A GENERAL-PURPOSE BINARY DATA FORMAT

John Roush

A thesis submitted in partial fulfillment of
the requirements for the degree of
Master of Science

Department of Computer Science

Central Michigan University
Mount Pleasant, Michigan
February 2017

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Patrick Seeling, for his moral and technical support. I would also like to thank the rest of my thesis committee for their time and attention. Finally, I would like to acknowledge the support of my family - without them, this would not have been possible.

ABSTRACT

A GENERAL-PURPOSE BINARY DATA FORMAT

by John Roush

The simplest non-trivial data format is a stream: an ordered sequence of symbols such as bytes. This is a natural representation for network interfaces like sockets, and is the generally accepted abstraction for files as well.

The majority of real-world digital data, however, is not naturally represented as a stream. Most data have an implicit hierarchical structure. The data formats used to represent this structure (often text-based) lack standardization and generally do not compose well with each other.

To address these issues I develop a simple binary data format that can:

- Express arbitrary hierarchical structure.
- Compose with other binary data without modification.
- Be annotated with arbitrary metadata.
- Be stored and traversed efficiently.

This format makes it simpler for standardized software tools to understand and respect the structure of data. It allows efficient composition of data with diverse semantics, including machine primitives like integers. It is more efficient than equivalent ad-hoc grammars in storage and transmission and faster for software to consume.

A formal specification of this data format is presented, along with a comparison against the JSON and BSON formats. The format performs well in both size and traversal-time benchmarks compared to raw JSON.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER I

BACKGROUND

Streams

The simplest non-trivial data format is a *stream*: an ordered list of data primitives such as bits, bytes, or characters. The length of a stream may be finite or infinite, or in some cases unknown.

Streams are the medium on which all common storage and communication formats are built. Streams of individual bits are used as the input to media encodings such as MPEG elementary streams, and to compression algorithms such as DEFLATE. They are also the abstraction presented by Layer 1 of the OSI networking model (i.e. networking hardware) [1].

Bytes (or more properly, octets) are the primitive unit of addressable memory on most modern computer architectures. They are consequently also the addressable unit for storage devices and for the filesystem. The POSIX standard defines a *regular file* as [2]:

> ...a randomly accessible stream of bytes, with no further structure imposed by the system.

and non-POSIX operating systems like Windows implicitly follow suit in their filesystem APIs. Berkley socket APIs are defined in a similar manner.

Bytes are also the unit of network transmission for OSI Layer 2 (the link layer in IP parlance) and higher. The frames transmitted by Link-layer protocols such as Ethernet (IEEE 802.3) and WLAN (IEEE 802.11) are byte-oriented, as are the packets used by network-layer protocols such as IP and transport-layer protocols like UDP. Due to the difficulty of guaranteeing order of delivery over a distributed network, these protocols do not offer a proper byte *stream* abstraction. TCP does, however, and it forms the basis of most non-media/gaming internet applications.

Text, of course, is a stream of characters. Just as bytes are composed of bits, characters are composed of bytes, although the precise definition of "character" and the encoding used to represent them are not nearly as standardized as the definition of the byte. Nevertheless, text is an extremely popular format for computerized data:

- POSIX **Streams** (e.g. `stdin` and `stdout`) are character-based.
- Windows filenames are unicode strings.

- Configuration file formats like ***.INI**, ***.cfg**, and the vast menagerie of files kept by UNIX utilities in `/etc` are all text-based.
- Many pure data formats use plain text with simple record separators. The **WARC** format used by web crawlers uses newlines, for example, and the **CSV** format uses commas.
- Virtually all programming languages are text-based, with some like C using a subset of ASCII and others like Java using Unicode [3].
- **HTTP**, and its parent **SGML** and cousin **XML** are all text-based [4].
- The core web languages - **HTML**, **CSS**, **Javascript**, and **JSON** - are all text based [5] [6].

Because it shares the stream structure, text maps naturally onto byte- and bit-stream formats, making it very convenient to transmit over network protocols and store in the filesystem.

Structured Data

Unfortunately, real-world data usually does *not* have a pure stream structure. Most of the examples above apply some structure above and beyond that imposed by the stream itself.

Example: A Typical Webserver Response

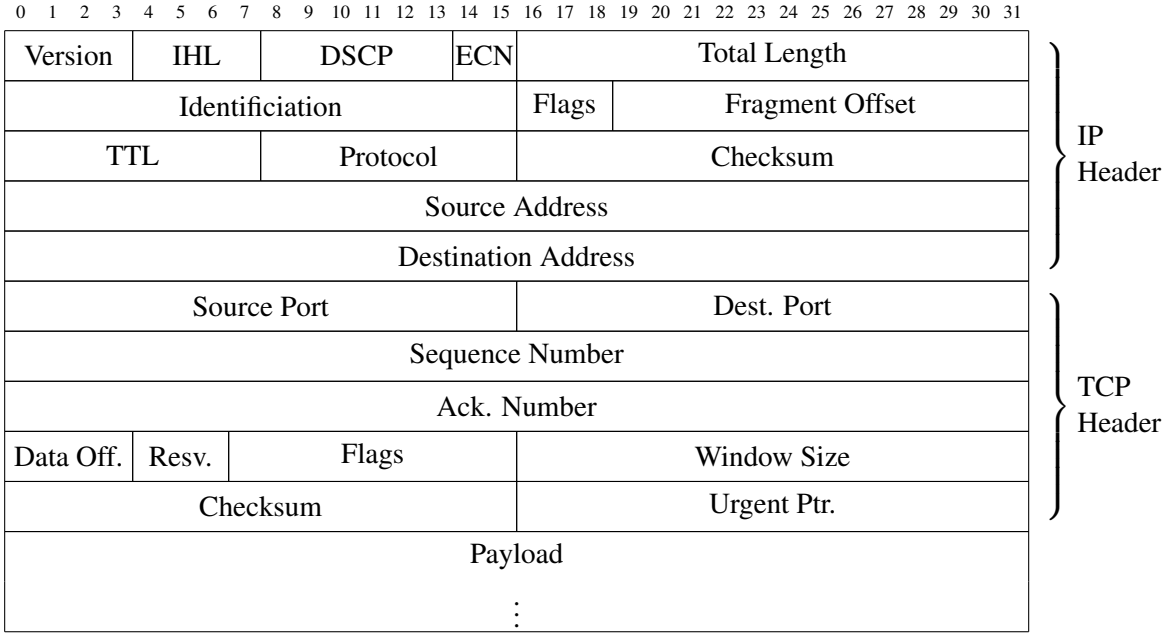

Figure I.1. The structure of a TCP packet.

Consider, as an example, a TCP packet from a typical web server response (Fig. I.1). The information in the IP and TCP packet headers is structured into discrete, rigidly-sized fields. These fields are obviously composed of bytes and are (mostly) byte-aligned, but they do not form a logical stream because *their order is irrelevant to the semantics of the protocol*, and of course because each field must occur exactly once. An IP or TCP header is more correctly thought of as a fixed-length *set* of data elements than as a stream.

```
1 HTTP/1.1 200 OK
2 Cache-Control: max-age=604800
3 Content-Type: text/html
4 Date: Mon, 20 Feb 2017 10:08:54 GMT
5 Content-Length: 1270
6
7 ... [response content] ...
```

Figure I.2. Example of an HTTP response.

The payload of the TCP packet is HTTP (Fig. I.2), and although it is text-based rather than byte-based it also encodes some structure. Note that HTTP uses regular keys to mark header fields rather than denoting them purely based on their offset like IP or TCP. This distinguishes the fields from one another, while also making the structure parsable by human readers (as opposed to using e.g. opaque numbers as keys).

The content of the HTTP request is an HTML document (Fig. I.3). HTML encodes yet another type of structure, much more complex than simple fields. HTML elements can be recursively embedded - note how one `<div>` element contains another `<div>` element. Despite being expressed through a text stream, the real structure of an HTML document is a tree.

Note also that this document contains a snippet of another format, CSS (Fig. I.4). Embedding one data format inside another is called *composition*. This is quite common, and in fact all four formats in this example (IP+TCP, HTTP, HTML, and CSS) were composed to form a single logical server response.

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title> <meta charset="utf-8" />
    <style type="text/css">
        body {
            background-color: #f0f0f2;
            font-family: "Helvetica", "Arial", "sans-serif";
        }
    </style>
</head>
<body><div>
    <div><h1>Example Domain</h1></div>
    <p>This domain is established to be used for illustrative examples
       in documents.</p>
</div></body>
</html>
```

Figure I.3. Example of an HTML document.

