A Survey of JSON-compatible Binary Serialization Specifications

Juan Cruz Viotti*
Department of Computer Science
University of Oxford
Oxford, GB OX2 6PN
juancruz.viotti@kellogg.ox.ac.uk

Mital Kinderkhedia
Department of Computer Science
University of Oxford
Oxford, GB OX2 6PN
mital.kinderkhedia@cs.ox.ac.uk

Abstract

In this paper, we present the recent advances that highlight the characteristics of JSON-compatible binary serialization specifications. We motivate the discussion by covering the history and evolution of binary serialization specifications across the years starting from 1960s to 2000s and onwards. We analyze the use cases of the most popular serialization specifications across the industries. Drawing on the schema-driven (ASN.1, Apache Avro, Microsoft Bond, Cap'n Proto, FlatBuffers, Protocol Buffers, and Apache Thrift) and schema-less (BSON, CBOR, FlexBuffers, MessagePack, Smile, and UBJSON) JSON-compatible binary serialization specifications, we compare and contrast their inner workings through our analysis. We explore a set of non-standardized binary integer encoding techniques (ZigZag integer encoding and Little Endian Base 128 variable-length integer encoding) that are essential to understand the various JSON-compatible binary serialization specifications. We systematically discuss the history, the characteristics, and the serialization processes of the selection of schema-driven and schema-less binary serialization specifications and we identify the challenges associated with schema evolution in the context of binary serialization. Through reflective exercise, we explain our observations of the selection of JSON-compatible binary serialization specifications. This paper aims to guide the reader to make informed decisions on the choice between schema-driven or schema-less JSON-compatible binary serialization specifications.

1 Introduction

1.1 Serialization and Deserialization

Serialization is the process of translating a data structure into a *bit-string* (a sequence of bits) for storage or transmission purposes. The original data structure can be reconstructed from the bit-string using a process called *deserialization*. Serialization specifications define the bidirectional translation between data structures and bit-strings. Serialization specifications support persistence and the exchange of information in a machine-and-language-independent manner.

Serialization specifications are categorized based on how the information is represented as a bit-string i.e. *textual* or *binary* and whether the serialization and deserialization processes require a formal description of the data structure i.e. *schema-driven* or *schema-less*. Before we go into a detailed discussion about textual and binary serialization specifications, we motivate by discussing the history and evolution of serialization specifications.

*https://www.jviotti.com

Preprint. Under review.

1.2 History and Evolution of Serialization Specifications

1960s. In 1969, IBM developed *GML* (Generalized Markup Language) [2], a markup language and schema-less textual serialization specification to define meaning behind textual documents. Decades later, XML [99] was inspired by GML.

1970s. In 1972, IBM OS/360 introduced a general-purpose schema-less serialization specification as part of their FORTRAN suite [64]. The IBM FORTRAN manuals referred to the serialization specification as *List-Directed Input/Output*. It consisted of comma-separated or space-separated values that now resemble the popular CSV [122] schema-less textual serialization specification.

1980s. Microsoft invented the *INI* general purpose schema-less textual serialization specification [3] as part of their MS-DOS operating system in the early 1980s (the exact year is unclear). In 2021, the Microsoft Windows operating system continues to make use of the INI specification. INI also inspired the syntax of configuration file formats in popular software products such as *git* [4] and *PHP* [5]. In 1983, the Osborne Executive portable computer Reference Guide [33] used the term *CSV* to refer to files containing comma-separated rows of data. In 1984, the *International Telecommunication Union* [6] specified the *ASN.1* schema-driven binary serialization specification as part of the [117] standard. The ASN.1 serialization specification became a standalone standard in 1988. In 1986, the *SGML* (Standard Generalized Markup Language), a descendant of IBM GML to define custom markup languages, was proposed as an ISO standard [38]. In the late 1980s, NeXT introduced the

[2] `http://www.sgmlsource.com/history/roots.htm`

[3] `https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2008-R2-and-2008/cc731332(v=ws.11)?redirectedfrom=MSDN`

[4] `https://git-scm.com`

[5] `https://www.php.net`

[6] `https://www.itu.int/ITU-T/recommendations/index.aspx?ser=X`

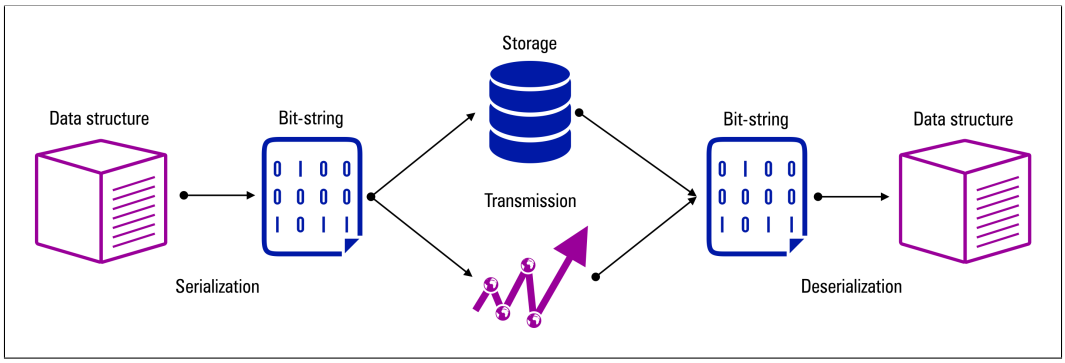

Figure 1: The process of translating a data structure to a bit-string is called *serialization*. The process of translating a bit-string back to its original data structure is called *deserialization*.

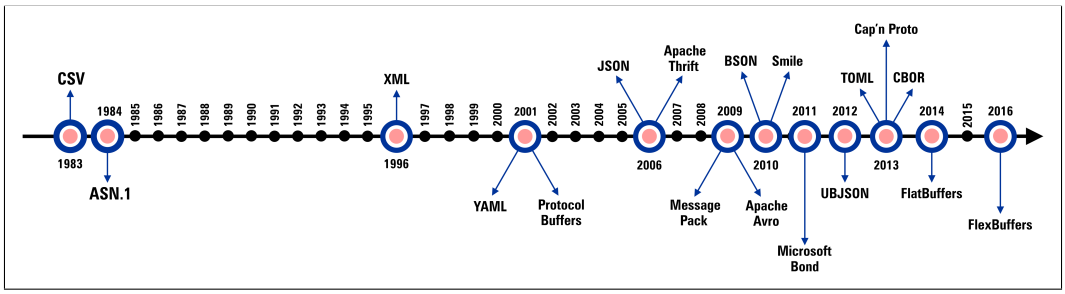

Figure 2: A timeline showcasing some of the most popular serialization specifications since the early 1980s.

schema-less textual ASCII-based [27] *Property List* serialization format [7] which is now popular in Apple's operating systems.

1990s. In 1996, the Netscape web browser used stringified representations of JavaScript data structures for data interchange [35]. This serialization approach would be standardized as JSON [18] a decade later. Also, the SGML [38] language inspired the first working draft of a general-purpose schema-less textual serialization specification named *XML* (Extensible Markup Language) [19]. In 1997, The Java programming language JDK 1.1 defined a *Serializable* interface [8] that provided binary, versioned and streamable serialization of Java classes and their corresponding state. This serialization specification is referred to as *Java Object Serialization* [9] and it is still in use. In 1998, [97] further improved this object serialization technique. In 1999, XML became a W3C (World Wide Web Consortium) [10] recommendation [19].

2000s. In 2000, Apple (who acquired NeXT in 1997) introduced a binary encoding for the *Property List* serialization specification [11]. A year later, Apple replaced the ASCII-based [27] original *Property List* encoding with an XML-based encoding [12]. In 2001, Google developed an internal schema-driven binary serialization specification and RPC protocol named *Protocol Buffers* [59]. In the same year, the first draft of the schema-less textual *YAML* [9] serialization specification was published as a human-friendly alternative to XML [99]. Refer to [44] for a detailed discussing of the differences between YAML and JSON. The widely-used *CSV* [122] schema-less textual serialization specification was standardized in 2005. The first draft of the *JSON* schema-less textual serialization specification was published in 2006 [34]. In the same year, Facebook developed an open-source schema-driven binary serialization specification and RPC protocol similar to Protocol Buffers [59] named *Apache Thrift* [126]. In 2008, Google open-sourced *Protocol Buffers* [59]. In 2009, the *MessagePack* [56] schema-less binary serialization specification was introduced by Sadayuki Furuhashi [13]. Two other binary serialization specifications were released in 2009: The Apache Hadoop [14] framework introduced the *Apache Avro* [52] schema-driven serialization specification. The MongoDB database [15] introduced a schema-less serialization alternative to JSON [18] named *BSON* (Binary JSON) [86].

Advances since 2010. Two new schema-less binary serialization specification alternatives to JSON [18] were conceived in 2010 and 2012: *Smile* [116] and *UBJSON* [17], respectively. In 2011, Microsoft developed *Bond* [87], a schema-driven binary serialization specification and RPC protocol inspired by Protocol Buffers [59] and Apache Thrift [126]. In 2013, the lessons learned from Protocol Buffers [59] inspired one of its original authors to create an open-source schema-driven binary serialization specification and RPC protocol named *Cap'n Proto* [142]. Two schema-less serialization specifications were created on 2013: a textual serialization specification inspired by INI named *TOML* [109] and a binary serialization specification designed for the Internet of Things named *CBOR* [14]. In 2014, Google released *FlatBuffers* [139], a schema-driven binary serialization specification that was later found to share some similarities to Cap'n Proto [142]. In 2015, Microsoft open-sourced *Bond* [87]. In 2016, Google introduced *FlexBuffers* [140], a schema-less variant of FlatBuffers [139].

[7] `https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/PropertyLists/OldStylePlists/OldStylePLists.html`

[8] `https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html`

[9] `https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`

[10] `https://www.w3.org`

[11] `https://opensource.apple.com/source/CF/CF-1153.18/CFBinaryPList.c.auto.html`

[12] `https://web.archive.org/web/20140219093104/http://www.appleexaminer.com/MacsAndOS/Analysis/PLIST/PLIST.html`

[13] `https://github.com/frsyuki`

[14] `https://hadoop.apache.org`

[15] `https://www.mongodb.com`

Table 1: A non-exhaustive list of companies that publicly state that they are using the binary serialization specifications discussed in this paper.

Serialization Specification	Companies
ASN.1	Broadcom, Cisco, Ericsson, Hewlett-Packard, Huawei, IBM, LG Electronics, Mitsubishi, Motorola, NASA, Panasonic, Samsung, Siemens [16]
Apache Avro	Bol, Cloudera, Confluent, Florida Blue, Imply, LinkedIn, Nuxeo, Spotify, Optus, Twitter [17]
Microsoft Bond	Microsoft
Cap'n Proto	Sandstorm, Cloudflare [18]
FlatBuffers / FlexBuffers	Apple, Electronic Arts, Facebook, Google, Grafana, JetBrains, Netflix, Nintendo, NPM, NVidia, Tesla [19], [20]
Protocol Buffers	Alibaba, Amazon, Baidu, Bloomberg, Cisco, Confluent, Datadog, Dropbox, EACOMM, Facebook, Google, Huawei, Intel, Lyft, Microsoft, Mozilla, Netflix, NVidia, PayPal, Sony, Spotify, Twitch, Uber, Unity, Yahoo, Yandex [21], [22]
Apache Thrift	Facebook, Cloudera, Evernote, Mendeley, last.fm, OpenX, Pinterest, Quora, RapLeaf, reCaptha, Siemens, Uber [23]
BSON	MongoDB
CBOR	Intel [24], Microsoft [25], Outfox [26], Pelion [27]
MessagePack	Amazon, Atlassian, CODESYS, Datadog, Deliveroo, GitHub, Google, GoSquared, LinkedIn, Microsoft, Mozilla, NASA, National Public Radio, NPM, Pinterest, Sentry, Shopify, Treasure Data, Yelp [28], [29], [30], [31]
Smile	Ning, Elastic [32]
UBJSON	Teradata [33], Wolfram [34]

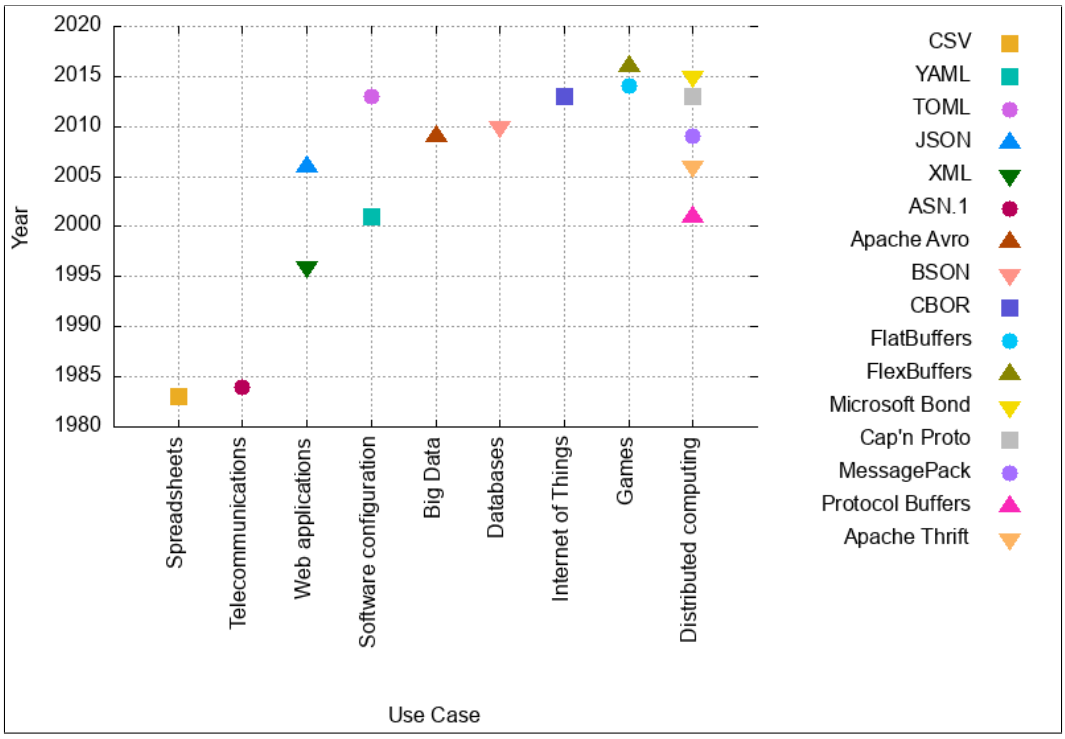

Figure 3: The most popular serialization specifications by their use case.

[16] https://www.oss.com/company/customers.html

[17] https://lists.apache.org/thread.html/rc11fcfbc294bb064c6e59167f21b38f3eb6d14e09b9af60970b237d6%40%3Cuser.avro.apache.org%3E

[18] https://www.linkedin.com/in/kenton-varda-5b96a2a4/

[19] https://github.com/google/flatbuffers/issues/6424

[20] https://github.com/google/flatbuffers/network/dependents

[21] https://groups.google.com/g/protobuf/c/tJVbWK3y_TA/m/vpOiSFfqAQAJ

[22] https://github.com/protocolbuffers/protobuf/network/dependents

[23] https://thrift.apache.org/about#powered-by-apache-thrift

[24] https://www.npmjs.com/package/tinycbor

[25] https://github.com/OneNoteDev/GoldFish

[26] https://github.com/outfoxx/PotentCodables

[27] https://github.com/PelionIoT/cbor-sync

[28] https://github.com/msgpack/msgpack-node/network/dependents

[29] https://github.com/msgpack/msgpack-ruby/network/dependents

[30] https://github.com/msgpack/msgpack-javascript/network/dependents

[31] https://github.com/msgpack/msgpack/issues/295

[32] https://github.com/FasterXML/smile-format-specification/issues/15

[33] https://docs.teradata.com/reader/C8cVEJ54PO4~YXWXeXGvsA/b9kd0QOTMB3uZp9z5QT2aw

[34] https://reference.wolfram.com/language/ref/format/UBJSON.html

1.3 Textual and Binary Serialization Specifications

A serialization specification is *textual* if the bit-strings it produces correspond to sequences of characters in a text encoding such as ASCII [27], EBCDIC/CCSID 037 [65], or UTF-8 [32], otherwise the serialization specification is *binary*.

We can think of a textual serialization specification as a set of conventions within the boundaries of a text encoding such as UTF-8 [32]. The availability and diversity of computer tools to operate on popular text encodings makes textual serialization specifications to be perceived as human-friendly. In comparison, binary serialization specifications are not constrained by a text encoding. This flexibility typically results in efficient representation of data at the expense of requiring accompanying documentation and specialized tools.

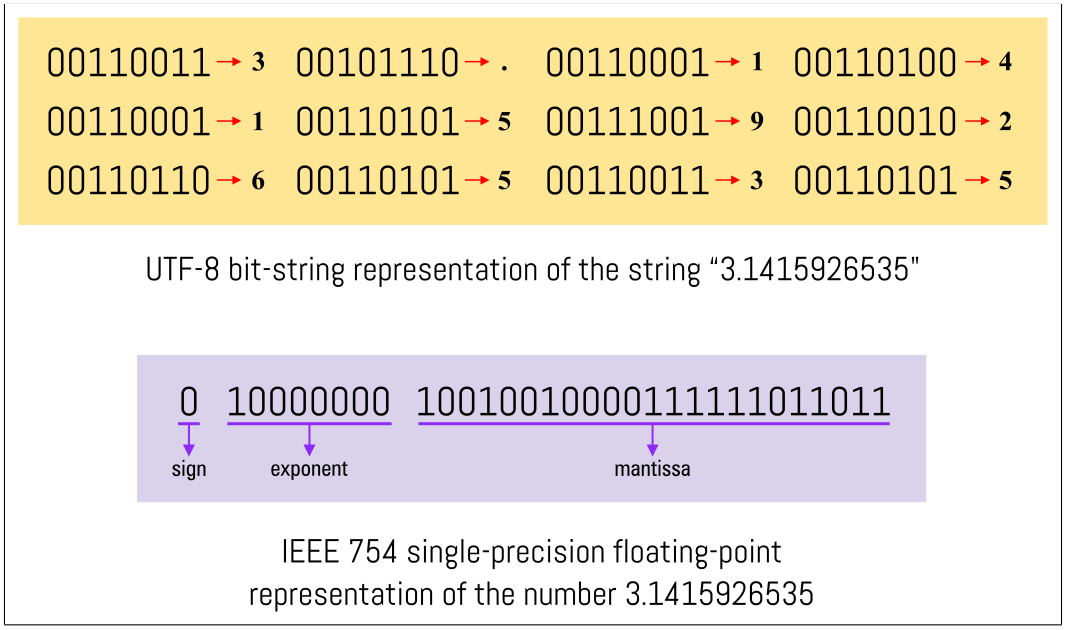

Figure 4: Textual and binary representations of the decimal number 3.1415926535. The textual representation encodes the decimal number as a 96-bits sequence of numeric characters ("3" followed by ".", followed by "1", and so forth) that we can easily inspect and understand using a text editor. On the other hand, the binary representation encodes the decimal number in terms of its sign, exponent, and mantissa. The resulting bit-string is only 32 bits long - three times smaller than the textual representation. However, we are unable to understand it using generally-available text-based tools.

1.4 Schema-less and Schema-driven Serialization Specifications

A *schema* is a formal definition of a data structure. For example, a schema may describe a data structure as consisting of two Big Endian IEEE 754 single-precision floating-point numbers [51]. A serialization specification is *schema-less* if it produces bit-strings that can be deserialized without prior knowledge of its structure and *schema-driven* otherwise.

Implementations of schema-less serialization specifications embed the information provided by a schema into the resulting bit-strings to produce bit-strings that are standalone with respect to deserialization. In comparison to schema-driven serialization specifications, schema-less serialization specifications are perceived as easier to use because receivers can deserialize any bit-string produced by the implementation and not only the ones the receivers know about in advance. Alternatively, schema-driven specification implementations can avoid encoding certain structural information into the bit-strings they produce. This typically results in compact space-efficient bit-strings. For this reason, network-efficient systems tend to adopt schema-driven serialization specifications [107]. Schema-driven serialization specifications are typically concerned with space efficiency and therefore tend to be binary. However, [25] propose a textual JSON-compatible schema-driven serialization specification. In the case of communication links with large bandwidths or small datasets, the gains

are negligible but considering slow communication links or large datasets which could be terabytes in size, the choice of serialization specification could have a big impact.

Writing and maintaining schema definitions is a core part of using schema-driven serialization specifications. Most schema-driven serialization specifications implement a custom schema language that is not usable by any other schema-driven serialization specification. A schema-driven serialization specification may use a low-level or a high-level schema definition language. Low-level schema definition languages such as *PADS* [49], *BinX* [147], and *Data Format Description Language* (DFDL) [147] are concerned with describing the contents of bit-strings while high-level schema definition languages such as *ASN.1* [118] and *JSON Schema* [152] abstractly describe data structures and depend on the serialization specification implementation to provide the corresponding encoding rules.

Often, schemas are auto-generated from formal definitions such as database tables [35], other schema languages [36], or inferred from the data [54] [74] [6] [36] [151] [43] [22] [128] [5]. There are also examples of domain-specific schema-driven serialization specifications where the schema definition is implicitly embedded into the serialization and deserialization implementation routines, such as the SOL binary representation for sensor measurements [20].

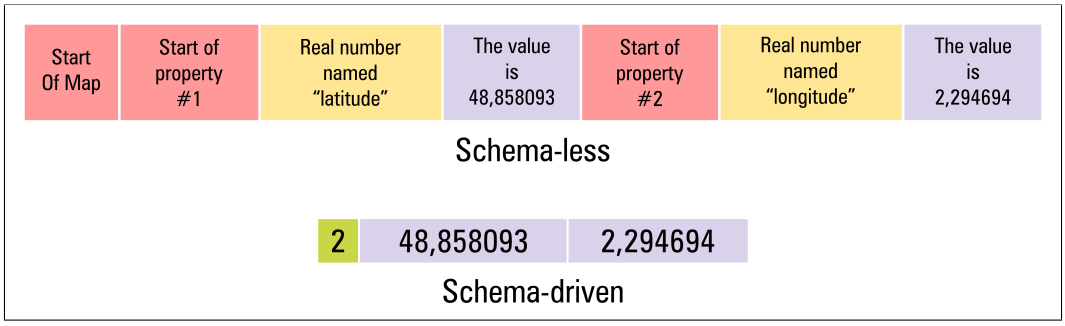

Figure 5: An associative array (also known as a map) that consists of two decimal number properties, "latitude" and "longitude", serialized with fictitious schema-less and schema-driven representations. The schema-less representation (top) is self-descriptive and each property is self-delimited. Alternatively, schema-driven representations (bottom) omit most self-descriptive information except for the length of the associative array as an integer prefix. A reader cannot understand how the schema-driven representation translates to the original data structure without additional information such as a schema definition.

1.5 Schema-less as a Subset of Schema-driven

Schema-driven serialization specifications avoid embedding structural information into the resulting bit-strings for space-efficiency purposes. If a schema fails to capture essential structural information then the serialization specification has to embed that information into the resulting bit-strings. We can reason about schema-less serialization specifications as schema-driven specifications where the schema is generic enough that it describes any bit-string and as a consequence carries no substantial information about any particular instance. For example, a schema that defines bit-strings as sequences of bits can describe any bit-string while providing no useful information for understanding the semantics of such bit-strings.

The amount of information included in a schema can be thought as being *inversely proportional* to the information that needs to be encoded in a bit-string described by such schema. However, schema-driven serialization specifications may still embed redundant information into the bit-strings with respect to the schema for runtime-efficiency, compatibility or error tolerance. We can rank schema-driven serialization specifications based on the extent of information that is necessary to include in the bit-strings.

[35] `https://github.com/SpringTree/pg-tables-to-jsonschema`
[36] `https://github.com/okdistribute/jsonschema-protobuf`

1.6 Related Work

Table 2 and Table 3 list existing literature that discusses different sets of serialization specifications, both textual and binary, schema-less and schema-driven. However, many of these publications discuss serialization specifications that are either not JSON-compatible, cannot be considered general-purpose serialization specifications, or are out of date. For example, Java Object Serialization, as its name implies, is only concerned with serialization of object instances in the Java programming language. The first Protocol Buffers [59] version 3 release was published on GitHub in 2014 [37], yet there are several publications listed in Table 2 and Table 3 released before that year discussing the now-obsolete Protocol Buffers version 2 [149] [58] [84] [127] [141] [46]. As another example, [88] discusses an ASN.1 [118] encoding (LWER) that has been abandoned in the 1990s [79].

Furthermore, many of the publications listed in Table 2 and Table 3 are concerned with benchmarking. They tend to describe the respective specifications in a high-level manner and do not get into the encoding details of non-trivial examples, if any.

1.7 Contributions

To the best of our knowledge, there exists gaps in the current literature resulting in a lack of discussion on a wide range of JSON-compatible binary serialization specifications in depth. We aim to fill that gap by providing a detailed comparative analysis of the most popular JSON-compatible binary serialization specifications. Through the process of conducting a literature review, we identified and resolved 13 issues with the documentation and specifications of the Apache Avro [52], Apache Thrift [126], FlatBuffers [139], FlexBuffers [140], Microsoft Bond [87], and Smile [116] open-source binary serialization specifications. Our fixes, the corresponding patches and links are listed in Table 4 and Table 5.

[37] https://github.com/protocolbuffers/protobuf/releases/tag/v3.0.0-alpha-1
[38] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[39] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[40] https://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/index.html
[41] https://github.com/RuedigerMoeller/fast-serialization
[42] http://hessian.caucho.com/doc/hessian-serialization.html
[43] https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html
[44] https://github.com/EsotericSoftware/kryo
[45] https://github.com/protostuff/protostuff
[46] https://www.rdfhdt.org
[47] https://github.com/niXman/yas

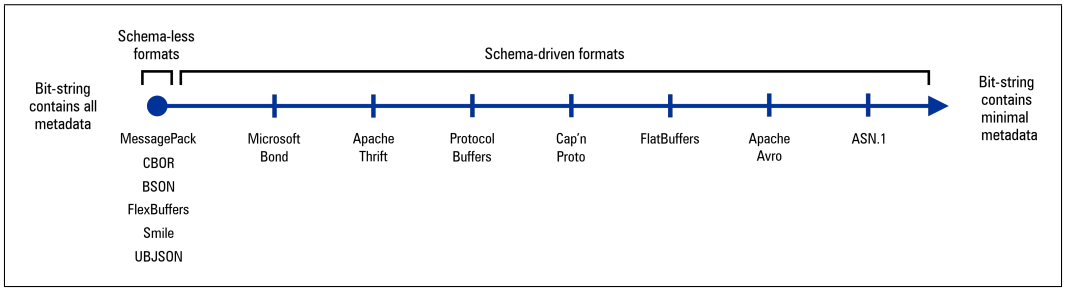

Figure 6: We can compare schema-less and schema-driven serialization specifications based on how much information their resulting bit-strings contain. Schema-less specifications are equivalent in that they all consist of implicit schemas that convey no information. However, not all schema languages supported by schema-driven specifications can describe the same amount of information. For this reason, some schema-driven specifications need to encode more structural information into their resulting bit-strings than others, placing them on the left hand side of the line. Schema-driven specifications that enable meticulously defined schemas are placed on the right hand side of the line.

1.8 Paper Organisation

In section 2, we introduce the JSON serialization specification and discuss its role and limitations. In section 3, we list the binary serialization specifications that are discussed in depth in this paper and our approach to analyzing them. In section 4, we explore a set of non-standardized binary integer encoding techniques that are essential for understanding the inner workings of the various binary serialization specifications. In section 5 and section 6, we systematically study the history, the characteristics, and the serialization processes of the selection of schema-driven and schema-less binary serialization specifications. In section 7, we introduce the challenges associated with schema evolution. Further, we discuss the schema evolution features that the selection of schema-driven serialization specifications provide. In section 8, we reflect over what we learned in the process of closely inspecting the selection of JSON-compatible binary serialization specifications. In section 9, we provide instructions for the reader to recreate the bit-strings analyzed in the survey. Finally, in section 10, we discuss a set of next-steps to continue and broaden the understanding of JSON-compatible binary serialization specifications.

Table 2: A list of publications that discuss binary serialization specifications. This table is continued in Table 3.

Publication	**Year**	**Context**	**Discussed Serialization Specifications**
An overview of ASN.1 [92]	1992	Networking	ASN.1 BER [120], ASN.1 PER [119]
Efficient encoding rules for ASN.1-based protocols [88]	1994	Networking	ASN.1 BER [120], ASN.1 CER [120], ASN.1 DER [120], ASN.1 LWER [79], ASN.1 PER [119]
Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication [149]	2011	Microblogging	JSON [41], Protocol Buffers [59]
Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones [58]	2011	Mobile	JSON [41], Protocol Buffers [59], XML [99]
Performance evaluation of object serialization libraries in XML, JSON and binary formats [84]	2012	Java Object Serialization	Apache Avro [52], Apache Thrift [126], Java Object Serialization [38], JSON [41], Protocol Buffers [59], XML [99]
A comparison of data serialization formats for optimal efficiency on a mobile platform [127]	2012	Mobile	Apache Thrift [126], JSON [41], Protocol Buffers [59], XML [99]
Performance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats [141]	2012	Web Services	Apache Avro [52], Java Object Serialization [39], JSON [41], MessagePack [56], Protocol Buffers [59], XML [99]
Google protocol buffers research and application in online game [46]	2013	Gaming	Protocol Buffers [59]
Integrating a System for Symbol Programming of Real Processes with a Cloud Service [76]	2015	Web Services	JSON [41], MessagePack [56], XML [99], YAML [9]

Table 3: Continuation of Table 2.

Publication	**Year**	**Context**	**Discussed Serialization Specifications**
Serialization and deserialization of complex data structures, and applications in high performance computing [155]	2016	Scientific Computing	Apache Avro [52], Boost::serialize [40], Cap'n Proto [142], OpenFPM Packer/Unpacker [66], Protocol Buffers [59]
Smart grid serialization comparison: Comparison of serialization for distributed control in the context of the internet of things [104]	2017	Internet of Things	Apache Avro [52], BSON [86], CBOR [14], FST [41], Hessian [42], Java Object Serialization [43], Kryo [44], MessagePack [56], Protocol Buffers [59], ProtoStuff [45], Smile [116], XML [99], YAML [9]
Binary Representation of Device Descriptions: CBOR versus RDF HDT [115]	2018	Internet of Things	CBOR [14], JSON [41], RFD HDT [46]
Evaluating serialization for a publish-subscribe based middleware for MP-SoCs [63]	2018	Embedded Development	FlatBuffers [139], MessagePack [56], Protocol Buffers [59], YAS [47]
Analytical Assessment of Binary Data Serialization Techniques in IoT Context [10]	2019	Internet of Things	BSON [86], FlatBuffers [139], MessagePack [56], Protocol Buffers [59]
FlatBuffers Implementation on MQTT Publish/Subscribe Communication as Data Delivery Format [108]	2019	Internet of Things	CSV [122], FlatBuffers [139], JSON [41], XML [99]
Enabling Model-Driven Software Development Tools for the Internet of Things [71]	2019	Internet of Things	Apache Avro [52], Apache Thrift [126], FlatBuffers [139], JSON [41], Protocol Buffers [59]
Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV [110]	2020	Automobile	Cap'n Proto [142], FlatBuffers [139], Protocol Buffers [59]
Performance Analysis and Optimization of Serialization Techniques for Deep Neural Networks [100]	2020	Machine Learning	FlatBuffers [139], Protocol Buffers [59]

Table 4: A list of open-source contributions made by the authors in the process of writing this survey paper. This table is continued in Table 5.

Specification	Repository	Commit	Description
Apache Avro [52]	`https://github.com/apache/avro`	`afe8fa1`	Improve the encoding specification to clarify that records encode fields even if they equal their explicitly-set defaults and that the `default` keyword is only used for schema evolution purposes
Apache Thrift [126]	`https://github.com/apache/thrift`	`2e7f39f`	Improve the Compact Protocol specification to clarify the Little Endian Base 128 (LEB128) variable-length integer encoding procedure and include a serialization example
Apache Thrift [126]	`https://github.com/apache/thrift`	`47b3d3b`	Improve the Compact Protocol specification to clarify that strings are not delimited with the *NUL* ASCII [27] character
FlatBuffers [139]	`https://github.com/google/flatbuffers`	`4aff119`	Extend the binary encoding specification to document how union types are encoded
FlatBuffers [139]	`https://github.com/google/flatbuffers`	`7b1ee31`	Improve the documentation to clarify that the schema language does not permit unions of scalar types but that the C++ [68] implementation has experimental support for unions of structs and strings
FlatBuffers [139]	`https://github.com/google/flatbuffers`	`52e2177`	Remove from the documentation an outdated claim that Protocol Buffers [59] does not support union types
FlatBuffers [139]	`https://github.com/google/flatbuffers`	`796ed68`	Improve the FlatBuffers [139] and FlexBuffers [140] encoding specification to clarify that neither specifications deduplicate vector elements but that vectors may include more than one offset pointing to the same value

Table 5: Continuation of Table 4.

Specification	Repository	Commit	Description
Microsoft Bond [87]	`https://github.com/microsoft/bond`	`4acf83b`	Improve the documentation to explain how to enable the Compact Binary version 2 encoding in the C++ [68] implementation
Microsoft Bond [87]	`https://github.com/microsoft/bond`	`0012d99`	Clarify the Compact Binary encoding specification to clarify that ID bits are encoded as Big Endian unsigned integers, that signed 8-bit integers use Two's Complement [50], formalize the concept of variable-length integers as Little Endian Base 128 (LEB128) encoding, clarify that real numbers are encoded as IEEE 764 floating-point numbers [51], and that enumeration constants are encoded as signed 32-bit integers
Smile [116]	`https://github.com/FasterXML/smile-format-specification`	`ac82c6b`	Fix the fact that the specification refers to ASCII [27] string of 33 to 64 bytes and Unicode [32] strings of 34 to 65 bytes using two different names
Smile [116]	`https://github.com/FasterXML/smile-format-specification`	`7a53b0a`	Improve the specification by adding an example of how real numbers are represented using the custom 7-bit variant of IEEE 764 floating-point number encoding [51]
Smile [116]	`https://github.com/FasterXML/smile-format-specification`	`95133dd`	Improve the specification to clarify how the byte-length prefixes from the *Tiny Unicode* and *Small Unicode* string encodings are computed differently compared to their ASCII [27] counterparts
Smile [116]	`https://github.com/FasterXML/smile-format-specification`	`c56793f`	Clarify that the encoding attempts to reserve the `0xff` byte for message framing purposes but that reserving such byte is no longer a requirement to make the format suitable for use with WebSockets

2 JSON

2.1 History and Characteristics

JSON (JavaScript Object Notation) is a standard *schema-less* and *textual* serialization specification *inspired* by a subset [48] of the JavaScript programming language [42]. Douglas Crockford [49], currently a Distinguished Architect at PayPal, described the JSON serialization specification online [50] in 2002 and published the first draft of the JSON serialization specification in 2006 [34]. Douglas Crockford claims he *discovered* and *named* JSON, whilst Netscape was already using an unspecified variant as an interchange format as early as in 1996 in their web browser [35].

```
{
  "Image": {
    "Width": 800,
    "Height": 600,
    "Title": "View from 15th Floor",
    "Thumbnail": {
      "Url": "http://www.example.com/image/481989943",
      "Height": 125,
      "Width": 100
    },
    "Animated": false,
    "IDs": [ 116, 943, 234, 38793 ]
  }
}
```

Figure 7: A JSON document example taken from the official specification [18].

