Semi-Indexing Semi-Structured Data in Tiny Space

Giuseppe Ottaviano
Dipartimento di Informatica
Università di Pisa
ottavian@di.unipi.it

Roberto Grossi
Dipartimento di Informatica
Università di Pisa
grossi@di.unipi.it

ABSTRACT

Semi-structured textual formats are gaining increasing popularity for the storage of document collections and rich logs. Their flexibility comes at the cost of having to load and parse a document entirely even if just a small part of it needs to be accessed. For instance, in data analytics massive collections are usually scanned sequentially, selecting a small number of attributes from each document.
We propose a technique to attach to a raw, unparsed document (even in compressed form) a "semi-index": a succinct data structure that supports operations on the document tree at speed comparable with an in-memory deserialized object, thus bridging textual formats with binary formats.
After describing the general technique, we focus on the JSON format: our experiments show that avoiding the full loading and parsing step can give speedups of up to 12 times for on-disk documents using a small space overhead.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; E.4 [**Coding and Information Theory**]: Data Compaction and Compression

General Terms

Algorithms, Performance

Keywords

Semi-index, Semi-structured data, Succinct data structures

1. INTRODUCTION

Semi-structured data formats have enjoyed popularity in the past decade and are virtually ubiquitous in Web technologies: extensibility and hierarchical organization—as opposed to flat tables or files—made them the format of choice for documents, data interchange, document databases, and configuration files.

CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

The field of applications of semi-structured data is rapidly increasing. These formats are making their way into the realm of storage of *massive* datasets. Their characteristics of being schema-free makes them a perfect fit for the mantra *"Log first, ask questions later"*, as the document schema is often evolving. Natural applications are crawler logs, query logs, user activity in social networks, to name a few.

In this domain JSON (*JavaScript Object Notation*, see [21]) in particular has been gaining momentum: the format is so simple and self-evident that its formal specification fits in a single page, and it is much less verbose than XML. In fact, both CouchDB[4] and MongoDB [24], two of the most used ([5, 25]) modern large-scale distributed schema-free document databases, are based on JSON, and Jaql [19] and Hive JSON SerDe [15] implement JSON I/O for Hadoop. These systems all share the same paradigm:

(a) Data is conceptually stored as a sequence of records, where each record is represented by a single (schema-free) JSON document.

(b) The records are processed in MapReduce [7] fashion: during the Map phase the records are loaded sequentially and parsed, then the needed attributes are extracted for the computation of the Map itself and the subsequent Reduce phase.

In part (b) the extracted data is usually a *small* fraction of the records actually loaded and parsed: in logs such as the ones mentioned above, a single record can easily exceed hundreds of kilobytes, and it has to be loaded entirely even if just a single attribute is needed. If the data is on disk, the computation time is dominated by the I/O.

A typical way of addressing the problem of parsing documents and extracting attributes is to change the data representation by switching to a *binary*, easily traversable format. For instance XML has a standard binary representation, Binary XML ([3]), and more sophisticated schemes that enable more powerful traversal operations and/or compression have been proposed in the literature (see [30]). Likewise MongoDB uses BSON, a binary representation for JSON. However, switching from a textual format to an ad-hoc binary format carries some drawbacks.

- In large-scale systems, the producer is often decoupled from the consumer, which gets the data through append-only or immutable, possibly compressed, distributed filesystems, so using simple self-evident standard formats is highly preferable.

- Binary data is not as easy to manually inspect, debug or process with scripting languages as textual data.
- If input/output is textual, back-and-forward conversions are needed.
- If existing infrastructure is based on textual formats, changing the storage format of already stored data can be extremely costly.

In fact, despite their advantages binary formats have not gained widespread adoption.

Surprisingly, even with a binary format it is not easy to support all the tree operations without incurring a significant space overhead. For example, BSON prepends to each element its size in bytes, enabling fast forward traversal by allowing to "skip" elements, but accessing the ith element of an array cannot have sublinear I/O complexity.

Our contribution. In this paper we introduce the notion of *semi-indexing* to speed up the access to the attributes of a textual semi-structured document without altering its storage format; instead, we accompany it with a small amount of redundancy.

A *semi-index* is a succinct encoding of the parse tree of the document together with a positional index that locates the nodes of the tree on the unparsed document. Navigation of the document is achieved by navigating the succinct parse tree and parsing on the fly just the leaf nodes that are needed, by pointing the parser at the correct location through the positional index. This way, a small part of the document has to be accessed: the I/O time is greatly reduced if the documents are large, and on a slow medium such as a disk or a compressed or encrypted filesystem. Specifically, the I/O time is proportional to the number of tree queries, regardless of the document size.

No explicit parsing tree is built, instead we employ a well-known balanced parenthesized representation and a suitable directory built on the latter. The resulting encoding is so small that it can be computed once and stored along with the document, without imposing a significant overhead.

We call our approach "semi-index" because it is an index on the *structure* of the document, rather than on its *content*: it represents a middleground between full indexing (where the preprocessing time and space can be non-negligible because the full content is indexed) and streaming (where data are not indexed at all).

The main novelty is that the document in its textual semi-structured format (or *raw data*) is not altered in any way, and can be considered a *read-only random access oracle*. The combination of *raw data + semi-index* can thus support the same operations as an optimized binary format, while maintaining the advantages of keeping the raw data unaltered.

- Backward-compatibility: Existing parsers can just ignore the semi-index and read the raw data.
- The semi-index does not need to be built by the producer: the consumer can build/cache it for later use.
- The raw data does not need to be given in explicit form, provided that a random-access primitive is given, while the semi-index is small enough that it can easily fit in fast memory. For example a compression format with random access can be used on the documents. We demonstrate this feature in the experimental analysis by compressing blockwise the data with *zlib*.

A semi-index can be engineered in several ways depending on the format grammar and the succinct data structures adopted for the purpose. Although the semi-indexing schema is general, we focus on a concrete implementation using JSON as the underlying format for clarity.