```
body {
    background-color: #f0f0f2;
    font-family: "Helvetica", "Arial", "sans-serif";
}
```

Figure I.4. Example of a CSS document.

Grammars

The example in Section illustrates two features of structured data that are not directly expressible with stream-based formats:

Semantic order-independence Data that are logically a collection of elements, but not necessarily an ordered sequence.

Non-linear topology Data in which the elements are naturally connected in one-to-many (tree) or many-to-many (graph) relationships.

Nevertheless, structured data formats that aim to be compatible with stream-based infrastructure like files and networks must be encodable to and parsable from a stream.

Streams are ordered by definition, hence to encode unordered data into a stream it must be coerced into some particular ordering. Extracting unordered data from a stream is of course much simpler - the ordering information is simply discarded.

Encoding non-linear structure onto a stream is more complex. Hierarchical structures can be serialized recursively through a set of *production rules*, which together form a *recursively enumerable grammar*, or just a "grammar." These rules can also be used to recover the structure from a stream, or recognized a stream with no recoverable structure. Some care must be taken in designing the grammar, however, to ensure that the serialization is unique and unambiguous.

An excerpt from a simplified grammar for HTML might look like this [7]: The symbols in lower-

```
1  document     := (head)? (body)? ;
2    ...
3  body         := BODY-BEGIN ( content )* BODY-END ;
4  BODY-BEGIN   := "<body>" ;
5  BODY-END     := "</body>" ;
6  content      := tag | TEXT ;
7  tag          := paragraph | list | preformatted | div |
8                  center | blockquote | HR | table |
9                   ... ;
10 div          := DIV-BEGIN ( content )* DIV-END ;
11 DIV-BEGIN    := "<div>" ;
12 DIV-END      := "</div>" ;
```

Figure I.5. Fragment from a grammar for HTML.

case, called *non-terminal symbols*, represent a non-leaf node in the hierarchy. To encode the structure, each non-terminal on the left side of a production rule is recursively replaced with the symbols on the right side of the rule. This production is not necessarily unique - `content`, for example, can be replaced with `TEXT` or `tag` depending on the actual structure of the document. The recursion terminates with the production rules in all-caps, called *terminal symbols*.

Some terminal symbols like `BODY-BEGIN` and `BODY-END` are defined as literal strings. These are the basic building blocks that a parser for HTML uses to recognize and reconstruct the hierarchical structure from a stream. When the parser sees "<body>", it knows that this was produced by a `BODY-BEGIN` rule and therefore that it marks the beginning of a `body` element. In order for this type of parsing to work, however, there can *only be one production rule that produces the string "<body>"*. Other production rules must

never produce, directly or indirectly, that specific combination of characters[1]. Not all terminal symbols are subject to this constraint, but the grammar must be designed to accommodate those that are.

Escape Productions

One of the terminal symbols in Fig. I.5 is `TEXT`, which is the "payload" element of HTML. Content composed from other formats, such as CSS or human-readable text like "Example Domain", are included as `TEXT` nodes in an HTML document. This text is loosely defined to be any character stream, but not all text streams can be `TEXT` nodes in HTML. For example, the following string (which might plausibly appear in some UNIX shell listings) is not permissible as HTML text:

tee <body>b.log (1)

To work around this limitation, HTML provides an *escape* mechanism. This grammar production, called `entity` in HTML, is defined in Fig. I.6. Productions of this form have special meaning to the parser.

```
1 entity       := ENT-BEGIN ( ENT-AMP | ENT-LT | ENT-GT | ... ) ENT-END ;
2 ENTITY-BEGIN := "\&" ;
3 ENTITY-END   := ";" ;
4 ENT-AMP      := "amp" ;
5 ENT-lT       := "lt" ;
6 ENT-GT       := "gt" ;
7  ...
```

Figure I.6. Grammar fragment for HTML entities.

When parsing HTML from a stream, "<" will be interpreted as a single character of `TEXT` '<'. Likewise ">" is read as '>' and "&" and '&'. This means that the string

tee <body>b.log (2)

when read by an HTML parser, will be unambiguously transformed into the shell command (1). In fact, for any "illegal" text string s in HTML, there is another "escaped" string $E(s)$ that is transformed to s when read

[1] This is a simplification. HTML allows strings like "<body>" to appear in comments, quoted strings, and CDATA sections. Most grammars make such exceptions for usability reasons, even though they complicate the design of parsers. This does not change the conclusions above though - comments, quoted strings, and CDATA sections all have their own terminal symbols which cannot appear un-escaped in their content.

by the parser. Therefore, it is possible to compose *arbitrary* text streams into HTML, provided that they are first escaped with entities. Most[2] grammars that rely on unambiguous terminal symbols to parse will include a similar mechanism. Common "escape" characters include '\', '^[' (an actual Escape character), and of course '&'.

Not all grammars require such gymnastics to compose arbitrary content. The grammar for the TCP packet in Fig. I.1 does not have *any* restricted non-terminal symbols, and does not require escaping of its payload content. Instead, it restricts the exact size and order of the expected fields in the stream. This approach is discussed further in Chapter IV.

[2] but not all - POSIX shebang lines are a notable grammar that has no production rules to escape special characters in paths.

CHAPTER II

MOTIVATION

This chapter discusses issues and strengths common in extant data formats, and in Section lays out the formal requirements for a new structured data format that attempts to improve on them.

Problems with Escape Productions

Section defines the role of grammars in encoding and parsing structured data. Some grammars use restricted terminal symbols to denote structure, and this limits the content that can be composed with that grammar (it cannot contain any of the restricted symbols). This problem is usually circumvented with a suitably designed escape production, which in turn requires that all composed content first be transformed ("escaped").

Escaping content poses several problems in practice. The first is simple readability. Consider, for example, the path to a Windows file containing the text of a book:

```
C:\Data\A Man Named "Nobody".txt
```

Now consider the equivalent strings, as escaped for composition into C/C++ and LaTeX:

```
"C:\\Data\\A Man Named \"Nobody\".txt"
C:\textbackslash{}Data\textbackslash{}A Man Named "Nobody".txt
```

The meaning of the path quickly becomes buried by the escapes, and the effect grows combinatorially when composing more than two grammars:

```
"C:\textbackslash{}\textbackslash{}Data\textbackslash{}\textbackslash{}A Man
  Named \textbackslash{}"Nobody\textbackslash{}".txt"
```

A more serious problem is the potential implications of *forgetting* to escape content before composing it into another format. Consider the effect of this code, which composes arbitrary user-provided input into an SQL query:

```
1 query = "SELECT * FROM records WHERE username = '" + userName + "';"
2 sqlDB.runQuery(query)
```

If the variable *userName* is not escaped for inclusion into SQL, then it is possible for an enterprising user to engineer a situation like this:

```
1 userName = "X'; DROP TABLE records; --"
2 query = "SELECT * FROM records WHERE username = '" + userName + "';"
3 sqlDB.runQuery(query)    # Deletes the 'records' table
```

This kind of "injection" attack has been the bain of poorly designed user interfaces for decades. Preventing these attacks requires a great deal of "sanitation" discipline when composing user input, and this in turn undermines the convenience and expedience of using a text-based query format like SQL in the first place.

Finally, there is the problem of efficiency. Escaping content usually increases it's size - sometimes by a few bytes, and sometimes by a considerable fraction. Base-64 encoding is often used to arbitrary binary data for composition into text-only formats like JSON or SMTP messages. Base-64, unfortunately, increases the size of the composed content by 30%. Even when size is not important, having to escape during composition and subsequently remove the escapes again during parsing consumes computing time and memory. Given a base-64-encoded email attachment for example, email client must decode the original binary image and then store it somewhere, separate from the email, for presentation to the user. The overhead of this resource allocation and management effectively limit the level of composition that is practical for grammars with escape productions.

Metadata

Despite the problems described above, grammars with restricted terminal symbols and escape productions are common, and in particular text-based formats with loose, often ad-hoc syntax are quite popular.

There are two important contributing factors to this. The first is that there is a huge ecosystem of tools for viewing, editing, and manipulating text (or at least, the anglo-centric subset of it commonly used in computing). Text Editors are of course ubiquitous, but there exist tools for searching text (like `grep` or other regex processors), comparing text (like `diff`), transforming text (like `awk` and `sed`), and many others. This represents a huge incumbent advantage over newer or more niche data formats.

The more important reason for the popularity of text, however, is it's sheer flexibility. Humans reading and writing text can encode *multiple*, independent structures in the text. Consider the snippet of C code in Fig. II.1. The author is uses the C grammar to express the semantics of the algorithm for the benefit