JSON is a human-readable open standard specification that consists of structures built on key-value pairs or list of ordered items. The data types it supports are objects, arrays, numbers, strings, `null`, and boolean constants `true` and `false`. A data structure encoded using JSON is referred to as a *JSON document*. [41] states that JSON documents are serialized as either UTF-8, UTF-16, or UTF-32 strings [32]. However, [18] mandate the use of UTF-8 for interoperability purposes. The serialization process involves recursively converting keys and values to strings and optionally removing meaningless new line, tab, and white space characters (a process known as *minification*) as shown in Figure 8.

```
{"Image":{"Width":800,"Height":600,"Title":"View from 15th Floor", "Thumbnail":{
"Url":"http://www.example.com/image/481989943","Height":125,"Width":100},
"Animated":false,"IDs":[116,943,234,38793]}}
```

Figure 8: A *minified* and semantically-equivalent version of the JSON document from Figure 7.

2.2 Relevance of JSON

JSON [41] is popular interchange specification in the context of cloud computing. In 2019, [143] found that JSON documents constitute a growing majority of the HTTP [47] responses served by Akamai, a leading Content Delivery Network (CDN) that serves around 3 trillion HTTP requests daily. Gartner [51], a business insight research and advisory firm, forecasts that the cloud services market will grow 17% in 2020 to total $266.4 billion and that SaaS will remain the largest market segment [52]. SaaS systems typically provide *application programming interfaces* (APIs) and JSON

[48] `http://timelessrepo.com/json-isnt-a-javascript-subset`

[49] `https://www.crockford.com/`

[50] `https://www.json.org`

[51] `https://www.gartner.com`

[52] `https://www.gartner.com/en/newsroom/press-releases/2019-11-13-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-17-percent-in-2020`

was found to be the most common request and response format for APIs [53]. According to their study, JSON was used more than XML [99]. JSON popularity over XML can be attributed to the fact that in comparison to XML, JSON results in smaller bit-strings and in runtime and memory efficient serialization and deserialization implementations [94].

There is an on-going interest in JSON within the research community. [15] describe the first formal framework for JSON documents and introduce a query language for JSON documents named *JSON Navigational Logic* (JNL). There is a growing number of publications that discuss JSON in the context of APIs [151] [43] [45] and technologies that emerged from the JSON ecosystem such as the JSON Schema specification [106] [85] [61] [55]. Apart from cloud computing, JSON is relevant in areas such as databases [31] [74] [8] [54] [105], big data [154], analytics [98] [82], mobile applications [127] [58], 3D modelling [81], IoT [146] [93], biomedical research [70], and configuration files, for example [54]. [7] presents a high-level overview of the JSON ecosystem including a survey of popular schema languages and implementations, schema extraction technologies and novel parsing tools.

2.3 Shortcomings

Despite its popularity, JSON is neither a runtime-efficient nor a space-efficient serialization specification.

Runtime-efficiency. Serialization and deserialization often become a bottleneck in data-intensive, battery-powered, and resource-constrained systems. [98] state that big data applications may spend up to 90% of their execution time deserializing JSON documents, given that deserialization of textual specifications such as JSON is typically expensive using traditional state-machine-based parsing algorithms. [58] and [127] highlight the impact of serialization and deserialization speed on mobile battery consumption and resource-constrained mobile platforms. As a solution, [12] propose a promising JSON encoder and decoder that infers JSON usage patterns at runtime and self-optimizes by generating encoding and decoding machine code on the fly. Additionally, [78] propose a novel approach to efficiently parse JSON document by relying on SIMD processor instructions. [82] claim that applications parse entire JSON documents but typically only make use of certain fields. As a suggestion for optimization, they propose a lazy JSON parser that infers schemas for JSON documents at runtime and uses those schemas to speculate on the position of the fields that an application requires in order to avoid deserializing unnecessary areas of the JSON documents.

Space-efficiency. Cloud services are typically accessed over the internet. As a result, these types of software systems are particularly sensitive to substandard network performance. For example, in 2007, Akamai, a global content delivery network (CDN), found out that "*a 100-millisecond delay in website load time can hurt conversion rates by 7 percent*" and that "*a two-second delay in web page load time increases bounce rates by 103 percent*" [55]. Cloud services frequently run on top of *infrastructure as a service* (IaaS) or *platform as a service* (PaaS) providers such as Amazon Web Services (AWS) [56]. These providers typically operate on a pay-as-you-go model where they charge per resource utilization. Therefore, transmitting data over the network directly translates to operational expenses. In comparison to JSON, [107] found that using a custom binary serialization specification reduced the overall network traffic by up to 94%. [11] conclude that JSON is not an appropriate specification for bandwidth-constrained communication systems citing the size of the documents as the main reason. [111] states that network traffic is one of the two biggest causes for battery consumption on mobile devices and therefore a space-efficient serialization specification could have a positive impact on energy savings.

There are several JSON-based specifications that highlight a need for compact JSON encodings:

- JSON Patch [21] is a specification for expressing a sequence of operations on JSON documents. [23] describe an algorithm called JDR to compute JSON Patch differences between two JSON documents optimized to keep the number of JSON Patch operations to a minimum for space-efficiency reasons.

[53] `https://www.programmableweb.com/news/json-clearly-king-api-data-formats-2020/research/2020/04/03`

[54] `https://www.typescriptlang.org/docs/handbook/tsconfig-json.html`

[55] `https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp`

[56] `https://aws.amazon.com`

- CityGML is an XML-based specification to represent 3D models of cities and landscapes. [81] introduce a JSON-based alternative to CityGML called CityJSON citing that CityGML documents are large enough that makes it difficult or even impossible to transmit and process them on the web. In comparison to CityGML, CityJSON results in smaller document size. However, the authors are looking for a binary JSON encoding to compress the CityJSON documents even further. Additionally, [138] explores how CityJSON documents can be compressed further using binary serialization specifications such as BSON [86] and CBOR [14].

- In the context of bioinformatics, *mmJSON* is a popular serialization specification used to encode representations of macromolecular structures. [16] introduce *MMTF*, a binary serialization specification to encode macromolecular structures based on MessagePack [56] to address the space-efficiency and runtime-efficiency concerns of using *mmJSON* to perform web-based structure visualization. In particular, using *mmJSON*, even after applying GZIP [39] compression, results in large macromolecular structures that are slow to download. Due to their size, the largest macromolecular structures results in deserialization memory requirements that exceed the amount of memory typically available in web browsers.

3 Methodology

Our approach to extend the body of literature through meticulous study of JSON-compatible binary serialization specifications is based on the following methodology. While several serialization specifications have characteristics outside of the context of JSON [18], the scope of this study is limited to those characteristics that are relevant for encoding JSON documents.

1. **Identify JSON-compatible Binary Serialization Specifications.** Research and select a set of schema-driven and schema-less JSON-compatible binary serialization specifications.

2. **Craft a JSON Test Document.** Design a sufficiently complex yet succinct JSON document in an attempt to highlight the challenges of encoding JSON data. This JSON document is referred to as the *input data*.

3. **Write Schemas for the Schema-driven Serialization Specifications.** Present schemas that describe the *input data* for each of the selected schema-driven serialization specifications. The schemas are designed to produce space-efficient results given the features documented by the corresponding specifications.

4. **Serialize the JSON Test Document.** Serialize the *input data* using each of the selected binary serialization specifications.

5. **Analyze the Bit-strings Produced by Each Serialization Specification.** Study the resulting bit-strings and present annotated hexadecimal diagrams that guide the reader in understanding the inner workings of each binary serialization specification.

6. **Discuss the Characteristics of Each Serialization Specification.** For each selected binary serialization specification, discuss the characteristics, advantages and optimizations that are relevant in the context of serializing JSON [41] documents.

3.1 Serialization Specifications

We selected a set of general-purpose schema-driven and schema-less serialization specifications that are popular within the open-source community. Some of the selected schema-driven serialization specifications support more than one type of encoding. In these cases, we chose the most space-efficient encoding. The implementations used in this study are freely available under open-source licenses with the exception of ASN.1 [118], for which a proprietary implementation is used. The choice of JSON-compatible serialization specifications, the selected encodings and the respective implementations are documented in Table 6 and Table 7.

As part of this study, we chose not to discuss serialization specifications that could not encode the *input data* JSON document from Figure 10 without significant changes, such as *Simple Binary Encoding* (SBE) [63] and Apple's *Binary Property List* (BPList) [64] or that could not be considered general-purpose serialization specifications, such as *Java Object Serialization* [65] and *YAS* [66]. We also chose not to discuss serialization specifications that remain unused in the industry at present, such as *PalCom Object Notation* (PON) [93] and as a consequence lack a well-documented and usable implementation, such as *SJSON* [3] and the *JSON-B*, *JSON-C* and *JSON-D* family of schema-less serialization specifications [62].

3.2 Input Data

We designed a test JSON [41] document that is used to showcase the challenges of serializing JSON [41] data and attempt to highlight the interesting aspects of each selected serialization specification. The JSON document we created, presented in Figure 10, has the following characteristics:

- It contains an empty array, an empty object, and an empty string.
- It contains nested objects and homogeneous and heterogeneous arrays.
- It contains an array of scalars with and without duplicate values.
- It contains an array of composite values with duplicate values.

[57] `https://avro.apache.org/docs/current/spec.html#binary_encoding`
[58] `https://microsoft.github.io/bond/reference/cpp/compact__binary_8h_source.html`
[59] `https://capnproto.org/encoding.html#packing`
[60] `https://google.github.io/flatbuffers/flatbuffers_internals.html`
[61] `https://developers.google.com/protocol-buffers/docs/encoding`
[62] `https://github.com/apache/thrift/blob/master/doc/specs/thrift-compact-protocol.md`
[63] `https://github.com/real-logic/simple-binary-encoding`
[64] `https://opensource.apple.com/source/CF/CF-550/CFBinaryPList.c`
[65] `https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html`
[66] `https://github.com/niXman/yas`

Table 6: The schema-driven binary serialization specifications, encodings and implementations discussed in this study.

Specification	Implementation	Encoding
ASN.1	OSS ASN-1Step Version 10.0.1 (proprietary)	PER Unaligned [119]
Apache Avro	Python `avro` (pip) 1.10.0	Binary Encoding [57] with no framing
Microsoft Bond	C++ library 9.0.3	Compact Binary v1 [58]
Cap'n Proto	`capnp` command-line tool 0.8.0	Packed Encoding [59]
FlatBuffers	`flatc` command-line tool 1.12.0	Binary Wire Format [60]
Protocol Buffers	Python `protobuf` (pip) 3.13.0	Binary Wire Format [61]
Apache Thrift	Python `thrift` (pip) 0.13.0	Compact Protocol [62]

Table 7: The schema-less binary serialization specifications, encodings and implementations discussed in this study.

Specification	Implementation
BSON	Python `bson` (pip) 0.5.10
CBOR	Python `cbor2` (pip) 5.1.2
FlexBuffers	`flatc` command-line tool 1.12.0
MessagePack	`json2msgpack` command-line tool 0.6 with `MPack` 0.9dev
Smile	Python `pysmile` (pip) 0.2
UBJSON	Python `py-ubjson` (pip) 0.16.1

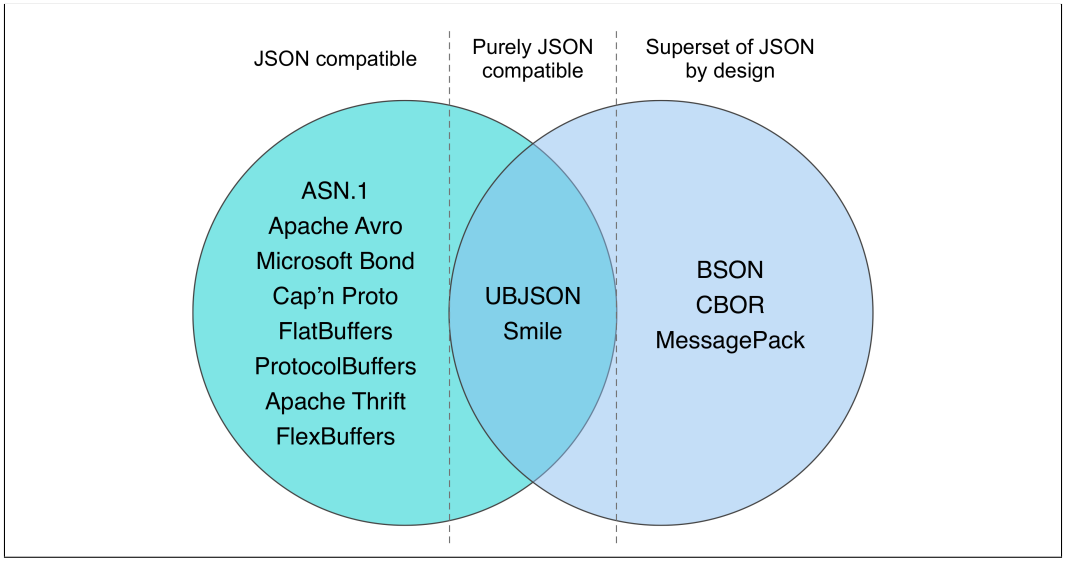

Figure 9: The binary serialization specifications discussed in this study divided by whether they are purely JSON compatible (center), whether they consider JSON compatibility but are a superset of JSON (right), or whether we found them to be JSON compatible but that's not one of their design goals (left).

- It contains a set and an unset nullable string.
- It contains positive and negative integers and floating-point numbers.
- It contains true and false boolean values.

The *input data* is not representative of every JSON document that the reader may encounter in practice. We hope that the characteristics of the input data and output demonstrate how serialization specifications perform with respect to various JSON documents.

3.3 System Specification

The implementations of the serialization specifications were executed on a MacBook Pro 13" Quad-Core Intel Core i7 2.7 GHz with 4 cores and 16 GB of memory (model identifier `MacBookPro15,2`) running macOS Catalina 10.15.7, Xcode 11.3 (11C29), clang 1100.0.33.16, and Python 3.7.3.

Table 8: Each serialization specification report will include a summary reference table following this layout.

Website	The website address of the serialization specification project
Company / Individual	The company or individual behind the serialization specification
Year	The year the serialization specification was conceived
Specification	A link or pointer to official specification of the serialization specification
License	The software license used to release the specification and implementations
Schema Language	The schema language used by the serialization specification, if schema-driven
Layout	The overall structure of the bit-strings created by the serialization specification
Languages	The programming languages featuring official and/or third-party implementations of the serialization specification
Types	A brief description of the data types supported by the serialization specification

```
{
  "tags": [],
  "tz": -25200,
  "days": [ 1, 1, 2, 1 ],
  "coord": [ -90.0715, 29.9510 ],
  "data": [
    { "name": "ox03", "staff": true },
    {
      "name": null,
      "staff": false,
      "extra": { "info": "" }
    },
    { "name": "ox03", "staff": true },
    {}
  ]
}
```

Figure 10: The JSON test document that will be used as a basis for analyzing various binary serialization specifications.

4 Encoding Processes

4.1 Little Endian Base 128 Encoding

Little Endian Base 128 (LEB128) variable-length integer encoding stores arbitrary large integers into a small variable amount of bytes. Decoders do not need to know how large the integer is in advance. The LEB128 encoding was introduced by the DWARF [125] debug data format and the concept of variable-length integers in the context of serialization was popularized by the Protocol Buffers [59] schema-driven serialization specification.

The encoding process consists of:

- Obtaining the Big Endian binary representation of the input integer.
- Padding the bit-string to make it a multiple of 7-bits.
- Splitting the bit-string into 7-bits groups, prefixing the most-significant 7-bit group with a zero-bit.
- Prefixing the remaining groups with set bits.
- Storing the result as Little Endian.

The decoding process is the inverse of the encoding process. The decoder knows when to stop parsing the variable-length integer bytes through the most-significant bit in each group: the most-significant bit in the last byte is equal to 0.

4.1.1 Unsigned Integer

In this example, we consider the unsigned integer 50399 in Figure 11.

The Big Endian unsigned integer representation of the number 50399 is 16-bits wide. 16 is not a multiple of 7, therefore we left-pad the bit-string with five zeroes to make it 21-bit wide. Then, we split the bit-string into 7-bit groups. Next, we prefix the most-significant group with `0` and the rest of the groups with `1`. The Little Endian representation of the bit-string equals `0xdf 0x89 0x03` which is the Little Endian Base 128 (LEB128) encoding of the unsigned integer 50399 as shown in Figure 11.

```
50399 = 1100 0100 1101 1111          Unsigned Big Endian integer representation
      = 00000 1100 0100 1101 1111    Left-padding (5-bits)
      = 0000011 0001001 1011111      7-bit groups
      = 00000011 10001001 11011111   Most-significant-bit prefixes
      = 11011111 10001001 00000011   Little Endian representation
      = 0xdf 0x89 0x03               Hexadecimal representation
```

Figure 11: Little Endian Base 128 (LEB128) encoding of the unsigned 32-bit integer 50399.

4.1.2 Signed Integer Using Two's Complement

The canonical approach to encoding signed integers using Little Endian Base 128 (LEB128) encoding is to apply Two's Complement [50] and proceed as if the integer is unsigned.

For example, consider the signed integer -25200 as shown in Figure 12.

```
-25200 = 1001 1101 1001 0000          Two's Complement 16-bit representation
       = 00000 1001 1101 1001 0000    Left-padding (5-bits)
       = 0000010 0111011 0010000      7-bit groups
       = 00000010 10111011 10010000   Most-significant-bit prefixes
       = 10010000 10111011 00000010   Little Endian representation
       = 0x90 0xbb 0x02               Hexadecimal representation
```

Figure 12: Little Endian Base 128 (LEB128) encoding of the signed 32-bit integer -25200.

4.2 ZigZag Integer Encoding

ZigZag integer encoding, pioneered by Protocol Buffers [59], is an approach to map signed integers to unsigned integers. The goal of ZigZag integer encoding is that the absolute values of the resulting unsigned integers are relatively close to the absolute values of the original negative integers. In comparison, Two's Complement [50] converts negative integers into large unsigned integers. The encoding works in a *zig-zag* fashion between positive and negative integers. ZigZag integer encoding does not affect the range of signed integer values that can be encoded in a given number of bits. *Little Endian Base 128* (LEB128) (4.1) variable-length integer encoding encodes unsigned integers using a number of bytes that are proportial to the absolute value of the unsigned integer. Therefore, ZigZag integer encoding results in space-efficiency when encoding negative integers in combination with Little Endian Base 128 (LEB128) encoding.

ZigZag Encoding:

- Encode the signed integer using Two's Complement [50].
- Left-shift the bit-string by one position.
- Right-shift the bit-string by the number of desired bits minus one.
- Calculate the exclusive disjunction (XOR) between both bit-shifted bit-strings.

ZigZag Decoding:

- Right-shift the unsigned integer bit-string by one position.
- Calculate the bitwise conjunction (AND) between the bit-string and 1.
- Calculate the Two's Complement [50] of the result of negating the integer represented by the bit-string.

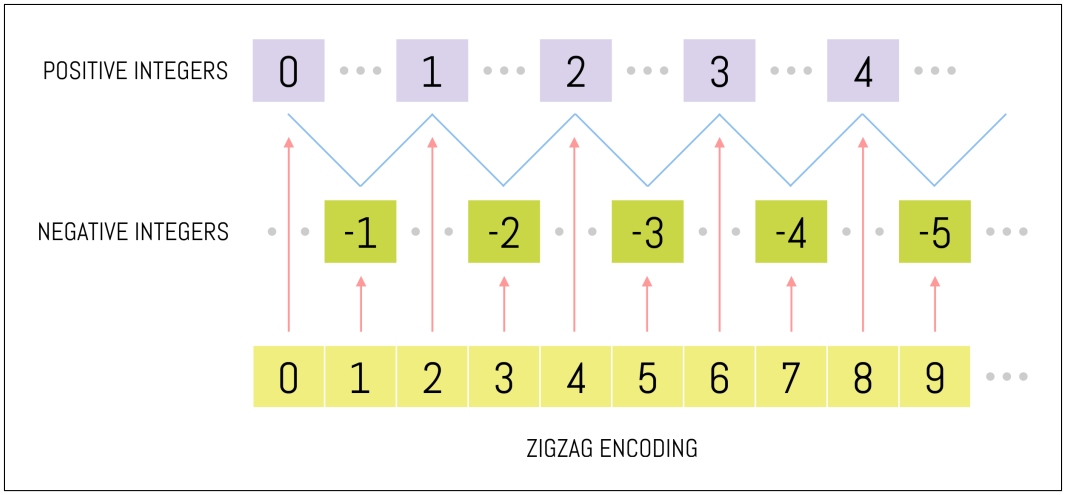

Figure 13: ZigZag integer encoding "zig-zags" between positive and negative integers.

- Calculate the exclusive disjunction (XOR) between both numbers.

Alternative definition of ZigZag integer encoding is that positive integers are multiplied by two. Negative integers are equal to their absolute values multiplied by two, minus one. ZigZag integer decoding is the inverse of the encoding process.

$$
\begin{array}{|l}
\hline
[Bits] \\
ZigZagEncode : \mathbb{Z} \to \mathbb{N} \\
ZigZagDecode : \mathbb{N} \to \mathbb{Z} \\
\hline
\forall\, i \in \mathbb{Z} \bullet ZigZagEncode(i) = (i << 1) \oplus (i >> (Bits - 1)) \\
\forall\, i \in \mathbb{N} \bullet ZigZagDecode(i) = (i >> 1) \oplus -(i \wedge 1) \\
ZigZagDecode \circ ZigZagEncode = id_{\mathbb{Z}} \\
\hline
\end{array}
$$

Figure 14: A formal definition of ZigZag integer encoding using Z notation [67]. We capture the relationship between the encoding and decoding functions by stating that the functional composition of `ZigZagDecode` and `ZigZagEncode` equals the identity function. Decoding an encoded integer equals the original integer.

4.2.1 ZigZag Encoding

The formal definition from Figure 14 states that the ZigZag integer encoding of the 32-bit signed negative integer -25200 is equal to $(-25200 << 1) \oplus (-25200 >> 31)$.

First, we calculate the Two's Complement [50] of the 32-bit signed integer -25200 which involves calculating the unsigned integer representation of its absolute value, calculating its binary complement, and adding 1 to the result as shown in Figure 15.

Then, we left-shift `0xffff9d90` by one position and right-shift `0xffff9d90` by 31 positions (32-bits minus 1) as shown in Figure 16.

Finally, we calculate the exclusive disjunction (XOR) between the two hexadecimal strings obtained before as shown in Figure 17.

Therefore, the ZigZag integer encoding of the signed 32-bit integer -25200 is `0x0000c4df`, which is the 32-bit unsigned integer 50399.

```
-25200 = ~|-25200| + 1
       = ~25200 + 1
       = ~ 0000 0000 0000 0000 0110 0010 0111 0000 + 1
       = 1111 1111 1111 1111 1001 1101 1000 1111 + 1
       = 1111 1111 1111 1111 1001 1101 1001 0000
       = 0xffff9d90
```

Figure 15: The Two's Complement [50] of the 32-bit signed integer -25200 is `0xffff9d90`.

```
0xffff9d90 << 1 = 1111 1111 1111 1111 1001 1101 1001 0000 << 1
                = 1111 1111 1111 1111 0011 1011 0010 0000
                = 0xffff3b20

0xffff9d90 >> 31 = 1111 1111 1111 1111 1001 1101 1001 0000 >> 31
                 = 1111 1111 1111 1111 1111 1111 1111 1111
                 = 0xffffffff
```

Figure 16: Bit-shifting the Two's Complement [50] of the 32-bit signed integer -25200 as per the formal definition of ZigZag integer encoding from Figure 14.

```
  1111 1111 1111 1111 0011 1011 0010 0000   = 0xffff3b20
⊕ 1111 1111 1111 1111 1111 1111 1111 1111   = 0xffffffff
  ---------------------------------------------------------
  0000 0000 0000 0000 1100 0100 1101 1111   = 0x0000c4df
```

Figure 17: The exclusive disjunction (XOR) between `0xffff3b20` and `0xffffffff` results in `0x0000c4df`.

4.2.2 ZigZag Decoding

The formal definition from Figure 14 states that the ZigZag integer decoding of the bit-string `0x0000c4df` we obtained in subsubsection 4.2.1 is equal to $(\texttt{0x0000c4df} >> 1) \oplus -(\texttt{0x0000c4df} \wedge 1)$.

First, we right-shift `0x0000c4df` by one position as shown in Figure 18.

```
0x0000c4df >> 1 = 0000 0000 0000 0000 1100 0100 1101 1111 >> 1
                = 0000 0000 0000 0000 0110 0010 0110 1111
                = 0x0000626f
```

Figure 18: Righ-shifting the ZigZag integer encoding of the 32-bit signed integer -25200 by one position results in `0x0000626f`.

Then, we calculate the bitwise conjunction (AND) between `0x0000c4df` and `0x00000001` as shown in Figure 19.

```
  0000 0000 0000 0000 1100 0100 1101 1111   = 0x0000c4df
∧ 0000 0000 0000 0000 0000 0000 0000 0001   = 0x00000001
  ─────────────────────────────────────────
  0000 0000 0000 0000 0000 0000 0000 0001   = 0x00000001
```

Figure 19: The bitwise conjunction (AND) between `0x0000c4df` and `0x00000001` results in `0x00000001`.

Next, we calculate the Two's Complement [50] of `-0x00000001` as shown in Figure 20.

```
-0x00000001 = ~|-0x00000001| + 1
            = ~0x00000001 + 1
            = ~ 0000 0000 0000 0000 0000 0000 0000 0001 + 1
            = 1111 1111 1111 1111 1111 1111 1111 1110 + 1
            = 1111 1111 1111 1111 1111 1111 1111 1111
            = 0xffffffff
```

Figure 20: The Two's Complement [50] of `-0x00000001` is `0xffffffff`.

Finally, we calculate the exclusive disjunction (XOR) between `0x0000626f` and `0xffffffff` as shown in Figure 21.

```
  0000 0000 0000 0000 0110 0010 0110 1111   = 0x0000626f
⊕ 1111 1111 1111 1111 1111 1111 1111 1111   = 0xffffffff
  ─────────────────────────────────────────
  1111 1111 1111 1111 1001 1101 1001 0000   = 0xffff9d90
```

Figure 21: The exclusive disjunction (XOR) between `0x0000626f` and `0xffffffff` results in `0xffff9d90`.

The result is `0xffff9d90` which is the Two's Complement [50] of the negative 32-bit signed integer -25200.

5 Schema-driven Specifications

5.1 ASN.1

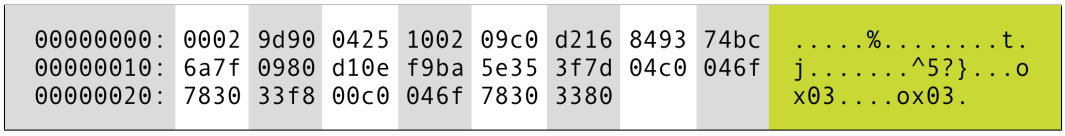

Figure 22: Hexadecimal output (`xxd`) of encoding Figure 10 input data with ASN.1 PER Unaligned (44 bytes).

History. ASN.1 [118] is a standard schema language used to serialize data structures using an extensible set of schema-driven encoding rules. ASN.1 was originally developed in 1984 as a part of the [117] standard and became a International Telecommunication Union recommendation and an ISO/IEC international standard [132] in 1988. The ASN.1 specification is publicly available [67] and there are proprietary and open source implementations of its standard encoding rules. The ASN.1 PER Unaligned encoding rules were designed to produce space-efficient bit-strings while keeping the serialization and deserialization procedures reasonably simple.

Characteristics.

- **Robustness.** ASN.1 is a mature technology that powers some of the highest-integrity communication systems in the world [68] [69] such as Space Link Extension Services (SLE) [26] communication services for spaceflight and the LTE S1 signalling service application protocol [1]. Refer to [135] for an example of formal verification of the encoding/decoding code produced by an ASN.1 PER Unaligned compiler (Galois [70]) in the automobile industry.
- **Standardization.** In comparison to informally documented serialization specification, ASN.1 is specified as a family of ITU Telecommunication Standardization Sector (ITU-T) recommendations and ISO/IEC international standards and has gone through extensive technical review.
- **Flexible Encoding Rules.** ASN.1 supports a wide range of standardised encodings for different use cases: BER (Basic Encoding Rules) based on tag-length-value (TLV) nested structures [120], DER (Distinguished Encoding Rules) and CER (Canonical Encoding Rules) [120] for restricted forms of BER [120], PER (Packed Encoding Rules) for space-efficiency [119], OER (Octet Encoding Rules) for runtime-efficiency [131], JER (JSON Encoding Rules) for JSON encoding [134], and XER (XML Encoding Rules) for XML encoding [133].

Layout. An ASN.1 PER Unaligned bit-string is a sequence of untagged values, sometimes nested where the ordering of the values is determined by the schema. ASN.1 PER Unaligned encodes values in as few bits as reasonably possible and does not align values to a multiple of 8 bits as the name of the encoding implies. ASN.1 PER Unaligned only encodes runtime information that cannot be inferred from the schema, such as the length of the lists or union type.

ASN.1 PER Unaligned encodes unbounded data types and bounded data types whose logical upper bound is greater than 65536 using a technique called *fragmentation* where the encoding of the value consists of one or more consecutive fragments each consisting of a length prefix followed by a series of items. The nature of each item depends on the type being encoded. For example, an item might be a character, a bit, or a logical element of a list. A value encoded using fragmentation consists of 0 or more fragments of either 16384, 32768, 49152, or 65536 items followed by a single fragment of 0 to 16383 items where each fragment is as large as possible and no fragment is larger than the preceding fragment. Refer to Table 9 for details on fragment length prefix encoding.

Numbers. ASN.1 PER Unaligned supports integer data types of arbitrary widths. The schema-writer may constraint the integer data type with a lower and upper bound:

[67] `https://www.itu.int/rec/T-REC-X.680/en`
[68] `https://www.itu.int/en/ITU-T/asn1/Pages/Application-fields-of-ASN-1.aspx`
[69] `https://www.oss.com/asn1/resources/standards-use-asn1.html`
[70] `https://galois.com`

- **If the integer type has no bounds or if the integer type only has an upper bound.** ASN.1 PER Unaligned encodes the value as a Big Endian Two's Complement [50] signed integer prefixed by its byte-length as a unsigned 8-bit integer.
- **If the integer type has a lower bound but not an upper bound.** ASN.1 PER Unaligned substracts the lower bound from the value and encodes the result as a variable-length Big Endian unsigned integer prefixed by its byte-length as a unsigned 8-bit integer. For example, the value -18 of an integer type whose lower bound is -20 is encoded as the unsigned integer $2 = -18 - (-20)$ prefixed with the byte-length definition `0x01`.
- **If the integer type has both a lower and an upper bound.** ASN.1 encodes the difference between the value and the lower bound using the smallest possible fixed-length Big Endian unsigned integer that can encode the difference between the upper and the lower bound. For example, an integer type constrained between the values 5 and 6 encodes the values as an unsigned 2-bit integer where 0 corresponds to 5 and 1 corresponds to 6.

In terms of real numbers, ASN.1 PER Unaligned does not support IEEE 764 floating-point numbers [51]. Instead, ASN.1 PER Unaligned encodes a real numbers as concatenation of its sign, base, scale, exponent, and mantissa where the real value equals $\textbf{sign} \times \textbf{mantissa} \times \textbf{2}^{\textbf{scale}} \times \textbf{base}^{\textbf{exponent}}$. Refer to Figure 23 for details on the encoding.

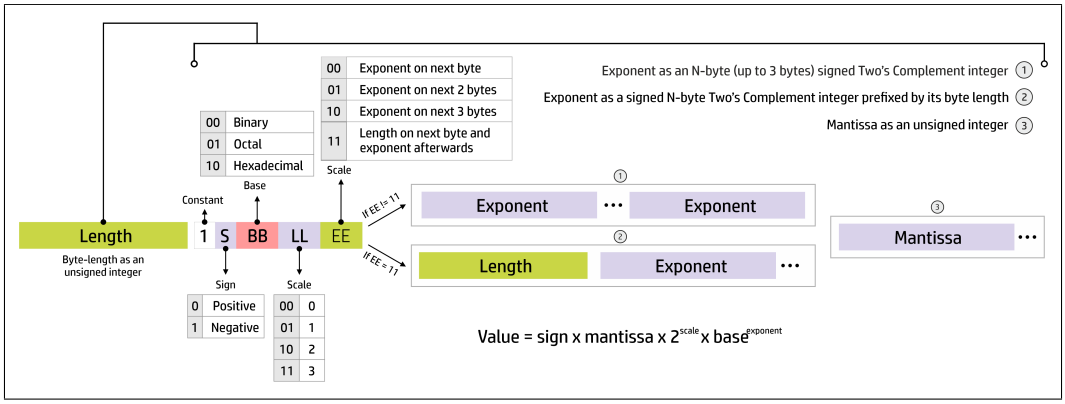

Figure 23: A visual representation of the `REAL` data type encoding inspired by [40], page 401. ASN.1 PER Unaligned encodes a real number as an 8-bit unsigned integer representing the byte-length of the real number, the sign (positive or negative) as 1-bit, the base as 2-bits (binary, octal, or hexadecimal), the scale as 2-bits (0, 1, 2, or 3), the exponent as a variable-length signed integer prefixed by its byte-length, and the mantissa as a Big Endian variable-length unsigned integer whose width is bounded by the data type length prefix.

Strings. ASN.1 supports a rich set of string types that are not *NUL*-delimited. The `IA5String` represents the full ASCII [27] range. The `VisibleString` subtype represents the subset of ASCII [27] that does not include control characters. The `NumericString` subtype represents digits and spaces. Finally, the `UTF8String` represents Unicode strings that are encoded in UTF-8 [32]. Schema-authors may subtype the supported string types to contrain the permitted alphabet. In the case of

Table 9: ASN.1 PER Unaligned fragment length prefixes depending on the number of items in the fragment as determined by the *From* and *To* ranges.

From	To	Fragment prefix
0	127	Length as an 8-bit unsigned integer
128	16383	Length as the 2-bits `10` followed by a Big Endian 14-bit unsigned integer
16384	16384	`1100 0001`
32768	32768	`1100 0010`
49152	49152	`1100 0011`
65536	65536	`1100 0100`

unconstrained string types and constrained string subtypes whose permitted alphabet contains more than 64 characters, each character is represented by its standard codepoint. Otherwise, ASN.1 PER Unaligned creates an ordered list of permitted characters (its alphabet) and encodes each character as an index of such list represented using the smallest possible Big Endian unsigned integer that can represent the set of permitted characters. ASN.1 PER Unaligned encodes strings using *fragmentation* when using string types in which the byte-length of the string is not always a multiple of the logical length of the string like `UTF8String`, when the string type has no size upper bound, or when the string type has a size upper bound which is greater than 65536. Otherwise, the string is prefixed with the string logical length as a bounded integer encoded as described in the *Numbers* section whose lower and upper bounds correspond to the size bounds of the string type.

Booleans. ASN.1 PER Unaligned encodes booleans as the bit constants 0 (False) and 1 (True).

Enumerations. ASN.1 PER Unaligned represents enumeration constants using the smallest Big Endian unsigned integer width that can encode the range of values in the enumeration.

Unions. ASN.1 supports a union operator called `CHOICE`. ASN.1 PER Unaligned prefixes the encoded value with the index to the choice in the union data type as the smallest-width Big Endian unsigned integer that can represent the available choices. ASN.1 also supports the concept of an *open type*. An open type is a container that holds an arbitrary value of a type known by the serializer and the deserializer applications. An open type is encoded as the encoding of the arbitrary value using *fragmentation*. ASN.1 PER Unaligned does not encode the type of the arbitrary value. Therefore, the byte length information allows a deserializer to skip the field if it does not know how to decode it.