In our experiments (Section 6) we show that query time is very fast, and speedup using the precomputed semi-index on a MapReduce-like computation ranges from 2 to 12 times. Using a block-compressed input file further improves the running time when the document size is large, by trading I/O time for CPU time. This comes at the cost of a space overhead caused by the storage of the semi-index, but on our datasets the overhead does not exceed (and is typically much less than) around 10% of the raw data size.

When comparing against the performance of BSON, our algorithm is competitive on some datasets and better on others. Overall, raw data + semi-index is never worse than BSON, despite the latter is an optimized binary format.

To our surprise, even if the semi-index is built on the fly right before attribute extraction, it is faster than parsing the document: thus semi-indexing can be also thought of as a fast parsing algorithm.

The main drawback of the semi-index is that it has a fixed additive overhead of 150–300 bytes (depending on the implementation), making it unsuitable for very small individual documents. This overhead can be however amortized for a *collection* of documents. In our experiments we follow this approach.

In summary, our contribution in this paper is to show how to exploit our notion of semi-indexing to speed up sequential access to a collection of semi-structured documents in a very simple way. We believe that our paradigm is quite general and can be applied to other formats as well.

Paper organization. In Section 2 we compare our approach with existing literature. In Section 3 we review the tools and notations used in the paper. In Section 4 we overview a general technique to build and query the semi-index. In Section 5 we describe a specific representation for JSON, adopting some data structures from the state of the art ([2, 29]). In Section 6 we discuss the experimental results and the practicality of the approach. Finally, in sections 7 and 8 we introduce a different application of the technique and conclude giving future work directions.

2. RELATED WORK

A similar approach for comma-separated-values files is presented in [17]. The authors describe a database engine that skips the ordinary phase of loading the data into the database by performing queries directly on the flat textual files. To speed up the access to individual fields, a (sampled) set of pointers to the corresponding locations in the file is maintained, something similar to our *positional index*. This approach, however, is suitable only for tabular data.

Virtually all the work on indexing semi-structured data focuses on XML, but most techniques are easily adaptable to other semi-structured data formats, including JSON. For example, AgenceXML [6] and MarkLogic[16] convert JSON documents internally into XML documents, and Saxon [23] plans to follow the same route.

To the best of our knowledge, no work has been done on indexes on the *structure* of the *textual* document, either for XML or other formats. Rather, most works focus on providing indexes to support complex tree queries and queries

on the *content*, but all of them use an ad-hoc binary representation of the data (see [13] for a survey on XML indexing techniques).

For the storage of XML data several approaches were proposed that simultaneously compress XML data while supporting efficient traversal, and they usually exploit the separation of tree structure and content (see [30] for a survey on XML storage schemes).

Some storage schemes employ succinct data structures: for example [8] uses a succinct tree to represent the XML structure, and [12] exploits compressed non-binary dictionaries to encode both the tree structure and the labels, while supporting subpath search operations.

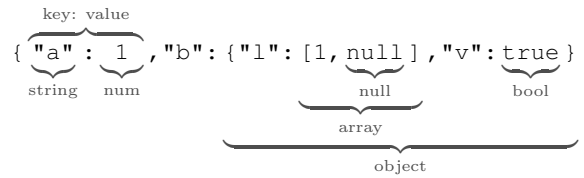
The work in [35] is the closest to this paper, as it uses a balanced-parentheses succinct tree representation of the document tree, but like the others it re-encodes the contents of the document to a custom binary format and discards the unparsed form.

Industrial XML parsers such as Xerces2 [1] keep in memory only a summary of the tree structure with pointers to the textual XML, while parsing only the elements that are needed. This technique, known as Lazy XML parsing, needs however to scan the full document to parse the tree structure every time the document is loaded, hence the I/O complexity is no better than performing a full parse. Refinements of this approach such as double-lazy parsing [11] try to overcome the problem by splitting the XML file in several fragments stored in different files that link to each other. This however requires to alter the data, and it is also XML-specific. Besides, each fragment that is accessed has to be fully scanned. Semi-indexing is similar to lazy parsing in that a pre-parsing is used to speed up the access to the document, but the result of semi-index preprocessing is small enough that can be saved along the document, while in lazy parsing the preprocessing has to be done every time the document is loaded.

3. BACKGROUND AND TOOLS

3.1 JSON format

JSON (JavaScript Object Notation) is a small fragment of the Javascript syntax used to represent semi-structured data. A JSON *value* can either be an atom (i.e. a string, a number, a boolean, or a *null*), an *object*, or an *array*. An object is an unordered list of key/value pairs, where a key is a string. An array is an ordered list of values (so one can ask for the ith value in it). A JSON *document* is just a value, usually an object. The following figure shows an example of a JSON document and its parsing.



The document tree of a JSON value is the tree where the leaf nodes are the atoms and internal nodes are objects and arrays. The queries usually supported on the tree are the basic traversal operations, i.e. *parent* and *ith* child (and *labeled child* for objects). We use the Javascript notation to denote path queries, so for example in this document `a` is `1` and `b.l[1]` is `null`.

3.2 Succinct data structures

To encode and query the semi-index we employ succinct data structures. A succinct data structure stores the input data in the informational theoretical minimum number of bits, and still supports some given operations in constant time. We use two such data structures.

Elias-Fano encoding. The *Elias-Fano* representation of monotone sequences [9, 10] is an encoding scheme to represent a non-decreasing sequence of m integers in $[0..n)$ occupying $2m+m\left\lceil \log \frac{n}{m} \right\rceil + o(m)$ bits, while supporting constant-time access to the ith integer. The representation can be used to represent sparse bitvectors (i.e. where the number m of `1`s is small with respect to the size n of the bitvector), by encoding the sequence of the positions of the `1`s. In fact, the representation can support all the operations defined by Fully Indexable Dictionaries (FID, see [28]).