```
static inline uint64_t parse_uint(const void *data, uint64_t bytes) {
    switch (bytes) {
        case 0:     return 0;
        case 1:     return         *(uint8_t*) data;
        case 2:     return ntohs (*(uint16_t*) data);
        case 4:     return ntohl (*(uint32_t*) data);
        case 8:     return ntohll(*(uint64_t*) data);
        default:
            assert( false ); // invalid integer size
            return 0;
    }
}
```

Figure II.1. A fragment of C code.

of the compiler, but at the same time uses the (independent) structure of whitespace, alignment, and the grouping of items to convey information about the relationship between the various expressions to a human reader.

While more concise data formats may save on space and speed, they often overlook the importance to human users of the ability to encode extra, useful information alongside the main content of a document or file. This secondary content is, collectively, called **metadata**. The ability to include (nearly) arbitrary metadata is an important strength of textual grammars.

Goals of this Project

The goal of this research effort is to design a structured data format which addresses the conflict between escape productions and composition, while retaining the freedom to include arbitrary metadata.

In addition, it is important to address the question of efficiency. All other things being equal, a data format that is smaller, faster to traverse, and simpler to understand will be applicable in more (and more extreme) situations. A general-purpose data format, therefore, should strive to be as efficient as possible to allow for the broadest possible range of applications.

With that in mind, the formal requirements for this data format are as follows:

1. It **must** have the ability to express **arbitrary hierarchical structure**. This includes self-recursive and mutually recursive structures.
2. It **must** have the ability to **compose with arbitrary binary data**.

3. Composition **must** be done **without escaping or transforming composed content**.
4. It **must** offer the ability to annotate data out-of-band, that is the ability to **include arbitrary metadata**. At a minimum, it must be able to **describe its own hierarchical structure**.
5. It **must** be able to traverse to any element in a time proportional to it's depth and constant with respect to all other factors. This implies that **each node must be reachable from its parent in constant time**.
6. It **should** be as efficient as possible to store and transmit, meaning the it **should not** encode unnecessary information.
7. It **should** offer the possibility of predictable in-memory alignment of data elements, meaning that it **should not** include unnecessary or implicit padding bytes outside the control of the user.
8. It **should** be as simple as possible to understand and implement, given the other constraints.

A design that meets these constraints is described in Chapter IV.

CHAPTER III

EXISTING DATA FORMATS

This chapter presents three existing data formats that serve as influences and useful points of contrast.

JSON

JSON (ECMA 404 [6]) is a lightweight text-based format for storing arbitrary hierarchical data. It is based on a subset of Javascript syntax for denoting data literals.

The JSON gammar defines 5 primitive and 2 compound data types:

- **null**, **true**, and **false**
- **numbers** including integers and floats.
- **strings** of unicode characters.
- **Arrays** of ordered heterogeneous values.
- Unordered sets of *name* : *value* pairs called **Objects**.

The syntax is drawn directly from JavaScript, and hence from other C-style languages. `Strings` are enclosed in double quotes '"' and use standard backslash escape productions ('\n', '\t', '\"', etc.). `Numbers` are in floating-point format, with optional leading sign, decimal points, and exponential notation using 'e' or 'E'. `Arrays` are enclosed with square brackets '[ ]' and `Objects` with curly braces '{ }'. Array elements and object entries are separated with commas ',', and key-value pairs are divided by a colon ':'. Whitespace outside of strings is permitted and ignored.

`Array` elements and `Object` values may be any JSON value, including nested `Arrays` or `Objects`. JSON is thus a recursive grammar and can represent any hierarchical data structure.

JSON is distinct from other text-based data formats in its focus on simplicity. It provides no mechanism for metadata, or even for comments. The formal grammar specification can fit on a postcard. This simplicity has led to widespread usage and a large number of independent implementations. It has also led to a number of extensions to add metadata and comments to the grammar.

BSON

BSON (Binary JSON) is a bytestream-based format developed by MongoDB[8] for use in their document store database.

BSON supports 6 fixed-size primitive data types and 3 variable size primitive types:

- **byte**, a single byte.
- **int32**, a 4-byte signed integer.
- **int64**, an 8-byte signed integer.
- **uint64**, an 8-byte unsigned integer.
- **double**, an 8-byte IEEE 754 base-2 float.
- **decimal128**, a 16-byte IEEE 754 base-10 float.
- **cstring**, a null-terminated stream of UTF-8 encoded characters, which *must not* contain '\0'.
- **string**, a null-terminated stream of arbitrary UTF-8 encoded characters.
- **binary**, an arbitrary stream of bytes.

All multi-byte values are serialized in little-endian byte order (the native order for x86-based processor architectures).

The single compound type in BSON is called **Document**: a set of (*typetag*, *name*, *value*) tuples, where *typetag* is a single-byte constant indicating the type of *value*. The *name* is a `cstring` corresponding to the *key* in a JSON `Object`. There are 18 entry typetags, including one for 8 of the primitive types (notably excluding `cstring`). There are also 3 typetags corresponding to the JSON primitives `null`, `true`, and `false`, indicating that *value* is empty (since the information conveyed by these values is contained entirely in the typetag). In addition, BSON supports some entry types which combine multiple primitives or add additional semantics and meaning to them:

- **datetime**, an `int64` containing a time denoted in UTC seconds from the UNIX epoc.
- **javascript**, a `string` containing JavaScript source code.
- **regex**, a pair of `cstrings` containing a regular expression and a set of regex flags, respectively.
- **objectid**, a 12-byte structure used to index database objects.
- **timestamp**, a `uint64` containing a time in a database-specific format.

Finally, there are two typetags indicating that an entry *value* is a nested `Document`: either an arbitrary document or an `Array`-style document whose entry *names* correspond to the array index ('0', '1', '2', etc.).

When encoded into a bytestream, BSON structure uses a mixture of fixed-length formatting (such as `int64` et al.), restricted terminal symbols (specifically the null-bytes used to terminate `cstrings`), and **length-prefixing** (for `string`, `binary`, and `Document`). In the latter case, the encoded value is prefixed with its length, in bytes, represented as an `int32`.

BSON's eclectic mixture of dataypes and grammar strategies reflects its origin as a bespoke proprietary format. MongoDB needed an extension to JSON that was also compact and fast to traverse [8]. The use of size-prefixing, in particular, allows BSON to compose arbitrary bytestreams as content, and the use of fixed-size typetags allows the parser to quickly resolve the type of each entry in a document before attempting to parse (or traverse over) it. Despite its ad-hoc design, BSON offers excellent traversal performance in practice.

Protocol Buffers

Protocol Buffers, often abbreviated as "protobuf", is a general-purpose format developed by Google for serializing structured data [9]. Protocol Buffers are more expressive and complex than JSON or BSON, and are closer to the ideal of an efficient, general-purpose data format.

Protocol Buffers defines 7 fixed-size primitive types and 4 variable-length primitive types:

- **bool**, a single byte containing 0 or 1.
- **uint32** and **uint64**, 4- and 8-byte unsigned integers.
- **sint32** and **sint64**, 4- and 8-byte signed integers.
- **float** and **double**, 4- and 8-byte IEEE 754 binary floats.
- **varint**, an unsigned integer of variable length.
- **svarint**, a signed integer of variable length.
- **string**, a stream of UTF-8 encoded characters.
- **bytes**, a stream of arbitrary bytes.

As with BSON, fixed-size integer and float types are serialized in little-endian byte order.

One notable innovation in the primitive types are the variable-length integers. These use a clever encoding to interleave the size and content of the integer: the most significant bit of the last byte in an integer will be 0, and the msb of all preceding bytes will be 1. For `varint`, the remaining seven bits of each byte are concatenated to recover the binary integer value. The signed `svarint` version is more complex, as the conventional twos complement representation of negative numbers requires a fixed number of bits. Instead, variable-length signed integers are stored in a "zig-zg" representation of alternating positive and negative values [9]. This allows integers will lower absolute values to consume less space, and Google asserts that the size gains are significant enough that all size fields, discriminants, and type ids in Protocol Buffers use this variable length integer encoding.

The core datatype of Procol Buffers is the **Message**. Protobuf messages are quite complex, but in practice each message can be thought of as an ordered list of entries in the form (*tag*, *wiretype*, *value*). Messages are compound types, but unlike `Objects` in JSON/BSON, `Messages` are not simply defined as a recursive, heterogeneous array or key-value set. Instead, they are *user-defined* compound types built from the primitive types and other messages.

The *tag* serves a similar purpose as the *name* in a BSON `Document` entry. Rather than a string it is a `varint`, however, and the precise semantics of each tag (i.e. the type of *value* and the order and number of times each tag may/must appear in a message) are defined by the user. This is done through the use of a type description language whose exact features are beyond the scope of this work (see [10]). The salient point is that the exact semantics of a message *are not encoded with it*. Instead, the user is expected to store and communicate their type descriptions out-of-band.