Lists. ASN.1 PER Unaligned supports an heterogeneous list type called `SEQUENCE` and an homogeneous list type called `SEQUENCE OF`. Both sequence types can be bounded or unbounded. Unbounded sequences are encoded using *fragmentation* and empty unbounded sequences are encoded as an empty fragment. Bounded lists are encoded as the sequence of its elements with no length metadata. If an heterogeneous sequence (`SEQUENCE`) contains N optional values, then the sequence is prefixed by a sequence of N bits that determine whether each optional value is set.

```
TestSchema DEFINITIONS AUTOMATIC TAGS ::= BEGIN
Test ::= SEQUENCE {
    tags SEQUENCE OF UTF8String,
    tz INTEGER,
    days SEQUENCE OF INTEGER (0..6),
    coord SEQUENCE OF REAL,
    data SEQUENCE OF SEQUENCE {
        name CHOICE { alt1 UTF8String, alt2 NULL } OPTIONAL,
        staff BOOLEAN OPTIONAL,
        extra SEQUENCE { info UTF8String } OPTIONAL
    }
}
END
```

Figure 24: ASN.1 schema to serialize the Figure 10 input data.

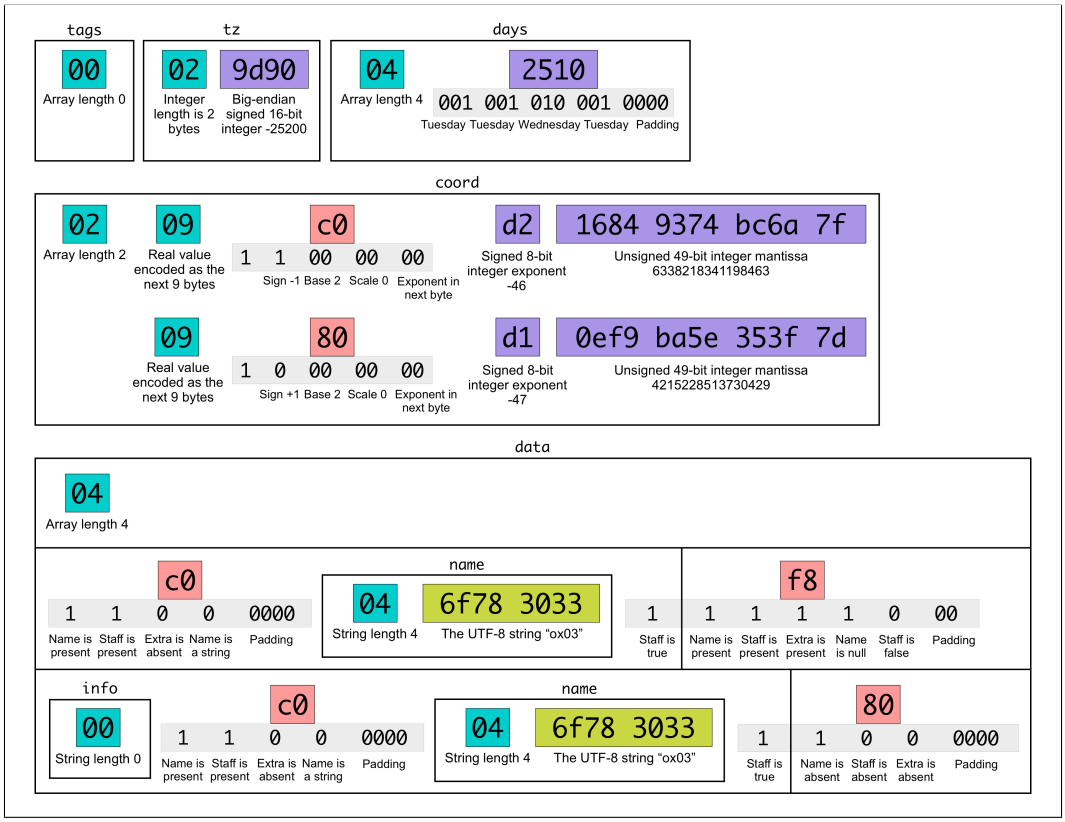

Figure 25: Annotated hexadecimal output of serializing the Figure 10 input data with ASN.1 PER Unaligned.

Table 10: A high-level summary of the ASN.1 PER Unaligned schema-driven serialization specification.

Website	`https://www.itu.int/rec/T-REC-X.680/en`
Company / Individual	International Telecommunication Union
Year	1984
Specification	ITU-T X.680-X.693 [118]
License	Implementation-dependent
Schema Language	ASN.1
Layout	Sequential and order-based
Languages	C, C++, C#, Java, Ada/Spark, Python, Erlang
Types	
Numeric	Big Endian Two's Complement [50] signed integers of user-defined length Big Endian unsigned integers of user-defined length Real numbers consisting of up to 255 bytes encoding the base, scale, exponent, and mantissa Arbitrary-length ASCII-encoded decimal numbers
String	ASCII [27], UTF-8 [32]
Composite	Choice, Enum, Set, Sequence
Scalars	Boolean, Null
Other	Octet string (byte array) Bit-string (arbitrary-length bit array) Date, Time, Date-time [69]

5.2 Apache Avro

```
00000000: 00df 8903 0802 0204 0200 047f 6abc 7493  ............j.t.
00000010: 8456 c0fa 7e6a bc74 f33d 4000 0800 086f  .V..~j.t.=@....o
00000020: 7830 3300 0100 0200 0002 0000 086f 7830  x03..........ox0
00000030: 3300 0100 0202 0000                      3.......
```

Figure 26: Hexadecimal output (`xxd`) of encoding Figure 10 input data with Apache Avro Binary Encoding with no framing (56 bytes).

History. Apache Avro [52] is a schema-driven binary serialization specification introduced by Douglass Cutting [71] during 2009 while working at Yahoo! [72]. Apache Avro is part of the Apache Hadoop [73] framework and is also deeply integrated with other projects from the Big Data field such as Apache Spark [74] and Apache Kafka [75]. Apache Avro is part of the Apache Software Foundation [76] and its released under the Apache License 2.0 [77]. [2] provides a detailed discussion of the role of the Apache Software Foundation in the Big Data industry including Apache Avro.

Characteristics.

- **Optional Code Generation.** Apache Avro implementations can perform serialization and deserialization by taking the input schema at runtime. This enables programs to work with new types of data without a recompilation step. In comparison, schema-driven serialization specifications such as Apache Thrift [126] require a code generation step at build time for each schema.
- **Compactness.** Apache Avro Binary Encoding with no framing produces considerably space-efficient bit-strings as it only encodes minimal metadata [78].

Layout. An Apache Avro Binary Encoding with no framing bit-string is a sequence of values, sometimes nested. The ordering of the values is determined by the schema. Apache Avro Binary Encoding only encodes runtime information that cannot be inferred from the schema, such as the length of the lists or union type information. Fields are encoded even if they equal their default values.

Numbers. Apache Avro Binary Encoding supports 32-bit and 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) signed integers. In terms of real numbers, Apache Avro Binary Encoding supports Little Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51] and arbitrary-precision signed decimal numbers. Arbitrary-precision signed decimal numbers are encoded using variable-length or fixed-length byte arrays. The byte array represents a Big Endian signed integer using Two's Complement [50] whose width is defined by the byte array length prefix. The schema declares the scale and precision as integers. The resulting decimal number equals the unscaled integer multiplied by $10^{-\text{scale}}$. The precision integer determines the maximum number of decimals stored in the data type.

Strings. Apache Avro Binary Encoding strings are encoded using UTF-8 [32] without *NUL* delimiters. Apache Avro Binary Encoding prefixes strings with 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length signed integers that represent the number of code-points in the string. Empty strings are encoded with a length prefix of 0 with no following characters. Apache Avro Binary Encoding does not attempt to deduplicate multiple occurrences of the same string.

Booleans. Apache Avro Binary Encoding encodes booleans using the byte constants `0x00` (False) and `0x01` (True).

[71] `https://github.com/cutting`
[72] `https://yahoo.com/`
[73] `https://hadoop.apache.org`
[74] `http://spark.apache.org/index.html`
[75] `https://kafka.apache.org`
[76] `https://www.apache.org`
[77] `http://www.apache.org/licenses/LICENSE-2.0.html`
[78] `https://avro.apache.org/docs/current/spec.html`

Enumerations. Apache Avro Binary Encoding represents enumeration constants using 32-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length signed integers.

Unions. An Apache Avro schema may declare an ordered list of potential types for a single field. In these cases, Apache Avro Binary Encoding prefixes the value with a 32-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length signed integer that corresponds to an index of the ordered list of types. Refer to Figure 27 for a visual example.

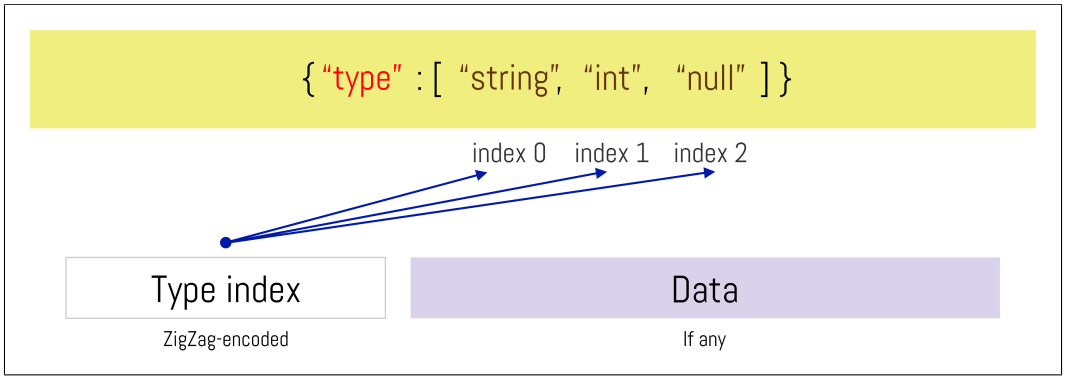

Figure 27: A field consisting of more than one possible type is encoded as the type index followed by the data, if any. For example, if the schema defines a field whose type is `[ "string", "int", "null" ]`, then the value will be prefixed with `0x00` if the value is a string, with `0x02` (the ZigZag-encoded (4.2) integer 1) if the value is an integer, or with `0x04` (the ZigZag-encoded (4.2) integer 2) if the value is null.

Lists. Apache Avro Binary Encoding encodes a list (array) as a series of blocks. Each block is prefixed with its logical size as a 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length signed integer followed by its elements in order. As a runtime optimization, if the logical size signed integer is negative, then the block size is the absolute value of the integer and the sequence of elements is further prefixed with the byte-length of the block as another 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length signed integer. Apache Avro Binary Encoding does not attempt to deduplicate multiple occurrences of the same element in an array.

[79] `https://avro.apache.org/docs/current/spec.html#Duration`

```
{
  "namespace": "schema.avro",
  "type": "record",
  "name": "Test",
  "fields": [
    { "name": "tags", "type": { "type": "array", "items": "string" } },
    { "name": "tz", "type": "int" },
    { "name": "days", "type": {
      "type": "array", "items": { "type": "enum", "name": "Day", "symbols": [
        "MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY", "SATURDAY", "SUNDAY"
      ] }
    } },
    { "name": "coord", "type": { "type": "array", "items": "double" } },
    { "name": "data", "type": {
      "type": "array", "items": {
        "type": "record", "name": "DataEntry", "fields": [
          { "name": "name", "type": [ "string", "null" ], "default": null },
          { "name": "staff", "type": [ "boolean", "null" ], "default": null },
          { "name": "extra",
            "default": null,
            "type": [ "null", {
              "type": "record",
              "name": "Metadata",
              "fields": [ { "name": "info", "type": "string" } ]
            } ]
          }
        ]
      }
    } }
  ]
}
```

Figure 28: Apache Avro schema to serialize the Figure 10 input data.

Table 11: A high-level summary of the Apache Avro Binary Encoding with no framing schema-driven serialization specification.

Website	`https://avro.apache.org`
Company / Individual	Apache Software Foundation
Year	2009
Specification	`https://avro.apache.org/docs/current/spec.html`
License	Apache License 2.0
Schema Language	Avro IDL
Layout	Sequential and order-based
Languages	C, C++, C#, Java
Types	
Numeric	32-bit and 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length integers Arbitrary-precision Two's Complement [50] signed decimal numbers Little Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	Array, Enum, Map, Record, Union
Scalars	Boolean, Null
Other	Bytes (variable-length byte array) Fixed (fixed-length byte array) UUID [80] Date (days from the UNIX Epoch) [89] Time (milliseconds and microseconds) [69] Timestamp (milliseconds and microseconds) [89] Duration [79]

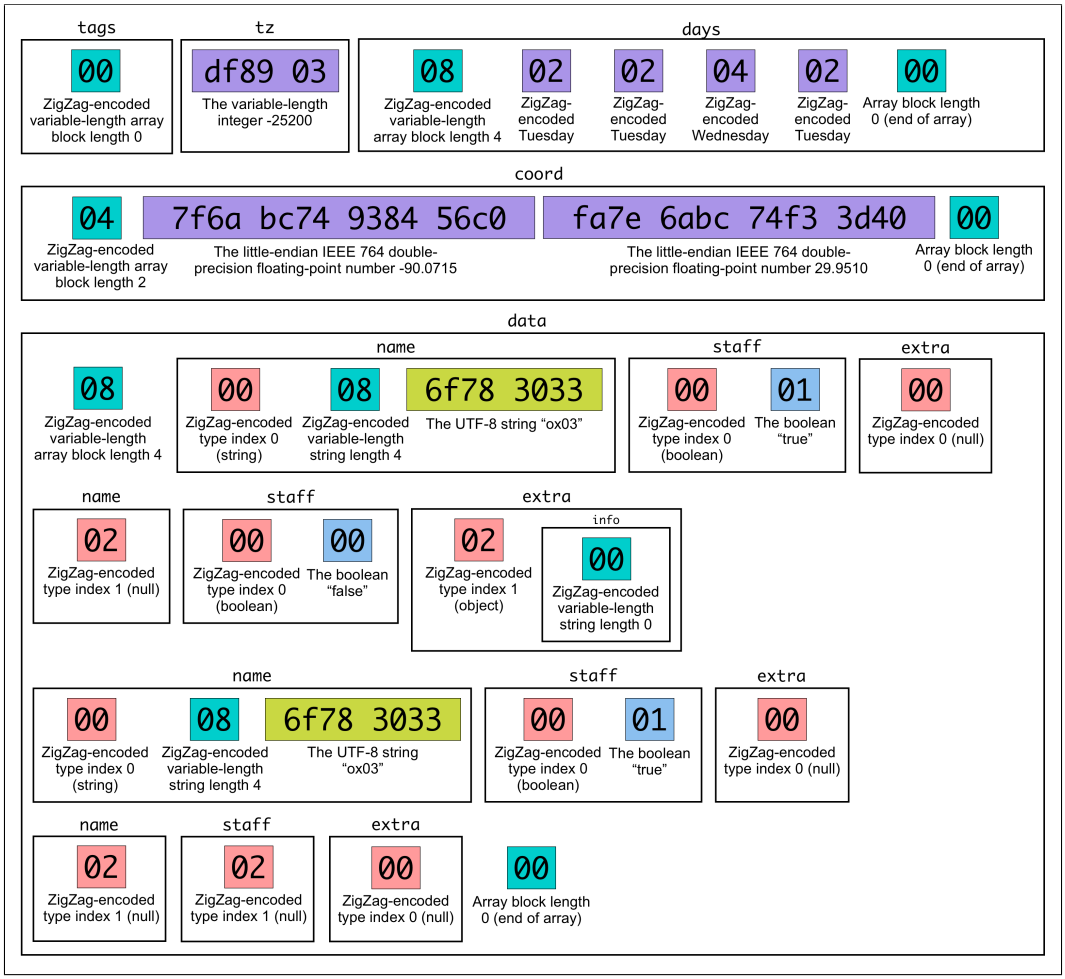

Figure 29: Annotated hexadecimal output of serializing the Figure 10 input data with Apache Avro Binary Encoding with no framing.

5.3 Microsoft Bond

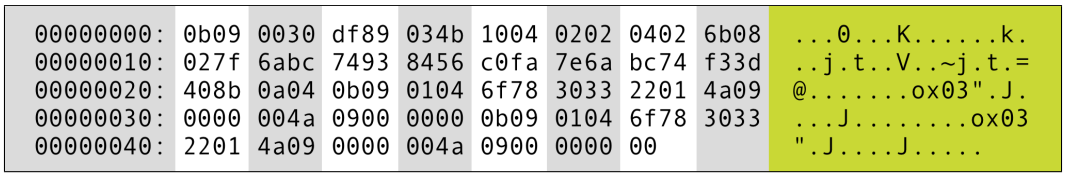

Figure 30: Hexadecimal output (`xxd`) of encoding Figure 10 input data with Microsoft Bond Compact Binary version 1 (77 bytes).

History. Microsoft Bond [87] is an RPC protocol and schema-driven serialization specification developed by Microsoft in 2011 and was made open-source in 2015. The Microsoft Bond project was started by Adam Sapek [80], a Principal Software Engineer at Microsoft Research, while working on the Microsoft Bing [81] search engine. Microsoft Bond is used at the core of many Microsoft services and it is released under the MIT license [82].

Characteristics.

- **Rich Type System.** The Microsoft Bond schema language supports generic types, inheritance and a wide range of scalar and composite data types such as sets and nullable types.
- **Custom Type Mappings.** To ease integration, Microsoft Bond supports statically mapping the types supported by the schema language to any compatible programming language type.
- **Opt-in Lazy Deserialization.** For runtime-performance reasons, the Microsoft Bond schema language supports marking certain fields to not be de-serialized automatically. These fields can be de-serialized when needed or omitted altogether.

Layout. A Microsoft Bond Compact Binary version 1 (v1) bit-string is a sequence of values, sometimes nested. Each value is prefixed with a type definition and the length of the value, where applicable. Each value has a numeric identifier that must be unique on the current nesting level. Microsoft Bond Compact Binary v1 only encodes a field if the field is required or if its value is not equal to the default hence resulting in efficient use of space.

Types. A Microsoft Bond Compact Binary v1 type definition consist of an positive absolute unique identifier integer and a type identifier constant as shown in Figure 31. Microsoft Bond Compact Binary v1 defines three type definition encodings depending on the length of the unique identifier. Refer to Table 12.

Table 12: The three type definition encodings that Microsoft Bond Compact Binary v1 supports depending on the value of the unique field identifier as determined by the *From* and *To* ranges. Note that Microsoft Bond Compact Binary v1 cannot encode unique field identifiers larger than 65535.

From	To	Size	First 3-bits	Next 5-bits	Remaining bits
0	5	1 byte	The unique identifier as a 3-bit unsigned integer	The field type identifier constant as shown in Figure 31	None
6	255	2 bytes	`110`	The field type identifier constant as shown in Figure 31	The unique identifier as an 8-bit unsigned integer
256	65535	3 bytes	`111`	The field type identifier constant as shown in Figure 31	The unique identifier as a Big Endian 16-bit unsigned integer

[80] `https://github.com/sapek`
[81] `https://www.bing.com`
[82] `https://opensource.org/licenses/MIT`

Microsoft Bond schemas can instruct implementations to not de-serialize a field automatically by marking the field as *bonded* at the schema level. Instead, the consumer de-serializes *bonded* fields when and if needed by the application. This results in runtime efficiency.

Numbers. Microsoft Bond Compact Binary v1 supports Little Endian IEEE 764 32-bit and 64-bit floating point-numbers [51]. In terms of integers, Microsoft Bond Compact Binary v1 supports Little Endian 8-bit fixed-length unsigned integers, 16-bit, 32-bit, and 64-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integers, 8-bit fixed-length signed integers with Two's Complement [50], and 16-bit, 32-bit, and 64-bit Little Endian Base 128 (LEB128) (4.1) variable-length ZigZag-encoded (4.2) signed integers.

Strings. Microsoft Bond can produce strings with UTF-8 and Little Endian UTF-16 encodings [32] without *NUL* delimiters. Refer to the `BT_STRING` and `BT_WSTRING` data types from Figure 31. Each string is prefixed with a 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integer determining the number of code-points in the string. Microsoft Bond Compact Binary v1 does not attempt to deduplicate multiple occurrences of the same string.

Booleans. Microsoft Bond Compact Binary v1 encodes booleans using the 1-byte constants `0x00` (False) and `0x01` (True).

Enumerations. Microsoft Bond Compact Binary v1 represents enumeration constants using 32-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length signed integers.

Unions. Microsoft Bond supports the *nullable* type union. Nullable fields are encoded as lists of zero or one value. Refer to Figure 32 for a visual example. Microsoft Bond Compact Binary v1 does not encode unset optional fields. Therefore schema-writers can approximate unions by defining a structure containing multiple fields and ensuring that only one of them is set at a time. Another common pattern to approximate unions is to rely on polymorphism and bonded types. In this case, a schema-writer could define a base structure including a single field: an enumeration denoting

[83] `https://github.com/microsoft/bond/blob/8d0fe6c00cbcd7ea9c54b1f1e947174caff596e4/idl/bond/core/bond_const.bond`

```
// Enumerator of Bond meta-schema types
enum BondDataType
{
    BT_STOP       = 0,
    BT_STOP_BASE  = 1,
    BT_BOOL       = 2,
    BT_UINT8      = 3,
    BT_UINT16     = 4,
    BT_UINT32     = 5,
    BT_UINT64     = 6,
    BT_FLOAT      = 7,
    BT_DOUBLE     = 8,
    BT_STRING     = 9,
    BT_STRUCT     = 10,
    BT_LIST       = 11,
    BT_SET        = 12,
    BT_MAP        = 13,
    BT_INT8       = 14,
    BT_INT16      = 15,
    BT_INT32      = 16,
    BT_INT64      = 17,
    BT_WSTRING    = 18,
    BT_UNAVAILABLE= 127
}
```

Figure 31: Microsoft Bond type identifiers definition [83].

the union choice and extend the base structure with subclasses defining each of the choices. Other structures refer to the union structure as a bonded field, so that the client can first deserialize the enumeration and then deserialize the rest of the fields depending on the enumeration constant value. The serializer program is responsible for correctly setting the enumeration constant. In Figure 33, a heterogeneous list using this technique is shown.

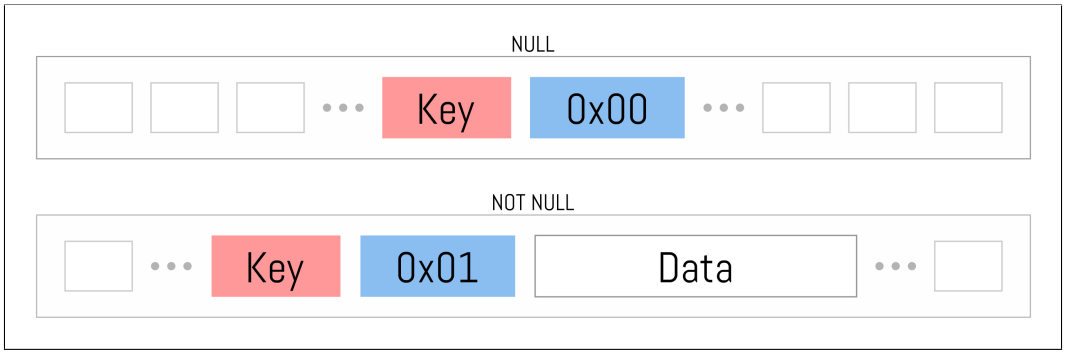

Figure 32: A visual representation of Microsoft Bond Compact Binary v1 *nullable* types. If the value is *NULL* (top), then it is encoded as the type definition followed by the length `0x00`. If the value is not null (bottom), then it is encoded as the type definition followed by the length `0x01`, followed by the value.

```
enum Kind { Unknown; WithString; WithBool; }

struct Base {
  0: Kind kind = Unknown;
}

struct Struct1 : Base {
  0: string extra;
}

struct Struct2 : Base {
  0: bool extra;
}

struct Example {
  0: list<bonded<Base>> items;
}
```

Figure 33: An adapted example of a polymorphic list definition [84].

Lists. Microsoft Bond Compact Binary v1 encodes a list as the type definition, followed by the list definition, followed by the elements encoded in order. The list definition consists in a byte where the five least-significant bits encode the element type identifier constant as shown in Figure 31 followed by the length of the list as a 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integer. Microsoft Bond Compact Binary v1 does not attempt to deduplicate multiple occurrences of the same element in a list.

[84] `https://github.com/microsoft/bond/blob/8d0fe6c00cbcd7ea9c54b1f1e947174caff596e4/examples/cpp/core/polymorphic_container/polymorphic_container.bond`

```
namespace Test

enum Day {
  MONDAY = 0, TUESDAY = 1, WEDNESDAY = 2, THURSDAY = 3,
  FRIDAY = 4, SATURDAY = 5, SUNDAY = 6
}

struct Metadata {
  0: required string info;
}

struct DataEntry {
  0: nullable<string> name;
  1: bool staff;
  2: Metadata extra;
}

struct Test {
  0: required list<string> tags;
  1: required int32 tz;
  2: required list<Day> days;
  3: required list<double> coord;
  4: required list<DataEntry> data;
}
```

Figure 34: Microsoft Bond schema to serialize the Figure 10 input data.

Table 13: A high-level summary of the Microsoft Bond Compact Binary v1 schema-driven serialization specification.

Website	`https://microsoft.github.io/bond/`
Company / Individual	Microsoft
Year	2011
Specification	`https://microsoft.github.io/bond/reference/cpp/compact__binary_8h_source.html`
License	MIT
Schema Language	Bond IDL
Layout	Sequential with field identifiers
Languages	C++, C#, Java, Python
Types	
Numeric	16-bit, 32-bit, and 64-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integers 16-bit, 32-bit, and 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (LEB128) (4.1) variable-length signed integers Fixed-length 8-bit unsigned integers Fixed-length 8-bit Two's Complement [50] signed integers Little Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8, Little Endian UTF-16 [32]
Composite	List, Maybe, Nullable, Set, Struct, Vector
Scalars	Boolean
Other	Blob (byte array)

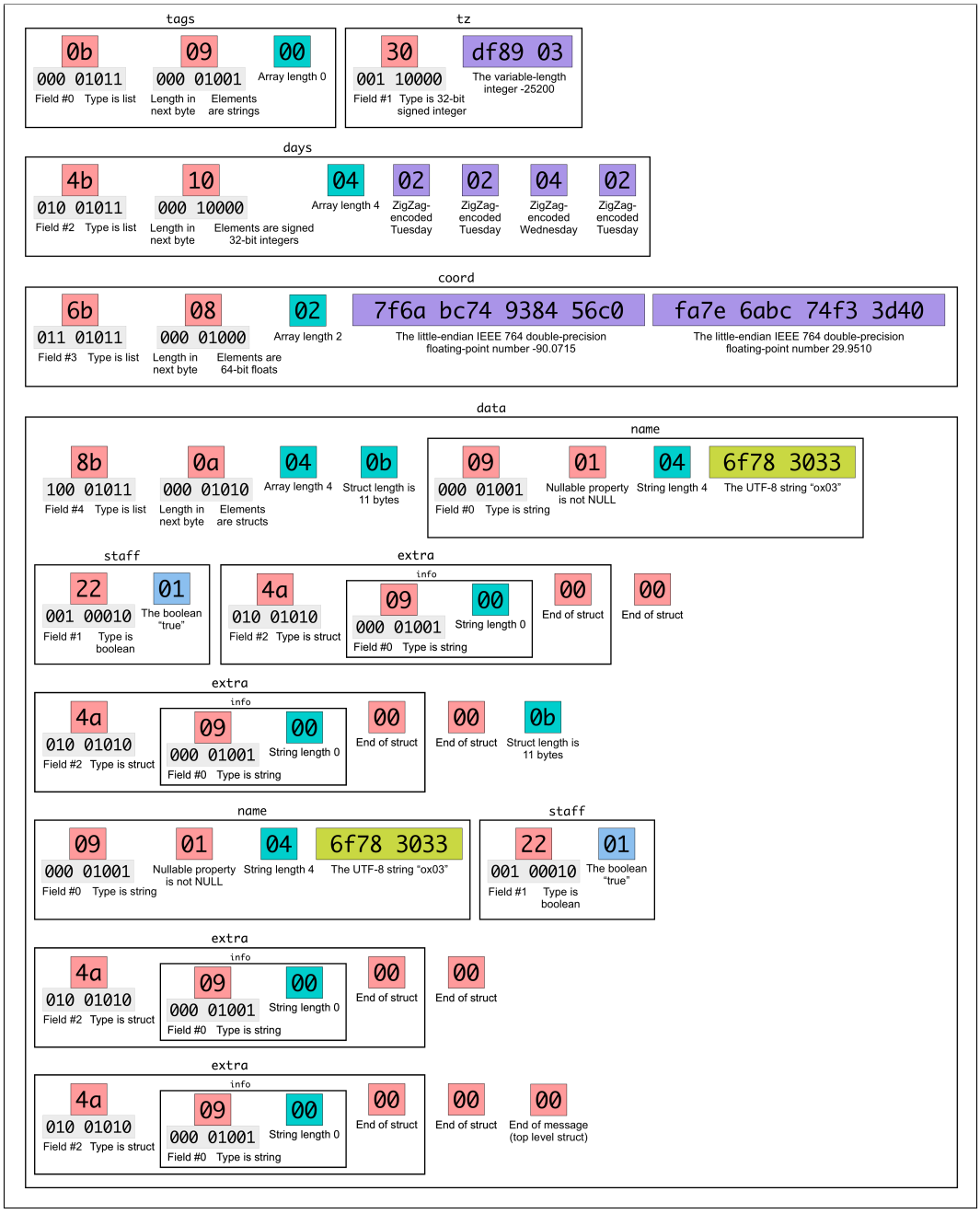

Figure 35: Annotated hexadecimal output of serializing the Figure 10 input data with Microsoft Bond Compact Binary v1.

5.4 Cap'n Proto

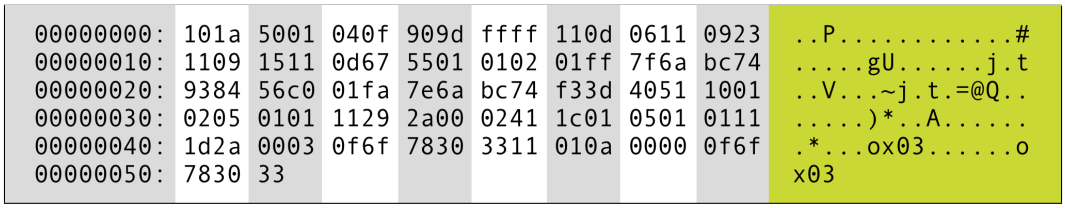

Figure 36: Hexadecimal output (`xxd`) of encoding Figure 10 input data with Cap'n Proto Packed Encoding (83 bytes).

History. Cap'n Proto [142] is an RPC protocol and schema-driven binary serialization specification created in 2013 by Kenton Varda [85], the primary author of Protocol Buffers [59] version 2, while working as a Technology Lead at Sandstorm [86]. Cap'n Proto is designed to support memory-efficient serialization and deserialization. Cap'n Proto is extensively used at Sandstorm and at high-profile companies such as Cloudflare [87], where Kenton Varda is currently employed as a Principal Engineer. Cap'n Proto is released under the MIT license [88].

Characteristics.

- **Efficient Reads.** Cap'n Proto produces implementations that perform runtime-efficiency and memory-efficient incremental and random-access reads as noted by [155].
- **Small Code Footprint.** Cap'n Proto includes a small runtime library with minimal dependencies and generates small amounts of serialization and deserialization code.

Layout. A Cap'n Proto bit-string consist of a tree hierarchy of pointers that eventually points at scalar types. These pointers are scattered across the bit-string close to the data that they point to for cache locality purposes. For runtime-performance reasons, Cap'n Proto values are aligned to 64-bit words. As a consequence, Cap'n Proto bit-strings tend to contain significant zero-byte padding. As a solution, Cap'n Proto defines a simple compression scheme called *Packed Encoding* where each 64-bit word in the bit-string is replaced by a tag byte followed by up to 8 content bytes. The position of the bits set in the tag byte determines the location of each content byte in the uncompressed 64-bit word. Refer to Figure 37 for a visual example. Additionally, Cap'n Proto Packed Encoding compresses sequences of zero-byte 64-bit using the `0x00` byte followed by the amount of zero-byte 64-bit words minus 1 as an 8-bit unsigned integer. The compression scheme can encode unpacked data prefixing the unpacked 64-bit word with the `0xff` byte and suffixing it with the amount of unpacked words to follow as an 8-bit unsigned integer.

Cap'n Proto structures consist of a 64-bit type definition, followed by **N** 64-bit words of scalar values, followed by **M** 64-bit pointers to composite values. The structure type definition and the remaining 64-bit words do not need to be contiguous in memory as the most-significant 30-bits of the structure type definition consists of a pointer to the data section. The next 2-bits equal 00 to declare that the type definition corresponds to a structure. The remaining 32-bits encode two Little Endian 16-bit unsigned integers corresponding to the word-lengths of the data and pointer sections, respectively.

Numbers. Cap'n Proto supports Little Endian IEEE 754 32-bit and 64-bit floating-point numbers [51]. In terms of integers, Cap'n Proto supports Little Endian 8-bit, 16-bit, 32-bit, and 64-bit unsigned integers and Little Endian 8-bit, 16-bit, 32-bit, and 64-bit Two's Complement [50] signed integers.

Strings. Cap'n Proto encodes strings as lists of UTF-8 [32] characters. Cap'n Proto strings are delimited with the *NUL* ASCII [27] character. However, the *NUL* character is typically packed by the Cap'n Proto word compression scheme. The list definition corresponding to the string encodes the byte-length of the string as a Little Endian 30-bit unsigned integer. Cap'n Proto does not attempt to deduplicate multiple occurrences of the same string.

[85] `https://github.com/kentonv`
[86] `https://sandstorm.io`
[87] `https://www.cloudflare.com`
[88] `https://opensource.org/licenses/MIT`

Booleans. Cap'n Proto encodes booleans as the bits 0 (False) or 1 (True) aligned to a multiple of their size on the structure they are defined in.

Enumerations. Cap'n Proto represents enumeration constants using aligned Little Endian 16-bit unsigned integers. As a consequence, Cap'n Proto does not support negative enumeration constants.

Unions. Cap'n Proto relies on unique field identifiers to implement union types. Each alternative in the union type must have a different field identifier and Cap'n Proto enforces that only one of such unique field identifiers is present at a given time. Cap'n Proto prefixes the encoded value with an 8-bit unsigned integer that determines the corresponding union field identifier.

Lists. Cap'n Proto encodes lists as a 64-bit list type definition with the elements encoded in order. The list type definition and the list elements do not need to be contiguous in memory. Cap'n Proto does not attempt to deduplicate multiple occurrences of the same value in a list. Cap'n Proto defines two list type definition encodings depending on whether the elements are scalar or composite as shown in Table 14.

Table 14: The two 64-bit list type definition encodings supported by Cap'n Proto depending on whether the elements of the list are scalar or composite values.