By analogy to FIDs, we call the access operation to the ith integer Select(i), and we denote its implementation in the pseudocode by `select`.

The scheme is very simple and elegant, and efficient practical implementations are described in [14, 27, 33][1].

BP. It is an acronym for *balanced parentheses*. They are inductively defined as follows: an empty sequence is BP; if α and β are sequences of BP, then also `(`α`)`β is a sequence of BP, where `(` and `)` are called *mates*. For example, `((()(()()))` is a sequence of BP. Note that a sequence of BP implicitly represents a tree, where each node corresponds to a pair of mates. BP sequences are represented as bitvectors, where `1` represents `(` and `0` represents `)`.

A sequence S of $2m$ BP can be encoded in $2m + o(m)$ bits [18, 26] so that the following operations, among others, are supported in constant or nearly-constant time.

- Access(i) returns $S[i]$; we denote its implementation with the square brackets operator `[]`.
- FindClose(i), for a value i such that $S[i] =$`(`, returns the position $j > i$ such that $S[j] =$`)` is its mate; we denote its implementation by `find_close`.
- FindOpen(i), for a value i such that $S[i] =$`)`, returns the position $j < i$ such that $S[j] =$`(` is its mate.
- $\text{Rank}_{(}(i)$ returns the pre-order index of the node corresponding to the parenthesis at position i and its mate; note that this is just the number of open parentheses before i, and we denote its implementation by `rank`.
- Excess(i) returns the difference between the number of `(`s and that of `)`s in the first $i+1$ positions of S. Note that since the parentheses are balanced this value is always non-negative, and it is easy to show that it equals $2\text{Rank}_{(}(i) - i$.
- Child(i, q) returns the parenthesis that opens the qth child of the node represented by the open parenthesis at position i.

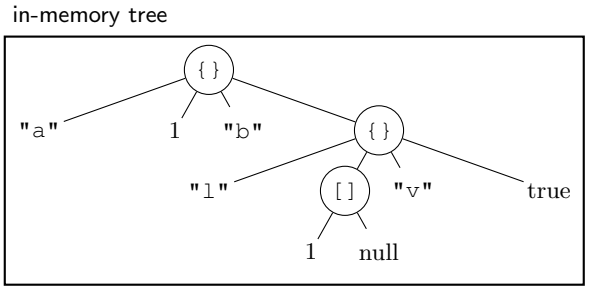
4. SEMI-INDEXING TECHNIQUE

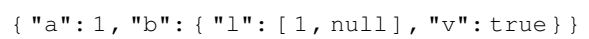
We illustrate our technique with the JSON document shown in the example of Section 3.1.

The most common way of handling textual semi-structured data is to *parse* it into an in-memory tree where the leaf

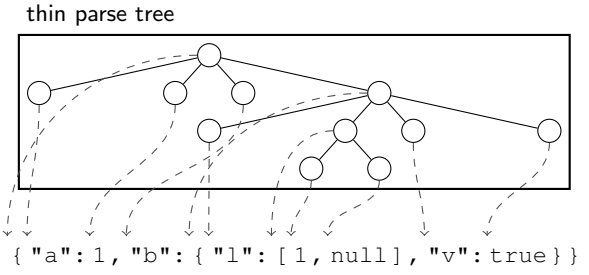
[1] In [27] the data structure is called *SDArray*.

nodes contain the parsed atoms. The tree is then queried for the requested attributes. Since the tree contains all the relevant data, the raw document is no longer needed, as shown in the figure below.



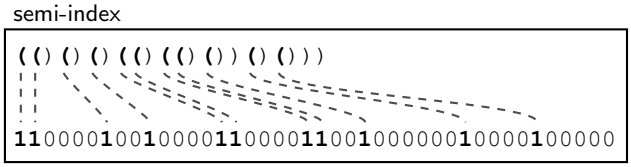


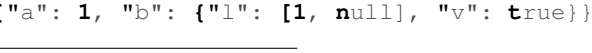
We would like to create a structure that allows us to navigate the document *without having to parse it*. One possible approach could be to dump the parse tree, and to replace the values in the nodes with *pointers* to the first character of their corresponding phrases in the document. This clearly requires to store also the raw data along with the resulting "thinner" tree.



We can now navigate the parse tree, and parse just the values corresponding to the leaves that are actually needed. Note that if the grammar is LL(1)[2], we do not need to store the node type: it suffices to look at the first character of the node to recognize the production. So we are left with the tree data structure representing the topology of the parse tree, and a pointer for each node to the first character of its phrase in the raw text. This is very similar to the approach adopted by lazy parsers for XML.

Still, this requires building explicitly the parse tree every time the document is loaded. Instead, we will show how a quick scan of the document is sufficient to produce a sequence of balanced parentheses plus a bitvector as shown in the figure below (see Section 5 for details).

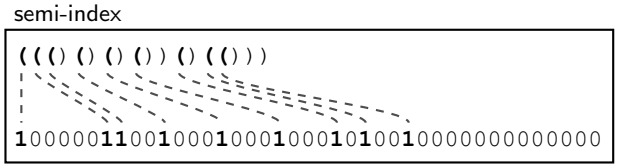


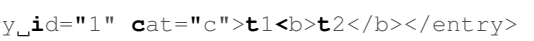


[2]Most semi-structured data formats are LL(1), including XML and JSON. If the grammar is not LL(1) an additional $\log P$ bits per node may be needed, where P is the number of productions.

This way we encode the same information as the thinner parse tree without parsing the document and building its parse tree. As we discuss in the experiments of Section 6, scanning is much faster than parsing. We merely store two binary sequences, encoding each parenthesis with a single bit, augmented with the machinery to support the operations described in Section 3.2. Besides, the binary sequence with the positions is sparse, so easy to encode in compressed format using very small space. Thanks to their small encoding, the two sequences can be computed just once and then be stored for future use.

This scheme can be applied to other formats. For instance, the XML semi-index would look like the following figure.