Because the user is responsible for providing separate information about the order and type of each *value*, and because users are prone to error, Protocol Buffers is designed to be resilient against missing (or outdated) type descriptions. This is where the *wiretype* comes into play - it contains just enough information for Protocol Buffers to deduce the size of *value*, so that the remainder of the message can still be parsed if the type of *value* is unknown.

The encoding used for message entries is, again, quite clever. There are four legal values for the wiretype:

- **varint**, if the type is `varint`, `svarint` or `bool` (which nicely encodes as a `varint` of length 1).

- **fixed32**, if the type is `sint32`, `uint32`, or `float`.
- **fixed64**, if the type is `sint64`, `uint64`, or `double`.
- **length-prefixed**, if the type is `string`, `bytes`, or an embedded `Message`. In this case the value is prefixed with its length encoded as a `varint`.

Each `message` entry is encoded with the *tag* and *wiretype* first, combined into a single `varint`, followed by the length of *value* if necessary, and then the *value* itself.

Although the semantics it offers to users through the type-description language are quite rich, the Protobuf encoding itself is fairly similar to BSON. `Messages` are essentially key-value sets by another name, albeit with much more complex (and efficient) keys. This is because Protocol Buffers is primarily designed for short, transient messages like Remote Procedure Calls over self-contained enterprise networks. Shorter messages make the overhead of metadata and self-description relatively more expensive, which is why Protobuf encodes as little structural metadata as possible. At the same time, RPC schemas are prone to frequent iterative changes as their usage is refined, so Protobuf eschews complex and fragile encodings in favor of the simple key-value structure even when it might be less efficient. Semantics that could theoretically use more efficient encodings (like required message fields, which could be encoded positionally without needing type tags) have been *removed* from the format because they might interfere with forward compatibility. The same trade-offs are, broadly speaking, made by other RPC formats like Apache Thrift and Cap'N'Proto[11],

CHAPTER IV

STRUCTURED DATA FORMAT

This chapter describes the design of a general **Structured Data Format** (henceforth abbreviated SDF) subject to the requirements enumerated in Section .

Given the desire to make the format as efficient and general as possible (Req. 6), it will not be text-based like other common data formats (JSON, XML, etc.). Instead, the primitive underlying SDF will be a stream of bytes, such as a file or network packet. The stream must have a known length (see Section), but is otherwise unrestricted.

Blocks

Given Requirement 1, SDF will be constructed of logical elements called **blocks**, related to one another in a tree structure. Section describes one way to divide a stream into logical elements using a grammar with restricted terminal symbols. This is unsuitable because it requires composed content to be escaped (Req. 3).

Instead, the stream will be broken down into blocks with a **fixed size**, where the **size of every block must be knowable without parsing its content**. This satisfies both Req. 2 and 3 by allowing blocks to contain unrestricted data without required escaping. In particular, blocks can be *compound*, that is contain nested child blocks, satisfying Req. 1.

Fixed-size Blocks

In other binary formats like BSON, certain types like `int64` have a fixed size. It is of course unnecessary to encode any explicit size information for such **primitive** (non-compound) blocks, and BSON does not. SDF follows suit in this regard - fixed-size primitive blocks are are encoded as the serialization of their content, with no additional metadata.

SDF extends the fixed-size concept to compound block types as well. **Any block containing only fixed-size children is itself fixed-size**. Such compound blocks need not encode any size information about their children - if the children are fized-size and stored in some deterministic order then the offset of any child block is directly computable. Therefore, a compound SDF block with only fixed-size children will encode its children sequentially without any additional metadata.

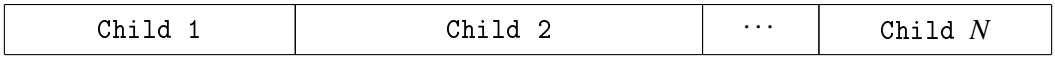

Figure IV.1. A block with N fixed-size children, laid out sequentially.

The layout of the IP packet header from Sec is a good example of what this looks like - it is also a valid SDF block[1]!

Child Block Offsets

Of course, not *every* SDF block will have a fixed size. There must be a mechanism to encode the size of variable-length content explicitly, preferably before the content itself to avoid minimize lookahead in the parser. One possibility, used by BSON (Sec.) is to prefix each block with a short fixed-length field containing it's size:

Figure IV.2. A block with a fixed-length size prefix.

The problem with this scheme is that fixed-length prefixes limit the maximum size of each block (BSON documents, incidentally, are limited to 2GB). The prefix is often too large for common sizes (e.g. most strings stored in JSON and BSON are less than 100 characters; a 4-byte size prefix is 3 bytes too many in that case), meaning wasted space. The prefix is also simultaneously too small for extreme sizes, making the format brittle and limiting its applicability.

An alternative is to use variable-length size prefixes, like the `vint` from Protocol Buffers (Sec.). This is more space efficient, albeit more complex.

A bigger problem with size prefixes in general is that they do not satisfy Req. 5. Traversing to a child block from the parent should be a constant-time operation independent of the size of both child and parent. Consider a parent block with N variable-length child blocks, where the children are laid out sequentially in memory and each is prefixed with a 2-byte size prefix (Fig. IV.3). To find the offset of block N starting from its parent at offset 0, one must visit each child block 1 ... $N-1$ and parse its size in order to

[1] Given a suitable block Type definition, of course.

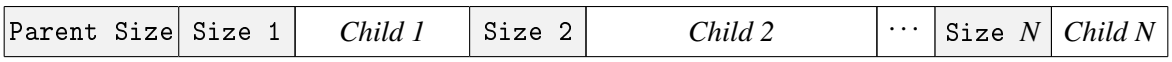

Figure IV.3. A block with N variable-size children, using fixed-length size prefixes.

jump to the next child. This takes time *proportional* to N.

The solution to this is to drop the size prefix entirely, and instead prefix the entire collection of child blocks with an array of fixed-length *offsets* (Fig. IV.4). Obtaining the offset to child block M is then a trivial

Figure IV.4. A block with N variable-size children, using N fixed-length offsets.

constant-time operation, and the size can be recovered by subtracting from the the offset of block $M+1$ (or the size of the parent block, for $M = N$). This decoupling of block size from block content turns out to be a very elegant solution overall.

As an additional bonus, since the child blocks start immediately after the last offset value, it is not necessary to explicitly store and offset for the first child[2] (Fig. IV.5 and IV.6). This is an especially useful

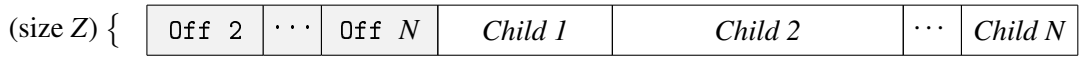

Figure IV.5. A block with N variable-size children, using $N-1$ fixed-length offsets (Off. 1 being implicit).

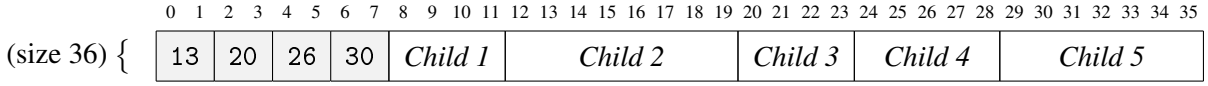

Figure IV.6. A block with 5 variable-size children, using 4 2-byte offsets.

optimization when the parent block has only one variable-sized child - in this case, no offsets are stored, and the size of the child is the size of the parent.

In summary, SDF variable-length blocks are not encoded with any explicit size information. The size is encoded by the *parent* block, indirectly, in a table of offsets. The size of a variable-length root block, which has no parent, must be supplied by the context from which the block was obtained. For SDF blocks

[2] Assuming, of course, that the number of children N is also known along with the size. See Section for a discussion of blocks with a variable number of children.

read from a file, for example, the size of the file would specify the size of the root block.

Blocks with Mixed-size Children

For blocks that contain both fixed-size and variable-size content, a combination of the previous two strategies is adopted. SDF maintains the logical order of the child blocks. Fixed-size child blocks are encoded sequentially, without repositioning or any additional metadata. Where a variable-sized child block other than the first would appear, a fixed-size offset to its content is encoded instead. The offset of the first variable-sized child block is implicitly given by the total size of the fixed-size children and offsets. The content of all variable-size blocks is encoded sequentially at the end of the parent block, after the fixed-size content and offsets.

As an example, consider is a block with 5 children having sizes:

`fixed` (5), *variable* (7), *variable* (4), `fixed` (4), *variable* (6)

The SDF encoding for this block (Fig. IV.7) would place the content of Child 1 first, followed by the offset (assumed here to be a 2-byte integer) to Child 3. Then follows the content of Child 4, then the offset to Child 5, and finally the content of Children 2, 3, and 5 in that order. It remains possible (although somewhat

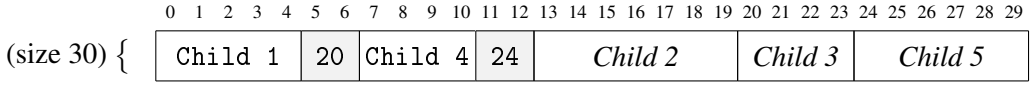

Figure IV.7. A block with 2 fixed- and 3 variable-size children, using 2 fixed-length offsets.

tedious to do by hand) to compute the offset and size of each child in constant time, without parsing any of the child block contents. Further, this strategy reduces to those described in Sections and in the case that the block has only fixed-size or variable-sized children, respectively.

Block Types

Each SDF block has a type. Up to this point, the term "type" has been used loosely to encompass everything from the size and hierarchical structure of the block to its semantics and presentation to the user (e.g. the distinction between the BSON "types" `int64` and `uint64`). However, in regards to SDF blocks, the **type** represents knowledge of:

- Whether the block has fixed or variable size, and the exact size if fixed.
- How many children the block has, or for blocks with a variable number of children, the means by which this quantity is encoded.
- The type of each child, or for children whose type varies, the means by which the reified type is encoded.

As with sizes, **the type of every block must be knowable without parsing its content**. This constraint is trivially true for formats like JSON and BSON, which support a very limited set of types defined explicitly in their respective specifications. Protocol Buffers (Sec.), on the other hand, allows user-defined types. It requires users to supply these type definitions to the protobuf encoder and parser out-of-band in order to process messages. Only a few bits of type information are encoded alongside the data as a sanity check. This reflects the common usage of protocol buffers as a medium for network communication such as RPC, where messages are short and transient and a single entity controls both endpoints and the software running on them.