Element type	**First 30-bits**	**Next 2-bits**	**Next 29-bits**	**Remaining 3-bits**
Scalar	Pointer to the start of the list	`01`	Number of elements in the list as a Little Endian 29-bit unsigned integer	The element length definition as a 3-bit unsigned integer as shown in Figure 38
Composite	Pointer to the composite tag word definition	`01`	Number of 64-bit words in the list as a Little Endian 29-bit unsigned integer	`111`

If the list consists of composite elements, then the list definitions points at a 64-bit word that describes each element. This 64-bit word starts with the number of elements in the list as a Little Endian 30-bit unsigned integer, followed by the 2-bit constant `00`, followed by the number of 64-bit scalar words in the element as a Little Endian 16-bit unsigned integer, followed by the number of 64-bit pointers in the element as another Little Endian 16-bit unsigned integer.

[89] `https://github.com/capnproto/capnproto/blob/bbea19f0e0b2ee1ed28d0836b778d8cf3995597d/c%2B%2B/src/capnp/common.h`

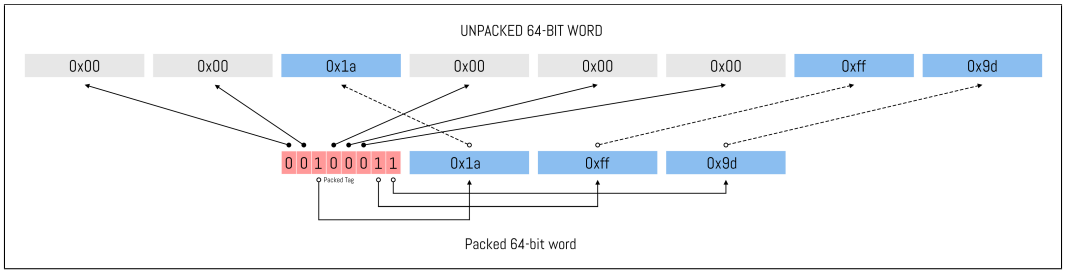

Figure 37: Cap'n Proto Packed Encoding compresses a 64-bit word by encoding a tag byte where its bits represent the unpacked bytes followed by up to 8 content bytes. The number of content bytes equal the number if bits set in the tag byte. If the bit from the tag byte is zero, the corresponding unpacked byte is zero. If the bit from the tag byte is set, then the byte is the corresponding unpacked byte following the tag byte.

```
enum class ElementSize: uint8_t {
  // Size of a list element.

  VOID = 0,
  BIT = 1,
  BYTE = 2,
  TWO_BYTES = 3,
  FOUR_BYTES = 4,
  EIGHT_BYTES = 5,

  POINTER = 6,

  INLINE_COMPOSITE = 7
};
```

Figure 38: Cap'n Proto list element size definitions [89].

```
@0x814a1b775e7ad635;

enum Day {
  monday @0; tuesday @1; wednesday @2; thursday @3;
  friday @4; saturday @5; sunday @6;
}

struct Metadata { info @0 :Text; }
struct DataEntry {
  union { nullname @0 :Void; name @1 :Text; }
  staff @2 :Bool;
  extra @3 :Metadata;
}

struct Test {
  tags @0 :List(Text);
  tz @1 :Int32;
  days @2 :List(Day);
  coord @3 :List(Float64);
  data @4 :List(DataEntry);
}
```

Figure 39: Cap'n Proto schema to serialize the Figure 10 input data.

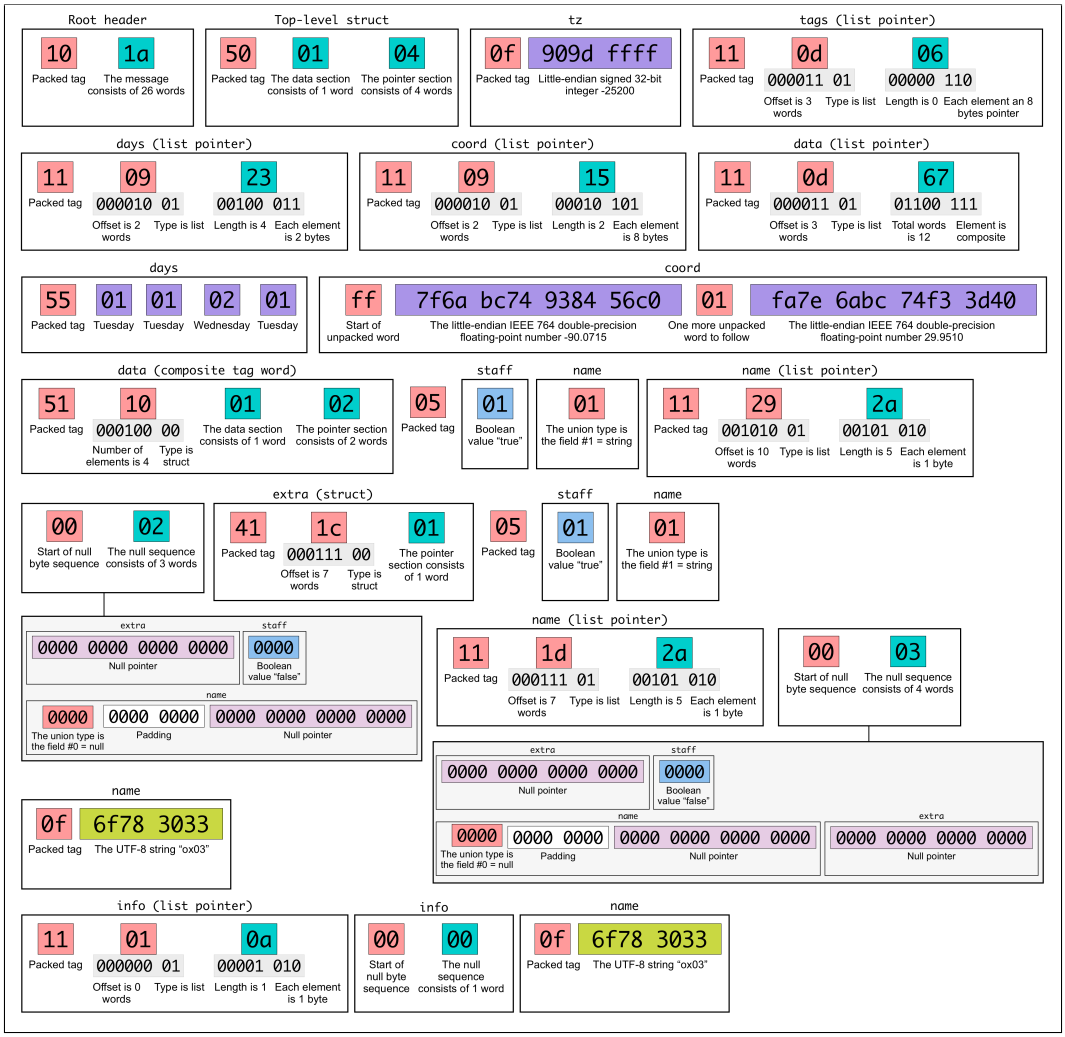

Figure 40: Annotated hexadecimal output of serializing the Figure 10 input data with Cap'n Proto Packed Encoding.

Table 15: A high-level summary of the Cap'n Proto schema-driven serialization specification.

Website	`https://capnproto.org`
Company / Individual	Sandstorm
Year	2013
Specification	`https://capnproto.org/encoding.html`
License	MIT
Schema Language	Cap'n Proto IDL
Layout	Pointed-based and order-based
Languages	C, C++, C#, D, Erlang, Go, Haskell, Java, JavaScript, Lua, Nim, OCaml, Python, Ruby, Rust, Scala
Types	
Numeric	Little Endian 8-bit, 16-bit, 32-bit, and 64-bit Two's Complement [50] signed integers Little Endian 8-bit, 16-bit, 32-bit, and 64-bit unsigned integers Little Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	Enum, List, Struct, Union
Scalars	Bool, Void
Other	Data (byte array)

5.5 FlatBuffers

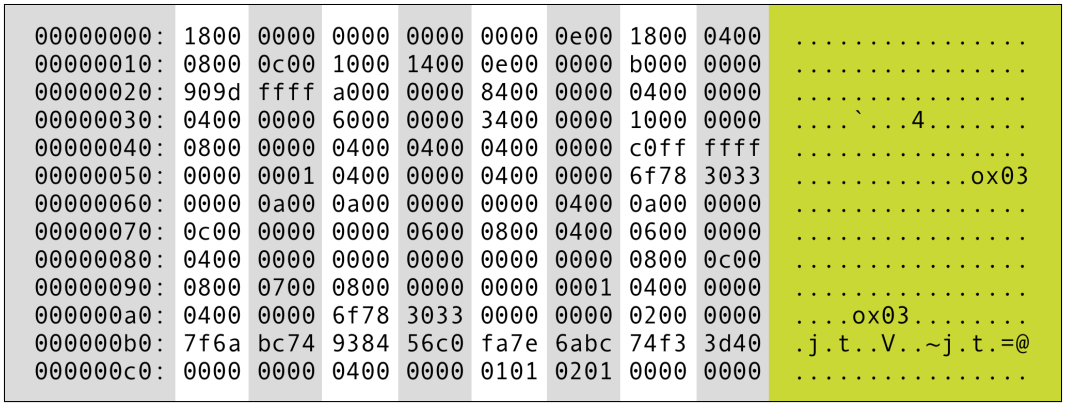

Figure 41: Hexadecimal output (`xxd`) of encoding Figure 10 input data with FlatBuffers Binary Wire Format (208 bytes).

History. FlatBuffers [139] is a schema-driven serialization specification created at Google in 2014 by the *Fun Propulsion Labs* (FPL) group whose mission was to improve game-related technologies for Android. FlatBuffers has been designed to support memory-efficient serialization and deserialization in the context of games and mobile. The project was started by Wouter van Oortmerssen [90], a Software Engineer at Google, and was released under the Apache License 2.0 [91]. FlatBuffers is also used in the context Machine Learning as part of the TensorFlow Lite [92] framework for mobile and IoT devices developed by Google [137] and in the DOS spatial system [103].

Characteristics.

- **Efficient Reads.** FlatBuffers produces implementations that perform runtime-efficiency and memory-efficient incremental and random-access reads as noted by [110], [10], [108], [100], and [71]. Given its efficient deserialization process, [136] proposes FlatBuffers as the specification in a system architecture for data and video streaming with unmanned aerial vehicles for natural disaster management.
- **Small Code Footprint.** FlatBuffers includes a small runtime library with minimal dependencies and generates small amounts of serialization and deserialization code.

Layout. A FlatBuffers Binary Wire Format bit-string consist of a tree hierarchy of 32-bit relative pointers that eventually point at scalar types. A FlatBuffers bit-string starts with a pointer to the root element. The FlatBuffers core data structure is a *Table*. A FlatBuffers Table is an ordered sequence of aligned values prefixed with a pointer to a *vTable* structure that defines the layout of the Table. A vTable consist of two Little Endian 16-bit unsigned integers describing the byte-lengths of the vTable and the Table. The size declarations are followed by a sequence of Little Endian 16-bit unsigned integer offsets to each element in the Table relative to the vTable pointer. Refer to Figure 42 for a visual example. As a space optimisation, multiple Tables sharing the same layout may point to the same vTable. FlatBuffers does not encode values that equal their default.

As an alternative to Tables, FlatBuffers supports the concept of *structs*. A struct is a more space-efficient alternative to a Table, however a struct can only include scalar values and other structs, and lacks the versioning and extensibility features of a Table. FlatBuffers encodes structs as the sequence of its members aligned to the largest scalar element it contains.

Numbers. FlatBuffers supports Little Endian IEEE 754 32-bit and 64-bit floating-point numbers [51]. In terms of integers, FlatBuffers supports Little Endian 8-bit, 16-bit, 32-bit and 64-bit unsigned integers and Little Endian 8-bit, 16-bit, 32-bit and 64-bit Two's Complement [50] signed integers.

[90] `https://github.com/aardappel`

[91] `http://www.apache.org/licenses/LICENSE-2.0.html`

[92] `https://www.tensorflow.org/lite/api_docs/cc/class/tflite/flat-buffer-model`

Strings. FlatBuffers produces *NUL*-delimited UTF-8 [32] strings. FlatBuffers strings are prefixed with a Little Endian 32-bit unsigned integer that represents the byte-length of the string without taking the *NUL* delimiter into consideration. Empty strings are encoded with a length 0 followed by the *NUL* character. By default, FlatBuffers does not attempt to deduplicate multiple occurrences of the same string. However, its serialization interface allows the application to track and share duplicated string values.

Booleans. FlatBuffers encodes booleans as the Little Endian unsigned integers 0 (False) and 1 (True) aligned to their own size.

Enumerations. FlatBuffers lets the schema-writer decide the data type to represent enumeration constants. A common choice is the `byte` type that represents an 8-bit signed Two's Complement [50] integer.

Unions. FlatBuffers encodes union data types as the combination of two fields: an enumeration that represents the union alternative choices and the offset to the union value. FlatBuffers reserves the union identifier 0 to mean that the value is not set. FlatBuffers unions do not support scalar data types. However, a FlatBuffers union may include a *struct* consisting of a single scalar value encoded with no space overhead.

Lists. A FlatBuffers list (vector) consists of the concatenation of its elements prefixed by the logical size of the vector as a Little Endian 32-bit unsigned integer. A vector of composite elements is encoded as a list of 32-bit pointers. By default, FlatBuffers does not attempt to deduplicate multiple occurrences of the same element in a vector. However, its serialization interface allows the application to track and share duplicated vector composite elements.

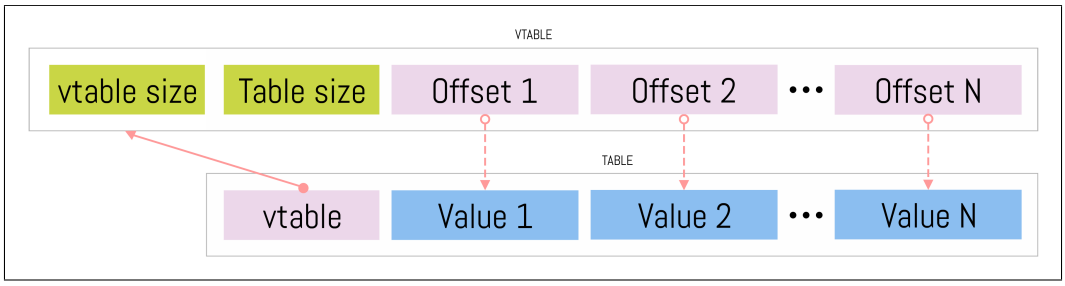

Figure 42: A Table is an aligned sequence of values. Tables are prefixed with a pointer to a *vTable* structure that defines the layout of the Table by specifying the offsets to each element.

```
namespace TestDocument;

table Metadata {
  info: string;
}

table DataEntry {
  name: string;
  staff: bool;
  extra: Metadata;
}

enum Day:byte {
  Monday = 0, Tuesday = 1, Wednesday = 2, Thursday = 3,
  Friday = 4, Saturday = 5, Sunday = 6
}

table Test {
  tags: [string];
  tz: int;
  days: [Day];
  coord: [double];
  data: [DataEntry];
}

root_type Test;
```

Figure 43: FlatBuffers schema to serialize the Figure 10 input data.

Table 16: A high-level summary of the FlatBuffers Binary Wire Format schema-driven serialization specification.

Website	`https://google.github.io/flatbuffers/`
Company / Individual	Google
Year	2014
Specification	`https://google.github.io/flatbuffers/flatbuffers_internals.html`
License	Apache License 2.0
Schema Language	FlatBuffers IDL
Layout	Pointer-based and order-based
Languages	C, C++, C#, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust, Swift
Types	
Numeric	Little Endian 8-bit, 16-bit, 32-bit, and 64-bit unsigned integers Little Endian 8-bit, 16-bit, 32-bit, and 64-bit Two's Complement [50] signed integers Little Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	Array, Enum, Struct, Table, Union, Vector
Scalars	Boolean

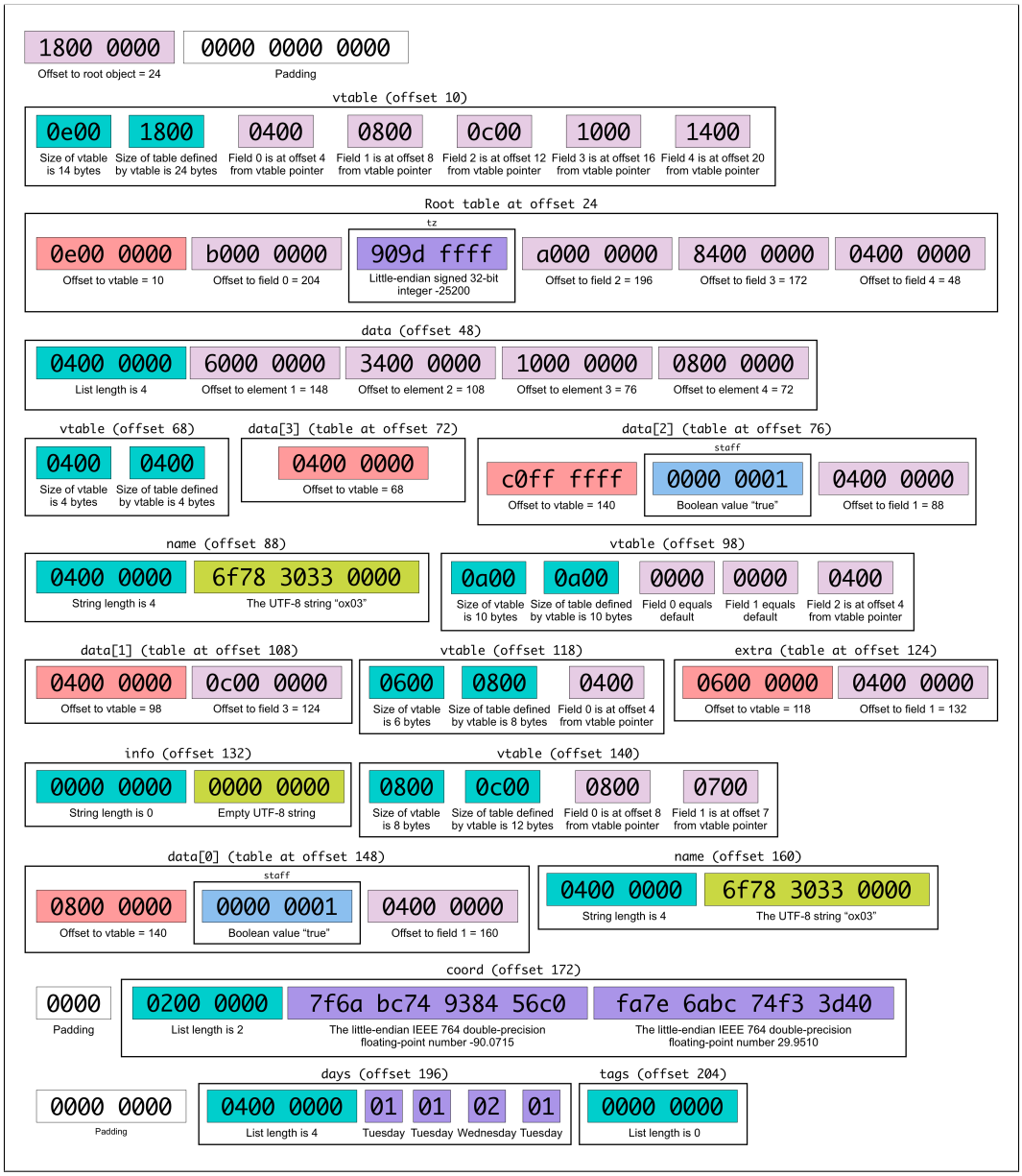

Figure 44: Annotated hexadecimal output of serializing the Figure 10 input data with FlatBuffers Binary Wire Format.

5.6 Protocol Buffers

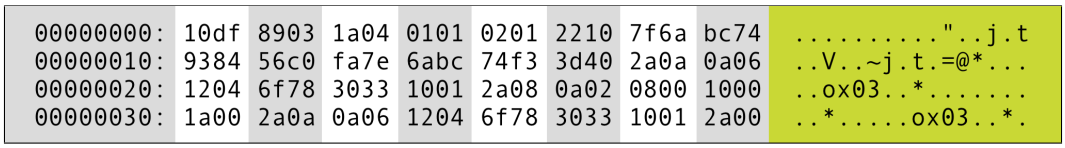

Figure 45: Hexadecimal output (`xxd`) of encoding Figure 10 input data with Protocol Buffers Binary Wire Format (64 bytes).

History. Protocol Buffers [59] is an RPC protocol and schema-driven binary serialization specification developed by Google in 2001 [93] and open-sourced under the 3-clause BSD license [94] in 2008 [95]. Protocol Buffers was initially maintained by Jon Skeet [96], a Staff Software Engineer at Google. Google uses Protocol Buffers for nearly all of its storage and transmission needs.

Related Literature. Protocol Buffers has been extensively optimized in the context of data centers. [101] further improve runtime efficiency of the official Protocol Buffers C++ implementation using hardware acceleration through a programmed co-processor. [77] describes *ProtoML*, a tool that takes complex constrains definitions and generates code to validate and potentially correct Protocol Buffers messages at runtime. [153] successfully developed a formally verified subset (most notably missing unions and recursive messages) of Protocol Buffers version 3 using the Coq [97] formal proof assistant. Protocol Buffers is used to encode certain parts of the *ProMC* specification to store representations of high-energy physics data for space-efficiency reasons [30]. Protocol Buffers is also a core component of the Caffe [72] deep learning framework [98] used by popular projects such as OpenPose, an open-source real-time multi-person keypoint detection library [24].

Characteristics.

- **Robustness.** The Protocol Buffers schema-driven serialization specification and official implementations have been battle-tested by Google in high-scale production environments.
- **Security.** Protocol Buffers has been designed with security in mind and has undergone reviews by the Google security team.
- **Popularity.** Protocol Buffers is one of the most popular schema-driven binary serialization specifications. As a result, it features excellent documentation, relevant tools and an active open source community.

Layout. A Protocol Buffers Binary Wire Format bit-string is a sequence of values, sometimes nested. Each value is prefixed with a type definition and the length of the value, if applicable. Each value has a numeric identifier that must be unique on the current nesting level. As a space-optimization, Protocol Buffers only encodes a field if its value is not equal to the default or if the field is explicitly marked as optional. The order of fields in a Protocol Buffers Binary Wire Format bit-string is non-deterministic.

Types. A Protocol Buffers Binary Wire Format type definition is a 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length integer encoding of the concatenation of a Big Endian arbitrary-length unsigned integer identifier and its 3-bit type category identifier. Protocol Buffers groups the data types it supports into a set of type categories depending on their length characteristics as shown in Table 17. The de-serializer refers to the schema for the specific data type.

For example, a type definition consisting of 64-bit value with a unique field identifier 5 is encoded as `0x2a = 00101 (5) 001 (wire type)` and a type definition consisting of a length-delimited value with a unique field identifier 17 is encoded as `0x9a 0x02`, the Little Endian Base 128 (LEB128) (4.1) variable-length integer encoding of `100011 (35) 010 (wire type)`.

[93] `https://developers.google.com/protocol-buffers/docs/faq`
[94] `https://opensource.org/licenses/BSD-3-Clause`
[95] `https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html`
[96] `https://github.com/jskeet`
[97] `https://coq.inria.fr`
[98] `https://caffe.berkeleyvision.org/tutorial/layers.html`

Numbers. Protocol Buffers supports Little Endian IEEE 754 32-bit and 64-bit floating-point numbers [51]. In terms of integers, Protocol Buffers supports 32-bit and 64-bit fixed-length and Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integers, 32-bit and 64-bit fixed-length and Little Endian Base 128 (LEB128) (4.1) variable-length Two's Complement [50] signed integers, and 32-bit and 64-bit Little Endian Base 128 (LEB128) (4.1) variable-length ZigZag-encoded (4.2) signed integers. Fixed-length numbers make use of the *32-bit values* or *64-bit values* wire types while variable-length integers make use of the *variable-length integer values* wire type as shown in Table 17.

Strings. Protocol Buffers Binary Wire Format produces UTF-8 [32] strings that are not *NUL* delimited encoded using the *length-delimited* wire type as shown in Table 17. A Protocol Buffers Binary Wire Format string is prefixed with a 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integer that declares the byte-length of the string. Protocol Buffers Binary Wire Format encodes empty strings with a zero-length prefix and no additional data. Protocol Buffers Binary Wire Format does not deduplicate multiple occurrences of the same string.

Booleans Protocol Buffers Binary Wire Format encodes booleans using the *variable-length integer* wire type as shown in Table 17 and the 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length integers 0 (False) and 1 (True). In practice, these variable-length integers are encoded as the 8-bit constants `0x00` and `0x01`, respectively.

Enumerations. Protocol Buffers Binary Wire Format represents enumeration constants using 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length Two's Complement [50] signed integers.

Unions. Protocol Buffers relies on unique field identifiers to implement union types called *oneof*. Each alternative in the union type must have a different field identifier and Protocol Buffers enforces that only one of such unique field identifiers is present at a given time. The de-serializer knows the type of the encoded value by comparing its field identifier against the union definition.

Lists. Protocol Buffers Binary Wire Format encodes a list (a *repeated* field) by encoding more than one value with the same unique field identifier. The type definitions corresponding to each element in the list use the *length-delimited* wire type as shown in Table 17, and include the byte-length of the value as a 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integer followed by the value. Lists that consist of scalar values of the same type can be encoded as a single *length-delimited* field, followed by the cummulative byte-length of the elements as a 32-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integer, followed by the elements encoded in order. Protocol Buffers does not natively support heterogeneous or multi-dimensional lists. As a workaround, schema-writers may define lists of structures including unions or lists. Protocol Buffers Binary Wire Format does not deduplicate multiple occurrences of the same element in a list.

[99] `https://developers.google.com/protocol-buffers/docs/proto3#json`

Table 17: The type categories (wire types) that Protocol Buffers supports. The two other wire types that are deprecated and unused at the time of this writing are not considered.

Name	Identifier	Data types
Variable-length integer values	`000`	32-bit and 64-bit Little Endian Base 128 (LEB128) (4.1) variable-length signed and unsigned integers, booleans, enumerations
64-bit values	`001`	64-bit fixed-length signed and unsigned integers, IEEE 764 64-bit floating-point numbers [51]
Length-delimited values	`010`	Strings, bytes, messages, lists
32-bit values	`101`	32-bit fixed-length signed and unsigned integers, IEEE 764 32-bit floating-point numbers [51]

```
syntax = "proto3";
import "google/protobuf/struct.proto";

message NullableString {
  oneof kind {
    google.protobuf.NullValue null = 1;
    string value = 2;
  }
}

message Test {
  repeated string tags = 1;
  sint32 tz = 2;

  enum Day {
    MONDAY = 0; TUESDAY = 1; WEDNESDAY = 2; THURSDAY = 3;
    FRIDAY = 4; SATURDAY = 5; SUNDAY = 6;
  }

  repeated Day days = 3;
  repeated double coord = 4;

  message DataEntry {
    optional NullableString name = 1;
    optional bool staff = 2;
    message Metadata { string info = 1; }
    optional Metadata extra = 3;
  }

  repeated DataEntry data = 5;
}
```

Figure 46: Protocol Buffers schema to serialize the Figure 10 input data.

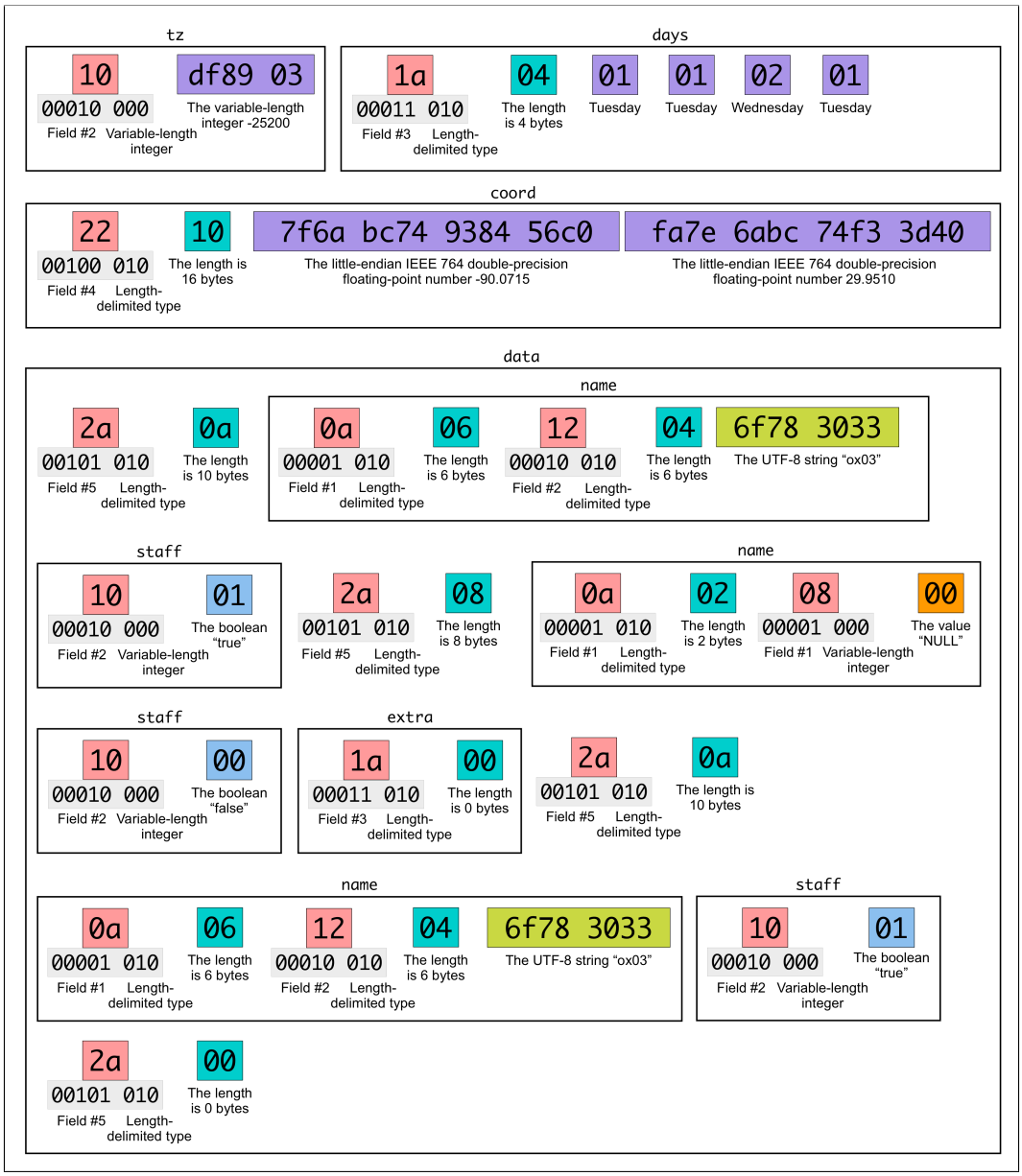

Figure 47: Annotated hexadecimal output of serializing the Figure 10 input data with Protocol Buffers Binary Wire Format.

Table 18: A high-level summary of the Protocol Buffers Binary Wire Format schema-driven serialization specification.

Website	`https://developers.google.com/protocol-buffers/`
Company / Individual	Google
Year	2001
Specification	`https://developers.google.com/protocol-buffers/docs/encoding`
License	3-clause BSD
Schema Language	Proto3
Layout	Sequential with field identifiers
Languages	C++, C#, Dart, Go, Java, Objective-C, Python, Ruby
Types	
Numeric	32-bit and 64-bit Two's Complement [50] Little Endian Base 128 (LEB128) (4.1) variable-length signed integers
	32-bit and 64-bit Little Endian Base 128 (LEB128) (4.1) variable-length unsigned integers
	32-bit and 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (4.1) variable-length signed integers
	Little Endian 32-bit and 64-bit fixed-length unsigned integers
	Little Endian 32-bit and 64-bit fixed-length Two's Complement [50] signed integers
	Little Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	Any, Enum, List, Map, Message, Oneof, Struct
Scalars	Boolean
Other	Bytes (byte array)
	Timestamp [75]
	Duration [99]

5.7 Apache Thrift

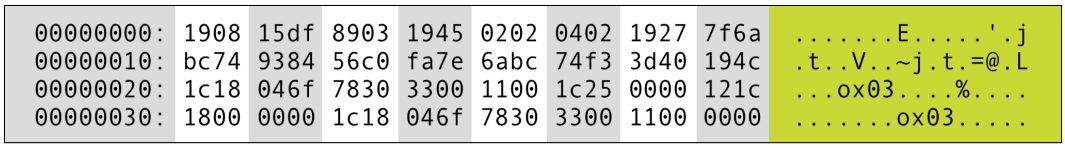

Figure 48: Hexadecimal output (`xxd`) of encoding Figure 10 input data with Apache Thrift Compact Protocol (64 bytes).

History. Apache Thrift [126] is an RPC protocol and schema-driven binary serialization specification developed at Facebook in 2006 and donated to the Apache Software Foundation [100]. Apache Thrift graduated from the Apache Incubator in 2010 and is released under the Apache License 2.0 [101]. Apache Thrift is used in a large number of scalable backend services at Facebook. Apache Thrift is also used as the transmission format of the Carat [96] large-scale research project to collect energy-related analytics from iOS and Android devices which collected 1.5 TB of data as of 2016 [102].

Related Literature. [83] attempts to re-implement Apache Thrift using model-driven engineering technologies such as Xtext [102], Eclipse Modeling Framework (EMF) [103], and Eclipse Epsilon [104] resulting in a significantly more concise implementation in terms of lines of code. The results are published on GitHub [105]. [121] explores automatically generating Apache Thrift service definitions as space and runtime efficient proxies to XML-based [99] SOAP [60] web services. [4] proposes an architecture to compose Apache Thrift services with with other Apache Thrift, SOAP [60], and REST [48] services using the Web Services Business Process Execution Language (BPEL) [130]. [29] performs high-performance large-scale datacenter backups using Apache Thrift on the Apache HBase [106] and Apache Cassandra [107] NoSQL databases. [57] explores the implications of a microservices architecture based on Apache Thrift for a movie renting, streaming, and reviewing system comprised of 33 microservice. [28] proposes an offline and online database system based on Conflict-free Replicated Data Types (CRDT) [123] which uses Apache Thrift as the middleware serialization specification.

Characteristics.

- **Native Type Mappings.** To ease integration, Apache Thrift implementations do not introduce Apache Thrift-specific types or wrapper objects. Instead, the implementations make use of programming language native types.
- **Portability.** Apache Thrift has well-maintained official implementations for a large number of programming languages.

Layout. An Apache Thrift Compact Protocol bit-string is a sequence of values, sometimes nested. Each value is prefixed with a type definition and the length of the value, if applicable. Each value has a numeric identifier that must be unique on the current nesting level. Based on our observations, Apache Thrift Compact Protocol encodes fields even if their values equal their explicitly-set defaults. Apache Thrift Compact Protocol structures are suffixed with the constant byte `0x00`.

Keys. Apache Thrift Compact Protocol type definitions consist in a unique field identifier and a type identifier as shown in Figure 50. Apache Thrift Compact Protocol supports two type definition encodings: the *Short form* and the *Long form* depending on whether the field identifiers are encoded in a relative or absolute manner. Refer to Table 19 for details.

[100] `https://www.apache.org`
[101] `http://www.apache.org/licenses/LICENSE-2.0.html`
[102] `https://www.eclipse.org/Xtext/`
[103] `https://www.eclipse.org/modeling/emf/`
[104] `https://www.eclipse.org/epsilon/`
[105] `https://github.com/SMadani/ThriftMDE/`
[106] `https://hbase.apache.org`
[107] `http://cassandra.apache.org`

Figure 49 illustrates a visual representation of the field identifier delta approach from the *Short form* encoding. Apache Thrift Compact Protocol implementations will choose the *Short form* delta-based encoding unless the unique identifier delta exceeds the value 15 or if the unique identifier is negative. Apache Thrift encourages schema-writer to set explicit unique field identifiers, which must be positive. However, Apache Thrift will automatically assign negative unique field identifiers by default.

