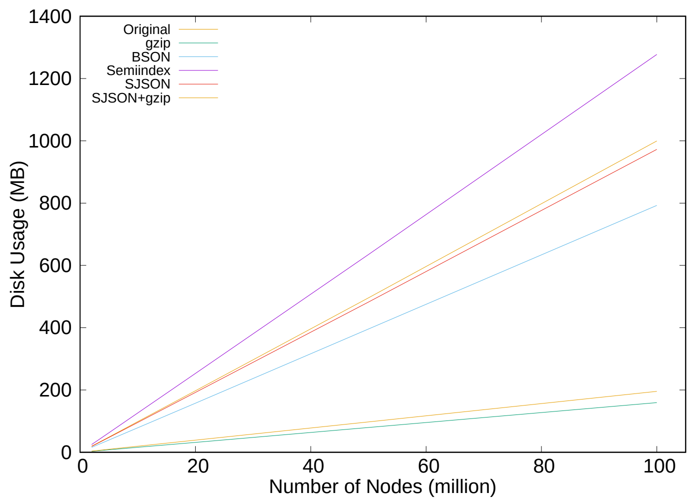
4.4. Disk Usage and Serialization Time

In Figures 13 and 14 we illustrate the disk usage of SJSON compared to the original file size and to *gzip*. Our scheme is able to compress a JSON document by up to 61% in synthetic files, and up to 28% in real-world corpora. From the figure, we can observe that disk usage is also proportional to the number of elements in the document. SJSON effectively reduces file size especially for `array` and `object`.

Although our compression is not as good as *gzip* with sizes about 2 to 9 times larger, it is easier to reload the compressed file generated by SJSON back to memory than to decompress and parse the *gzip*ped file. Note that once deserialized in RAM, we do not need to maintain the content stored on disk for future use. We also provide *gzip*ped result of the SJSON serialization, which



(a) $n = 10,000,000$.



(b) Varying n.

Figure 13: Disk usage of SJSON compared to the original file size and to *gzip*, for synthetic data.

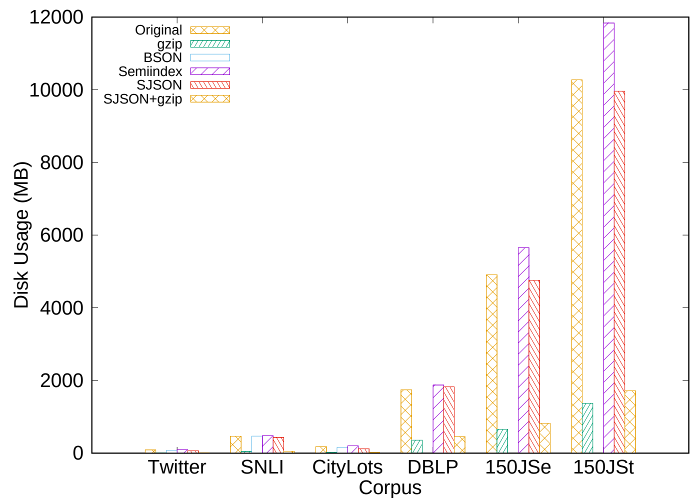


Figure 14: Disk usage of SJSON compared to the original file size and to *gzip*, for real world corpora.

further reduces the disk usage without penalizing the performance.

It is shown in the figure that BSON decreases disk usage by up to 33%. Unfortunately, the BSON library provided by MongoDB was not able to convert most of the real-world datasets, since it only supports UTF-8 characters.

Corpus	Time (s)
`array`	0.148
`bool`	0.204
`double`	0.255
`int`	0.238
`null`	0.214
`object`	0.211
`string`	0.338

Corpus	Time (s)
`Twitter`	0.168
`SNLI`	1.345
`Citylots`	0.472
`DBLP`	3.168
`150JS-evaluation`	11.17
`150JS-training`	52.76

Table 5: Serialization time of SJSON.

Table 5 summarizes serialization time of SJSON processed result. We can see that time needed to serialize corpora is proportional to their size.

4.5. Query Time

In Section 3.3 several types of queries are discussed, and SJSON implements most of those emulated as tree operations supported in the SDSL library.

Corpus	`listObjNames`	`getObjValue`	`cntArrElems`	`getArrValue`
`string` ($n = 10,000,000$)	209	17.2	-	-
`Twitter`	89.3	16.8	9.8	19.8
`SNLI`	106	17.2	10.3	20.7
`CityLots`	135	16.7	9.8	20.6
`DBLP`	131	16.8	10.2	19.7
`150JS-evaluation`	92.6	17.1	9.9	20.9
`150JS-training`	94.4	17.3	10.1	20.8

Table 6: Query time of SJSON. Units are in microseconds.