Our approach is to employ a set of succinct structures to replace the functionality of the thinner parse tree, and obtain faster construction and query time (see Section 6). We can thus define the *semi-index*.

Definition 4.1 *A* semi-index *for a document D is a succinct encoding of (i) the topology of the parse tree T of D, and (ii) the pointers that originate from each node of T to the beginning of the corresponding phrase in D.*

Let m denote the number of nodes in T. The general template to build the semi-index using an event parser is illustrated in Algorithm 1[3]. By *event parser* we mean a parser that simulates a depth-first traversal of the parse tree, generating an `open` event when the visit enters a node and a `close` event when it leaves it. (An example is the family of SAX parsers for XML.) If the event parser uses constant memory and is one-pass, so does Algorithm 1. Thus it is possible to build the semi-index without having to build an explicit parse tree in memory. In the pseudocode, (i) `bp` is the balanced parentheses tree structure, and (ii) `positions` is the Elias-Fano representation of the pointers.

```
def build_semi_index(s):
    positions = EliasFanoSequence()
    bp = BalancedParentheses()
    for event, pos in parse_events(s):
        if event == 'open':
            bp.append(1)
            positions.append(pos)
        elif event == 'close':
            bp.append(0)
```

Algorithm 1: Construction of semi-index using an event parser

For the construction algorithm to be correct we need the following observation.

[3]The pseudocode is actually working Python code, but we omitted the auxiliary functions and classes for the sake of presentation.

Observation 4.2 *The sequence of pointers in a pre-order visit of the parse tree T induces a non-decreasing sequence of m positions in D. In other words, the sequence of positions of* `open` *events in an event parsing is non-decreasing.*

Observation 4.2 allows us to use the Elias-Fano encoding, whose implementation is referred to as `EliasFanoSequence` in Algorithm 1, for the positions.

Algorithm 2 shows the pseudocode for some tree operations. Operation `get_node_pos` returns the position of the phrase in the document D corresponding to the node of T represented by the parenthesis at position `par_idx`. Operation `first_child` returns the position of the parenthesis corresponding to the first child of the current node. Operation `next_child` returns the position of the next sibling (if any).

```
def get_node_pos(bp, positions, par_idx):
    node_idx = bp.rank(node)
    pos = positions.select(node_idx)
    return pos

def first_child(bp, par_idx):
    return par_idx + 1

def next_sibling(bp, par_idx):
    next = bp.find_close(par_idx) + 1
    if bp[next]:
        return next
    else:
        return None
```

Algorithm 2: Some tree operations on the semi-index

We now discuss the space usage in our encoding. As shown in our example, the tree topology can be encoded with the balanced parentheses representation, thus taking $2m+o(m)$ bits. Nodes are identified by the open parentheses, so that $\text{Rank}_(i)$ gives the pre-order index of node i.

The pointers can be thus encoded in pre-order by using the Elias-Fano representation, taking another $2m+m\left\lceil \log \frac{n}{m} \right\rceil + o(m)$ bits. Summing the two figures leads to the following lemma.

Lemma 4.3 *A semi-index of a document D of n bytes such that the parse tree T has m nodes can be encoded in*

$$4m + m\left\lceil \log \frac{n}{m} \right\rceil + o(m) \qquad (1)$$

bits, while supporting each of the tree navigational operations in $O(1)$ time.

The bound in (1) compares favorably against an explicit representation of the tree T and its text pointers: even if space-conscious, it would require 2 node pointers plus one text pointer, i.e. $m(2\log m + \log n)$ bits. For example, for a reasonably sized 1MB document with density 0.2 (1 node for each 5 bytes on average), the size of the data structure would be 1.4MB, 140% of the document itself!

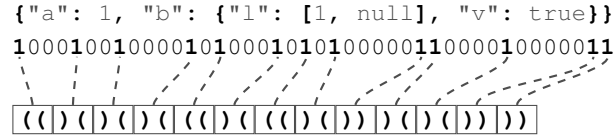
In practical implementation, the data structures that we adopt use approximately $5.5m + m\left\lceil \log \frac{n}{m} \right\rceil$ bits and have $O(\log n)$ complexity, but they are practically faster than data structures requiring theoretically constant time (see [2]). The encoding of the example above then takes 262kB, just 26.2% of the raw document. Even in case of a pathological high density document, i.e. $n = m$, the data structure would occupy $5.5m$ bits, i.e. an 68.7% overhead. Real-world documents, however, have very low densities (see Section 6).

5. ENGINEERING THE JSON *SEMI-INDEX*

In this section we describe a semi-index specifically tailored for JSON. It slightly deviates from the general schema presented in Section 4, since it exploits the simplicity of the JSON grammar to gain a few more desirable properties, as we will see shortly. As in the general scheme, we associate two bitvectors to the JSON document, `bp` and `positions`.

- The structural elements of the document, i.e. the curly brackets `{}`, the square brackets `[]`, the comma `,` and the colon `:` are marked in the bitvector `positions`, which is encoded with the Elias-Fano representation.
- For each structural element a pair of parentheses is appended to the `bp` (balanced parentheses) vector:
  - Brackets `{` and `[` open their own node (the *container*) and the first element of the list, so their encoding is `((`.
  - Brackets `}` and `]` close the last element of the list and their own node, so their encoding is `))`.[4]
  - Comma `,` closes the current element and opens the next, so its encoding is `)(`.
  - Colon `:` is treated like the comma, so key/value pairs are encoded simply as consecutive elements.

An example of the encoding is shown below: the JSON document (top), the `positions` bitvector (middle), and the `bp` bitvector (bottom). We implement `bp` as a binary sequence where `(` is encoded by `1`, and `)` is encoded by `0`.