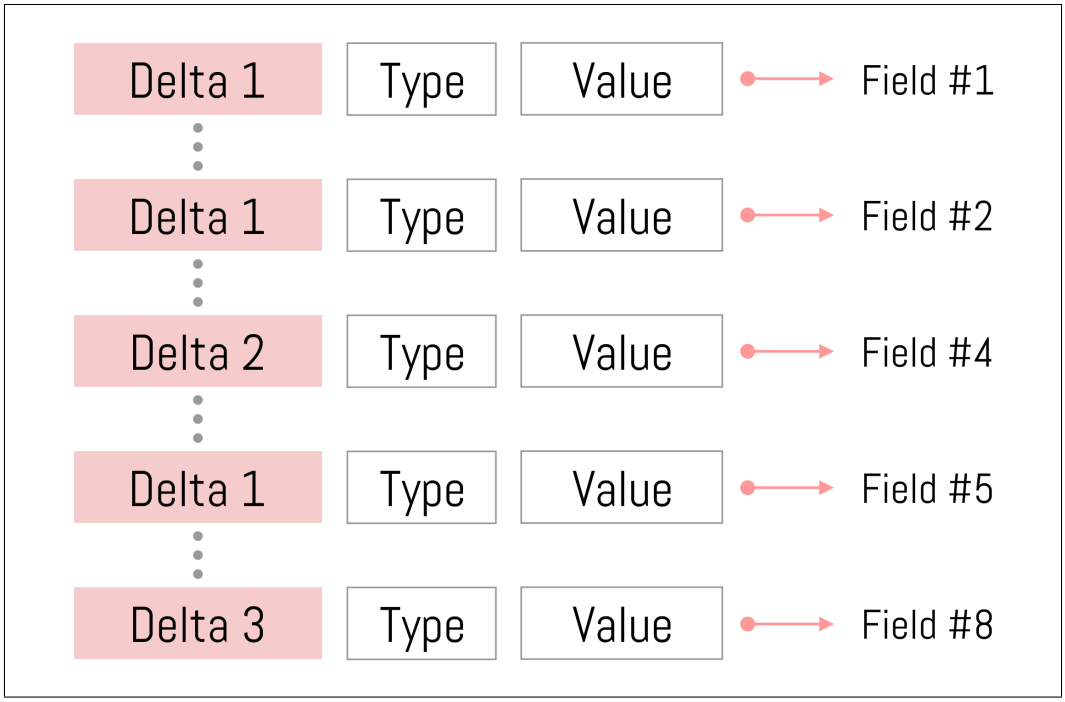

Figure 49: We can think of Apache Thrift fields as triplets consisting of a field delta, a field type, and a field value. Field deltas determine the unique identifier of the current field based on the previous delta values. A field identifier is the sum of all the deltas up to that field.

Numbers. Apache Thrift supports 16-bit, 32-bit, and 64-bit Little Endian Base 128 (LEB128) (4.1) variable-length ZigZag-encoded (4.2) signed integers. Apache Thrift does not support fixed-length integer types. However, Apache Thrift supports a `byte` type that can represent a fixed-length 8-bit

[108] `https://github.com/apache/thrift/blob/05bb55148608b4324a8c91c21cf9a6a0dff282fa/lib/cpp/src/thrift/protocol/TCompactProtocol.tcc`

Table 19: Apache Thrift Compact Protocol supports two structure type definition encodings. The *Long form* encoding represents the field identifier as an absolute integer while the *Short form* encoding represents the field identifier as the difference from the previous field identifier as a space optimization.

Name	Size	First 4-bits	Next 4-bits	Remaining bits
Short form	1 byte	The unique identifier delta as a 4-bit unsigned integer greater than 0	The field type identifier constant as shown in Figure 50	None
Long form	2 to 4 bytes	0000	The field type identifier constant as shown in Figure 50	The unique identifier as a 16-bit Little Endian Base 128 (LEB128) (4.1) variable-length ZigZag-encoded (4.2) signed integer

ZigZag-encoded (4.2) signed integer. In terms of real numbers, Apache Thrift supports Little Endian IEEE 764 64-bit floating-point numbers [51].

Strings. Apache Thrift Compact Protocol produces UTF-8 [32] strings that are not *NUL* delimited. Apache Thrift Compact Protocol strings are prefixed with positive 32-bit Little Endian Base 128 (4.1) variable-length ZigZag-encoded (4.2) signed integers declaring the byte-length of the string. Apache Thrift Compact Protocol encodes empty strings with a zero-length prefix and no additional data. Apache Thrift Compact Protocol does not attempt to deduplicate multiple occurrences of the same string.

Booleans. Apache Thrift Compact Protocol encodes a boolean field and a list of booleans in different manners. If the value is a standalone field, then Apache Thrift Compact Protocol encodes the boolean value at the type definition level using the `CT_BOOLEAN_TRUE` and `CT_BOOLEAN_FALSE` data types shown in Figure 50. If the boolean value is a member of a list, then it is encoded using the 8-bit ZigZag-encoded (4.2) signed integers `0x02` (True) and `0x00` (False).

Enumerations. Apache Thrift Compact Protocol represents enumeration constants using positive 32-bit ZigZag-encoded (4.2) Little Endian Base 128 (4.1) variable-length signed integers.

Unions. Apache Thrift relies on unique field identifiers to implement union types. Each alternative in the union type must have a different field identifier and Apache Thrift enforces that only one of such unique field identifiers is present at a given time. The de-serializer knows the type of the encoded value by comparing its field identifier against the union definition.

Lists. Apache Thrift Compact Protocol encodes a list as the type definition, followed by the list definition, followed by the elements encoded in order. Apache Thrift Compact Protocol specifies two encodings for the list definition depending on the length of the list as shown in Table 20. Apache Thrift Compact Protocol does not attempt to deduplicate multiple occurrences of the same element in a list. Apache Thrift natively supports lists of lists and list of unions as a mechanism to support heterogeneous lists.

[110] `https://github.com/apache/thrift/blob/05bb55148608b4324a8c91c21cf9a6a0dff282fa/lib/cpp/src/thrift/protocol/TCompactProtocol.tcc`

```
enum Types {
  CT_STOP           = 0x00,
  CT_BOOLEAN_TRUE   = 0x01,
  CT_BOOLEAN_FALSE  = 0x02,
  CT_BYTE           = 0x03,
  CT_I16            = 0x04,
  CT_I32            = 0x05,
  CT_I64            = 0x06,
  CT_DOUBLE         = 0x07,
  CT_BINARY         = 0x08,
  CT_LIST           = 0x09,
  CT_SET            = 0x0A,
  CT_MAP            = 0x0B,
  CT_STRUCT         = 0x0C
};
```

Figure 50: Apache Thrift Compact Protocol struct type identifiers definition [108].

Table 20: The two list definition encodings supported by Apache Thrift Compact Protocol depending on the list length as determined by the *From* and *To* ranges. Note that Apache Thrift Compact Protocol cannot encode lists containing more than $2^{32} - 1$ elements.

From	To	Size	First 4-bits	Next 4-bits	Remaining bits
0	14	1 byte	Length of list as a 4-bit unsigned integer	Element type identifier constant as shown in Figure 51	None
15	$2^{32} - 1$	2 to 6 bytes	1111	Element type identifier constant as shown in Figure 51	Length of list as a positive 32-bit Little Endian Base 128 (4.1) variable-length ZigZag-encoded (4.2) signed integer

```
const int8_t TTypeToCType[16] = {
  CT_STOP, // T_STOP
  0, // unused
  CT_BOOLEAN_TRUE, // T_BOOL
  CT_BYTE, // T_BYTE
  CT_DOUBLE, // T_DOUBLE
  0, // unused
  CT_I16, // T_I16
  0, // unused
  CT_I32, // T_I32
  0, // unused
  CT_I64, // T_I64
  CT_BINARY, // T_STRING
  CT_STRUCT, // T_STRUCT
  CT_MAP, // T_MAP
  CT_SET, // T_SET
  CT_LIST, // T_LIST
};
```

Figure 51: Apache Thrift Compact Protocol list type identifiers definition [110]. The indexes represent the type identifier constants. For example, the `CT_DOUBLE` type is encoded as `0x04` because it is the fourth element of the `TTypeToCType` array.

```
enum Day {
  MONDAY = 0, TUESDAY = 1, WEDNESDAY = 2, THURSDAY = 3,
  FRIDAY = 4, SATURDAY = 5, SUNDAY = 6
}

enum Null { NULL = 0 }
struct Metadata { 1: string info; }
union NullableString { 1: string value, 2: Null null }

struct DataEntry {
  1: optional NullableString name,
  2: optional bool staff,
  3: optional Metadata extra
}

struct Test {
  1: required list<string> tags,
  2: required i32 tz,
  3: required list<Day> days,
  4: required list<double> coord,
  5: required list<DataEntry> data
}
```

Figure 52: Apache Thrift schema to serialize the Figure 10 input data.

Table 21: A high-level summary of the Apache Thrift Compact Protocol schema-driven serialization specification.

Website	`https://thrift.apache.org`
Company / Individual	Apache Software Foundation
Year	2006
Specification	`https://github.com/apache/thrift/tree/master/doc/specs`
License	Apache License 2.0
Schema Language	Thrift IDL
Layout	Sequential with field identifiers
Languages	ActionScript, C, C++, C#, Common LISP, D, Dart, Erlang, Haskell, Haxe, Go, Java, JavaScript, Lua, OCaml, Perl, PHP, Python, Ruby, Rust, Smalltalk, Swift
Types	
Numeric	16-bit, 32-bit, and 64-bit ZigZag-encoded (4.2) Little Endian Base 128 (4.1) variable-length signed integers Little Endian 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	List, Map, Set, Struct, Union
Scalars	Boolean
Other	Binary (byte array) Byte

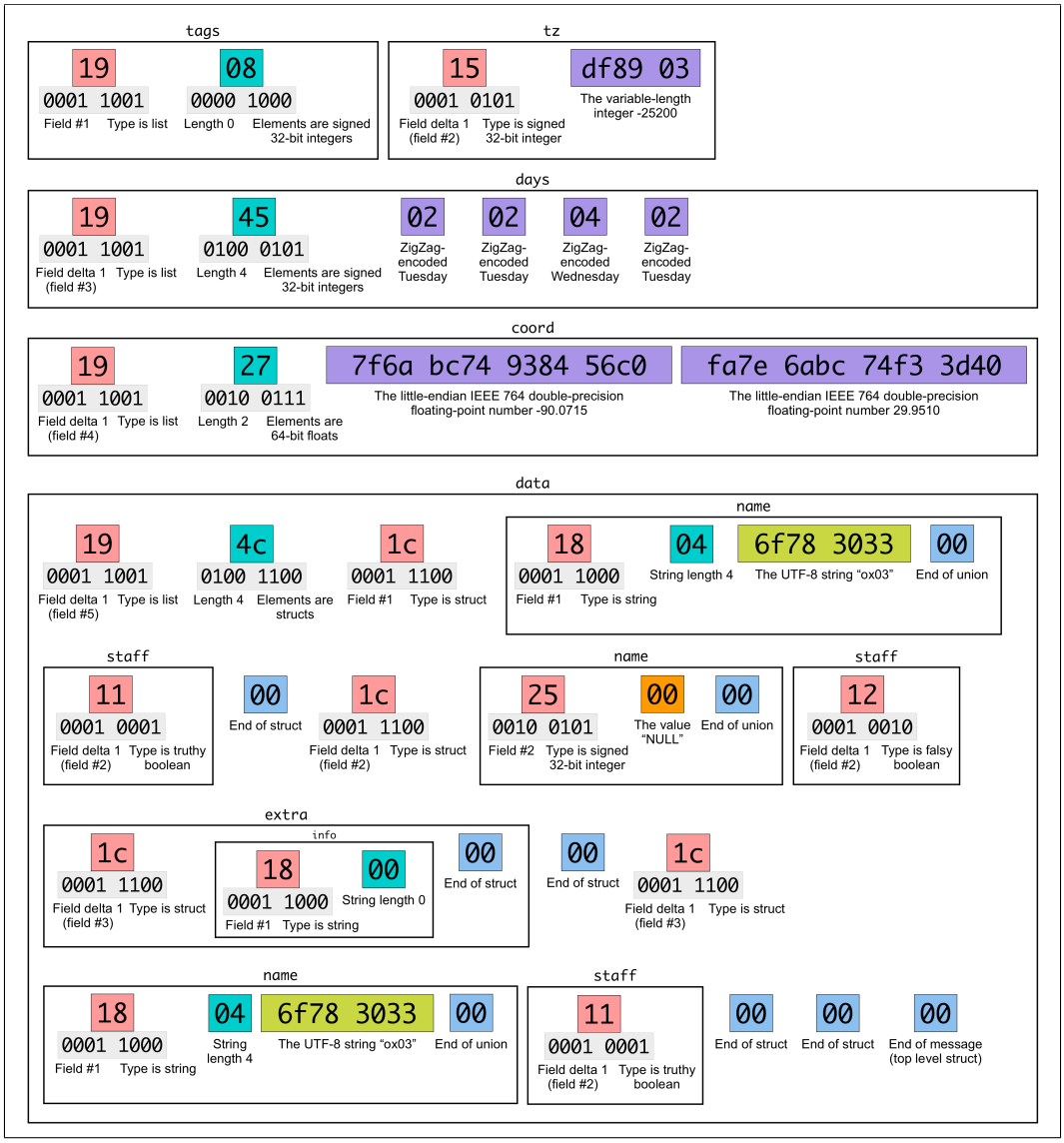

Figure 53: Annotated hexadecimal output of serializing the Figure 10 input data with Apache Thrift Compact Protocol.

6 Schema-less Specifications

In this section, we discuss the history and characteristics of JSON-compatible schema-less binary serialization specifications: BSON [86], CBOR [14], FlexBuffers [140], MessagePack [56], Smile [116] and UBJSON [17].

6.1 BSON

```
00000000: df00 0000 0474 6167 7300 0500 0000 0010  .....tags.......
00000010: 747a 0090 9dff ff04 6461 7973 0021 0000  tz......days.!..
00000020: 0010 3000 0100 0000 1031 0001 0000 0010  ..0......1......
00000030: 3200 0200 0000 1033 0001 0000 0000 0463  2......3.......c
00000040: 6f6f 7264 001b 0000 0001 3000 7f6a bc74  oord......0..j.t
00000050: 9384 56c0 0131 00fa 7e6a bc74 f33d 4000  ..V..1..~j.t.=@.
00000060: 0464 6174 6100 7800 0000 0330 001c 0000  .data.x....0....
00000070: 0002 6e61 6d65 0005 0000 006f 7830 3300  ..name.....ox03.
00000080: 0873 7461 6666 0001 0003 3100 2a00 0000  .staff....1.*...
00000090: 0a6e 616d 6500 0873 7461 6666 0000 0365  .name..staff...e
000000a0: 7874 7261 0010 0000 0002 696e 666f 0001  xtra......info..
000000b0: 0000 0000 0000 0332 001c 0000 0002 6e61  .......2......na
000000c0: 6d65 0005 0000 006f 7830 3300 0873 7461  me.....ox03..sta
000000d0: 6666 0001 0003 3300 0500 0000 0000 00    ff....3........
```

Figure 54: Hexadecimal output (`xxd`) of encoding Figure 10 input data with BSON (223 bytes).

History. BSON [86] is a schema-less binary serialization specification created by MongoDB, Inc [111] in 2009, which was called *10gen* at that time [112]. BSON is a superset of JSON [18] as it includes additional data types relevant to the MongoDB NoSQL database. BSON is a core component of the MongoDB NoSQL database storage layer and drivers. The BSON project was started by Mike Dirolf [113], a Software Engineer at 10gen. The BSON specification is released to the public domain under the Creative Commons 1.0 license [114]. There is also a minimal BSON C third-party implementation targetted at embedded devices called *BINSON* [115].

Characteristics.

- **Traversal Runtime-performance.** The MongoDB NoSQL database stores documents using BSON. Database data access patterns typically involve searching for fields and values within documents and [148] shows that BSON documents can be traversed at least four times faster than JSON [41]. However, in the context of Big Data, [129] note that BSON collections are often larger than their JSON [41] and CSV [122] counterparts due to the additional metadata included to support fast traversals.
- **In-place Updates.** BSON has been designed to support updates that do not involve re-serializing the entire document.

Layout. A BSON bit-string (a document) is a sequence of key-value pairs prefixed with the byte-size of the rest of the document as a positive Little Endian signed Two's Complement [50] 32-bit integer. Each key-value pair consists of a 1-byte type definition, followed by the property name as a *NUL*-delimited UTF-8 string [32], followed by the value. BSON documents are suffixed with the `0x00` byte.

Types. BSON declares 29 1-byte type definitions as shown in Figure 55. Many of these type definitions correspond to MongoDB-specific extensions to JSON [18] such as the `DBPointer` and

[111] `https://www.mongodb.com`
[112] `https://www.mongodb.com/press/10gen-announces-company-name-change-mongodb-inc`
[113] `https://github.com/mdirolf`
[114] `https://creativecommons.org/publicdomain/zero/1.0/`
[115] `https://github.com/alialavia/binson`

`ObjectId` (OID) [116] data types. BSON also supports binary values annotated with a sub-type such as UUID [80] and MD5 [112].

```
/**
 * bson_type_t:
 *
 * This enumeration contains all of the possible types within a BSON document.
 * Use bson_iter_type() to fetch the type of a field while iterating over it.
 */
typedef enum {
   BSON_TYPE_EOD = 0x00,
   BSON_TYPE_DOUBLE = 0x01,
   BSON_TYPE_UTF8 = 0x02,
   BSON_TYPE_DOCUMENT = 0x03,
   BSON_TYPE_ARRAY = 0x04,
   BSON_TYPE_BINARY = 0x05,
   BSON_TYPE_UNDEFINED = 0x06,
   BSON_TYPE_OID = 0x07,
   BSON_TYPE_BOOL = 0x08,
   BSON_TYPE_DATE_TIME = 0x09,
   BSON_TYPE_NULL = 0x0A,
   BSON_TYPE_REGEX = 0x0B,
   BSON_TYPE_DBPOINTER = 0x0C,
   BSON_TYPE_CODE = 0x0D,
   BSON_TYPE_SYMBOL = 0x0E,
   BSON_TYPE_CODEWSCOPE = 0x0F,
   BSON_TYPE_INT32 = 0x10,
   BSON_TYPE_TIMESTAMP = 0x11,
   BSON_TYPE_INT64 = 0x12,
   BSON_TYPE_DECIMAL128 = 0x13,
   BSON_TYPE_MAXKEY = 0x7F,
   BSON_TYPE_MINKEY = 0xFF,
} bson_type_t;

/**
 * bson_subtype_t:
 *
 * This enumeration contains the various subtypes that may be used in a binary
 * field. See http://bsonspec.org for more information.
 */
typedef enum {
   BSON_SUBTYPE_BINARY = 0x00,
   BSON_SUBTYPE_FUNCTION = 0x01,
   BSON_SUBTYPE_BINARY_DEPRECATED = 0x02,
   BSON_SUBTYPE_UUID_DEPRECATED = 0x03,
   BSON_SUBTYPE_UUID = 0x04,
   BSON_SUBTYPE_MD5 = 0x05,
   BSON_SUBTYPE_USER = 0x80,
} bson_subtype_t;
```

Figure 55: BSON type definitions [117].

Numbers. BSON supports Little Endian 64-bit and 128-bit IEEE 764 floating-point numbers [51]. In terms of integers, BSON supports Little Endian 32-bit and 64-bit signed Two's Complement [50] integers and Little Endian 64-bit unsigned integers. BSON implementations pick the smallest data type that can encode the given number.

Strings. BSON produces *NUL*-delimited UTF-8 [32] strings for property names and for string values. In comparison to property names, string values are prefixed with the byte-length of the string as a positive Little Endian signed Two's Complement [50] 32-bit integer. BSON does not attempt to deduplicate multiple occurrences of the same property name or string value.

[116] `https://docs.mongodb.com/manual/reference/method/ObjectId/`

[117] `https://github.com/mongodb/libbson/blob/ffc8d983ecf6b46d5404f5cc20e756a85dfcbfd2/src/bson/bson-types.h`

Booleans. Booleans values are encoded with the 1-byte type tag `0x08` as shown in Figure 55 followed by the byte constants `0x00` (False) or `0x01` (True).

Lists. BSON treats arrays as JSON objects [18] with stringified integral keys. For example, the JSON document `{ "foo": [ true, false ] }` is encoded as `{ "foo": { "1": true, "2": false } }`. BSON can distinguish an array from a user-supplied JSON object with stringified integral keys by their different 1-byte type definition prefixes. BSON does not attempt to deduplicated multiple occurrences of the same value in an array.

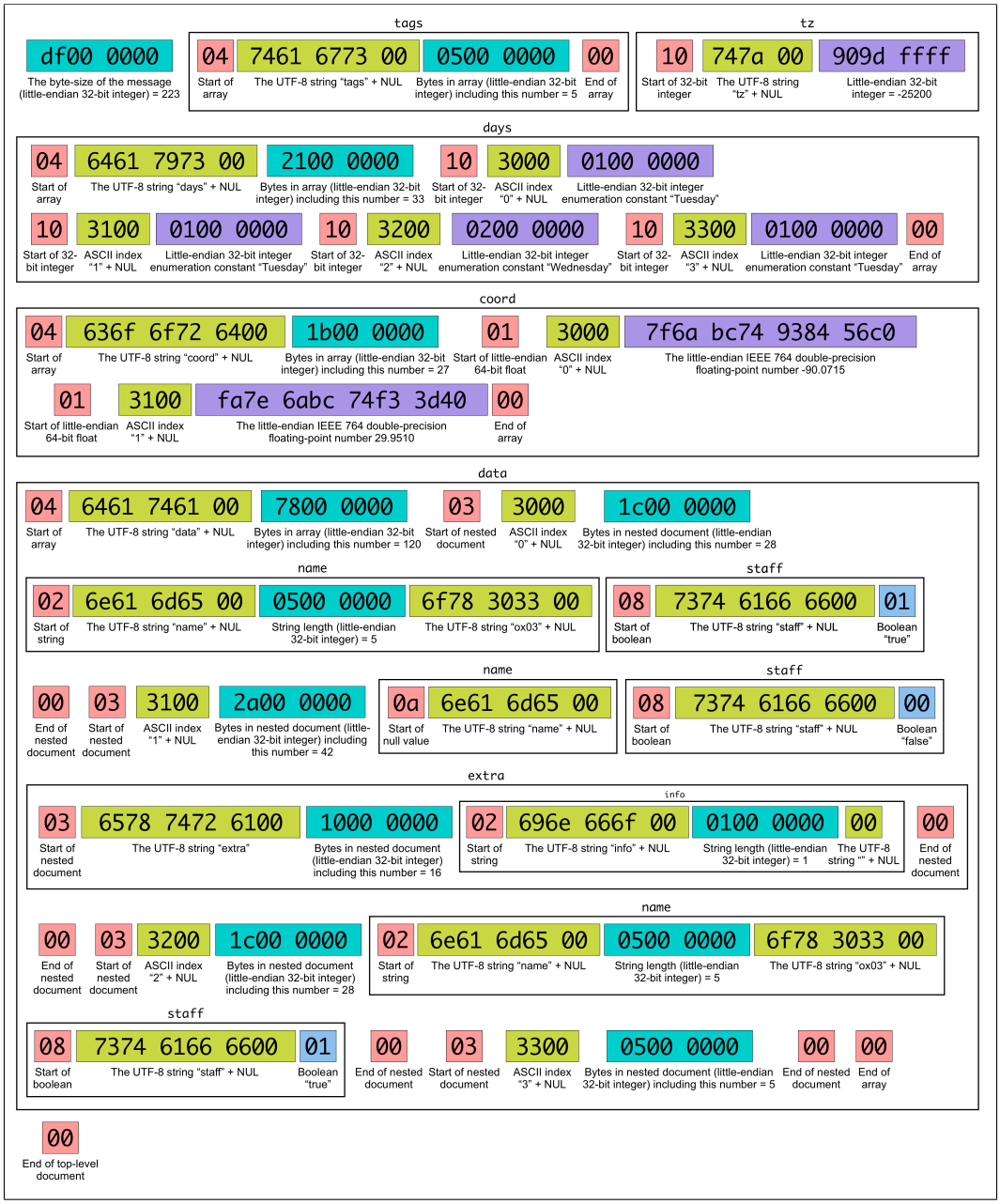

Figure 56: Annotated hexadecimal output of serializing the Figure 10 input data with BSON.

[118] `http://dochub.mongodb.org/core/objectids`

Table 22: A high-level summary of the BSON schema-less serialization specification.

Website	`http://bsonspec.org`
Company / Individual	MongoDB
Year	2010
Specification	`http://bsonspec.org/spec.html`
License	Creative Commons 1.0
Layout	Sequential
Languages	C, C++, C#, Go, Java, Node.js, Perl, PHP, Python, Ruby, Scala
Types	
Numeric	Little Endian Two's Complement [50] signed 32-bit and 64-bit integers
	Little Endian unsigned 64-bit integers
	Little Endian 64-bit and 128-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	Array, Document (object)
Scalars	Boolean, Null, Undefined
Other	Binary data (byte array)
	ObjectId [118]
	Epoch [89]
	DBPointer
	JavaScript code
	Function
	UUID [80]
	MD5 [112]
	Symbol
	MongoDB Timestamp
	Regular expression

6.2 CBOR

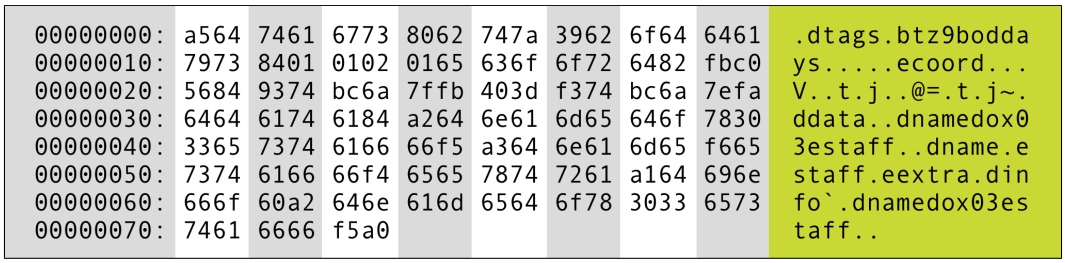

Figure 57: Hexadecimal output (`xxd`) of encoding Figure 10 input data with CBOR (118 bytes).

History. Concise Binary Object Representation (CBOR) [14] is a schema-less binary serialization specification first published by Carsten Bormann [119] and Paul Hoffman [120] as an Internet Engineering Task Force (IETF) [121] document in 2013. CBOR has been primarily designed for the Internet of Things. In fact, CBOR is the recommended serialization layer for the CoAP [124] Internet of Things transfer protocol. CBOR is natively supported as part of the RIOT operating system for the Internet of Things [13]. Outside of IoT, CBOR is also used as the serialization specification behind the *piChain* fault-tolerant distributed state machine [90]. [113] provides an alternative introduction to the CBOR specification and describes how to translate XML documents [99] into CBOR.

Characteristics.

- **Resource Efficient.** CBOR has been designed to produce implementations to run on memory and processor constrained devices [122].
- **Standardization.** In comparison to informally documented serialization specifications, CBOR is specified as an IETF RFC document [14] and has gone through extensive technical review as a result.

Layout. A CBOR bit-string is a sequence of key-value pairs, sometimes nested. Each CBOR key or value starts with a type definition. A CBOR key-value pair is a concatenation of the key followed by the value and a CBOR map is a concatenation of key-value pairs.

Types. CBOR groups the types it supports into 8 *major types* as shown in Figure 58. A CBOR type definition consists of 1-byte whose most-significant 3-bits encode the major type. The remaining 5-bits encode type-specific information as shown in Table 23.

Numbers. CBOR supports Big Endian IEEE 764 16-bit, 32-bit, and 64-bit floating-point numbers [51]. Floating-point numbers are defined with the **major type 7** as shown in Table 23 and with the floating-point precision as a sub-type. CBOR makes a distinction between positive integers (**major type 0**) and negative integers (**major type 1**) at the type level. In terms of positive integers, CBOR supports Big Endian 5-bit, 8-bit, 16-bit, 32-bit and 64-bit unsigned integers. Negative integers do not use Two's Complement [50] nor ZigZag encoding (4.2). Instead, CBOR encodes the negative number as a Big Endian 5-bit, 8-bit, 16-bit, 32-bit, or 64-bit unsigned integer where the final value is equal to minus one minus the unsigned integer. For example, the negative integer $-\mathbf{25200}$ is encoded as $\mathbf{25199}$ given that $-\mathbf{1} - \mathbf{25199} = -\mathbf{25200}$.

Additionally, CBOR supports arbitrary-length positive and negative integers and arbitrary-precision decimal numbers. Arbitrary-length integers are encoded as unsigned integers represented as potentially-indefinite byte strings prefixed with a **major type 6** as shown in Table 23 and a positive or negative sub-type annotation. Negative arbitrary-length integers are encoded similarly to fixed-length negative numbers: as unsigned integers where the value is equal to minus one minus

[119] `https://www.linkedin.com/in/cabo/`

[120] `https://datatracker.ietf.org/person/paul.hoffman@vpnc.org`

[121] `https://www.ietf.org`

[122] `https://tools.ietf.org/html/rfc7049#section-1.1`

[123] `https://github.com/intel/tinycbor/blob/fc42a049853b802e45f49588f8148fc29d7b4d9c/src/cborinternal_p.h`

the unsigned integer. Arbitrary-precision decimal numbers are encoded as arrays of exactly two integer elements: the scale as a fixed-length positive or negative integer, and the mantissa as a fixed-length or arbitrary-length positive or negative integer. The resulting decimal number is equal to **mantissa** $\times$ $\mathbf{10}^{\mathbf{scale}}$.

Strings. CBOR property names and string values are encoded using UTF-8 [32] without *NUL* delimiters. CBOR can encode fixed-length strings where the type definition prefix includes the byte-length of the string as a Big Endian 5-bit, 16-bit, 32-bit, or 64-bit unsigned integer. Additionally, CBOR can encode arbitrary-length strings by splitting the input string into a sequence of fixed-length string values suffixed with the `0xff` stop code. CBOR does not attempt to deduplicate multiple occurrences of the same property name or string value.

Booleans CBOR encodes boolean values at the type level through a type definition byte that consists of the **major type 7** as shown in Table 23 and the Big Endian 5-bit unsigned integer sub-types 20 (False) or 21 (True). For example, the boolean value *True* is encoded as `0xf5 = 111 (major type 7) 10101 = 21`.

Lists. CBOR supports fixed-length and arbitrary-length arrays both constisting of a **major type 4** as shown in Table 23 type definition and the array elements encoded in order. Fixed-length arrays use the least-significant 5-bits of the type definition to store the logical length of the array as a Big

Table 23: A definition of CBOR major types as specified in [14].

Major type	Description	Remaining 5-bits
0	Unsigned integers	The integer itself or the byte-length of the integer to follow
1	Negative integers	The integer itself or the byte-length of the integer to follow
2	Byte strings	The length of the byte-string to follow
3	Text strings	The length of the string to follow
4	Arrays	The logical length of the array or the byte-length of the integer representing the logical length of the array following the type definition
5	Maps	The number of keys in the map or the byte-length of the integer representing the number of keys in the map following the type definition
6	Optional metadata	Additional semantic information
7	Floats and types without content	The sub-type followed by the content if the value is a floating-point number

```c
/*
 * CBOR Major types
 * Encoded in the high 3 bits of the descriptor byte
 * See http://tools.ietf.org/html/rfc7049#section-2.1
 */
typedef enum CborMajorTypes {
    UnsignedIntegerType = 0U,
    NegativeIntegerType = 1U,
    ByteStringType = 2U,
    TextStringType = 3U,
    ArrayType = 4U,
    MapType = 5U,           /* a.k.a. object */
    TagType = 6U,
    SimpleTypesType = 7U
} CborMajorTypes;
```

Figure 58: CBOR major types definition [123].

Endian 5-bit unsigned integer or to store the byte-length of the logical length of the array following the type definition represented as a Big Endian unsigned 16-bit, 32-bit, or 64-bit integer.

In comparison to fixed-length arrays, arbitrary-length arrays do not encode the logical length of the array. Instead, all the least-significant 5-bits of the type definition are set and the array is suffixed with the `0xff` stop code. CBOR does not attempt to deduplicate multiple occurrences of the same scalar or composite value in an array.

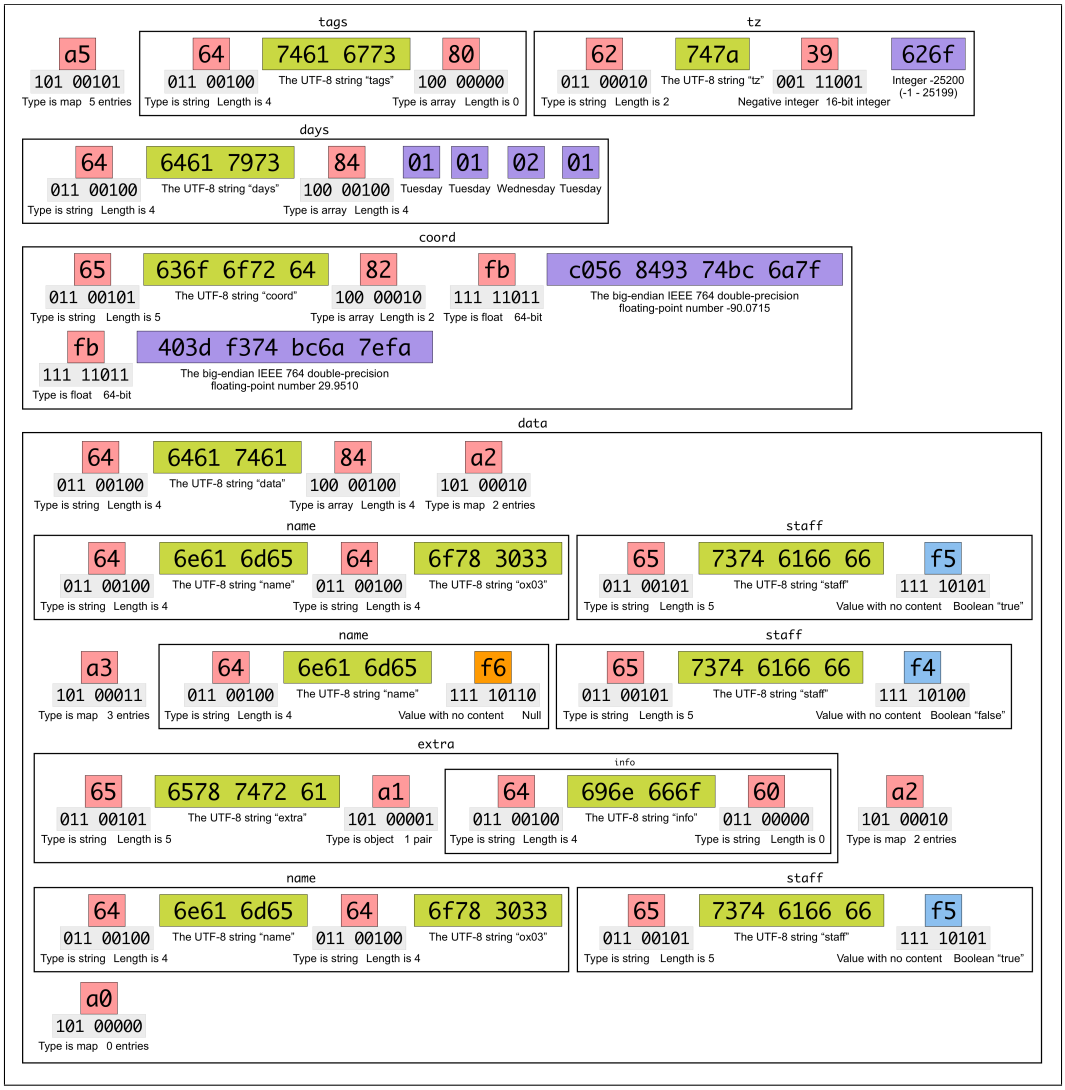

Figure 59: Annotated hexadecimal output of serializing the Figure 10 input data with CBOR.