Table 6 exhibits the query time experimental results of SJSON. Queries are invoked in various document locations, and their average time is calculated. For some corpora where arrays do not exist, only the object queries are run in the experiments. The time is mostly the same regardless of the location each query handles, because tree-navigational queries take constant time. Additionally, pointing the exact location in the bit indexed array takes constant time as well, by the constant-time implementation `rank` and `select` operations. For the `listObjNames` queries, the actual experimental time is highly affected by the degree of each element accessed.

We have also emulated the relevant queries as the native operations supported by the other libraries – JsonCpp, JSON for Modern C++, and RapidJSON. Figure 15 shows comparison of query time in the `Twitter` corpus among the four libraries. Since the succinct tree representation are slower in supporting the tree navigational operations compared to the pointer-based representations, our representation mostly gives the worse performance in query time. Nonetheless, the SJSON library allows querying through a large document which other

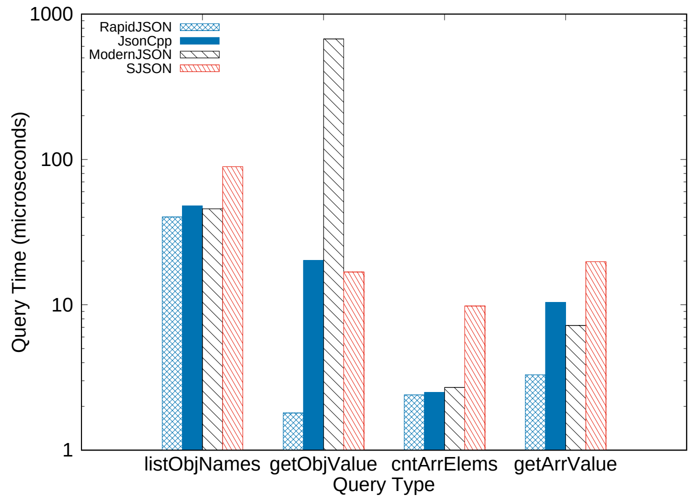


Figure 15: Query time of SJSON in the `Twitter` corpus, compared to different libraries.

frameworks fail to process, with almost identical processing time regardless of the size of the document.

Semi-index supports retrieval of values in an arbitrary location when a name is given. This is done by traversing the whole tree with the assistance of the constructed index. Although our library does not explicitly support the whole traversal as of now, it is remarkable that emulation of traversal would guarantee similar query time to that of semi-index.

4.6. Splitting the Document into a Collection of Chunks

Even though maintaining large size documents is one of the merits of our representation, to improve the RAM usage further, we also experimented by splitting a large JSON document into a collection of smaller chunks. More specifically, we performed several experiments based on chunk division, where a single JSON document acting as a corpus is divided into multiple smaller chunks. Each chunk has its own concrete representation of a tree and arrays, while all the chunks are connected as children of a virtual root node. (In all our datasets,

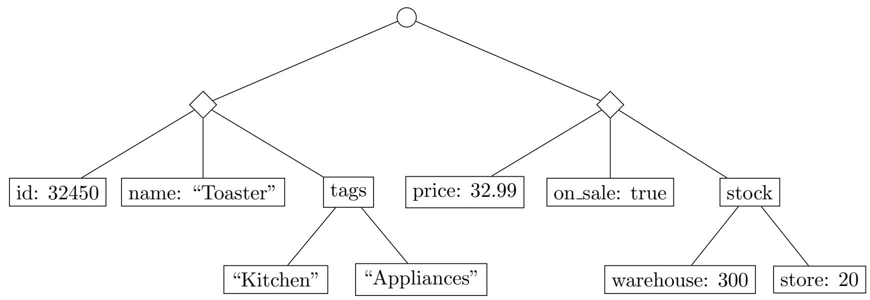


Figure 16: Document tree structure corresponding to the sample JSON document shown in Figure 2 divided into 2 chunks.

Corpus	Time (s)	Ratio
`string` ($n = 10,000,000$)	5.29	1.11
`SNLI`	7.78	1.13
`DBLP`	43.8	1.26

Table 7: Construction time of SJSON with chunk division enabled.

the tree structure is fairly shallow, with small depth, and hence this simple modification is enough to split the large document into several smaller chunks.) This enables efficient RAM usage and large-scale document processing because we do not need to maintain and store all the intermediate representations to make queries work. Nevertheless, this may increase the disk size, given that representations of chunks do not share pre-constructed `names` and `stringValues` arrays. By adding a virtual root node to the entire tree structure, each part of a document is considered as a child tree with its own suite of arrays. Figure 16 exhibits a document tree structure consisting of two JSON chunks.