This encoding allows a very simple algorithm for the construction of the semi-index: a one-pass scan of the document is sufficient, and it can be implemented in constant space (in particular, no stack is needed). As a result, building the semi-index is extremely fast.

```
def build_semi_index(json):
    positions = EliasFanoBitVector()
    bp = BalancedParentheses()
    i = 0
    while i < len(json):
        c = json[i]
        if c in '[{':
            positions.append(1)
            bp.extend([1, 1])
        elif c in '}]':
            positions.append(1)
            bp.extend([0, 0])
        elif c in ',:':
            positions.append(1)
            bp.extend([0, 1])
        elif c == '"':
            # consume_string returns the position of
            # the matching '"'
            new_i = consume_string(json, i)
            length = new_i - i + 1
            i = new_i
            positions.extend([0] * length)
        else:
            positions.append(0)
        i += 1
```

Algorithm 3: Construction of JSON semi-index

[4]The empty object `{}` and array `[]` have encoding `(())`, so they are special cases to be handled separately in navigation.

Our ad-hoc encoding gives us two further features. First, each bit **1** in the bitvector `positions` is in one-to-one correspondence with pairs of consecutive parentheses in `bp`: there is no need to support a Rank operation to find the position in `bp` corresponding to a **1** in `positions`, as it is sufficient to divide by 2. Second, since the positions of closing elements (**}**, **]**, **,**) are marked in `positions`, it is possible to locate in constant time both endpoints of the phrase that represents a value in the JSON document, not just its starting position.

Navigation inside a JSON document is as follows. Finding a key in an object is performed by iterating its subnodes in pairs and parsing the keys until the searched one is found. The pseudocode for this operation can be found in `object_get`, Algorithm 4.

```
def get_pos(node):
    pos = positions.select(node / 2)
    pos += node % 2
    return pos

def object_get(json, node, key):
    # if node is odd, it is a comma, so skip it
    node += node % 2
    opening_pos = get_pos(node)
    if json[opening_pos] != '{':
        # not an object
        return None
    # empty objects are a special case
    if json[opening_pos + 1] == '}':
        return None

    node = node + 1
    node_pos = opening_pos + 1

    while True:
        if bp[node] == 0:
            return None
        node_close = bp.find_close(node)
        # node position after the colon
        val_open = node_close + 1
        # check if current key matches
        if parse(json, node_pos) == key:
            return val_open
        # otherwise skip to next key/value pair
        val_close = bp.find_close(val_open)
        node = val_close + 1
        # skip the comma
        node_pos = get_pos(node) + 1
```

Algorithm 4: Get position and object child by key

The query algorithm makes a number of probes to the JSON document that is linear in the fanout of the object. This is not much of a problem since the fanout is usually small. Otherwise, if it is possible to ensure that the keys are in sorted order, binary search can be used to reduce the number of probes to the logarithm of the fanout.

Array access can be done similarly with forward iteration through FindClose, with backwards iteration by jumping to the parenthesis closing the container and iterating on the contents with FindOpen, or with the ith child if `bp` supports it. In any case, at most 3 accesses to the JSON document are made: the I/O complexity is constant even if the runtime complexity may be linear.

We remark that in the design of the engineered JSON semi-index we have chosen simplicity over theoretical optimality. In general, other space/time/simplicity tradeoffs can be achieved by composing together other succinct data structures chosen from the vast repertoire in the literature, thus giving rise to a wide range of variations of the semi-indexing framework.

6. EXPERIMENTAL ANALYSIS

In this section we discuss the experimental analysis of the semi-index described in Section 5. The benchmark is aimed at the task of attribute extraction described in Section 1.

- Each dataset consists in a text file whose lines are JSON documents. The file is read from disk.
- The query consists in a list of key/index paths, to define which we use the Javascript notation. For instance given the following document

  ```
  {"a": 1, "b": {"v": [2, "x"], "l": true}}
  ```

  the query `a,b.v[0],b.v[-1]` returns `[1, 2, "x"]`, i.e. the list of the extracted values encoded as a JSON list. Note that negative indices count from the end of the array, so `-1` is the last element.
- Each benchmark measures the time needed to run the query on each document of the dataset and write the returned list as a line in the output file.

Implementation and testing details. The algorithms have been implemented in C++ and compiled with g++ 4.4. The tests were run on a dual core Intel Core 2 Duo E8400 with 6MB L2 cache, 4GB RAM and a 7200RPM SATA hard drive, running Linux 2.6.35 - 64bit. Before running each test the kernel page caches were dropped to ensure that all the data is read from disk. When not performing sequential scan, the input files were memory-mapped to let the kernel load lazily only the needed pages. For the construction of the semi-index each dataset is considered as a single string composed by the concatenation of all the documents, so a single semi-index is built for each dataset and stored on a separate file. The positions in the positional index are thus absolute in the file, not relative to each single document.

The source code used for the experiments is available at the URL `https://github.com/ot/semi_index`.

Succinct data structures. For the Select bitvectors and the Elias-Fano encoding we implemented the broadword techniques described in [33], in particular `rank9` for the Rank (used in the Excess primitive in BP) and a one-level hinted binary search for the Select. For the balanced parentheses we implemented the Range Min-Max tree described in [2]. With respect to the parameters described in the paper, we traded some space for speed using smaller superblock sizes. The excess forward and backward search on 64-bit words, that is in the inner loop of all the tree navigation operations, was implemented using a combination of lookup tables and broadword techniques to eliminate the branches.

Document compression. To simulate the behavior on compressed file systems we implemented a very simple block compression scheme which we call `gzra` (for gzipped "random access"). The file is split into 16kB blocks which are compressed separately with `zlib` and indexed by an offset table. On decompression, blocks are decompressed as they are accessed. We keep an LRU cache of decompressed blocks (in our experiments we use a cache of 8 blocks). The on-disk representation is not optimized—it may be possible to shave the I/O cost by aligning the compressed blocks to the disk block boundaries.