SDF is also built around user-defined block types. Because it seeks to be a general-purpose data format, however, it is not reasonable to expect users to store and supply type definitions out-of-band like Protocol Buffers. Persistent data storage on disk, for example, needs more than simple sanity checks to rescue it if the original type definitions are lost or modified. In asymmetric communications like those of a typical public webserver, there simply *is* no out-of-band channel along which to send extra information. It is necessary for SDF to provide some mechanism for encoding type definitions directly. Of course, one of the fundamental requirements for SDF is the ability to encode *arbtrary metadata* (Req. 4). Block type definitions are just a special case of this feature.

Thus, SDF does not have a simple list of fundamental types like those in Sections and . All of the types in those sections are (or at least could be) user-defined types in SDF, but they need not be fundamental or built in to the format in any way. Instead, SDF has five type *Kinds* (in the type-theoretical sense): **Primitive**, **Union**, **Struct**, **Array**, and **Meta**. These are type constructors; they are the building blocks of type definitions, which in turn define the actual concrete block types.

Primitive Types

As in other formats, primitive types are those with no internal structure, or whose structure is opaque to the format. Consequently, primitives have no children. They are encoded as a simple stream of bytes.

A `Primitive` type may have a fixed size specified in its definition (including zero, for those types where it makes sense), or it may be of variable size.

Union Types

`Unions`, or "tagged enums" in C parlance, are sum-types. They represent a choice between several possible values, called **variants**. They have exactly one child, but the type of that child value depends on the selected variant. Unions are the foundation of variable-typed data structures and polymorphism in SDF.

`Unions` are encoded starting with a fixed-size **discriminant**: an integer specifying the variant. The content of the child block is encoded immediately afterwards (Fig. IV.8).

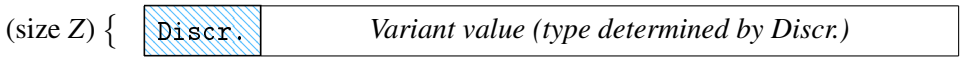

Figure IV.8. The encoding scheme for `Union` types.

A `Union` type is fixed-size if and only if (1) all of its variants are fixed-size, and (2) all of the variants have the *same* fixed size. This contrasts with, for example, C unions where the size is that of the *largest* variant. The decision to use a tighter constraint in SDF is due to the alignment requirement (Req. 7), which forbids unnecessary padding and filler that is not explicitly part of the type definition.

Struct Types

`Structs` are heterogeneous product-types, very similar to the C struct type. A `Struct` has a fixed number of children (possibly zero) called **fields**, each with an associated type.

`Structs` are encoded using the mixed-block strategy described in Section . As they have a fixed number of fields with fixed types, in a fixed order, so no additional information needs to be encoded. A `Struct` type has a fixed size if and only if all of its fields have a fixed size (in which case, the size of the parent struct is the sum of the field sizes).

Array Types

`Arrays` are exponential-types, again very similar to arrays in C. They represent an ordered sequence of child values called **elements**. While the type of all children must be fixed (and homogeneous), the number of children may be variable or fixed, depending on the type definition.

`Arrays` are encoded exactly like the equivalent `Struct` with the same number of children (see Sec.). Variable-count arrays are prefixed with a fixed-size field containing the number of elements. An `Array` type has a fixed size if (1) its element type has a fixed size and (2) it has a fixed number of elements.

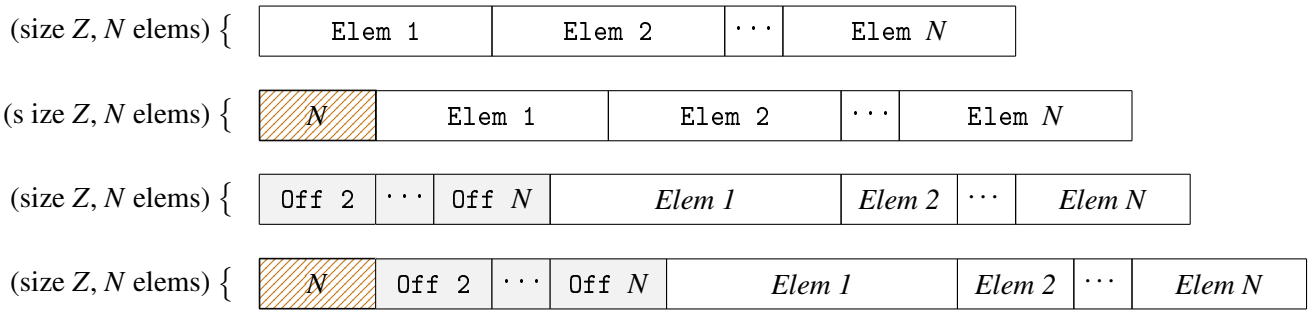

Figure IV.9. The encoding scheme for `Array` types.

Meta Types

`Meta` types are the final and most unique SDF type kind. They represent the definition of **metadata** (an `Array` of arbitrary blocks whose semantics are defined by the user) for some **data** block with a fixed type. Structurally, `Meta` blocks behave as if they had only one child, the data block. The metadata is available to be read, but is explicitly not part of the regular hierarchical structure of an SDF block tree.

`Meta` types are encoded as if they were a `Struct` with *two* children. The first, "hidden" child is the metadata block, a variable-count array of individual metadata blocks. The second child is the data block. A `Meta` type always has variable size, by virtue of the variable count of its internal meta array.

Figure IV.10. The encoding scheme for `Meta` types.

`Meta` blocks are the means by which SDF allows the inclusion of arbitrary metadata (Req. 4). The core use of `Meta`-type blocks is to encode the definition of SDF types. Any block type which is not built-in (see Sec.) must be defined in a `Meta` node which contains all blocks of that type.

Concrete Types

Because of the mandate for simplicity (Req. 8), and because of the support for user-defined types (see Section), SDF does not need to define many concrete types up-front. There are four block types which are necessary in order to allow type definitions:

typedef The type of type definitions. This is a very complex type, with five nested subtypes corresponding to the five different type kinds. `Typedefs` are recursive, meaning that they can contain nested type definitions. A full description of SDF typedefs can be found in the Appendix.

anydef A list of type definitions which may appear in the metadata of a `Meta` block. These types become new **top-level**, or "named" types in the scope of the meta's data block.

any A special `Union` type with a variant for *every top-level type*. `Any` is unique in that its definition changes as new types are added. This is the only type with this property, and it obviously must be hard-coded into the parser.

metaany A `Meta` type whose data type is `Any`. There obviously must be at least one `Meta` type pre-defined in order for documents to begin defining their own types.

In addition, it is necessary to define some kind of **unsigned integer** block type. Unsigned integers are implicitly required for the encoding of block offsets, `Union` discriminants, and `Array` element counts. The full set of built-in types in the Appendix actually defines 4 unsigned integer types - one each of size 8, 16, 32, and 64 bits. All of these types are encoded in Big-Endian (network-native) byte order.

CHAPTER V

APPLICATION TO JSON

This chapter examines the use of SDF to transcode documents from the JSON format (Sec.), and benchmark the transcoded versions against both the original JSON documents and an equivalent BSON documents (Sec.).

JSON was chosen as a basis for comparison for two reasons. First, JSON is a simple format and (unlike, say, HTML) publicly available JSON documents are unlikely to contain syntax errors or other pathological issues that would make benchmarking more difficult. Second, JSON is a very popular format and there are a large number of implementations and publicly available JSON datasets covering large variety of use cases.

Data Sources

About 180MiB of raw JSON data was collected from 5 different sources:

- **Github.com** [12], which provides public API endpoints to recent events (commits, forks, issues posted, etc.) involving source code repositories.
- **Reddit.com** [13], which provides public API endpoints to metadata about the front pages of several popular "subreddit" forums.
- **JSONStudio.com** [14], which hosts several curated JSON datasets for users to test its software with. These include historical stock market metadata and a database of emails.
- **Data.gov** [15], which curates public-sector datasets from Federal, State, and City level governments. The datasets used include demographics from New York City and Birth & Fertility rates measured by the CDC.
- **Common Crawl** [16], an community-run open web crawler that provides a complete snapshot of the WWW every six weeks. The metadata on each crawled page is available as a (very large!) JSON database.

SDF Type Definitions for JSON

In order to represent data from a JSON document, it is necessary to define SDF block types corresponding to the core JSON data types. In fact, three slightly different SDF schemas were defined and benchmarked

independently.

- The values `null`, `true`, and `false` were defined as new `Primitive` types with a fixed size of zero, similarly those used in BSON (Sec.).
- Integer-valued `Numbers` were split amongst four SDF types `int8`, `int16`, `int32`, `int64`, representing signed integers and encoded as fixed-size `Primitive` types in network byte order. Each integer parsed from the JSON was encoded in the smallest type that is large enough to hold it.
- Non-integer `Numbers` were split amongst two SDF types `float32`, `float64`, representing IEEE 754 binary floats and encoded as fixed-size `Primitive` types in network byte order. Each float parsed from the JSON was encoded in the smallest type that is large enough to hold it.
- `Numbers` too big to be represented losslessly in one of the other number types were represented as `vint`, a variable-size `Primitive` type encoding the string representation of the number as written in the original JSON. This type is necessary because JSON does not limit the size of numbers, but is rarely used in practice (in fact, it was not used at all in by any of the test data).
- `Strings` were represented as `str`, a variable-size `Primitive` type encoding the UTF-8 string.
  One of the alternative schemas also includes `stz`, a variable-size `Primitive` type encoding the string after compression with `zlib`. Strings in this schema are encoded into `str` or `stz`, whichever results in a smaller encoded block.
- `Arrays` were represented as `arr`, a variable-count `Array` type whose elements have type `jsonAny`.
- `Objects` were represented as `obj`, a variable-count `Array` of entries. Each entry is an anonymous `Struct` type with two fields, "key" of type `str`, and "val" of type `jsonAny`.
  The two alternative schemas replace the string key type with an anonymous `Union` type that has one variant for every unique object key in the JSON document, and whose values are all empty `Primitives`.
- `jsonAny` is a variable-sized `Union` type with one variant for each of the other types in the schema. This is the element type for `arr` and the entry-value type for `obj`.