We tested the effect of chunk division using both synthetic and real-world corpora. For synthetic dataset, we divided the string corpus ($n = 10,000,000$, disk size 111MB) into 10 chunks. For real-world corpora, we chose `SNLI` and `DBLP`, each divided into 50 and 100 chunks, respectively.

Since no intermediate procedure other than the serialization is needed, the construction time is almost identical to that of the original version, as in Table 7.

Corpus	RAM Usage (MB)	Ratio
`string` ($n = 10,000,000$)	29	0.18
`SNLI`	29	0.06
`DBLP`	62	0.03

Table 8: Memory usage of SJSON with chunk division enabled.

Corpus	Disk Usage (MB)	Ratio
`string` ($n = 10,000,000$)	144	1.23
`SNLI`	496	1.09
`DBLP`	1,877	1.03

Table 9: Disk usage of SJSON with chunk division enabled.

From Table 8 it is clear that chunk division allows only a portion of RAM is needed to process the whole document. This intermediate representation is flushed to disk, so only a small amount of RAM is required even for a big JSON document. Note that these ratios are not inversely proportional to the number of chunks, since duplicate values among two individual chunks are not considered identical in the representation.

We have noticed negligible serialization and query time difference from the original representation since only one extra tree operation needs to be done. For queries, we assume that the entire data structure is already loaded into the RAM so that no extra de-serialization is needed during query processing. But as one can imagine, if the representation is not in the RAM, then the chunk division approach will support the queries significantly faster as it only needs to load a small portion of the data structure into the RAM to answer the query.

4.7. String Compression

We also integrated compressed suffix array data structure to the SJSON library, so that string values could be compressed efficiently while naive query support is guaranteed. In this subsection, we illustrate the details when the string compression is enabled, by comparing the experimental result to the

Corpus	Time (s)	Ratio
`string` ($n = 10,000,000$)	11.2	2.35
`Twitter`	13.3	5.99
`SNLI`	135	18.3
`CityLots`	17.7	1.61

Table 10: Construction time of SJSON with string compression enabled.

original representation.

Table 10 denotes construction time when string compression is applied to some of the corpora. Following the tendency of the theoretic time bounds suggested in Section 2.4, when a JSON document contains a large portion of strings, constructing its compressed suffix array takes most of the construction time. One alternative way to compress the `stringValues` array is to apply general-purpose compression schemes, however, the core penalty of this method is that the compressed form does not support random access without explicit decompression, which is significant overhead while querying.

Corpus	Disk Usage (MB)	Ratio
`string` ($n = 10,000,000$)	133	1.14
`Twitter`	49	0.72
`SNLI`	240	0.53
`CityLots`	106	0.95

Table 11: Disk usage of SJSON with string compression enabled.

Additional string compression using compressed suffix arrays drastically decreases the overall disk size – even competitive to *gzip*ped compression – if the original corpus contains a high portion of strings, summarized in Table 11. We claim that most JSON documents contain a large number of strings so that applying string compression to those guarantees less disk usage.

If no string compression is applied, extracting an arbitrary string value from the `stringValues` array does not rely on the length of the value. Nevertheless,

Corpus	`listObjNames`	`getObjValue`	`cntArrElems`	`getArrValue`
`string` ($n = 10,000,000$)	216	152	-	-
`Twitter`	92	158	10.6	143
`SNLI`	108	195	10.7	215
`CityLots`	137	144	11.1	189

Table 12: Query time of SJSON with string compression enabled. Units are in microseconds.

as dealt in Section 2.4, extracting a string from the compressed suffix array takes linear time proportional to the desired length of the string. This affects query time, which is illustrated in Table 12. Answers to the queries had 6 to 8 characters in average.

5. Conclusion and Future Work

JSON is the de facto prevalent data interchange document format for data transmission on web service APIs, besides being used to store or represent structured data in many modern software systems. However, no well-known queriable compression scheme tailored for JSON exists yet. In this paper, we have engineered and implemented SJSON, a succinct representation and compression scheme for parsing and storing JSON documents in a memory-efficient manner.