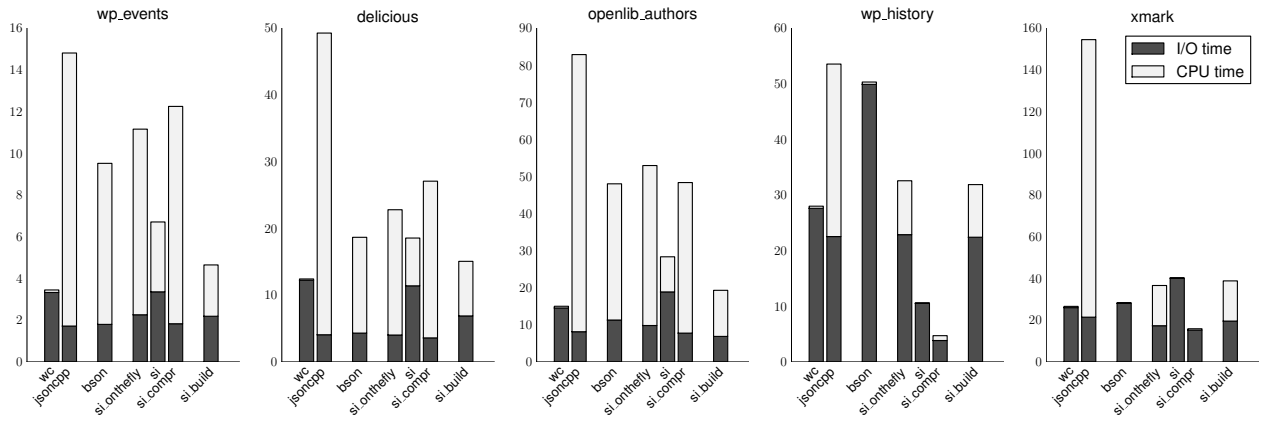


Figure 1: Wall clock times for each dataset as listed in Table 1. I/O time indicates the time the CPU waits for data from disk, while in CPU time the CPU is busy (and the kernel may be prefetching pages from disk).

Dataset	Wall clock time (seconds)						
	wc	jsoncpp	bson	si_onthefly	si	si_compr	si_build
wp_events	3.5	14.8 *(4.29)*	9.5	11.2 *(3.23)*	**6.7 *(1.94)***	12.2 *(3.55)*	4.7 *(1.35)*
delicious	12.4	49.3 *(3.96)*	18.7	22.8 *(1.83)*	**18.6 *(1.49)***	27.1 *(2.18)*	15.1 *(1.21)*
openlib_authors	15.0	82.9 *(5.52)*	48.0	53.0 *(3.53)*	**28.4 *(1.89)***	48.4 *(3.22)*	19.3 *(1.29)*
wp_history	28.0	53.5 *(1.91)*	50.3	32.6 *(1.16)*	10.6 *(0.38)*	**4.7 *(0.17)***	31.9 *(1.14)*
xmark	26.6	154.5 *(5.80)*	28.3	36.6 *(1.38)*	40.2 *(1.51)*	**15.9 *(0.60)***	38.9 *(1.46)*

Table 1: Running times for each dataset. Numbers in parentheses are the runtimes normalized on `wc` time. Numbers in bold are the ones within 10% from the best

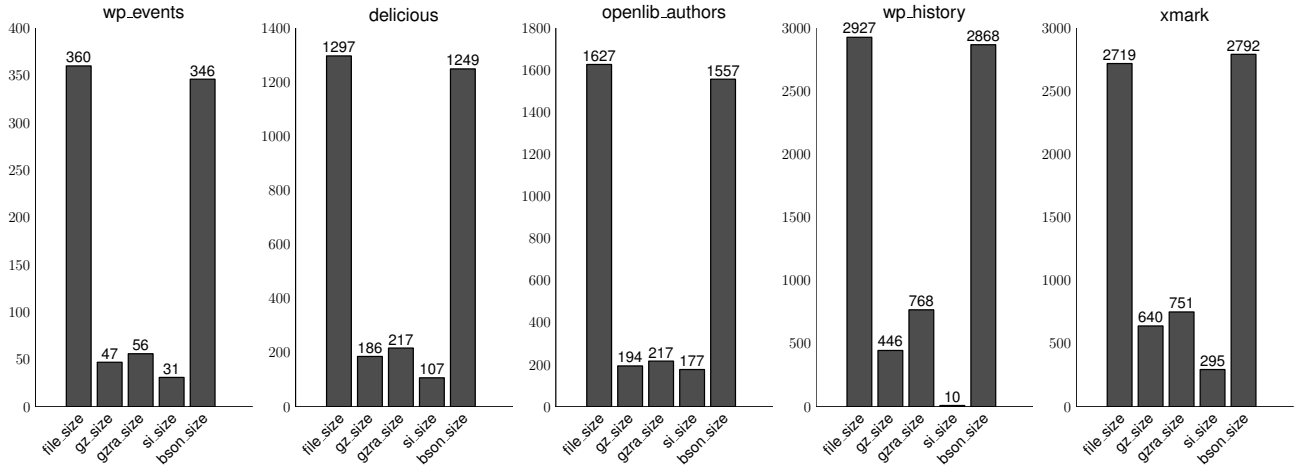


Figure 2: Space occupancy in MB of the file uncompressed (`file_size`), compressed with gzip (`gz_size`) and compressed with gzra (`gzra_size`), encoded in BSON (`bson_size`), and of the semi-index (`si_size`)

Dataset	Records	Average kBytes	Average nodes	Semi-index overhead
wp_events	1000000	0.36	24.82	8.85%
delicious	1252973	1.04	61.28	8.31%
openlib_authors	6406158	0.25	22.00	10.90%
wp_history	23000	127.27	203.87	0.34%
xmark	1000	2719.47	221221.48	10.86%

Table 2: Number of documents, average document size, average number of nodes, and semi-index space overhead (percent with respect to average document size) for each dataset used in the benchmark

Datasets. The experiments were performed on a collection of datasets of different average document size and density. On one extreme of the spectrum there are datasets with small document size (`wp_events`), which should yield little or no speedup, and very high density (`xmark`), which should give a high semi-index overhead. On the other extreme there is `wp_history` which has large documents and relatively small density. Specifically:

The Wikipedia data was obtained by converting to JSON the Wikipedia dumps [34], while we used the `xmlgen` tool from XMark [31] and converted the output to JSON to generate synthetic data of very high density.