The primary schema is a very straight-forward representation of JSON data in SDF. The only real difference is that numeric values are represented as fixed-size integers or floats when possible. In particular, the primary schema uses literal string keys for `Object` entries just as JSON does.

The first alternative schema does not use string keys - instead it builds a custom `Union` type with one variant for every unique `Object` key in the original JSON document. This union type is used for the Object keys, effectively replacing the literal strings with union discriminants. This can yield significant space savings for JSON documents with a small set of frequently-repeated keys (see Sec.). The second alternative type also uses `Union`-style object keys, and also includes a special compressed string type `stz` for compressed strings. This can yield significant space savings for documents with long string values, such as email bodies.

A small transcoder utility was written in C to parse the JSON test data and re-encode it in SDF using these schemas.

Benchmark: Encoded File Size

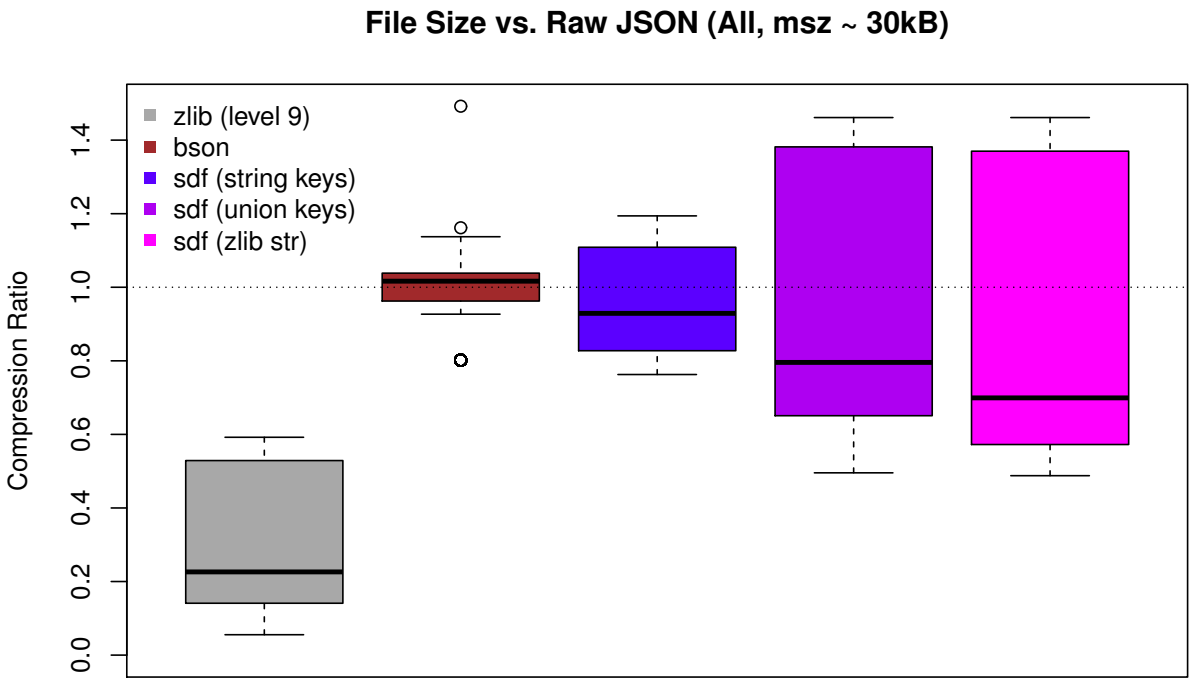

Figure V.1. Size ratio of JSON documents in various formats, relative to the original JSON.

The size of each JSON document in the data set was compared for each of 5 transcodings:

- The document after compression with `zlib -l9`.
- A BSON (Sec.) document.

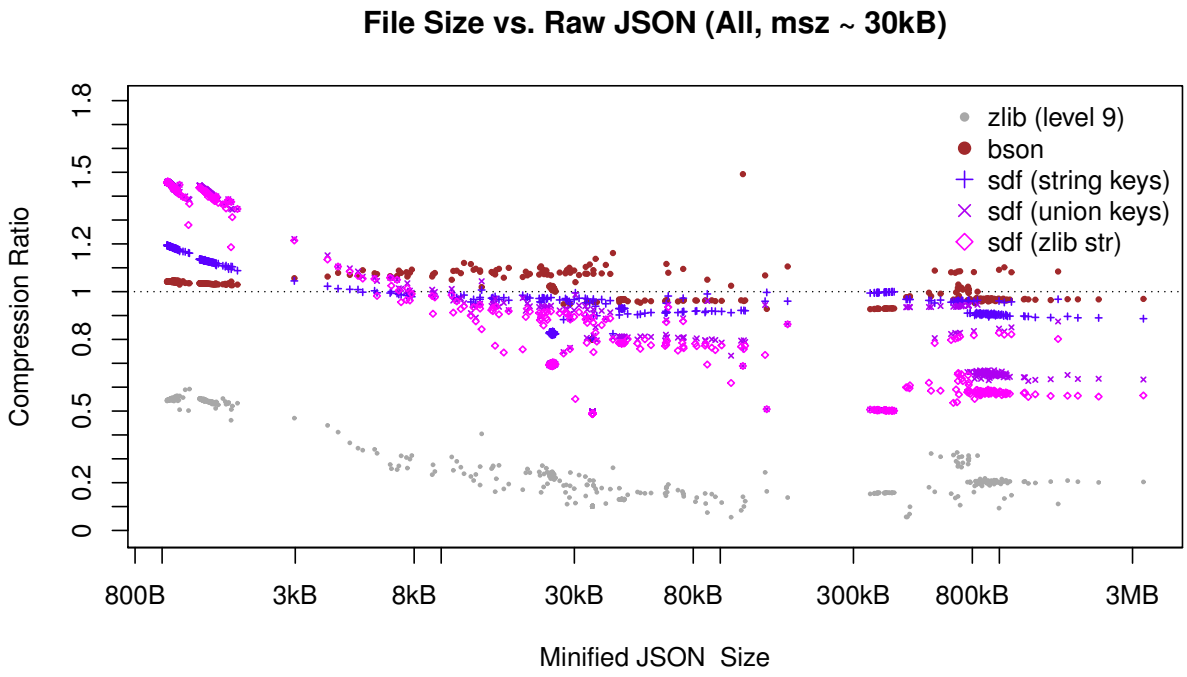

Figure V.2. Size ratio of JSON documents in various formats vs. the size of the original JSON.

- An SDF document in the primary schema.
- An SDF document using `Union`-typed object keys instead of strings.
- AN SDF document using `Union`-typed object keys and optional compression of long string values.

The basis for size comparison is the "minified" form of the original JSON document, with all whitespace characters stripped out. Although JSON is usually "prettified" using whitespace and indentation to clarify the structure, the whitesapce is redundant to the existing punctuation and can be added or removed mechanically. JSON documents are often minified before transmission to save bandwidth, and this was done for the test data to provide a fairer comparison.

Fig. V.1 shows the general distribution of transcoding sizes across all of the available test data. Zlib compression, of course, achieves excellent results with an median compression ration of 22.3% over the the entire data set. SDF performs more modestly, with a median compression ratio of 92.9%, 79.6%, and 69.9% for the three schemas respectively. Notable is that BSON does not do well - the median BSON document is actually 1.6% *larger* than the original JSON.

Aggregate statistics can be a bit misleading, however, as the actual compression ratio varies significantly

with the JSON document size for all five transcodings. Fig. V.2 plots the compression ratio for every source document individually, showing that all five formats are actually capable of compressing *large* JSON documents, but have differing levels of overhead that dominate smaller document sizes.

In particular, the three SDF schemas are fully self-describing - each file transcoded contains a complete definition of the block types for the schema, as described in Sec. . This ~300 bytes of extra metadata results in very poor compression ratios for JSON documents less than about 5kB in size. The effect is worse for the alternative SDF schemas which define object keys as a `Union` type in the metadata header. The format for type definitions is not particularly well optimized for compactness, and it turns out that each key takes up significantly more space when defined this way than it does as a string literal. For smaller documents, and particularly for those which did not repeat keys, the alternative schemas had extremely poor compression ratios. However, for large documents with many repeated keys, the alternative schemas offered compression ratios below 60%.

The second alternative schema, which allows long string values to be compressed, is worth discussion. In general it is more efficient to compress and entire document at once, rather than compressing single blocks individually. This is borne out by the very impressive `zlib` results. The downside of bulk compression, however, is that it requires the entire file to be decompressed before it can be traversed. When only leaf nodes in the document are compressed, on the other hand, traversal time is unaffected. Thus, although the compressed string type `stz` was not nearly as efficient as full-document compression, the benefits it did provide (sub-50% compression ratios for certain data) came essentially for free. The ability to encode such ad-hoc optimizations highlights the flexibility of the SDF format.

Benchmark: Parser Traversal Time

The time to parse each JSON document and extract a single node was compared for each of 5 transcodings (see Sec.). A representative node was chosen for each file, with the goal being to benchmark the "random access" traversal efficiency of the underlying format. Unfortunately, code quality or engineering decisions in the implementation of a parser can have a significant effect on the results of runtime benchmarks, potentially dwarfing the properties of the format itself. For this reason, *two* independent JSON parsers were benchmarked. The specific parsers are **cJSON**[17] and **parson**[18]. Both of these libraries are written in C and did well in a benchmark study of JSON libraries published by M. Yip[19].

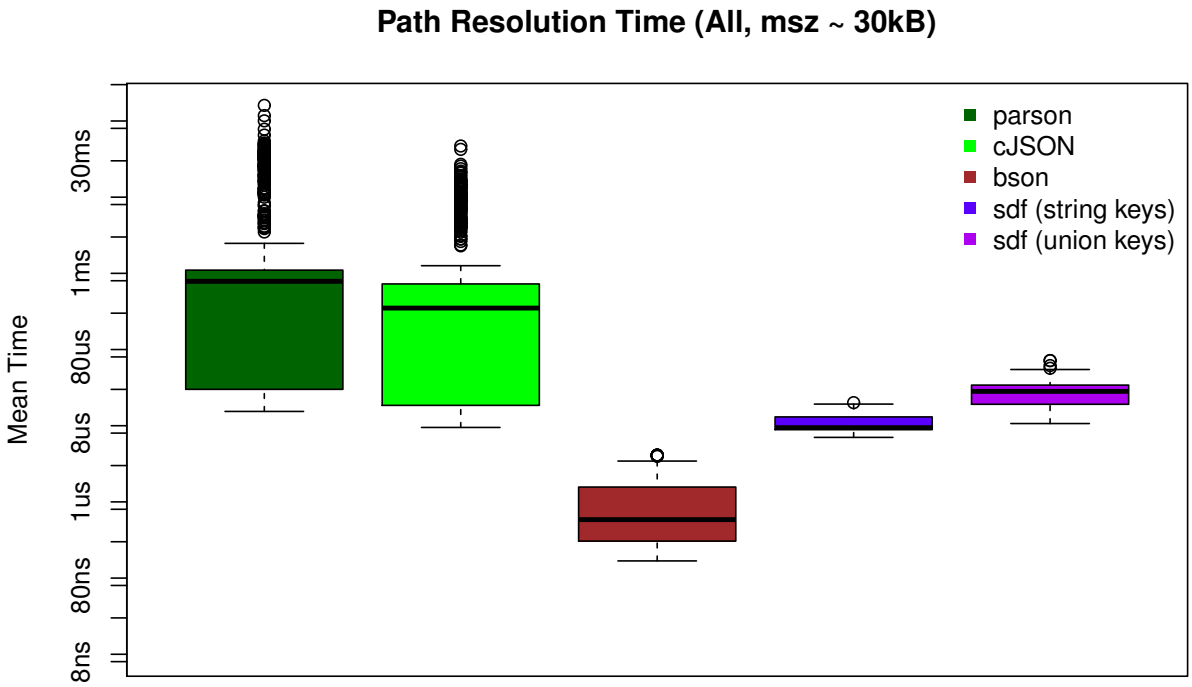

Figure V.3. Traversal time for JSON, BSON, and SDF, over all available datasets.

BSON traversal was benchmarked with the canonical BSON implementation developed by MongoDB[8]. As seen in the results, there is no reason to doubt the quality or optimization of this implementation.

A short C program was used to benchmark the SDF schemas. All three were tested, but the results of the two alternative schemas matched so closely that they are reported together in this section for brevity.

The benchmark results are summarized in Fig. V.3. The two JSON parsers performed comparably, suggesting the effect of implementation details was not significant. BSON is a clear leader in performance - despite the large size of the BSON files parsing is done on the order of microseconds per file. This very close to the resolution of the benchmark itself, meaning that true BSON performance numbers may actually be better. For the median document, BSON is able to traverse and extract arbitrary nodes from a document about 400 times faster than the JSON parsers searching for the same information.

The performance of SDF is, as with size, more modest. In the median case, it performed about 10-30 times faster than the JSON parsers, but about 20 times slower than the BSON implementation. That SDF is slower than BSON is somewhat vexing - it offers constant-time traversal of large nodes while BSON does not, and thus should have an edge. One thing to bear in mind, of course, is that the BSON parser is the result of significant corporate investment for use in a very performance-sensitive application. It should be expected

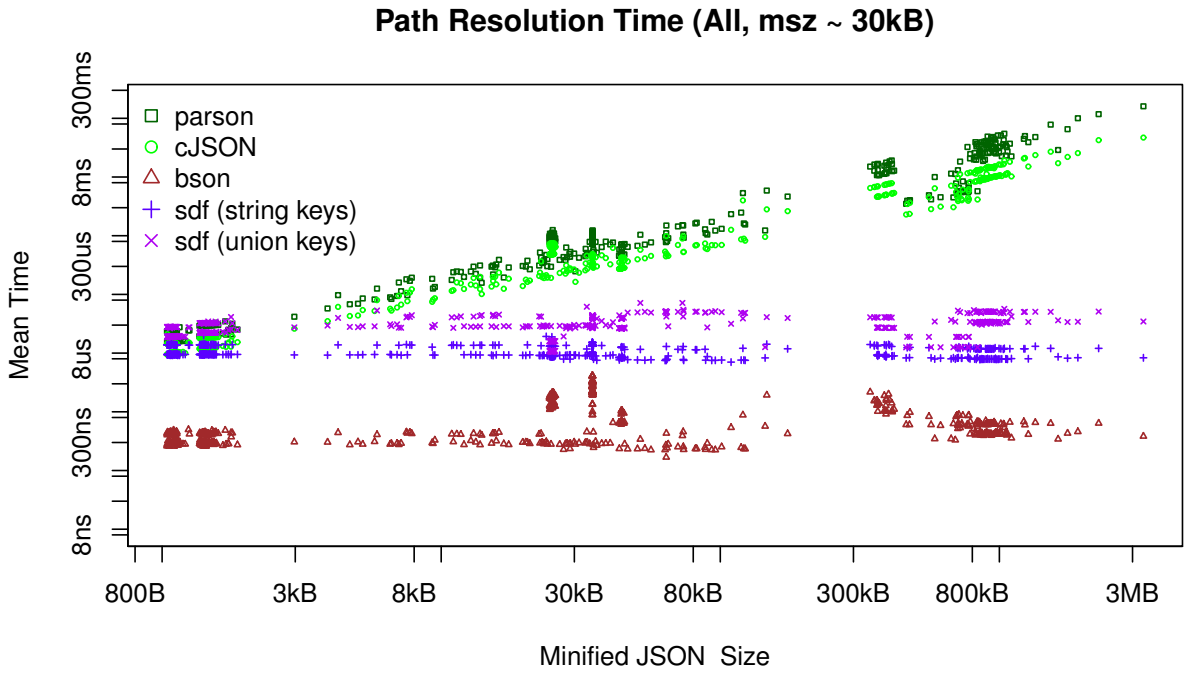

Figure V.4. Traversal time for JSON, BSON, and SDF, over all available datasets.

that further refinement of the SDF parser would at least close the gap somewhat. Careful examination of the results, however, indicates that the SDF parser *spends significant time parsing the metadata to extract type definitions*. This is a fundamental cost that is not easily mitigated. This is an important result - it highlights the real, measurable performance cost of the extra complexity that comes with a self-describing data format.

The difference between the three SDF schemas is quite consistent: using literal strings as `Object` keys leads 3-4 times faster traversal than the more complex `Union`-typed keys. This may be due to the extra layer of indirection involved, or it may simply be due to the extra time required to parse the larger type definitions. Either way, it serves to reinforce the (often unforeseen) cost of adding complexity to the schema.

Figure V.4 shows the traversal time as a function of raw JSON size. For both of the JSON parsers traversal time grows linearly with input size[1]. For SDF and BSON, however, traversal time is independent of document size and in fact quite consistent even across different paths (the exact node path traversed varied for each document). This illustrates one of the main strengths of the two formats.

[1] Fig. V.4 is plotted on log-log axes, so the fact that the growth is *linear* rather than some other power law is not obvious. The raw data, however, show a clear linear relationship.

APPENDIX

SDF BUILT-IN TYPES

This Appendix contains a full listing of the built-in types used in the implementation of SDF developed for this work. Some core block types are *required* by the SDF specification (see Sec.), but some of the details that go into their definition were omitted in that section.

In particular, it is mentioned that `Union` discriminants, `Array` element counts, and the offets of variable-size child blocks must be encoded as fixed-length unsigned integers in Big-Endian byte order. The exact length, however, was left unspecified. The particular `typedef` type given here allows individual type definitions to choose how large these internal fields should be.

`Uniondef` is itself is a union with four variants, one variant each for 8, 16, 32, and 64 bit discriminants. There is no "dynamic" option; each `Union` type must specify a fixed discriminant size directly in its definition.

`Arraydef` has a "counts" option. For fixed-count arrays, this option contains the actual number of array elements. For variable-count arrays, this option selects the count prefix to be an 8, 16, 32, or 64 bit integer. Variable-count arrays can also have a *dynamically calculated* count - the number of elements is calculated to be the size of the parent block divided by the element size. In this case no explicit count prefix is needed - but this option value is only legal for `Arrays` with fixed-size elements.

`Structdef`, `Arraydef`, and `Metadef` all have an "offsets" option which causes the offsets for blocks of that type to be 8, 16, 32, or 64 bit integers. This option also allows blocks to have a *dynamically calculated* length for their offsets - that is, offsets use the smallest of the four integer types that is large enough to contain the size of the parent block (which must be >= the offset of the last child block). This is allowed within the SDF Requirements because "dynamic" offset values still have a length that can be calculated from the parent block size without traversing or parsing any of the child content.