Table 24: A high-level summary of the CBOR schema-less serialization specification.

Website	`http://cbor.io`
Company / Individual	Carsten Bormann and Paul Hoffman
Year	2013
Specification	RFC-7049 [14]
License	Implementation-dependent
Layout	Sequential
Languages	C, C++, C#, Clojure, Crystal, D, Dart, Elixir, Erlang, Go, Haskell, Java, JavaScript, Julia, Lua, OCaml, Perl, PHP, Python, Ruby, Rust, Scala, Scala, Swift
Types	
Numeric	Big Endian 5-bit, 8-bit, 16-bit, 32-bit, and 64-big unsigned integers
	Big Endian 5-bit, 8-bit, 16-bit, 32-bit, and 64-big negative integers (encoded as -1 minus the value)
	Big Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51]
	Big Endian arbitrary-length positive and negative integers
	Arbitrary-precision signed decimals (mantissa and scale-based)
String	UTF-8 [32]
Composite	Array, Map
Scalars	Boolean, Null, Undefined
Other	Byte string
	Datetime [75]
	Epoch [89]
	Base64 [73]
	Regular expression
	MIME message [53]

6.3 FlexBuffers

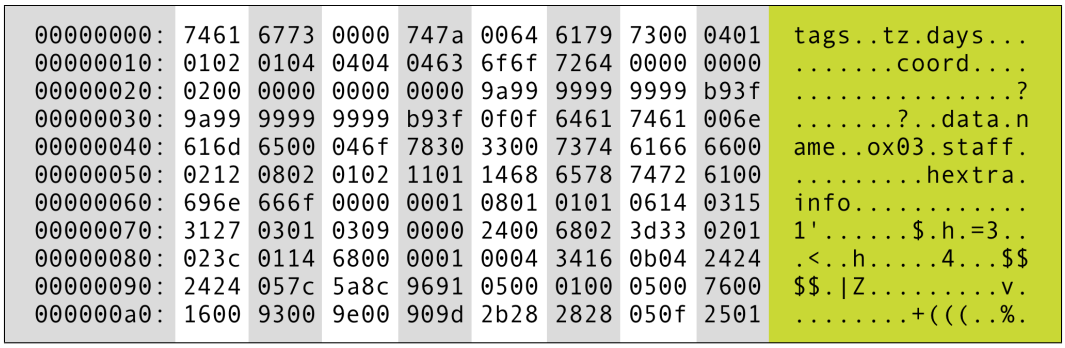

Figure 60: Hexadecimal output (`xxd`) of encoding Figure 10 input data with FlexBuffers (176 bytes).

History. FlexBuffers [140] is the official schema-less variant of the FlatBuffers [139] schema-driven binary serialization specification developed by Google. The FlexBuffers serialization specification was developed in 2016 by Wouter van Oortmerssen [124], the main author of FlatBuffers. FlexBuffers has the same use cases as FlatBuffers: memory-efficient serialization and deserialization mainly in the context of games and mobile. FlexBuffers is also released under the Apache License 2.0 [125].

Characteristics.

- **Efficient Reads.** FlexBuffers also produces implementations that perform runtime-efficiency and memory-efficient incremental and random-access reads.
- **Embeddable.** FlexBuffers can be used in conjunction with FlatBuffers [139] by storing a part of a FlatBuffers bit-string in the FlexBuffers specification.

Layout. A FlexBuffers bit-string consist of a tree hierarchy of pointers that eventually point at scalar types. The FlexBuffers core data structure is the *map*. A FlexBuffers map consist of a values vector and a keys vector as shown in Figure 61 and as follows:

- **Values Vector.** The values vector starts with a pointer to its corresponding keys vector, the byte-length of each key in the keys vector as a Little Endian unsigned integer, and the logical size of the values vector as a Little Endian unsigned integer. The values vector then sequentially encodes its elements followed by a sequence of 1-byte type definitions that correspond to each value in the map.
- **Keys Vector.** The keys vector consists of the logical size of the vector as a Little Endian unsigned integer followed by the sequence of keys, which are typically pointers to string values. Individual keys may be shared and multiple values vectors can point to the same keys vector if they share the same structure resulting in efficient use of space.

Every vector element is aligned to the largest element that the vector contains. The parent structure that points to the vector encodes the size of the elements. Scalar values are encoded at the beginning of the bit-string while map definitions are encoded at the end of the bit-string. FlexBuffers bit-strings end with a footer vector that consists of a pointer to the root element, the type definition of the root element, and the byte-length of each element in the footer vector as a Little Endian 8-bit unsigned integer.

Types. FlexBuffers uses 1-byte type definitions. The most-significant 6-bits encode the data type as an unsigned integer as shown in Figure 62. The least-significant 2-bits encode the bit-width of the data type as shown in Figure 63.

[124] `https://github.com/aardappel`

[125] `http://www.apache.org/licenses/LICENSE-2.0.html`

[126] `https://github.com/google/flatbuffers/blob/8778dc7c2bc20b3165a86d62e2e499d2b86912f0/include/flatbuffers/flexbuffers.h`

[126] `https://github.com/google/flatbuffers/blob/8778dc7c2bc20b3165a86d62e2e499d2b86912f0/include/flatbuffers/flexbuffers.h`

Numbers. FlexBuffers supports Little Endian 8-bit, 16-bit, 32-bit, and 64-bit unsigned and signed integers. Signed integers use Two's Complement [50]. FlexBuffers defines four bit-widths for all its types as shown in Figure 63. Therefore, FlexBuffers theoretically supports Little Endian 8-bit, 16-bit, 32-bit, and 64-bit IEEE 764 [51] floating-point numbers. While [51] does not define 8-bit floating-point numbers, these type of reduced-precision floats are useful in the context of artificial intelligence [145].

Strings. FlexBuffers produces *NUL*-delimited UTF-8 [32] strings for property names and for string values. In comparison to property names, string values are prefixed with the byte-length of the string as an 8-bit unsigned integer. By default, FlexBuffers does not attempt to deduplicate multiple occurrences of the same string. However, its serialization interface allows the application to track and share duplicated string values.

Booleans. FlexBuffers encodes booleans as the Little Endian unsigned integers 0 (False) and 1 (True). A type definition byte that describes a boolean value consists of the type 26 (`FBT_BOOL`) as shown in Figure 62. The bit-width declared in the least-significant 2-bits of the type definition byte defines the width of the unsigned integer that represents the boolean value depending on the alignment of the vector that includes the boolean value. For example, a truthy boolean that is part of a 16-bit elements vector is encoded as `0x00 0x01` with the type definition byte `0x69 = 011010 (FBT_BOOL) 01 (16-bits)`.

Lists. A FlexBuffers list (vector) consists of the concatenation of its elements and a variable amount of surrounding metadata. FlexBuffers supports arbitrary-length untyped vectors, arbitrary-length typed vectors of certain scalar types, and fixed-length (of 2, 3, or 4 elements) typed vectors of certain scalar types as shown in Figure 64.

- **Arbitrary-length Untyped Vectors.** These type of vectors are prefixed with their logical size as a Little Endian unsigned integer aligned to the size of the largest element of the vector. The concatenation of elements is suffixed with a sequence of 1-byte type definitions corresponding to each of the vector elements. The parent element pointing at the vector declares the generic type `FBT_VECTOR` as shown in Figure 62.

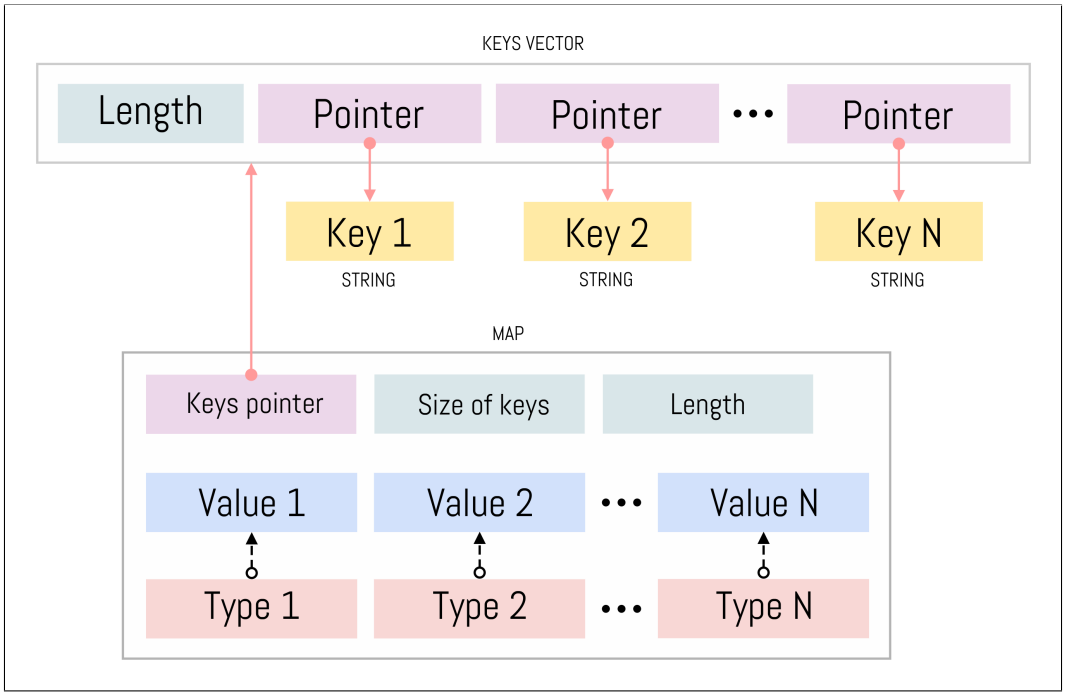

Figure 61: FlexBuffers maps consist of values vectors that include N number of elements followed by N type definitions. These vectors point to keys vectors consisting of N elements that typically consist of pointers to strings.

- **Arbitrary-length Typed Vectors.** As the *Arbitrary-length untyped vectors*, these type of vectors are prefixed with their logical size as a Little Endian unsigned integer aligned to the size of the largest element of the vector. However, the concatenation of elements is not suffixed with a list of type definitions. Instead, the parent element pointing at the vector declares one of the available typed vector types such as `FBT_VECTOR_INT` or `FBT_VECTOR_FLOAT` as shown in Figure 62.
- **Fixed-length Typed Vectors.** These type of vectors do not include the logical size unsigned integer nor a list of type definitions. Instead, the parent element pointing at the vector declares one of the available fixed-length typed vector types such as `FBT_VECTOR_UINT4` or `FBT_VECTOR_FLOAT3` as shown in Figure 62.

The elements of a vector are aligned to the largest element and the parent element pointing at the vector declares the byte-length of the elements. FlexBuffers encodes empty vectors with the 8-bit unsigned integer vector length 0 and no additional information. FlexBuffers does not attempt to deduplicate multiple occurences of the same element in a vector but a vector may include more than one pointer to the same value.

```
// These are used as the upper 6 bits of a type field to indicate the actual type.
enum Type {
  FBT_NULL = 0,
  FBT_INT = 1,
  FBT_UINT = 2,
  FBT_FLOAT = 3,
  // Types above stored inline, types below store an offset.
  FBT_KEY = 4,
  FBT_STRING = 5,
  FBT_INDIRECT_INT = 6,
  FBT_INDIRECT_UINT = 7,
  FBT_INDIRECT_FLOAT = 8,
  FBT_MAP = 9,
  FBT_VECTOR = 10,       // Untyped.
  FBT_VECTOR_INT = 11,  // Typed any size (stores no type table).
  FBT_VECTOR_UINT = 12,
  FBT_VECTOR_FLOAT = 13,
  FBT_VECTOR_KEY = 14,
  // DEPRECATED, use FBT_VECTOR or FBT_VECTOR_KEY instead.
  // Read test.cpp/FlexBuffersDeprecatedTest() for details on why.
  FBT_VECTOR_STRING_DEPRECATED = 15,
  FBT_VECTOR_INT2 = 16,  // Typed tuple (no type table, no size field).
  FBT_VECTOR_UINT2 = 17,
  FBT_VECTOR_FLOAT2 = 18,
  FBT_VECTOR_INT3 = 19,  // Typed triple (no type table, no size field).
  FBT_VECTOR_UINT3 = 20,
  FBT_VECTOR_FLOAT3 = 21,
  FBT_VECTOR_INT4 = 22,  // Typed quad (no type table, no size field).
  FBT_VECTOR_UINT4 = 23,
  FBT_VECTOR_FLOAT4 = 24,
  FBT_BLOB = 25,
  FBT_BOOL = 26,
  FBT_VECTOR_BOOL = 36,  // To Allow the same type of conversion of type to vector type
};
```

Figure 62: FlexBuffers type identifiers definition [126].

```
// These are used in the lower 2 bits of a type field to determine the size of
// the elements (and or size field) of the item pointed to (e.g. vector).
enum BitWidth {
  BIT_WIDTH_8 = 0,
  BIT_WIDTH_16 = 1,
  BIT_WIDTH_32 = 2,
  BIT_WIDTH_64 = 3,
};
```

Figure 63: FlexBuffers bit-width definition [126].

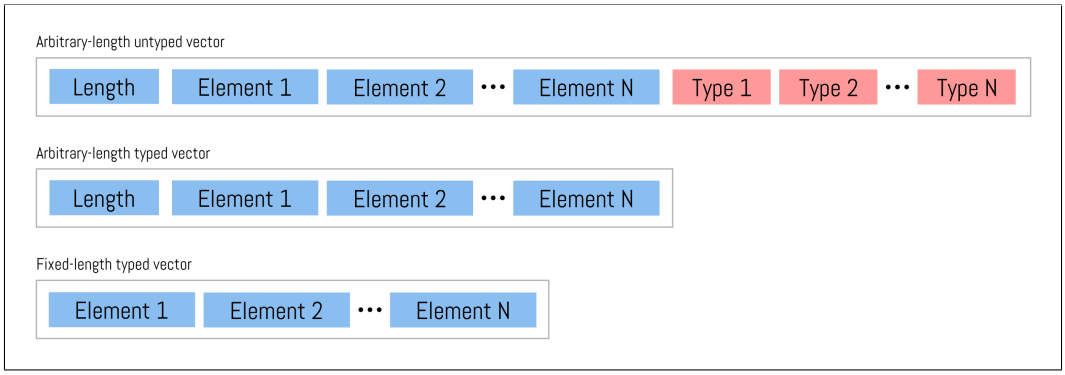

Figure 64: A visual representation of arbitrary-length untyped vectors (top), arbitrary-length typed vectors (middle), and fixed-length typed vectors (bottom).

Table 25: A high-level summary of the FlexBuffers schema-less serialization specification.

Website	`https://google.github.io/flatbuffers/flexbuffers.html`
Company / Individual	Google
Year	2016
Specification	`https://google.github.io/flatbuffers/flatbuffers_internals.html`
License	Apache License 2.0
Layout	Pointer-based
Languages	C++, Java
Types	
Numeric	Little Endian 8-bit, 16-bit, 32-bit, and 64-bit Two's Complement [50] signed integers Little Endian 8-bit, 16-bit, 32-bit, and 64-bit unsigned integers Little Endian 16-bit, 32-bit, and 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	Vector, Map
Scalars	Boolean, Null
Other	Blob (byte array)

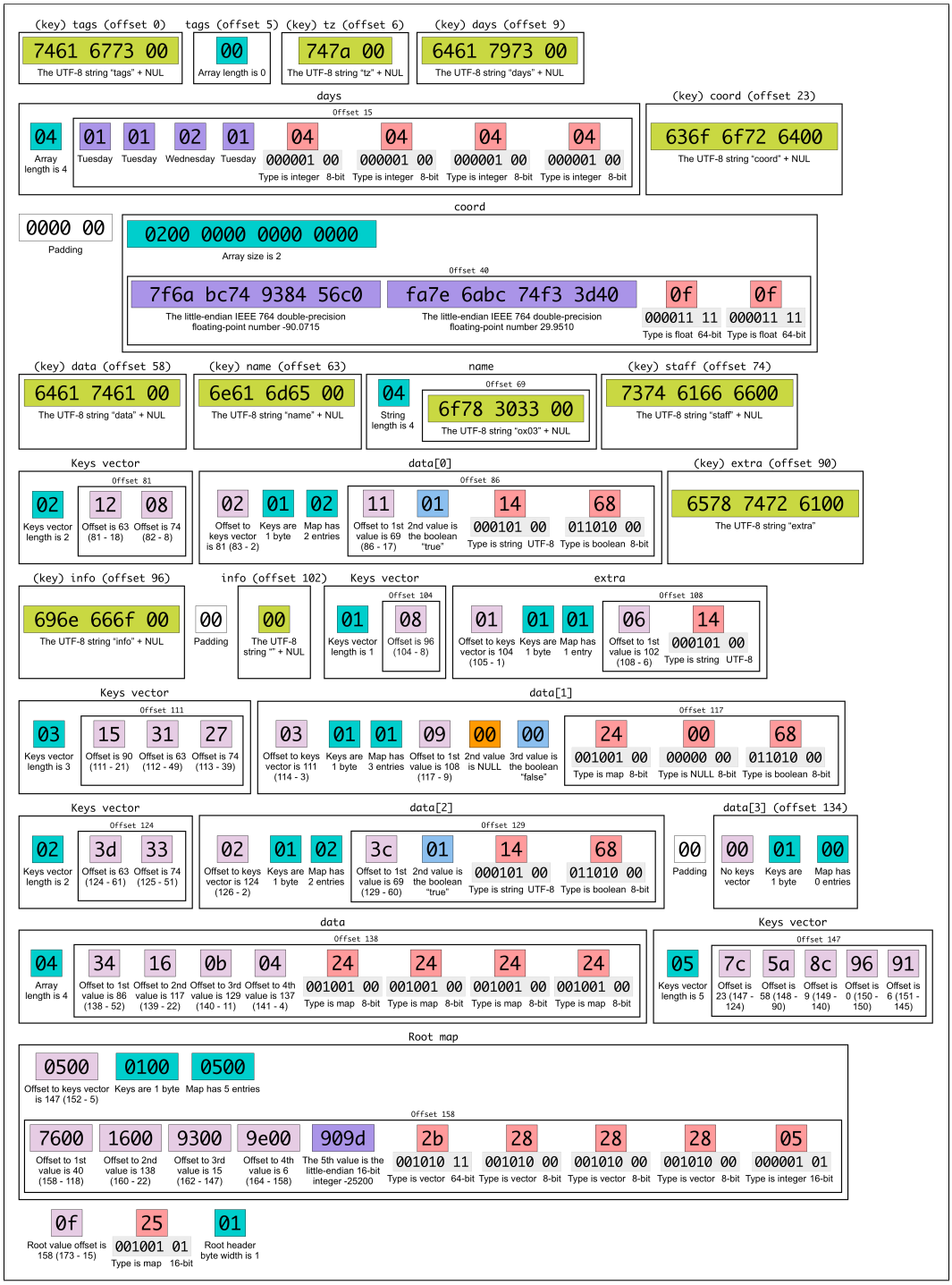

Figure 65: Annotated hexadecimal output of serializing the Figure 10 input data with FlexBuffers.

6.4 MessagePack

```
00000000: 85a4 7461 6773 90a2 747a d19d 90a4 6461  ..tags..tz....da
00000010: 7973 9401 0102 01a5 636f 6f72 6492 cbc0  ys......coord...
00000020: 5684 9374 bc6a 7fcb 403d f374 bc6a 7efa  V..t.j..@=.t.j~.
00000030: a464 6174 6194 82a4 6e61 6d65 a46f 7830  .data...name.ox0
00000040: 33a5 7374 6166 66c3 83a4 6e61 6d65 c0a5  3.staff...name..
00000050: 7374 6166 66c2 a565 7874 7261 81a4 696e  staff..extra..in
00000060: 666f a082 a46e 616d 65a4 6f78 3033 a573  fo...name.ox03.s
00000070: 7461 6666 c380                           taff..
```

Figure 66: Hexadecimal output (`xxd`) of encoding Figure 10 input data with MessagePack (118 bytes).

History. MessagePack [56] is a schema-less binary serialization specification designed by Sadayuki Furuhashi [127] in 2009 while working on the Kumofs [128] distributed key-value store. MessagePack is used at the core of services such as Fluentd [129], another popular project by the same author, and Pinterest [130]. MessagePack has been released under the Apache License 2.0 [131]. There also exists third-party MessagePack C implementations suited for embedded development such as [132], [133], and [134].

Characteristics.

- **Simplicity.** The MessagePack specification is easy to understand and implement from a developer point of view. [95] cites MessagePack's ease of use as the main reason why they chose MessagePack over Protocol Buffers [59] to transmit information obtained from the analysis of the video signal of their real-time position tracking system design.
- **Portability.** MessagePack popularity and simplicity resulted in a large amount of official and third-party implementations covering over forty programming languages. In comparison, FlexBuffers [140] and Microsoft Bond [87] support two and four programming languages, respectively.

Layout. A MessagePack bit-string is a sequence of key-value pairs, sometimes nested. Each key or value (an element) is prefixed with a type definition. A key-value pair is a concatenation of the key element followed value element. A map (an object) is the concatenation of its key-value pairs.

Types. MessagePack elements are prefixed with a type definition that occupies from 1 to 9 bytes depending on the type and the length of the element as shown in Table 26. If applicable, the type definition includes a Big Endian unsigned integer representing the logical size or byte-length of the element. The width of the unsigned integer is determined by the first part of the type definition.

Numbers. MessagePack supports Big Endian IEEE 754 32-bit (type definition `0xca`) and 64-bit (type definition `0xcb`) floating-point numbers [51]. In terms of integers, MessagePack supports Big Endian 8-bit (type definition `0xcc`), 16-bit (type definition `0xcd`), 32-bit (type definition `0xce`), and 64-bit (type definition `0xcf`) unsigned integers and 8-bit (type definition `0xd0`), 16-bit (type definition `0xd1`), 32-bit (type definition `0xd2`), and 64-bit (type definition `0xd3`) signed Two's Complement [50] integers.

Unsigned integers less than 128 and signed integers greater than or equal to -32 are encoded as 8-bit integers without a preceding type definition resulting in efficient use of space. MessagePack can distinguish these integers based on their constant most-significant bits (`0` for the unsigned integers

[127] `https://github.com/frsyuki`
[128] `https://github.com/etolabo/kumofs`
[129] `https://www.fluentd.org`
[130] `https://www.pinterest.com`
[131] `http://www.apache.org/licenses/LICENSE-2.0.html`
[132] `https://github.com/clwi/CWPack`
[133] `https://github.com/ludocode/mpack`
[134] `https://github.com/rtsisyk/msgpuck`

and 111 for the signed integers) as they are not re-used in any other type definition. MessagePack implementations pick the smallest data type that can encode the given number.

Strings. MessagePack produces UTF-8 [32] strings that are not delimited with the *NUL* character for both property names and string values. MessagePack supports four encoding variants of the type definition depending on the byte-length of the string:

- If the byte-length of the string is less than 32, then MessagePack prefixes the string with a type definition whose most-significant 3 bits equal 101 and whose remaining bits encode the string byte-length as an unsigned 5-bit integer.
- If the type definition is the byte 0xd9, then MessagePack prefixes the string with the type definition followed by the byte-length of the string as an 8-bit unsigned integer.
- If the type definition is the byte 0xda, then MessagePack prefixes the string with the type definition followed by the byte-length of the string as a Big Endian 16-bit unsigned integer.
- If the type definition is the byte 0xdb, then MessagePack prefixes the string with the type definition followed by the byte-length of the string as a Big Endian 32-bit unsigned integer.

MessagePack does not attempt to deduplicate multiple occurrences of the same property name or string value.

Booleans. Booleans are encoded at the type level using the type definition bytes 0xc2 (False) and 0xc3 (True).

Lists. MessagePack encodes lists (arrays) as the concatenation of its elements prefixed with a type definition that includes the logical size of the array. MessagePack supports three encodings variants of the array type definition depending on the size of the array:

- If the array contains less than 16 elements, then MessagePack prefixes the array with a type definition whose most-significant 4 bits equal 1001 and whose remaining bits encode the array logical size as an unsigned 4-bit integer.

Table 26: A list of MessagePack type definitions as specified in [56]. This table omits extension types which are not discussed in this paper.

Type	Type byte	Embedded in type byte	Length
Nil	0xc0	Nil	None
Boolean	0xc2 to 0xc3	False, True	None
Unsigned integer	0x00 to 0x7f	7-bit unsigned integer	In type byte
Unsigned integer	0xcc to 0xcf	Integer width: 8-bit, 16-bit, 32-bit, 64-bit	In type byte
Signed integer	0xe0 to 0xff	5-bit signed integer	In type byte
Signed integer	0xd0 to 0xd3	Integer width: 8-bit, 16-bit, 32-bit, 64-bit	In type byte
Float	0xca to 0xcb	Float precision: 32-bit, 64-bit	In type byte
String	0xa0 to 0xbf	5-bit unsigned integer	In type byte
String	0xd9 to 0xdb	String byte-length integer width: 8-bit, 16-bit, 32-bit, 64-bit	Big Endian 8-bit, 16-bit, or 64-bit unsigned integer
Byte array	0xc4 to 0xc6	Byte array length integer width: 8-bit, 16-bit, 32-bit	Big Endian 8-bit, 16-bit, or 64-bit unsigned integer
Array	0x90 to 0x9f	4-bit unsigned integer	In type byte
Array	0xdc to 0xdd	Array length integer width: 16-bit, 32-bit	Big Endian 16-bit or 32-bit unsigned integer
Map	0x80 to 0x8f	4-bit unsigned integer	In type byte
Map	0xde to 0xdf	Map length integer width: 16-bit, 32-bit	Big Endian 16-bit or 32-bit unsigned integer

- If the type definition is the byte `0xdc`, then MessagePack prefixes the array with the type definition followed by the logical size of the array as a Big Endian 16-bit unsigned integer.
- If the type definition is the byte `0xdd`, then MessagePack prefixes the array with the type definition followed by the logical size of the array as a Big Endian 32-bit unsigned integer.

MessagePack does not attempt to deduplicate multiple occurrences of the same element in an array.

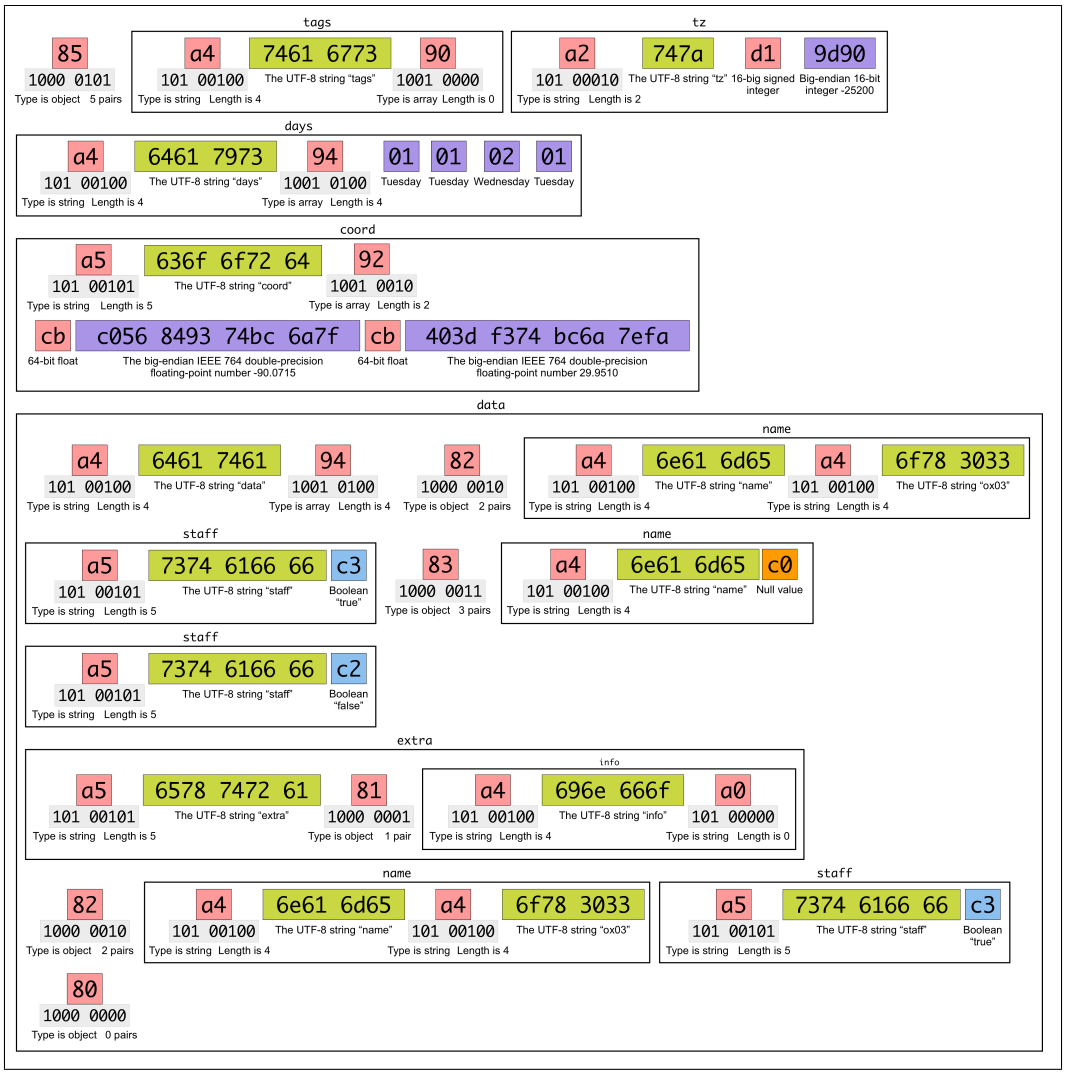

Figure 67: Annotated hexadecimal output of serializing the Figure 10 input data with MessagePack.

Table 27: A high-level summary of the MessagePack schema-less serialization specification.

Website	`https://msgpack.org`
Company / Individual	Sadayuki Furuhashi
Year	2009
Specification	`https://github.com/msgpack/msgpack/blob/master/spec.md`
License	Apache License 2.0
Layout	Sequential
Languages	ActionScript, C, C++, C#, Clojure, Crystal, D, Dart, Delphi, Elixir, Erlang, F#, Go, GNU Guile, Haskell, Haxe, HHVM, J, Java, JavaScript, Julia, Kotlin, Nim, MATLAB, OCaml, Objective-C, Pascal, PHP, Perl, Pony, Python, R, Racket, Ruby, Rust, Scala, Scheme, Smalltalk, SML, Swift
Types	
Numeric	Big Endian 5-bit, 8-bit, 16-bit, 32-bit, and 64-bit Two's Complement [50] signed integers Big Endian 7-bit, 8-bit, 16-bit, 32-bit, and 64-bit unsigned integers Big Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51]
String	UTF-8 [32]
Composite	Array, Map
Scalars	Boolean, Nil
Other	Bin (byte array) 32-bit, 64-bit, and 96-bit UNIX seconds and nanoseconds Epoch timestamps [89]

6.5 Smile

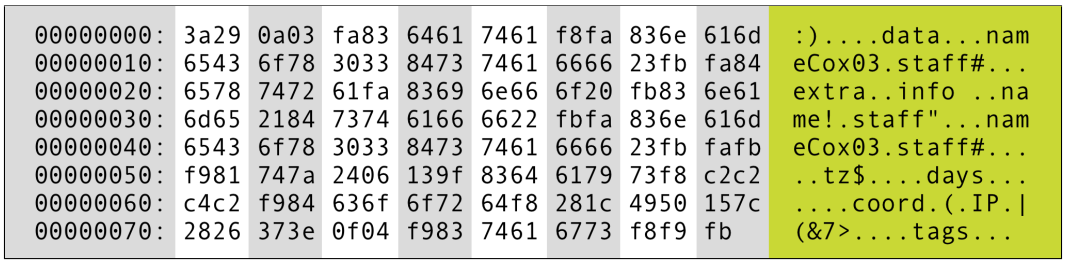

Figure 68: Hexadecimal output (`xxd`) of encoding Figure 10 input data with Smile (127 bytes).

History. Smile [116] is a schema-less binary serialization specification developed by the team behind the popular Jackson JSON parser [135]. Smile is an attempt to create a standard JSON [18] binary representation. Smile development started on 2010 led by Tatu Saloranta [136], founder of FasterXML [137] while also being a Principal Software Engineer at Salesforce. Smile is released under the 2-clause BSD license [138].

Characteristics.

- **Performance Efficiency.** Smile's observation is that serialization specifications typically sacrifice write performance to speed-up read operations. As a solution, Smile has been designed support equally runtime-efficient read and write operations.
- **Streaming.** Smile implementations can de-serialize bit-strings given a fixed amount of buffering.

Layout. A Smile bit-string is a self-delimited sequence of key-value pairs, sometimes nested. The bit-strings produced by Smile are prefixed with a header that consists of the ASCII string `0x3a 0x29 0x0a`, a smiling face as the ASCII string ":)" plus a new-line character. The header string is followed by a byte that consists of the version number as a 4-bit unsigned integer, followed by a reserved bit, followed by 3 bit flags:

- Whether the bit-string contains raw binary values.
- Whether string values may be shared.
- Whether property names may be shared.

Key-value pairs are encoded as the key element followed by the value element. Smile objects are encoded as the concatenation of their key-value pairs prefixed with the byte `0xfa` and suffixed with the byte `0xfb`. Smile messages are guaranteed to not contain the `0xff` byte as such byte is supported as an optional end-of-message marker for framing purposes.

Types. Smile data types are prefixed with a 1-byte type definition. The value might be embedded in the type definition if it is small enough resulting in efficient use of space. There are three groups of type definitions:

- Type bytes whose type is encoded in the 3 most-significant bits and the remaining 5-bits are type-dependent.
- Constants such as `0x21` (null).
- Structural markers such as `0xf8` (start of array).

Numbers. Smile supports Little Endian 5-bit, 32-bit, and 64-bit ZigZag-encoded 4.2 signed integers. 5-bit signed integers are encoded as the least-significant bits of a type byte whose most-significant 3

[135] `https://github.com/FasterXML/jackson`
[136] `https://github.com/cowtowncoder`
[137] `http://fasterxml.com`
[138] `https://opensource.org/licenses/BSD-2-Clause`

bits equal 110. Smile also supports a *BigInteger* type capable of encoding arbitrary-length signed integers. These type of integers are encoded using the type byte 0x30, followed by the byte-length of the *stringified* representation of the integer as an unsigned Little Endian Base 128 (LEB128) (4.1) variable-length integer, followed by the *stringified* UTF-8 [32] representation of the integer encoded as a byte array.