- `wp_events`: Each document represents the metadata of one edit on Wikipedia.
- `delicious` [20]: Each document represents the metadata of the links bookmarked on Delicious in September 2009.
- `openlib_authors` [32]: Each document represents an author record in The Open Library.
- `wp_history`: Each document contains the full history of a Wikipedia page, including the text of each revision.
- `xmark`: Each document is generated using `xmlgen` from XMark with scale factor chosen uniformly in the range $[0.025, 0.075)$.

Queries. The queries performed on each dataset are shown in Table 3. We have chosen the queries to span several depths in the document trees (the XPath dataset `xmark` has deeper trees that allow for more complex queries). Some queries access negative indices in arrays, to include the contribution of the performance of backwards array iteration in our tests.

Dataset	Queries

`wp_events`	`id`
	`timestamp`
	`title`

`delicious`	`links[0].href`
	`tags[0].term`
	`tags[-1].term`

`openlib_authors`	`name` `last_modified.value`

`wp_history`	`id`
	`title`
	`revision[0].timestamp`
	`revision[-1].timestamp`

`xmark`	`people.person[-1].name`
	`regions.europe.item[0].quantity`
	`regions.europe.item[-1].name`
	`open_auctions.open_auction[0].current`

Table 3: Queries performed on each dataset

Testing. For each dataset we measured the time needed to perform the following tasks.

- `wc`: The Unix command that counts the number of lines in a file. We use it as a baseline to measure the I/O time needed to scan sequentially the file without any processing of the data.
- `jsoncpp`: Query task reading each line, and parsing it using the JSONCpp library [22] (one of the most popular and efficient JSON C++ libraries). The requested values are output by querying the in-memory tree structure obtained from the parsing.
- `bson`: Query task using data pre-converted to BSON.
- `si_onthefly`: Query task reading each line, building on the fly the semi-index, and using it to perform the queries. Note that in this case the semi-index is not precomputed.
- `si`: Query task using a precomputed semi-index from a file on disk.
- `si_compr`: Like `si`, but instead of reading from the uncompressed JSON file, the input is read from a `gzra`-compressed file.
- `si_build`: Construction of the semi-index from the JSON file.

Results. We summarize the running times for the above tests in Figure 1 and Table 1, and the space overhead in Figure 2 and Table 2, which also reports the statistics for each file in the dataset. We now comment in some detail these experimental findings.

A common feature on all the datasets is that the standard load-and-parse scheme using the JSONCpp library, and implemented as `jsoncpp`, has the worst performance. If time efficiency is an issue, the other methods are preferable.

BSON is a good candidate in this sense, since it uses preconverted data, as implemented in `bson`, and always runs faster than `jsoncpp`. It is interesting to compare `bson` with our methods, which also run faster than `jsoncpp`.

Consider first the situation in which we do not use any preprocessing on the data: when performing the queries, we replace the full parsing of the documents with an on-the-fly construction of the semi-index, as implemented in `si_onthefly`. As shown in our tests, `si_onthefly` performs remarkably well even if it has to load the full document, as the semi-index construction is significantly faster than a full parsing. Compared to `bson`, which uses a prebuilt index, the running times are slightly larger but quite close, and the I/O times are simlar. Surprisingly, for file `wp_history`, we have that `si_onthefly` is faster than `bson`: a possible reason is that in the latter the value sizes are interleaved with the values, causing a large I/O cost, whereas the semi-index is entirely contained in few disk blocks.

We now evaluate experimentally the benefit of using a prebuilt semi-index. The running times for `si_build` show that the construction of the semi-index is very fast, and mainly dominated by the I/O cost (as highlighted by the comparison with `wc`).

Using the pre-computed semi-index, `si` can query the dataset faster than `bson`, except for file `xmark` where it

is slightly slower: as we shall see, when this file is in compressed format, the I/O time is significantly reduced. Also, on wp_history querying the last element of an array requires bson to scan all the file (as explained in the introduction), while si can jump to the correct position; overall si is 5 times faster than bson on this dataset. Note that contrarily to bson, si require less time than wc in some cases since it takes advantage of the semi-index to make random accesses to the file and retrieve just the needed bits.

The space overhead of the pre-built semi-index is reported in the last column of Table 2 and item si_size of Figure 2. The semi-index takes between 8% and 10% of the uncompressed input for all datasets except wp_history, where the overhead is practically negligible because data is sparse.

If the overall space occupancy is an issue, we can opt for a variant of our method si, as implemented in si_compr, where the dataset is kept compressed using the gzra format previously discussed. Note that this format, which is a variant of gzip, requires slightly more space but it allows for random block access to compressed data (e.g. compare items gz_size and gzra_size in Figure 2). When comparing to the space required by the binary format of bson (item bson_size in Figure 2), we obtain a significant saving, where the total space occupancy of the semi-index and the compressed dataset is the sum of the values of items gzra_size and size_size in Figure 2.

Regarding its time performance, si_compr is slighter slower than bson and si for sparse files, while it performs better for dense files such as wp_history and xmark: in the former case, the decompression cost dominates the access cost, while in the latter the I/O cost is dominant and the reduced file size improves it (still taking advantage of semi-indexing). This is also why si_compr is faster than wc on some files, and obtains a 12x speedup on wp_history over jsoncpp. Note that on xmark the running times of si and si_onthefly are comparable: the access pattern of the queries that we tested touches a large part of the file pages so there is no advantage in precomputing the semi-index, as (almost) all the file is accessed anyway.