Our library is able to consistently represent JSON documents across a range of synthetic and real-world datasets in up to 91% less RAM compared to popular libraries, most often using less space than the original file size. Furthermore, the empirical analysis shows that SJSON generated compressed files occupy up to 41% smaller space than the original document. In addition to the merits above, by using succinct tree structures, this suggested representation supports traversal and retrieval queries efficiently.

There still are details that could be improved in the library to further enhance its performance in time efficiency and conciseness.

Supporting Other Formats. Moreover, it would be feasible to support other semi-structured document formats in the library, such as XML described before.

For instance, XMill [25], TREECHOP [26], XQueC [27] and XBZipIndex [28] point out compression and querying in XML documents.

Dynamic and Online Modification. The second point of interest for future work would be to enrich the functionality to process JSON documents online and dynamic. Since data corpus grows in the era of big data, there are needs to add, remove, or alter elements of a JSON document occasionally. A naive method to support dynamic document processing is to rewrite the original JSON document and reconstruct the whole representation once per relevant modification, which is a significantly burdensome action from both memory and disk perspectives. For the succinct tree representation, utilizing dynamic representation could be an option to consider [52]. Manipulating the bit string indexed array needs several subprocedures. A core reason is that in most cases size of the modified value may differ from its original one, so re-indexing of the array is required.

Solving these future works would improve preprocessing time and make the library more appealing from a user standpoint.

References

[1] J. James, Data never sleeps 3.0, https://www.domo.com/blog/2015/08/data-never-sleeps-3-0/ (2015).

[2] P. B. Brandtzæg, Big data, for better or worse: 90% of world's data generated over last two years, https://www.sciencedaily.com/releases/2013/05/130522085217.htm (2013).

[3] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (3) (1977) 337–343.

[4] G. J. Jacobson, Succinct static data structures, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, aAI8918056 (1988).

[5] J. I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM Journal on Computing 31 (3) (2001) 762–776.

[6] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, S. S. Rao, Representing trees of higher degree, Algorithmica 43 (4) (2005) 275–292.

[7] J. Fischer, V. Heun, A new succinct representation of rmq-information and improvements in the enhanced suffix array, in: International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies, Springer, 2007, pp. 459–470.

[8] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2005, pp. 45–56.

[9] T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, F. Yergeau, Extensible markup language (xml) 1.0., World Wide Web Consortium.

[10] T. Bray, The javascript object notation (json) data interchange format, RFC 7159, RFC Editor (March 2014).
URL `https://www.rfc-editor.org/rfc/rfc7159.txt`

[11] J. C. Anderson, J. Lehnardt, N. Slater, CouchDB: The Definitive Guide Time to Relax, 1st Edition, O'Reilly Media, Inc., 2010.

[12] K. Chodorow, M. Dirolf, MongoDB: The Definitive Guide, 1st Edition, O'Reilly Media, Inc., 2010.

[13] Bson (binary json): Specification.
URL `http://bsonspec.org/spec.html`

[14] T. Inc., Twitter developers documentation on rest apis, https://dev.twitter.com/rest/public (2016).

[15] I. Google, Using json in the google data protocol, https://developers.google.com/gdata/docs/json (2014).

[16] G. Ottaviano, R. Grossi, Semi-indexing semi-structured data in tiny space, in: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, ACM, 2011, pp. 1485–1494.

[17] T. Rincy, R. Rajesh, Space efficient structures for json documents, International Journal of Computer Engineering and Technology 5 (12) (2014) 148–153.

[18] T. C. Casas, Jsonc-json compressor and decompressor, https://github.com/tcorral/jsonc (2015).

[19] L. P. Deutsch, Deflate compressed data format specification version 1.3.

[20] G. Jacobson, Space-efficient static trees and graphs, in: 30th Annual Symposium on Foundations of Computer Science, IEEE, 1989, pp. 549–554.

[21] A. Doan, A. Halevy, Z. Ives, Principles of Data Integration, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[22] F. Maddix, Books, http://www.cems.uwe.ac.uk/ fj-maddix/Books.xml.

[23] J. Clark, S. DeRose, et al., Xml path language (xpath) version 1.0., World Wide Web Consortium.

[24] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, P. Wadler, Xquery 1.0 and xpath 2.0 formal semantics, World Wide Web Consortium.

[25] H. Liefke, D. Suciu, Xmill: an efficient compressor for xml data, in: ACM Sigmod Record, Vol. 29, ACM, 2000, pp. 153–164.

[26] G. Leighton, T. Müldner, J. Diamond, Treechop: a tree-based query-able compressor for xml, in: Proceedings of the Ninth Canadian Workshop on Information Theory (CWIT 2005), Citeseer, 2005, pp. 115–118.