In terms of floats, Smile supports Little Endian 32-bit and 64-bit IEEE 764 [51] floating-point numbers. However, Smile encodes floating-point numbers using 7-bits to avoid including bytes that are reserved for other type definitions. The encoding process consists in:

1. Obtaining the Big Endian IEEE 764 [51] representation of the floating-point number.
2. Writing the least-significant 7 bits.
3. Right-shifting 7 bits.
4. Repeating the process until encoding the entire bit-string as shown in Figure 69.

Smile also supports a *BigDecimal* type capable of encoding arbitrary-precision decimal numbers. These type of decimal numbers are encoded using the type byte 0x20, followed by the scale as a ZigZag-encoded (4.2) 32-bit signed integer, followed by the *stringified* UTF-8 [32] representation of the integral part encoded as a byte array prefixed with its byte-length as an unsigned Little Endian Base 128 (LEB128) (4.1) variable-length integer.

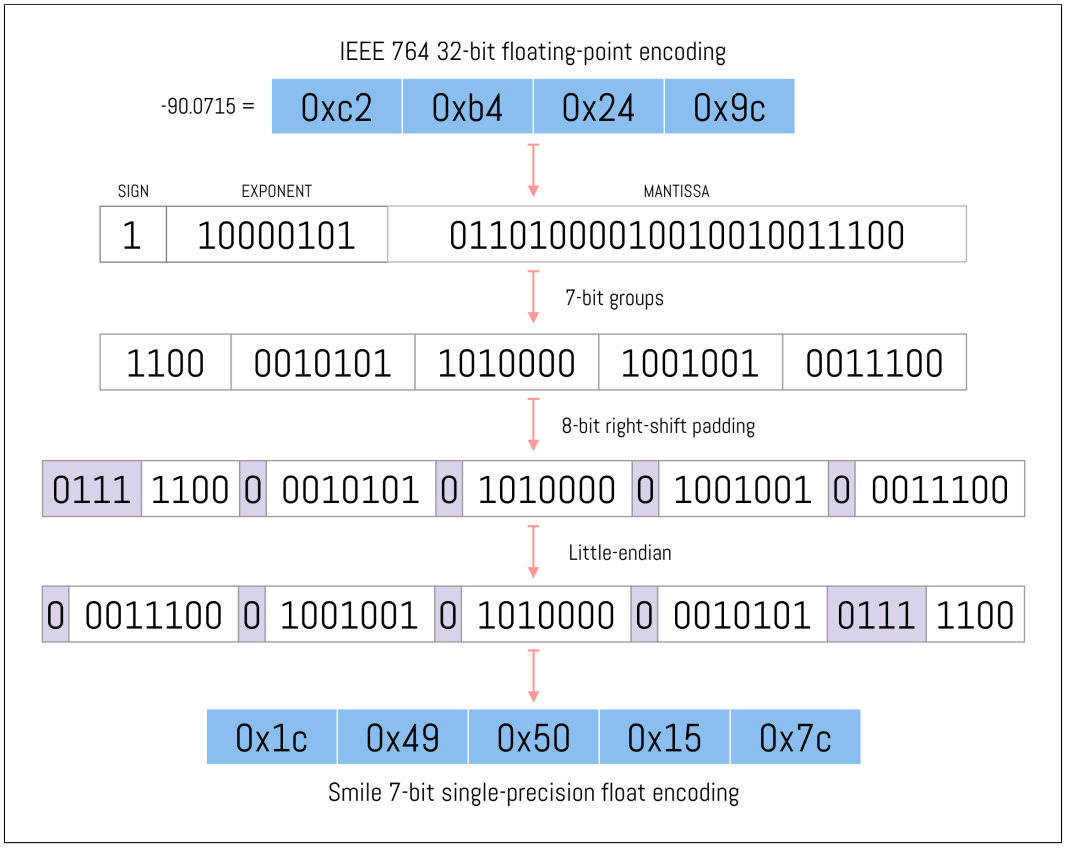

Figure 69: A visual representation of converting the -90.0715 IEEE 764 single-precision floating-point number [51] to Smile's 7-bit float encoding.

Strings. Smile supports strings using ASCII [27] and UTF-8 [32] encodings that are not *NUL*-delimited. Based from our observations, property names are encoded using UTF-8 while string values are encoded using ASCII unless the strings contain characters outside of the ASCII range. Smile provides an empty string type and three type definition variants for each of the supported encodings depending on the byte-length of the string as shown in Table 28.

For example, an ASCII [27] string consisting of 4 characters can be encoded as a *Tiny ASCII* string. Therefore, it is prefixed with the type byte `0x43 = 01000011`. The 5 least-significant bits are the unsigned integer 3 so the string length is $4 = 1 + 3$.

Alternatively, a UTF-8 [32] string consisting of 35 characters can be encoded as a *Small Unicode* string. Therefore, it is prefixed with the type byte `0xa1 = 10100001`. The 5 least-significant bits are the unsigned integer 1 so the string length is $35 = 34 + 1$.

Note that the *Tiny Unicode* string encoding group cannot encode a 1-byte string. A 1-byte UTF-8 [32] string must be a valid ASCII [27], therefore the *Tiny ASCII* encoding group is preferred.

Booleans. Booleans are encoded as the type definition level using the type bytes `0x22` (False) and `0x23` (True).

Lists. Smile encodes lists (arrays) as the concatenation of its elements prefixed with the constant byte `0xf8` and suffixed with the constant byte `0xf9`.

Table 28: The string encodings that Smile supports as specified in [116]. The *From* and *To* columns describe the string byte-lengths that each group can encode.

Name	Type byte	From	To	String length	Suffix
Empty string	`0x20`	0 bytes	0 bytes	0	None
Tiny ASCII	`0x40` to `0x5f`	1 byte	32 bytes	Least-significant 5-bits as unsigned integer + 1	None
Tiny Unicode	`0x80` to `0x9f`	2 bytes	33 bytes	Least-significant 5-bits as unsigned integer + 2	None
Small ASCII	`0x60` to `0x7f`	33 bytes	64 bytes	Least-significant 5-bits as unsigned integer + 33	None
Small Unicode	`0xa0` to `0xbf`	34 bytes	65 bytes	Least-significant 5-bits as unsigned integer + 34	None
Long ASCII	`0xe0`	1 byte	Any	Until suffix marker	`0xfc` end-of-string marker
Long Unicode	`0xe4`	2 bytes	Any	Until suffix marker	`0xfc` end-of-string marker

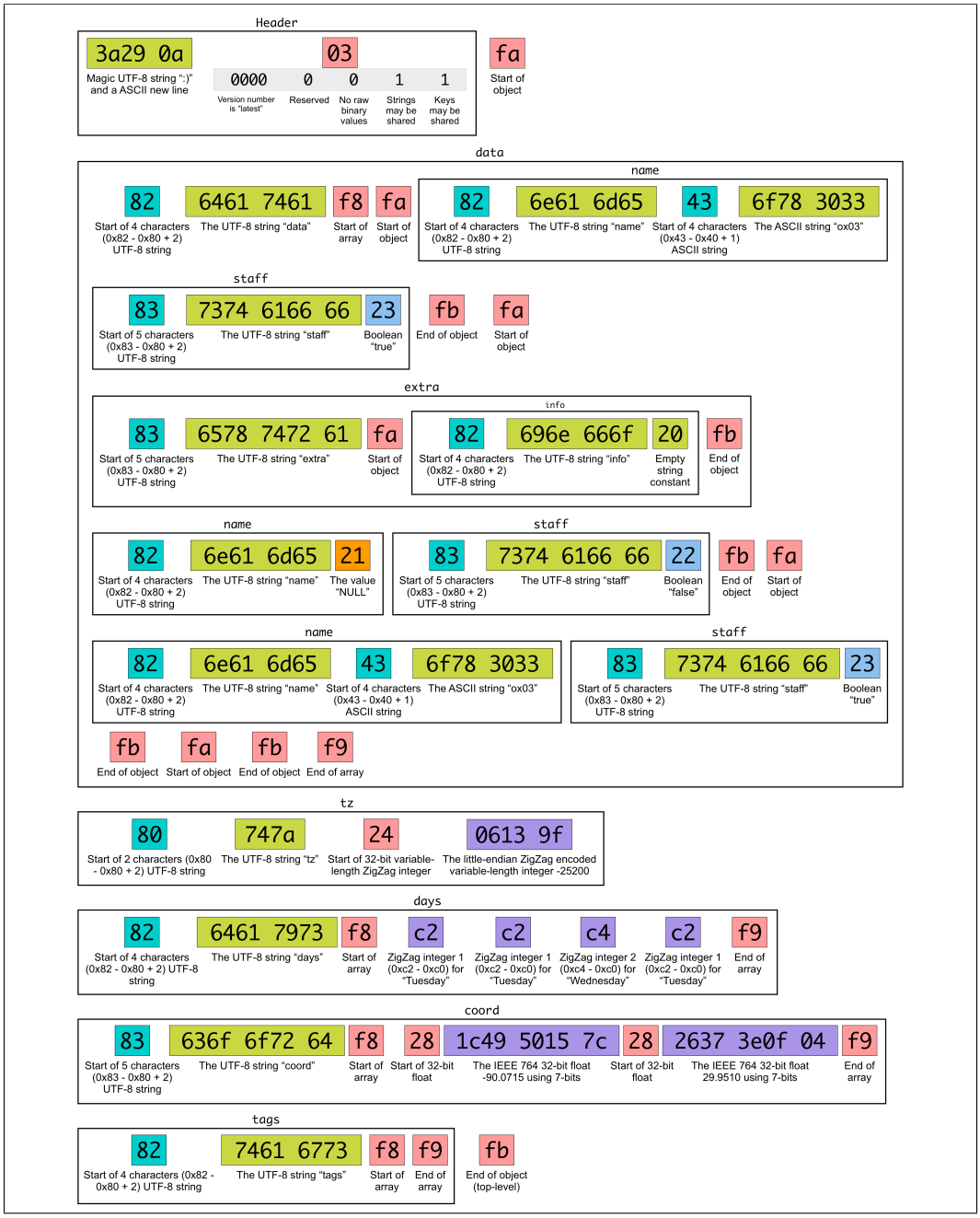

Figure 70: Annotated hexadecimal output of serializing the Figure 10 input data with Smile.

Table 29: A high-level summary of the Smile schema-less serialization specification.

Website	`https://github.com/FasterXML/smile-format-specification`
Company / Individual	FasterXML
Year	2010
Specification	`https://github.com/FasterXML/smile-format-specification/blob/master/smile-specification.md`
License	2-clause BSD
Layout	Sequential
Languages	C, Clojure, Go, Java, JavaScript, Python
Types	
Numeric	5-bit, 32-bit, and 64-bit ZigZag-encoded (4.2) signed Little Endian Base 128 (LEB128) (4.1) variable-length integers Little Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51] encoded using 7 bit groups Arbitrary-length stringified signed integers Arbitrary-precision decimals (with scale and stringified integral)
String	ASCII [27], UTF-8 [32]
Composite	Array, Object
Scalars	Boolean, Null
Other	Binary (byte array)

6.6 UBJSON

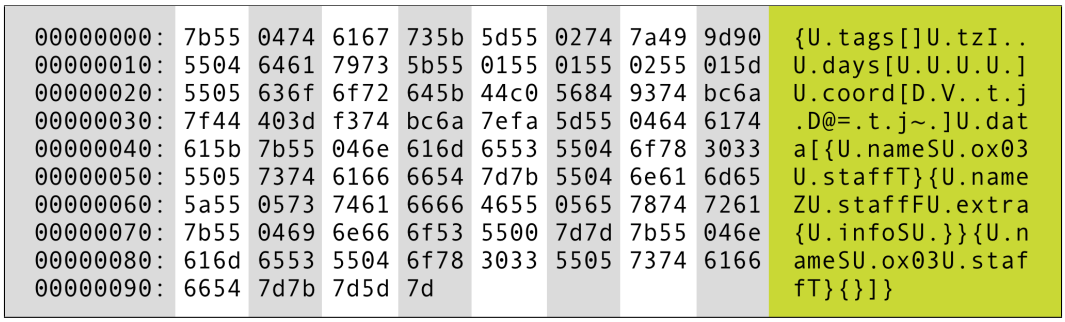

Figure 71: Hexadecimal output (`xxd`) of encoding Figure 10 input data with UBJSON (151 bytes).

History. UBJSON [17] is a schema-less binary serialization specification that is a purely-compatible binary alternative to JSON [18]. Riyad Kalla [139] started working on UBJSON in 2012 while working as a Principal Lead at Genuitec [140]. Many high-profile software solutions include UBJSON support, such as Teradata [141] and Wolfram Mathematica [142]. UBJSON is also natively supported as part of the RIOT operating system for the Internet of Things [13]. The UBJSON specification has been released under the Apache License 2.0 [143].

Characteristics.

- **Readability.** Despite being a binary serialization specification, UBJSON is comparatively human-readable as it makes use of printable ASCII [27] characters in field type definitions.
- **Simplicity.** The UBJSON specification is easy to understand and implement from a developer point of view as the specification is defined using a single core data structure throughout the entire specification
- **JSON Compatibility.** In comparison to serialization specifications such as BSON [86], UBJSON is strictly compatible with the JSON specification [18] and does not introduce additional data types.

Layout. A UBJSON bit-string is a sequence of key-value pairs, sometimes nested. Every UBJSON element shares the same structure: a 1-byte type definition, the content length if applicable, and the optional content data. A key-value pair is the sequence of a string element and its corresponding value element. UBJSON encodes objects as the concatenation of their key-value pairs prefixed with the byte `0x7b` (the ASCII character `{`) and suffixed with the byte `0x7d` (the ASCII character `}`).

Types. Each UBJSON element is prefixed with a 1-byte type definitions. Type definitions are encoded using characters from the *printable* ASCII [27] range. Each type definition character is a mnemonic of its respective type. Refer to Figure 72 for a complete list. For example, the *string* type makes use of the character `S`.

Numbers. UBJSON supports Big Endian 8-bit (type definition `i`), 16-bit (type definition `I`), 32-bit (type definition `l`), and 64-bit (type definition `L`) signed Two's Complement [50] integers and Big Endian 8-bit (type definition `U`) unsigned integers. UBJSON also supports Big Endian IEEE 764 [51] 32-bit (type definition `d`) and 64-bit (type definition `D`) floating-point numbers. Additionally, UBJSON supports high-precision arbitrarily-large UTF-8 [32] stringified numbers prefixed by the type definition `H` and the byte-length of the string. UBJSON implementations pick the smallest data type that can encode the given number.

[139] `https://github.com/rkalla`
[140] `https://www.genuitec.com`
[141] `https://www.teradata.co.uk`
[142] `https://www.wolfram.com/mathematica/`
[143] `http://www.apache.org/licenses/LICENSE-2.0.html`
[144] `https://github.com/WhiZTiM/UbjsonCpp/blob/7a7857f64247ce82b72072b04f87183b090fd554/include/types.hpp`

Strings. UBJSON produces UTF-8 [32] strings that are not delimited with the *NUL* character for both property names and string values. UBJSON strings are prefixed with the type definition `S` and the byte-length of the string as a standalone integer element with its own type definition. UBJSON does not attempt to deduplicate multiple occurrences of the same property name or string value. UBJSON supports a *character* type to encode a single-character string without encoding its byte-length resulting in efficient use of space.

Booleans. UBJSON encodes booleans at the type definition level. UBJSON supports two data types without content: `0x54` (True) and `0x46` (False) which stand for the ASCII [27] characters `T` and `F`, respectively.

Lists. UBJSON encodes lists (arrays) as the concatenation of its elements. UBJSON arrays are prefixed with the byte `0x5b` (the ASCII character `[`) and suffixed with the byte `0x5d` (the ASCII character `]`). UBJSON arrays are not prefixed with their logical size nor their byte-length. UBJSON does not attempt to deduplicate multiple occurrences of the same element in an array.

```
enum class Marker : byte
{
    Invalid = '\0',

    Null    = 'Z',
    No_Op   = 'N',
    Char    = 'C',
    True    = 'T',
    False   = 'F',

    Int8    = 'i',
    Uint8   = 'U',
    Int16   = 'I',
    Int32   = 'l',
    Int64   = 'L',
    Float32 = 'd',
    Float64 = 'D',
    HighPrecision = 'H',

    String  = 'S',
    Binary  = 'b',  //Extension

    Object_Start   = '{',
    Object_End     = '}',
    Array_Start    = '[',
    Array_End      = ']',

    Array,
    Object,

    Optimized_Type  = '$',
    Optimized_Count = '#'
};
```

Figure 72: UBJSON markers definition [144].

7 Schema Evolution

Schema evolution is the problem of updating a schema definition while ensuring that the programs relying on it can continue to operate. The study of schema evolution originated in the context of relational databases to evolve database schemas without disrupting client applications [114]. In the context of binary serialization specifications, schema evolution is concerned with how bit-string producers and bit-string consumers can intercommunicate despite future updates to the structure of the bit-strings they are concerned with.

As discussed in subsection 1.4, programs using schema-driven serialization specifications must know in advance the schema definitions of the messages they are expecting to interchange. This problem is exacerbated by the fact that schema definitions are typically updated in response to new or changing

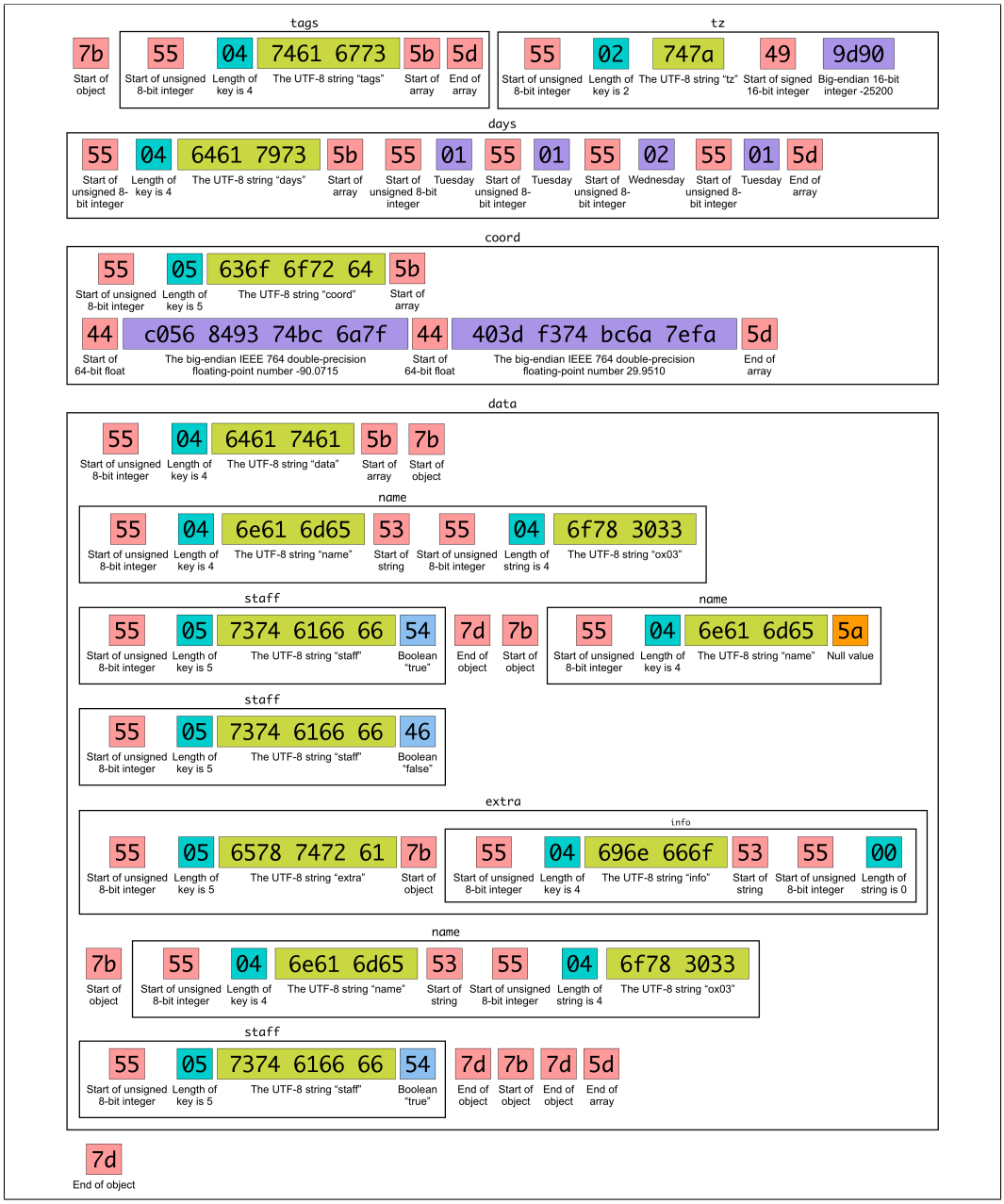

Figure 73: Annotated hexadecimal output of serializing the Figure 10 input data with UBJSON.

requirements. [91] state that software requirements continuously change and as a result of these changes software projects tend to fail. Schema evolution is an important topic in the context of schema-driven serialization specifications as updating a schema definition may result in a risky and expensive operation that requires re-compilation and coordinated re-deployment of all the programs relying on such schema.

Two schemas are *compatible* if one schema can deserialize a bit-string produced by the other schema and recover the original information. There are three levels of schema compatibility:

- **Backwards.** The first schema is backwards-compatible with respect to the second schema if the first schema can deserialize data produced by the second schema.

Table 30: A high-level summary of the UBJSON schema-less serialization specification.

Website	`https://ubjson.org`
Company / Individual	Riyad Kalla
Year	2012
Specification	`https://github.com/ubjson/universal-binary-json`
License	Apache License 2.0
Layout	Sequential
Languages	C, C++, C#, D, Go, Java, JavaScript, MATLAB, PHP, Python, Swift
Types	
Numeric	Big Endian 8-bit, 16-bit, 32-bit, and 64-bit Two's Complement [50] signed integers Big Endian 8-bit unsigned integers Big Endian 32-bit and 64-bit IEEE 764 floating-point numbers [51] Arbitrary-precision ASCII-encoded numbers
String	UTF-8 [32]
Composite	Array, Object
Scalars	Boolean, Null

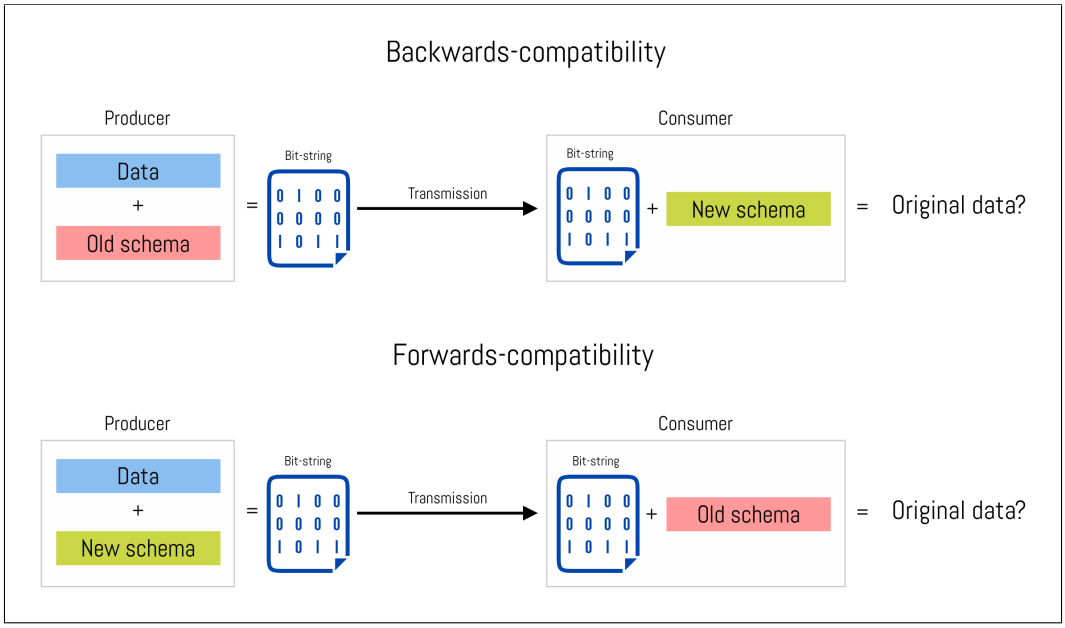

Figure 74: In both of these cases, the producer serializes a data structure using one version of the schema and the consumer attempts to deserialize the resulting bit-string using another version of the schema. Schema evolution is concerned with whether the consumer will succeed in obtaining the original data structure or not.

- **Forwards.** The first schema is forwards-compatible with respect to the second schema if the second schema can deserialize data produced by the first schema.
- **Full.** The new schema is fully-compatible with respect to the old schema if it is both backwards and forwards compatible with respect to the old schema.

We can think of a schema as a set of its valid instances where the following rules apply:

- A backwards or forwards compatible transformation to the schema *expands* or *confines* the set of its valid instances, respectively. A fully-compatible transformation to the schema keeps the set of its valid instances intact.
- A schema transformation results in an incompatible schema if neither of the sets is a subset of the other.
- A schema is *fully-compatible* with respect to itself.
- The first schema is *backwards-compatible* with respect to the second schema if and only if the second schema is *forwards-compatible* with respect to the first schema.
- Compatibility is a transitive property. A schema is transitive backwards, transitive forwards, or transitive fully-compatible with respect to a set of schemas if it is backwards, forwards, or fully compatible with respect to each of the schemas in the set, respectively.

7.1 Deploying Schema Transformations

Consider a system involving a set of consumers and a set of producers that exchange information using a schema-driven serialization specification. In such a system, the rules for deploying *compatible* schema transformations with zero-downtime are as described in Table 31. Deploying *incompatible* schema transformations typically involves re-deploying all consumers and producers at the same time or including multiple incompatible schemas in each of the programs and adding application-specific logic to decide which schema to use when.

The same program in the system might be both a producer and a consumer. In this case, consider the program to use one schema to produce data and another schema to consume data where the two schemas may be equal. Therefore, each of the schemas within the same program can be deployed separately using the rules described in Table 31.

Table 31: These are the rules for deploying compatible schema transformations with zero-downtime. For example, it is safe to deploy a forwards-compatible schema transformation to any producer. However, deploying a forwards-compatible schema transformation to any consumer requires first deploying the schema transformation to all producers.

	Backwards-compatible schema transformation	**Forwards-compatible schema transformation**	**Fully-compatible schema transformation**
Deploy to Producer	Deploy transformation to consumers first	Safe	Safe
Deploy to Consumer	Safe	Deploy transformation to producers first	Safe

7.2 Compatibility Analysis

We selected a set of structural and type conversion schema transformations and tested if they result in compatible changes using the schema-driven serialization implementations and encodings introduced in Table 6. A different encoding of the same schema-driven serialization specification may yield different results. The results of the structural schema transformations are presented in Table 33 and the results of the type conversion schema transformations are presented in Table 34. We mark the test results as shown in Table 32.

We found that sometimes the schema evolution features of a serialization specification are subtly affected by the data types being used and by the infinite possibilities of surrounding data. For this reason, we recommend schema-writers to use these results as a guide and to unit test the schema transformations they plan to apply before deploying them. We also encountered various cases of undocumented compatible schema transformations. These transformations may rely on accidental behaviour of either the serialization specification design or the chosen implementation and may carry no future guarantees. We encourage readers to consult the official schema evolution documentation and check if their serialization specification of choice satisfies the intended compatible transformation by design or by accident.

Table 32: Descriptions of how we will mark schema evolution transformation results.

Symbol	Description	When
`A`	Fully-compatible	The schemas are fully-compatible for all tested instances
`F`	Forwards-compatible	The schemas are forwards-compatible for all tested instances, despite backwards-compatibility failures or exceptions
`B`	Backwards-compatible	The schemas are backwards-compatible for all tested instances, despite forwards-compatibility failures or exceptions
`N`	Silently-incompatible	The schemas are not forwards nor backwards-compatible in at least one tested instance but no exception is thrown
`X`	Runtime exception	The schemas are neither forwards nor backwards-compatible for all tested instances and at least one exception is thrown
	Not-applicable	The schema transformation is not applicable to the serialization specification as it involves data types not supported by the serialization specification

Description from Table 33.

(1) Microsoft Bond [87] supports the concept of *required_optional* fields that are required at serialization time but optional at deserialization time. This concept enables schema-writers to make an optional field requires and viceversa in a fully-compatible manner through a two-step process: Changing an optional or required field to *required_optional*, deploying the schema update to both producers and consumers, and then changing the *required_optional* field to required or optional.

(2) ASN.1 PER Unaligned [119] and Cap'n Proto [142] support updating a list of scalars to a list of structures where the scalar is the first and only element in a fully-compatible manner. In the case of ASN.1 PER Unaligned, this transformation is possible because structures are list of values and a list of structures with a single required scalar is encoded in the same manner as a list of such scalars. In the case of Cap'n Proto, a list definition declares whether its element are scalars or composites as shown in Table 14. If the elements are composite, the list definition points to a 64-bit word that defines the composite elements, allowing the deserializer to determine if following the pointer or not yields a scalar of the same expected type. As an exception, Cap'n Proto does not support this schema transformation on a list of booleans for runtime-efficiency reasons [145].

(3) Protocol Buffers Binary Wire Format [59] supports transforming a field into a list of a compatible type in a backwards-compatible manner. Protocol Buffers Binary Wire Format encodes lists as multiple occurrences of the same field identifier or as a concatenation of the members prefixed with a length-delimited type definition in the case of packed field encoding. This design decision makes implementations using the new schema interpret a standalone value as a list consisting of one value.

(4) Serialization specifications based on field identifiers that implement unions without involving additional structures such as Protocol Buffers Binary Wire Format [59] support forwards-compatibility

[145] `https://capnproto.org/language.html#evolving-your-protocol`

when moving an optional field into an existing union. In this case, union choices and fields outside of the union share the same field identifier context. This means that an application using the older schema either leaves the union choices or the optional field outside the union unset. In comparison, FlatBuffers [139] requires creating a new data structure to hold the union type. As a result, it supports backwards-compatibility when moving an optional field into an existing union as an application using the newer schema will ignore the optional field outside of the union. The converse is true when extracting an optional field out of an existing union.

(5) Protocol Buffers [59] implements unions based on field identifiers on the current identifier context and supports unions of a single choice. Therefore, an optional field and a union of the single field are equivalent. A similar argument follows for Cap'n Proto [142], however Cap'n Proto does not support unions of a single choice, making this transformation only backwards-compatible. Apache Avro [52]

Table 33: A schema transformation result is annotated as shown in Table 32. The *Type* column documents whether a schema transformation confines (`C`), expands (`E`), changes (`!`), or preserves (`=`) the domain of the schema.

Category	Type	Schema Transformation	ASN.1 PER Unaligned	Apache Avro Binary Encoding	Microsoft Bond Compact Binary v1	Cap'n Proto Packed Encoding	FlatBuffers Binary Wire Format	Protocol Buffers Binary Wire Format	Apache Thrift Compact Protocol
Structures / Tables	E	Add an optional field to the end	A	A	A	A	A	A	A
	C	Remove an optional field from the end	A	A	A	A	A	A	A
	C	Add a required field	F	F	F		F		F
	E	Remove a required field	B	B	B		B		B
	C	Optional to required	F	F	A (1)		F		F
	E	Required to optional	B	B	A (1)		B		B
	!	Change field default	N	N	N	N	N		N
Lists	E	List of scalars to list of structures with scalar	A (2)	X	N	A (2)	X	X	X
	E	Scalar to list of scalars	X	X	X	N	N	B (3)	N
	E	Composite to list of composites	X	X	X	X	X	B (3)	N
Unions	!	Move optional field to existing union	X	X		N	B (4)	F (4)	N
	!	Extract optional field from existing union	X	X		N	F (4)	B (4)	N
	E	Move required field to existing union	X	X			B		N
	C	Extract required field from existing union	X	X			F		N
	!	Move optional field to a new union	X	B (5)		B (5)	N	A (5)	N
	E	Move required field to a new union	B	B			N		N
	E	Add choice to existing union	B	B		B	B	B	B
	C	Remove choice from existing union	F	F		F	F	F	F
Enums	C	Scalar to enumeration	F	X	F	F	F	F	F
	E	Enumeration to scalar	B	X	B	B	B	B	B
	E	Add enumeration constant	B	B	B	B	B	B	B
	C	Remove enumeration constant	F	F	F	F	F	F	F

supports unions of a single choice, however its schema resolution rules throw an exception on the forwards-compatible case.

Table 34: A schema transformation result is annotated as shown in Table 32. The *Type* column documents whether a schema transformation confines (`C`), expands (`E`), changes (`!`), or preserves (`=`) the domain of the schema.

Category	Type	Schema Transformation	ASN.1 PER Unaligned	Apache Avro Binary Encoding	Microsoft Bond Compact Binary v1	Cap'n Proto Packed Encoding	FlatBuffers Binary Wire Format	Protocol Buffers Binary Wire Format	Apache Thrift Compact Protocol
Type Conversions	E	Increase integer width	N	B	B	N	B	B	N
	C	Decrease integer width	N	F	F	N	F	F	N
	E	Increase float precision	B	B	B	N	N	N	
	C	Decrease float precision	F	F	F	N	N	N	
	E	Unsigned to larger signed integer	N		X	N	B	B	
	C	Signed to smaller unsigned integer	N		X	N	F	F	
	E	Signed integer to float	X	B	X	N	N	N	N
	C	Float to signed integer	X	F	X	N	N	N	N
	E	String to byte-array	B	X	X	B	B	B	B
	C	Byte-array to string	F	X	X	F	F	F	F
	E	Boolean to integer	X	X	X	N	B	B	N
	C	Integer to boolean	X	X	X	N	F	F	N
	=	Byte-array to array of 8-bit unsigned integers	A		A	A			

8 Conclusions

8.1 Use Cases

In Table 35, we identify a set of use-cases that binary serialization specifications tend to optimize for and the characteristics that typically enable those use-cases.

None of the binary serialization specifications from this study support every use-case listed in Table 35 as some enabling characteristics tend to conflict:

- The *Space-efficiency* use-case typically involves a schema-driven serialization specification. However, JSON [18] is a schema-less serialization specification. Therefore, the *Drop-in JSON replacement* requires a schema-less serialization specification.
- The *Space-efficiency* use-case requires bit-strings to be as compact as possible. However, the *Runtime-efficient deserialization* use-case may require aligned data types and alignment may involve significant padding. For example, Cap'n Proto [142] aligns data types to 64-bit words for runtime-performance reasons and supports a simple compression scheme to mitigate the additional space overhead.
- The *Space-efficiency* use-case typically requires bit-strings to contain minimal metadata. However, the *Partial reads* use-case may require a table of contents for the bit-string, which may result in more encoded metadata. The extra overhead is amortized when encoding large amounts of data sharing the same structures. The input data JSON document from

Figure 10 is a small data structure that consists of significant structure and relatively little data. In the case of the Cap'n Proto [142] and FlatBuffers [139] schema-driven serialization specifications, roughly half of the bit-strings produced by serializing the input data consists of pointers and structural information that represent a table of contents.

- The *Streaming serialization* use-case may involve serializing the scalar types before the composite types, like FlexBuffers [140], given that an implementation may not know the size of a composite data type before its members are encoded. However, this approach tends to conflict with the *Streaming deserialization* use-case as an implementation would have to wait until all scalar types are received before starting to understand how they interconnect.
- The *Runtime-efficient deserialization* and *Partial reads* use-cases typically involve a pointer-based table of contents of the bit-string. As a result, implementing *In-place updates* is usually not runtime-efficient as some updates might involve adjusting pointer references in multiple parts of the bit-string as noted by [100] when using the FlatBuffers [139] serialization specification. For example, adding a new field to a FlexBuffers [140] map may involve creating a new keys vector, updating the metadata and contents of the vector data section, and adjusting most of the pointers in the bit-string.