Summing up, for each dataset at least one among si and si_compr has at least a 2x speedup. The graphs suggest that compression enables better performance as the average document size increases.

7. MEMORY-EFFICIENT PARSING

In this section we describe an alternative application of the semi-indexing technique.

A fully deserialized document tree takes often much more memory than the unserialized document itself. Hence in case of big documents it is very likely that the textual (XML or JSON) document fits in main memory but its deserialized version doesn't.

Industrial parsers such as Xerces2 [1] work around this problem by loading in memory only the tree structure of the document and going back to the unparsed document to parse the required elements. This approach however requires at least a pair of pointers per node, and usually much more. As shown in Section 4 for dense documents a pointer-based representation of the document tree can be more expensive than the document itself.

Since the construction of the semi-index is extremely fast, we suggest that a semi-index can be used in place of pointer-based data structures for lazy parsing.

Figure 3 shows the running times for jsoncpp and si construction and querying when the document is already in main memory, hence with no I/O is involved. Note that query times using the semi-index for in-memory documents are just 10 times slower than by accessing a fully deserialized tree using jsoncpp. This is very reasonable, since the query time includes the time to parse the leaf attributes once the semi-index has identified their position in the unparsed document.

Thus the semi-index can be used as an alternative to explicit or lazy parsing in applications where memory is a concern, for example on mobile devices.

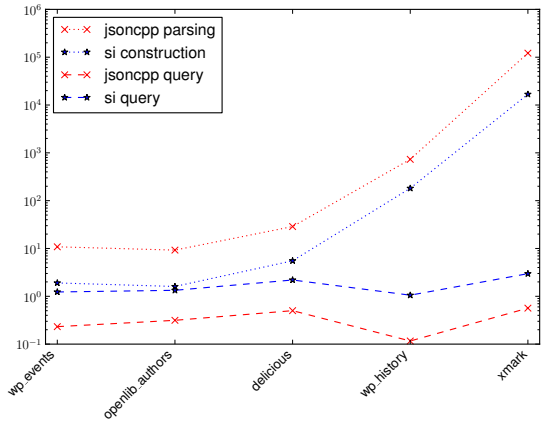


Figure 3: Timings (in microseconds, log-scale) for in-memory parsing and query operations.

8. CONCLUSIONS

We have described semi-indexing, a technique to build a data structure that enables navigation operations on textual semi-structured data without altering its representation. We engineered and implemented a specialization of the technique of the JSON format.

Our analysis demonstrates that the semi-index for JSON documents significantly outperforms the naive approach of parsing each document entirely, and has better performance than a binary format, without sacrificing too much storage space.

The technique we described only addresses basic traversal of the document tree. It would be interesting to devise more powerful semi-indexing schemes that support complex queries (à la XPath). Future work will focus on augmenting the semi-index with other structures to support new operations, while maintaining a small space overhead by exploiting the access to the raw document.

Acknowledgments

The authors would like to thank Luca Foschini for his insightful comments and suggestions on a preliminary draft.

9. REFERENCES

[1] Apache Xerces2 XML Parser. http://xerces.apache.org/xerces-j/.

[2] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *ALENEX*, pages 84–97, 2010.

[3] Binary XML. http://en.wikipedia.org/wiki/Binary_XML.

[4] CouchDB. http://couchdb.apache.org/.

[5] CouchDB in the wild. http://wiki.apache.org/couchdb/CouchDB_in_the_wild.

[6] A. Couthures. JSON for XForms. In *Proc. XMLPrague 2011*, 2011.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[8] O. Delpratt, R. Raman, and N. Rahman. Engineering succinct DOM. In *EDBT*, pages 49–60, 2008.

[9] P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[10] R. Fano. On the number of bits required to implement an associative memory. Memorandum 61. *Computer Structures Group, Project MAC, MIT, Cambridge, Mass., nd*, 1971.

[11] F. Farfán, V. Hristidis, and R. Rangaswami. 2LP: A double-lazy XML parser. *Inf. Syst.*, 34(1):145–163, 2009.

[12] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *J. ACM*, 57(1), 2009.

[13] G. Gou and R. Chirkova. Efficiently Querying Large XML Data Repositories: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[14] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

[15] Hive JSON SerDe. http://code.google.com/p/hive-json-serde/.

[16] J. Hunter. A JSON Facade on MarkLogic Server. In *Proc. XMLPrague 2011*, 2011.

[17] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *5th International Conference on Innovative Data Systems Research (CIDR)*, 2011.

[18] G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989.

[19] Jaql. http://code.google.com/p/jaql/.

[20] JSON dump of Delicious bookmarks, September 2009. http://infochimps.com/datasets/delicious-bookmarks-september-2009.

[21] JSON specification. http://json.org/.

[22] JsonCpp. http://jsoncpp.sourceforge.net/.

[23] M. Kay. Ten Reasons Why Saxon XQuery is Fast. *IEEE Data Eng. Bull.*, 31(4):65–74, 2008.

[24] MongoDB. http://www.mongodb.org/.

[25] MongoDB Production Deployments. http://www.mongodb.org/display/DOCS/Production+Deployments.

[26] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *FOCS*, pages 118–126, 1997.

[27] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, 2007.

[28] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding n-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[29] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *SODA*, pages 134–149, 2010.

[30] S. Sakr. XML compression techniques: A survey and comparison. *J. Comput. Syst. Sci.*, 75(5):303–322, 2009.

[31] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *In VLDB*, pages 974–985, 2002.

[32] The Open Library, JSON dump of author records. http://infochimps.com/datasets/the-open-library.

[33] S. Vigna. Broadword implementation of rank/select queries. In *WEA*, pages 154–168, 2008.

[34] Wikipedia database dumps. http://download.wikimedia.org/.

[35] R. K. Wong, F. Lam, and W. M. Shui. Querying and maintaining a compact XML storage. In *WWW*, pages 1073–1082, 2007.