```
1 | 1  'any'        : UNION(size=*, discr_sz=8)
2 |        [all top-level types are variants of 'any']
3 | 2  'metaany'    : META(size=*, off_sz=log(sz))
4 |        meta: metaarray
5 |        data: any
6 | 3  'metaarray' : ARRAY(size=*, elems=*, count_sz=4, off_sz=log(sz))
7 |        elem: any
8 | 8  'typedef'    : UNION(size=*, discr_sz=1)
9 |         0  'primdef'    : primdef
```

```
          1  'uniondef'   : uniondef
          2  'structdef' : structdef
          3  'arraydef'   : arraydef
          4  'metadef'    : metadef
         10  'typeref8'   : uint8
         11  'typeref16'  : uint16
         12  'typeref32'  : uint32
         13  'typeref64'  : uint64
 9  'primdef'    : STRUCT(size=*, fields=1, off_sz=log(sz))
        'size'        : UNION(size=*, discr_sz=1)
             0  'static'     : uint64
             1  'dynamic'    : unit
10  'uniondef'   : UNION(size=*, discr_sz=1)
          0  'fixed8'     : ARRAY(size=*, elems=*, count_sz=4, off_sz=4)
                elem: STRUCT(size=*, fields=3, off_sz=log(sz))
                    'discr'       : uint8
                    'name'        : utf8
                    'type'        : typedef
          1  'fixed16'    : ARRAY(size=*, elems=*, count_sz=4, off_sz=4)
          2  'fixed32'    : ARRAY(size=*, elems=*, count_sz=4, off_sz=4)
          3  'fixed64'    : ARRAY(size=*, elems=*, count_sz=4, off_sz=4)
                elem:  [see 'fixed8' above, but uint16/32/64 discr]
11  'structdef' : STRUCT(size=*, fields=2, off_sz=log(sz))
        'offsets'     : UNION(size=1, discr_sz=1)
             0  'fromsize'   : unit
            10  'fixed8'     : unit
            11  'fixed16'    : unit
            12  'fixed32'    : unit
            13  'fixed64'    : unit
        'fields'      : ARRAY(size=*, elems=*, count_sz=4, off_sz=4)
            elem: STRUCT(size=*, fields=2, off_sz=log(sz))
                'name'        : utf8
                'type'        : typedef
12  'arraydef'   : STRUCT(size=*, fields=3, off_sz=log(sz))
        'counts'      : UNION(size=*, discr_sz=1)
             0  'static'     : uint64
             1  'fromsize'   : unit
            10  'fixed8'     : unit
            11  'fixed16'    : unit
            12  'fixed32'    : unit
            13  'fixed64'    : unit
        'offsets'     : UNION(size=1, discr_sz=1)
             0  'fromsize'   : unit
            10  'fixed8'     : unit
            11  'fixed16'    : unit
            12  'fixed32'    : unit
            13  'fixed64'    : unit
        'type'        : typedef
13  'metadef'    : STRUCT(size=*, fields=2, off_sz=log(sz))
        'offsets'     : UNION(size=1, discr_sz=1)
             0  'fromsize'   : unit
```