[27] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, A. Pugliese, Xquec: Pushing queries to compressed xml data, in: Proceedings of the

29th International Conference on Very large Data Bases, VLDB Endowment, 2003, pp. 1065–1068.

[28] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and searching xml data via two zips, in: Proceedings of the 15th International Conference on World Wide Web, ACM, 2006, pp. 751–760.

[29] A. Kapoulkine, pugixml, https://pugixml.org (2006).

[30] O. Delpratt, S. Joannou, N. Rahman, R. Raman, The sixml project: Sixdom 1.2.

[31] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan, E. J. Shekita, Jaql: A scripting language for large scale semistructured data analysis, in: Proceedings of VLDB Conference, 2011.

[32] D. Florescu, G. Fourny, Jsoniq: The history of a query language, IEEE Internet Computing 17 (5) (2013) 86–90. `doi:http://doi.ieeecomputersociety.org/10.1109/MIC.2013.97`.

[33] D. Clark, Compact pat trees.

[34] J. I. Munro, Tables, in: International Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 1996, pp. 37–42.

[35] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct trees in practice, in: Proceedings of the Meeting on Algorithm Engineering & Expermiments, Society for Industrial and Applied Mathematics, 2010, pp. 84–97.

[36] J. Jansson, K. Sadakane, W.-K. Sung, Ultra-succinct representation of ordered trees with applications, Journal of Computer and System Sciences 78 (2) (2012) 619–631.

[37] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE transactions on Information Theory 24 (5) (1978) 530–536.

[38] T. A. Welch, A technique for high-performance data compression, Computer 17 (6) (1984) 8–19.

[39] P. Weiner, Linear pattern matching algorithms, in: 14th Annual Symposium on Switching and Automata Theory, IEEE, 1973, pp. 1–11.

[40] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing 22 (5) (1993) 935–948.

[41] R. Grossi, J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM Journal on Computing 35 (2) (2005) 378–407.

[42] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM (JACM) 52 (4) (2005) 552–581.

[43] M. Burrows, D. J. Wheeler, A block-sorting lossless data compression algorithm.

[44] R. Grossi, A. Gupta, J. S. Vitter, High-order entropy-compressed text indexes, in: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2003, pp. 841–850.

[45] F. Claude, G. Navarro, A. Ordónez, The wavelet matrix: An efficient wavelet tree for large alphabets, Information Systems 47 (2015) 15–32.

[46] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms (TALG) 3 (2) (2007) 20.

[47] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proceedings of the Meeting on Algorithm Engineering & Expermiments, Society for Industrial and Applied Mathematics, 2007, pp. 60–70.

[48] E. Anjos, J. Lee, S. R. Satti, Sjson: A succinct representation for javascript object notation documents, in: 2016 Eleventh International Conference on Digital Information Management (ICDIM), IEEE, 2016, pp. 173–178.

[49] Tencent, Rapidjson, https://github.com/miloyip/rapidjson (2015).

[50] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: Plug and play with succinct data structures, in: 13th International Symposium on Experimental Algorithms (SEA 2014), 2014, pp. 326–337.

[51] S. Vigna, Broadword implementation of rank/select queries, in: International Workshop on Experimental and Efficient Algorithms, Springer, 2008, pp. 154–168.

[52] G. Navarro, K. Sadakane, Fully functional static and dynamic succinct trees, ACM Transactions on Algorithms (TALG) 10 (3) (2014) 16.

[53] B. Lepilleur, JsonCpp, https://github.com/open-source-parsers/jsoncpp (2016).

[54] N. Lohmann, JSON for Modern C++, https://github.com/nlohmann/json (2016).

[55] S. R. Bowman, G. Angeli, C. Potts, C. D. Manning, A large annotated corpus for learning natural language inference, arXiv preprint arXiv:1508.05326.

[56] M. Zeiss, City lots san francisco, https://github.com/zeMirco/sf-city-lots-json (2012).

[57] E. Demaine, M. Hajiaghayi, Bigdnd: Big dynamic network data, http://projects.csail.mit.edu/dnd/DBLP/ (2014).

[58] M. Ley, The dblp computer science bibliography: Evolution, research issues, perspectives, in: International Symposium on String Processing and Information Retrieval, Springer, 2002, pp. 1–10.

[59] V. Raychev, P. Bielik, M. Vechev, A. Krause, Learning programs from noisy data, in: ACM SIGPLAN Notices, Vol. 51, ACM, 2016, pp. 761–774.