```
61 |             10  'fixed8'     : unit
62 |             11  'fixed16'    : unit
63 |             12  'fixed32'    : unit
64 |             13  'fixed64'    : unit
65 |         'datatype'    : typedef
66 | 14  'anydef'     : [identical to 'uniondef']
67 | 16  'bottom'     : PRIMITIVE(size=0)
68 | 17  'unit'       : PRIMITIVE(size=0)
69 | 18  'bool'       : UNION(size=1, discr_sz=1)
70 |           0  'false'      : unit
71 |           1  'true'       : unit
72 | 24  'uint8'      : PRIMITIVE(size=1)
73 | 25  'uint16'     : PRIMITIVE(size=2)
74 | 26  'uint32'     : PRIMITIVE(size=4)
75 | 27  'uint64'     : PRIMITIVE(size=8)
76 | 32  'int8'       : PRIMITIVE(size=1)
77 | 33  'int16'      : PRIMITIVE(size=2)
78 | 34  'int32'      : PRIMITIVE(size=4)
79 | 35  'int64'      : PRIMITIVE(size=8)
80 | 40  'float32'    : PRIMITIVE(size=4)
81 | 41  'float64'    : PRIMITIVE(size=8)
82 | 48  'bytes'      : PRIMITIVE(size=*)
83 | 49  'ascii'      : PRIMITIVE(size=*)
84 | 50  'utf8'       : PRIMITIVE(size=*)
```

REFERENCES

[1] T. Dean, *Network+ guide to networks*. Cengage Learning, 2012.

[2] A. J. et al., "The open group base specifications issue 7," The IEEE and The Open Group, Tech. Rep., 2016. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799.

[3] J. G. et al., "The java language specification," Oracle Inc., Tech. Rep., 2015. [Online]. Available: https://html.spec.whatwg.org/.

[4] R. Fielding and J. Reschke, "Hypertext transfer protocol (http/1.1): message syntax and routing," Internet Engineering Task Force, Tech. Rep., 2014.

[5] "Html living standard," WHATWG, Tech. Rep., 2017. [Online]. Available: https://html.spec.whatwg.org/.

[6] "The json data interchange format," Ecma International, Tech. Rep., 2013. [Online]. Available: http://json.org/.

[7] A. Hinds and T. Parr, ANTLR Grammar for HTML, 2004, [Online]. Available: http://www.antlr3.org/grammar/HTML/html.g.

[8] "Bson 1.1 specification," MongoDB, Inc., Tech. Rep., 2014. [Online]. Available: http://bsonspec.org/spec.html.

[9] "Protocol buffers: encoding," Google Developers, Tech. Rep., 2016. [Online]. Available: https://developers.google.com/protocol-buffers/docs/encoding.

[10] "Protocol buffers: language guide," Google Developers, Tech. Rep., 2016. [Online]. Available: https://developers.google.com/protocol-buffers/docs/proto3.

[11] K. Varda, "Cap'n proto: encoding spec," Tech. Rep., 2016. [Online]. Available: https://capnproto.org/encoding.html.

[12] (2017). Github data apis, [Online]. Available: https://api.github.com.

[13] (2017). Reddit api, [Online]. Available: https://www.reddit.com/dev/api.

[14] (2014). Json datasets, [Online]. Available: http://jsonstudio.com/resources.

[15] (2017). Data catolog - json, [Online]. Available: https://catalog.data.gov/dataset?res_format=JSON.

[16] (2017). Wat page metadata, [Online]. Available: http://commoncrawl.org/.

[17] D. Gamble and M. Bruckner. (2017). Cjson ultralightweight json parser in ansi c, [Online]. Available: https://github.com/DaveGamble/cJSON#welcome-to-cjson.

[18] K. Gabis. (2017). Parson lightweight json library written in c, [Online]. Available: http://kgabis.github.io/parson/.

[19] M. Yip. (2016). C/c++ json parser/generator benchmark, [Online]. Available: https://github.com/miloyip/nativejson-benchmark.