We could not identify a fundamental conflict involving the *Constrained devices* use-case. We believe that whether a binary serialization specification is a good fit for constrained devices tends to be a consequence of how it is implemented rather than a property of the serialization specification. For example, the official Protocol Buffers [59] implementations are typically not suitable for constrained

Table 35: Every serialization specification considered in this study supports a subset of these use-cases. The *Enabling characteristics* column describes certain characteristics that *tend* to result in a serialization specification that is a good fit for the respective use-case. The last column shows an example of a JSON-compatible binary serialization specification that has at least one of the respective enabling characteristics. However, the fact that a serialization specification has certain characteristics does not guarantee that its implementations make use of those characteristics to enable the respective use-cases, often for reasons other than technical.

Use case	**Enabling characteristics**	**Example**
Space-efficiency	The resulting bit-string embeds little metadata	ASN.1 PER Unaligned [119]
	Non-aligned data types	
Runtime-efficient deserialization	Deserialization without additional memory allocations	Cap'n Proto [142]
	Table of contents for the bit-string	
	Aligned data types	
Partial reads	Field byte-length serialized before content	FlatBuffers Binary Wire Format [139]
	Table of contents of the bit-string	
Streaming deserialization	Field byte-length serialized before content	Smile [116]
	Sequential and standalone-encoded fields	
Streaming serialization	No byte-length prefix metadata, mainly for nested structures	UBJSON [17]
	Content serialized before structure	
	Positional structural markers instead of length prefixes	
In-place updates	Field spatial locality	BSON [86]
	No byte-length field metadata	
	Positional structural markers instead of length prefixes	
Constrained devices	Simple specification and binary layout	CBOR [14]
	Small generated code and/or runtime library	
Drop-in JSON replacement	The resulting bit-string embeds all metadata	MessagePack [56]

devices as they tend to generate large amounts of code and incur significant binary size and memory allocation overheads. Kenton Varda, one of Protocol Buffers former authors, argues that the official Protocol Buffers implementations "*were designed for use in Google's servers, where binary size is mostly irrelevant, while speed and features (e.g. reflection) are valued*" [146]. However, *nanopb* [147] is a Protocol Buffers implementation targeted at 32-bit micro-controllers and other constrained devices. Refer to [10] for discussions and examples of *nanopb*.

8.2 Sequential and Pointer-based Serialization Specifications

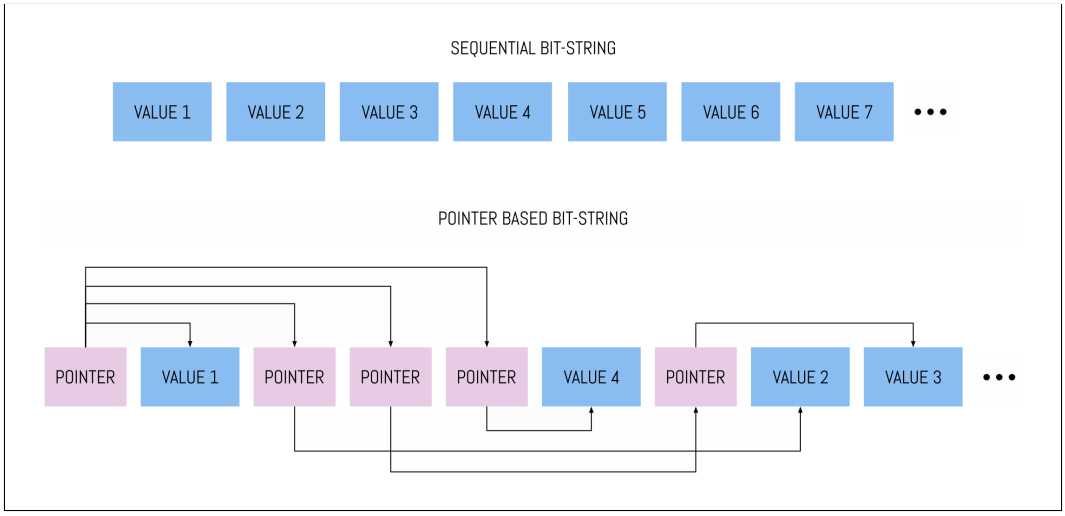

Figure 75: Visual representations of a sequential bit string (top) and a pointer-based bit string (bottom).

We found that serialization specifications can be categorized into that are orthogonal to whether a serialization specification is schema-driven or schema-less: whether the resulting bit-string is *sequential* or *pointer-based* as shown in Figure 75.

Sequential. Serialization specifications are *sequential* if the bit-strings they produce are concatenations of independently-encoded data types. The majority of the serialization specifications discussed in this study are sequential. As an example, Protocol Buffers [59] is a sequential serialization specification as its bit-strings consist of a non-deterministic concatenation of fields [148] that are standalone with respect to the rest of the message.

Pointer-based. Serialization specifications are *pointer-based* if the bit-strings they produce are tree structures where each node is either a scalar type or a composite value consisting of pointers to further nodes. In comparison to sequential serialization specifications, this layout typically results in larger bit-strings that are complicated to understand. However, pointer-based serialization specifications enable efficient streaming deserialization, efficient random access reads and no additional memory allocations during deserialization which translates to better deserialization runtime performance. The pointer-based serialization specifications discussed in this study are Cap'n Proto [142], FlatBuffers [139], and FlexBuffers [140].

8.3 Types of Schema Compatibility Resolution

Every schema-driven serialization specifications discussed in this study allow the schema-writer to perform certain schema transformations in compatible manners. We found that we can categorize schema-driven serialization specifications into two groups based on how they approach schema compatibility resolution: *data-based resolution* or *schema-based resolution*.

[146] `https://news.ycombinator.com/item?id=25586632`
[147] `https://github.com/nanopb/nanopb`
[148] `https://developers.google.com/protocol-buffers/docs/encoding#implications`

Data-based resolution. Every schema-driven serialization specification discussed in this study except for Apache Avro [52] fall into this category. In this type of schema compatibility resolution, the serialization specification tries to understand the data by deserializing the bit-string as if it was produced with the new schema and trying to accomodate to potential mismatches at runtime.

Schema-based resolution. This approach is pioneered by Apache Avro [52], which refers to it as *symbolic resolution*. In comparison to the other schema-driven serialization specifications analyzed in this study, an application deserializing an Apache Avro bit-string has to provide both the *exact schema* that was used to produce the bit-string and the new schema. The implementation attempts to resolve the differences between the schemas before deserializing the bit-string in order to determine how to adapt any instance to the new representation. The bit-string is deserialized using the old schema and transformed to match the new schema. [144] briefly discusses the problem of integrating heterogenous JSON datasets by resolving differences at the schema-level using a similar approach. [150] propose a similar approach based on version control systems where the codebase only maintains the latest schema definition and code to upgrade older bit-strings to the latest version is auto-generated based on the project commit history.

We found that implementations of *data-based resolution* schema-driven serialization specifications, with some exceptions, tend to perform little runtime checks to ensure data consistency, presumably for performance reasons. For example, if the schema declares that the piece of data to follow is a Little Endian 64-bit unsigned integer, then the deserialization specification may blindly try to interpret the next 64-bits of the bit-string as such, resulting in many cases in silently-incompatible unpredictable results rather than informative runtime exceptions. In comparison to *data-based resolution* schema-driven serialization specifications, we found that *schema-based resolution* tends to produce informative runtime exceptions rather than unpredictable silently-incompatible results. However, *schema-based resolution* specifications require the consumer to know the exact schema that was used to produce the data and have it available at the deserialization process which may result in more complicated schema transformation deployments.

Based on the schema evolution experiments performed in subsection 7.2, we conclude that none of these approaches produce specifications that are clearly more advantageous with regards to compatible schema transformations: with some specification-specific exceptions, most specifications tend to support the same compatible schema transformations.

8.4 Similarities

The bit-strings produced by the selection of binary serialization specifications from subsection 3.1 were more similar than the authors expected. Each serialization specification has a certain degree of unique characteristics and its tuned to particular use-cases. However, most serialization specifications share the same underlying ideas and approach to encoding. The only notable exception to this pattern were the sequential and pointer-based serialization specification groups discussed in subsection 8.2. Leaving that difference aside, we found that we could largely infer the overall structure of the bit-strings produced by a serialization specification without the need of a specification after studying a handful of serialization specifications in depth.

9 Reproducibility

The hexadecimal bit-strings discussed in this study can be recreated by the reader using the code files hosted on GitHub [149]. This GitHub repository contains a folder called *analysis* including the input data document from Figure 10 and the schema and code files for each binary serialization specification implementation discussed in Table 6 and Table 7. The repository includes a *Makefile* for serializing the input data document with each of the selected serialization specifications.

10 Future Work

Space-efficient benchmarks. We plan to run space-efficiency benchmarks involving the JSON-compatible binary serialization specifications discussed in this paper using a range of JSON documents

[149] `https://github.com/jviotti/binary-json-survey`

[18] differing in content, structure, and size. The goal of these space-efficiency benchmarks is dual. First, we want to understand what are the most space-efficient binary serialization alternatives to JSON at the time of this writing. More importantly, we want to understand what serialization specification characteristics and optimizations typically lead to more compact results and what are the space-related bottlenecks that the new generation of JSON-compatible binary serialization specifications need to solve to make a breakthrough in the context of space-efficiency.

Strict JSON-compatibility analysis. As discussed in subsection 3.1, we discarded binary serialization specifications that could not represent the *input data* JSON document from Figure 10 without changes. The fact that a serialization specification can encode the *input data* JSON document provides a loose guarantee that such serialization specification is JSON-compatible. Given the relevance of JSON at the time of this writing, we believe that it is important to formally analyze whether the serialization specifications discussed in this paper can represent *any* valid JSON document before claiming that they are JSON-compatible.

Formal schema evolution compatibility analysis. In subsection 7.2, we showcase a list of common schema transformations and try to determine the level of schema compatibility supported by the schema-driven serialization specifications discussed in this paper through simple test cases. We envision a formal analysis of the various schema languages and their schema transformation compatibility levels that can provide high-assurance and a more detailed view of what type of transformations are compatible under what contexts.

Schema semantic versioning. To the best of our knowledge, there is no human readable versioning scheme that can distingush between backwards and forwards compatible changes. Software libraries typically rely on *Semantic Versioning* [150] to succintly communicate whether a software library update is safe by distingushing between incompatible changes, backwards-compatible new functionality, and backwards-compatible bug fixes. We envision a similar versioning convention that is more applicable to schemas and distingushes between backwards, forwards, and fully compatible changes.

Schema-driven comparison metric. As discussed in subsection 1.5, whether a serialization specification is schema-driven is not a boolean characteristic. The schema-driven serialization specifications that we studied in this paper leverage their respective schemas to different degrees during the deserialization process. How much they leverage their schemas depends on the expressiveness of their schema languages and on the amount of metadata they embed into the bit-strings they produce. We can envision a metric that can be used to compare schema-driven serialization specifications in terms of *how much* schema-driven they are. We believe that such metric can formalize the understanding of why some schema-driven specifications are generally more space-efficient than others. We think that this metric is analogous to Big O-notation [37] from the context of algorithm analysis.

[150] `https://semver.org`

Acknowledgments and Disclosure of Funding

References

[1] 3GPP. 2021. *Evolved Universal Terrestrial Radio Access Network (E-UTRAN); S1 Application Protocol (S1AP)*. 3GPP. https://www.3gpp.org/ftp/Specs/archive/36_series/36.413/36413-g40.zip

[2] Aleem Akhtar. 2020. Role of Apache Software Foundation in Big Data Projects. arXiv:2005.02829 [cs.SE]

[3] Edman Anjos, Junhee Lee, and Srinivasa Rao Satti. 2016. SJSON: A succinct representation for JavaScript object notation documents. , 173–178 pages.

[4] Mansoureh Anvari, Mehdi Dehghan Takht, and Behrouz Sefid-Dashti. 2018. Thrift Service Composition: Toward Extending BPEL. In *Proceedings of the International Conference on Smart Cities and Internet of Things* (Mashhad, Iran) *(SCIOT '18)*. Association for Computing Machinery, New York, NY, USA, Article 13, 5 pages. https://doi.org/10.1145/3269961.3269973

[5] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Counting Types for Massive JSON Datasets. In *Proceedings of The 16th International Symposium on Database Programming Languages*. Association for Computing Machinery, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/3122831.3122837

[6] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Parametric schema inference for massive JSON datasets. *The VLDB Journal* 28, 4 (2019), 497–521.

[7] Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2019. Schemas and Types for JSON Data: From Theory to Practice. 2060–2063. https://doi.org/10.1145/3299869.3314032

[8] Rahwa Bahta and Mustafa Atay. 2019. Translating JSON Data into Relational Data Using Schema-Oblivious Approaches. In *Proceedings of the 2019 ACM Southeast Conference* (Kennesaw, GA, USA) *(ACM SE '19)*. Association for Computing Machinery, New York, NY, USA, 233–236. https://doi.org/10.1145/3299815.3314467

[9] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. 2009. *YAML Ain't Markup Language (YAML) Version 1.2*. YAML. https://yaml.org/spec/1.2/spec.html

[10] Amrit Kumar Biswal and Obada Almallah. 2019. *Analytical Assessment of Binary Data Serialization Techniques in IoT Context*. Master's thesis. Dipartimento di Elettronica, Informazione e Bioingegneria.

[11] Sebastian Bittl, Arturo Gonzalez, and Wolf Heidrich. 2014. Performance comparison of encoding schemes for ETSI ITS C2X communication systems.

[12] Daniele Bonetta and Matthias Brantner. 2017. FAD.Js: Fast JSON Data Access Using JIT-Based Speculative Optimizations. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1778–1789. https://doi.org/10.14778/3137765.3137782

[13] Tuhin Borgohain, Uday Kumar, and Sugata Sanyal. 2015. Survey of Operating Systems for the IoT Environment. arXiv:1504.02517 [cs.OS]

[14] C. Bormann and P. Hoffman. 2013. *Concise Binary Object Representation (CBOR)*. RFC. IETF. https://doi.org/10.17487/RFC7049

[15] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoč. 2017. JSON: Data Model, Query Languages and Schema Specification. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Chicago, Illinois, USA) *(PODS '17)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/3034786.3056120

[16] Anthony R. Bradley, Alexander S. Rose, Antonín Pavelka, Yana Valasatava, Jose M. Duarte, Andreas Prlić, and Peter W. Rose. 2017. MMTF—An efficient file format for the transmission, visualization, and analysis of macromolecular structures. *PLOS Computational Biology* 13, 6 (06 2017), 1–16. https://doi.org/10.1371/journal.pcbi.1005575

[17] Steven Braeger. 2016. *Universal Binary JSON Specification*. UBJSON. https://ubjson.org

[18] T. Bray. 2014. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC. IETF. https://doi.org/10.17487/RFC8259

[19] Tim Bray and C. M. Sperberg-McQueen. 1996. *Extensible Markup Language (XML)*. W3C Working Draft. W3C. https://www.w3.org/TR/WD-xml-961114.

[20] K. Brun-Laguna, T. Watteyne, S. Malek, Z. Zhang, C. Oroza, S. D. Glaser, and B. Kerkez. 2016. SOL: An end-to-end solution for real-world remote monitoring systems. In *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE, Valencia, Spain, 1–6. https://doi.org/10.1109/PIMRC.2016.7794864

[21] P. Bryan. 2013. *JavaScript Object Notation (JSON) Patch*. RFC. IETF. https://doi.org/10.17487/RFC6902

[22] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2013. Discovering Implicit Schemas in JSON Data. In *Web Engineering*, Florian Daniel, Peter Dolog, and Qing Li (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–83.

[23] Hanyang Cao, Jean-Rémy Falleri, Xavier Blanc, and Li Zhang. 2016. JSON Patch for Turning a Pull REST API into a Push. In *Service-Oriented Computing*, Quan Z. Sheng, Eleni Stroulia, Samir Tata, and Sami Bhiri (Eds.). Springer International Publishing, Cham, 435–449.

[24] Z. Cao, G. Hidalgo, T. Simon, S. E. Wei, and Y. Sheikh. 2021. OpenPose: Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43, 1 (2021), 172–186. https://doi.org/10.1109/TPAMI.2019.2929257

[25] David Carrera, Jonathan Rosales, and Gustavo A. 2018. Optimizing Binary Serialization with an Independent Data Definition Format. *International Journal of Computer Applications* 180 (03 2018), 15–18. https://doi.org/10.5120/ijca2018916670

[26] CCSDS. 2016. *Space Link Extension — Return Channel Frames Service Specification*. CCSDS. https://public.ccsds.org/Pubs/911x2b3.pdf

[27] V.G. Cerf. 1969. *ASCII format for network interchange*. STD. IETF. https://doi.org/10.17487/RFC0020

[28] E. Chandra and A. I. Kistijantoro. 2017. Database development supporting offline update using CRDT: (Conflict-free replicated data types). In *2017 International Conference on Advanced Informatics, Concepts, Theory, and Applications (ICAICTA)*. IEEE, Denpasar, 1–6. https://doi.org/10.1109/ICAICTA.2017.8090961

[29] Bao Rong Chang, Hsiu-Fen Tsai, Yo-Ai Wang, and Chin-Fu Kuo. 2015. Intelligent Adaptation to In-Cloud NoSQL Database Remote Backup between Data Centers. In *Proceedings of the ASE BigData & SocialInformatics 2015* (Kaohsiung, Taiwan) *(ASE BD&SI '15)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/2818869.2818892

[30] S. V. Chekanov. 2013. Next generation input-output data format for HEP using Google's protocol buffers. arXiv:1306.6675 [cs.CE]

[31] Alberto Hernandez Chillon, Diego Sevilla Ruiz, and Jesus Garcia Molina. 2020. Deimos: a model-based NoSQL data generation language. In *International Conference on Conceptual Modeling*. Springer, Springer, Vienna, Austria, 151–161.

[32] The Unicode Consortium. 2019. *The Unicode Standard, Version 12.1.0*. Standard. The Unicode Consortium, Mountain View, CA.

[33] Osborne Computer Corporation. 1983. *Osborne EXECUTIVE Reference Guide*. Osborne Computer Corporation, San Francisco, CA.

[34] Douglas Crockford. 2006. *JSON*. IETF. https://tools.ietf.org/html/draft-crockford-jsonorg-json-00

[35] Douglas Crockford. 2011. *Douglas Crockford: The JSON Saga*. Youtube. https://www.youtube.com/watch?v=-C-JoyNuQJs

[36] Javier Luis Cánovas Izquierdo and Jordi Cabot. 2016. JSONDiscoverer: Visualizing the schema lurking behind JSON documents. *Knowledge-Based Systems* 103 (2016), 52–55. https://doi.org/10.1016/j.knosys.2016.03.020

[37] P Danziger. 2010. Big O notation. *Source internet: http://www.scs.ryerson.ca/ mth110/Handouts/PD/bigO.pdf, Retrieve: April* 1, 1 (2010), 6.

[38] ISO/IEC JTC 1/SC 34 Document description and processing languages. 1986. *Standard Generalized Markup Language (SGML)*. Standard. International Organization for Standardization.

[39] P. Deutsch. 1996. *GZIP file format specification version 4.3*. RFC. https://doi.org/10.17487/RFC1952

[40] Olivier Dubuisson and Philippe Fouquart. 2001. *ASN.1: Communication between Heterogeneous Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[41] ECMA. 2017. *ECMA-404: The JSON Data Interchange Syntax*. ECMA, Geneva, CH. https://www.ecma-international.org/publications/standards/Ecma-404.htm

[42] ECMA. 2021. *ECMA-262: ECMAScript 2021 language specification*. ECMA, Geneva, CH. https://www.ecma-international.org/publications/standards/Ecma-262.htm

[43] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2017. Example-Driven Web API Specification Discovery. In *Modelling Foundations and Applications*, Anthony Anjorin and Huáscar Espinoza (Eds.). Springer International Publishing, Cham, 267–284.

[44] Malin Eriksson and Victor Hallberg. 2011. Comparison between JSON and YAML for data serialization. *The School of Computer Science and Engineering Royal Institute of Technology* 0, 0 (2011), 1–25.

[45] Paola Espinoza-Arias, Daniel Garijo, and Oscar Corcho. 2020. Mapping the Web Ontology Language to the OpenAPI Specification. In *International Conference on Conceptual Modeling*. Springer, Springer, Vienna, Austria, 117–127.

[46] Jianhua Feng and Jinhong Li. 2013. Google protocol buffers research and application in online game. In *IEEE Conference Anthology*. IEEE, China, 4.

[47] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1*. RFC. https://doi.org/10.17487/RFC2616

[48] Roy T Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, Irvine, California, US.

[49] Kathleen Fisher and Robert Gruber. 2005. PADS: A Domain-Specific Language for Processing Ad Hoc Data. *SIGPLAN Not.* 40, 6 (June 2005), 295–304. https://doi.org/10.1145/1064978.1065046

[50] Ivan Flores. 1963. *The Logic Of Computer Arithmetic*. Prentice-Hall, Newport Coast, CA, USA, 24.

[51] Working Group for Floating-Point Arithmetic. 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* 1, 1 (2019), 1–84.

[52] The Apache Software Foundation. 2012. *Apache Avro™ 1.10.0 Specification*. The Apache Software Foundation. https://avro.apache.org/docs/current/spec.html

[53] N. Freed and N. Borenstein. 1996. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. RFC. IETF. https://doi.org/10.17487/RFC2045

[54] A. A. Frozza, R. d. S. Mello, and F. d. S. d. Costa. 2018. An Approach for Schema Extraction of JSON and Extended JSON Document Collections. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, Salt Lake City, Utah, 356–363.

[55] Michael Fruth, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. 2020. Challenges in Checking JSON Schema Containment over Evolving Real-World Schemas. In *International Conference on Conceptual Modeling*. Springer, Springer, Vienna, Austria, 220–230.

[56] Sadayuki Furuhashi, Satoshi Tagomori, Yuichi Tanikawa, Stephen Colebourne, Stefan Friesel, René Kijewski, Michael Cooper, Uenishi Kota, and Gabe Appleton. 2013. *MessagePack Specification*. MessagePack. `https://github.com/msgpack/msgpack/blob/master/spec.md`

[57] Y. Gan and C. Delimitrou. 2018. The Architectural Implications of Cloud Microservices. *IEEE Computer Architecture Letters* 17, 2 (2018), 155–158. `https://doi.org/10.1109/LCA.2018.2839189`

[58] Bruno Gil and Paulo Trezentos. 2011. Impacts of Data Interchange Formats on Energy Consumption and Performance in Smartphones. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication* (Lisboa, Portugal) *(OSDOC '11)*. Association for Computing Machinery, New York, NY, USA, 1–6. `https://doi.org/10.1145/2016716.2016718`

[59] Google. 2020. *Protocol Buffers Version 3 Language Specification*. Google. `https://developers.google.com/protocol-buffers/docs/reference/proto3-spec`

[60] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. 2003. SOAP Version 1.2. *W3C recommendation* 24 (2003), 12.

[61] Andrew Habib, Avraham Shinnar, Martin Hirzel, and Michael Pradel. 2019. Type Safety with JSON Subschema. arXiv:1911.12651 [cs.PL]

[62] P. Hallam-Baker. 2018. *Binary Encodings for JavaScript Object Notation: JSON-B, JSON-C, JSON-D*. Technical Report. IETF. `https://tools.ietf.org/html/draft-hallambaker-jsonbcd-11#section-6`

[63] Jean Carlo Hamerski, Anderson RP Domingues, Fernando G Moraes, and Alexandre Amory. 2018. Evaluating serialization for a publish-subscribe based middleware for MPSoCs. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, IEEE, Bordeaux, 773–776.

[64] IBM. 1972. *IBM FORTRAN Program Products for OS and the CMS Component of VM/370 General Information*. IBM. `http://bitsavers.trailing-edge.com/pdf/ibm/370/fortran/GC28-6884-0_IBM_FORTRAN_Program_Products_for_OS_and_CMS_General_Information_Jul72.pdf`

[65] IBM. 1986. *Code Page CPGID 00037*. IBM. `ftp://ftp.software.ibm.com/software/globalization/gcoc/attachments/CP00037.pdf`

[66] Pietro Incardona, Antonio Leo, Yaroslav Zaluzhnyi, Rajesh Ramaswamy, and Ivo F Sbalzarini. 2019. OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. *Computer Physics Communications* 241 (2019), 155–177.

[67] their environments ISO/IEC JTC 1/SC 22 Programming languages and system software interfaces. 2002. *Z formal specification notation — Syntax, type system and semantics*. Standard. International Organization for Standardization.

[68] their environments ISO/IEC JTC 1/SC 22 Programming languages and system software interfaces. 2011. *Information technology — Programming languages — C++*. Standard. International Organization for Standardization.

[69] data elements ISO/TC 154, Processes, industry documents in commerce, and administration. 2004. *Data elements and interchange formats – Information interchange – Representation of dates and times*. Standard. International Organization for Standardization, Geneva, CH.

[70] Massimiliano Izzo. 2016. *The JSON-Based Data Model*. Springer International Publishing, Cham, 39–48. https://doi.org/10.1007/978-3-319-31241-5_3

[71] K. Jahed and J. Dingel. 2019. Enabling Model-Driven Software Development Tools for the Internet of Things. In *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*. IEEE, Montreal, QC, Canada, 93–99. https://doi.org/10.1109/MiSE.2019.00022

[72] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. Association for Computing Machinery, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[73] S. Josefsson. 2006. *The Base16, Base32, and Base64 Data Encodings*. RFC. IETF. https://doi.org/10.17487/RFC4648

[74] Meike Klettke, Uta Störl, and Stefanie Scherzinger. 2015. Schema extraction and structural outlier detection for JSON-based nosql data stores. In *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*, Thomas Seidl, Norbert Ritter, Harald Schöning, Kai-Uwe Sattler, Theo Härder, Steffen Friedrich, and Wolfram Wingerath (Eds.). Gesellschaft für Informatik e.V., Bonn, 425–444.

[75] G. Klyne and C. Newman. 2002. *Date and Time on the Internet: Timestamps*. RFC. IETF. https://doi.org/10.17487/RFC3339

[76] Pavel Kyurkchiev. 2015. Integrating a System for Symbol Programming of Real Processes with a Cloud Service. , 8 pages.

[77] M. Kálmán. 2013. ProtoML: A rule-based validation language for Google Protocol Buffers. In *8th International Conference for Internet Technology and Secured Transactions (ICITST-2013)*. IEEE, London, UK, 188–193. https://doi.org/10.1109/ICITST.2013.6750189

[78] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. , 941–960 pages.

[79] John Larmouth. 1999. *ASN.1 Complete*. Open Systems Solutions, 1748 Millstream Way, Henderson, NV 89014, Chapter 4.2 LWER - Light-Weight Encoding Rules, 316–319.

[80] P. Leach, M. Mealling, and R. Salz. 2005. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC. IETF. https://doi.org/10.17487/RFC4122

[81] H. Ledoux, G.A.K. Arroyo Ohori, K. Kavisha, B. Dukai, A. Labetski, and S. Vitalis. 2019. CityJSON: a compact and easy-to-use encoding of the CityGML data model. , urn:issn:2363–7501 pages.

[82] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proc. VLDB Endow.* 10, 10 (June 2017), 1118–1129. https://doi.org/10.14778/3115404.3115416

[83] Sina Madani and Dimitrios S Kolovos. 2016. Re-implementing Apache Thrift using model-driven engineering technologies: An experience report. In *CEUR Workshop Proceedings 1403*. York, The University of York, Heslington, York, YO10 5DD, United Kingdom, 149–156.

[84] K Maeda. 2012. Performance evaluation of object serialization libraries in XML, JSON and binary formats. , 177–182 pages.

[85] Benjamin Maiwald, Benjamin Riedle, and Stefanie Scherzinger. 2019. What Are Real JSON Schemas Like?. In *Advances in Conceptual Modeling*, Giancarlo Guizzardi, Frederik Gailly, and Rita Suzana Pitangueira Maciel (Eds.). Springer International Publishing, Cham, 95–105.

[86] Dwight Merriman, Stephen Steneker, Hannes Magnusson, Luke Lovett, Kevin Albertson, Kay Kim, and Allison Reinheimer Moore. 2020. *BSON Specification Version 1.1*. MongoDB. http://bsonspec.org/spec.html

[87] Microsoft. 2018. *Bond Compiler 0.12.0.1*. Microsoft. https://microsoft.github.io/bond/manual/compiler.html

[88] N. Mitra. 1994. Efficient encoding rules for ASN.1-based protocols. *AT T Technical Journal* 73, 3 (1994), 80–93. https://doi.org/10.1002/j.1538-7305.1994.tb00590.x

[89] T. Mizrahi, J. Fabini, and A. Morton. 2020. *Guidelines for Defining Packet Timestamps*. RFC. IETF. https://doi.org/10.17487/RFC8877

[90] Florian Morath. 2018. Implementing a Distributed Reliable Database.

[91] Muhammad Wasim Bhatti, F. Hayat, N. Ehsan, A. Ishaque, S. Ahmed, and E. Mirza. 2010. A methodology to manage the changing requirements of a software project. In *2010 International Conference on Computer Information Systems and Industrial Management Applications (CISIM)*. IEEE, Krakow, Poland, 319–322. https://doi.org/10.1109/CISIM.2010.5643642

[92] Gerald Neufeld and Son Vuong. 1992. An overview of ASN.1. *Computer Networks and ISDN Systems* 23, 5 (1992), 393–415. https://doi.org/10.1016/0169-7552(92)90014-H

[93] Mattias Nordahl and Boris Magnusson. 2015. A lightweight Data Interchange Format for Internet of Things in the PalCom Middleware Framework. , 284–291 pages.

[94] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. 2009. Comparison of JSON and XML data interchange formats: a case study. *Caine* 9 (2009), 157–162.

[95] Iakushkin Oleg, Ruslan Sevostyanov, Alexander Degtyarev, P. E. Karpiy, E. G. Kuzevanova, A. A. Kitaeva, and S. A. Sergiev. 2019. Position Tracking in 3D Space Based on a Data of a Single Camera. In *Computational Science and Its Applications – ICCSA 2019*, Sanjay Misra, Osvaldo Gervasi, Beniamino Murgante, Elena Stankova, Vladimir Korkhov, Carmelo Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, and Eufemia Tarantino (Eds.). Springer International Publishing, Cham, 772–781.

[96] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. 2013. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems* (Roma, Italy) *(SenSys '13)*. Association for Computing Machinery, New York, NY, USA, Article 10, 14 pages. https://doi.org/10.1145/2517351.2517354

[97] L. Opyrchal and A. Prakash. 1999. Efficient object serialization in Java. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*. IEEE, Austin, TX, USA, 96–101. https://doi.org/10.1109/ECMDD.1999.776421

[98] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter before You Parse: Faster Analytics on Raw Data with Sparser. *Proc. VLDB Endow.* 11, 11 (July 2018), 1576–1589. https://doi.org/10.14778/3236187.3236207

[99] Jean Paoli, François Yergeau, Tim Bray, Eve Maler, and Michael Sperberg-McQueen. 2006. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C Recommendation. W3C. https://www.w3.org/TR/2006/REC-xml-20060816/.

[100] Akshay Parashar, Payal Anand, and Arun Abraham. 2020. Performance Analysis and Optimization of Serialization Techniques for Deep Neural Networks. In *Computer Vision, Pattern Recognition, Image Processing, and Graphics*, R. Venkatesh Babu, Mahadeva Prasanna, and Vinay P. Namboodiri (Eds.). Springer Singapore, Singapore, 250–260.

[101] Dinesh Parimi, William Zhao, and Jerry Zhao. 2019. *Datacenter Tax Cuts: Improving WSC Efficiency Through Protocol Buffer Acceleration*. University of California, Berkeley.

[102] E. Peltonen, E. Lagerspetz, P. Nurmi, and S. Tarkoma. 2016. Too big to mail: On the way to publish large-scale mobile analytics data. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, Washington, DC, USA, 2374–2377. https://doi.org/10.1109/BigData.2016.7840871

[103] Shangfu Peng, Jagan Sankaranarayanan, and Hanan Samet. 2018. DOS: A Spatial System Offering Extremely High-Throughput Road Distance Computations. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Seattle, Washington) *(SIGSPATIAL '18)*. Association for Computing Machinery, New York, NY, USA, 199–208. https://doi.org/10.1145/3274895.3274898

[104] Bo Petersen, Henrik Bindner, Shi You, and Bjarne Poulsen. 2017. Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the internet of things. In *2017 Computing Conference*. IEEE, IEEE, London, UK, 1339–1346.

[105] Dušan Petković. 2020. Non-Native Techniques for Storing JSON Documents into Relational Tables. In *Proceedings of the 22nd International Conference on Information Integration and Web-Based Applications & Services* (Chiang Mai, Thailand) *(iiWAS '20)*. Association for Computing Machinery, New York, NY, USA, 16–20. https://doi.org/10.1145/3428757.3429103

[106] F. Pezoa, J.L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. 2016. Foundations of JSON schema. , 263–273 pages.

[107] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić. 2016. Performance evaluation of using Protocol Buffers in the Internet of Things communication. In *2016 International Conference on Smart Systems and Technologies (SST)*. International Conference on Smart Systems and Technologies, Osijek, HR, 261–265.

[108] M. A. Pradana, A. Rakhmatsyah, and A. A. Wardana. 2019. Flatbuffers Implementation on MQTT Publish/Subscribe Communication as Data Delivery Format. In *2019 6th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*. IEEE, Bandung, Indonesia, 142–146. https://doi.org/10.23919/EECSI48112.2019.8977050

[109] Tom Preston-Werner and Pradyun Gedam. 2020. *TOML v1.0.0-rc.2*. TOML. https://toml.io/en/v1.0.0-rc.2

[110] D. P. Proos and N. Carlsson. 2020. Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV. In *2020 IFIP Networking Conference (Networking)*. IEEE, Paris, France, 10–18.

[111] Ricardo Queirós. 2014. JSON on Mobile: is there an Efficient Parserl. In *3rd Symposium on Languages, Applications and Technologies (OpenAccess Series in Informatics (OASIcs), Vol. 38)*, Maria Jo ao Varanda Pereira, José Paulo Leal, and Alberto Sim oes (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 93–100. https://doi.org/10.4230/OASIcs.SLATE.2014.93

[112] R. Rivest. 1992. *The MD5 Message-Digest Algorithm*. RFC. IETF. https://doi.org/10.17487/RFC1321

[113] T. Rix, K. Detken, and M. Jahnke. 2016. Transformation between XML and CBOR for network load reduction. In *2016 3rd International Symposium on Wireless Systems within the Conferences on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS-SWS)*. IEEE, Offenburg, Germany, 106–111. https://doi.org/10.1109/IDAACS-SWS.2016.7805797

[114] John F. Roddick. 1995. A Survey of Schema Versioning Issues for Database Systems. *Information and Software Technology* 37 (1995), 383–393.

[115] Kristina Sahlmann, Alexander Lindemann, and Bettina Schnor. 2018. Binary Representation of Device Descriptions: CBOR versus RDF HDT. *Technische Universität Braunschw: Braunschweig, Germany* 0, 0 (2018), 4.

[116] Tatu Saloranta. 2017. *Efficient JSON-compatible binary format: "Smile"*. FasterXML. https://github.com/FasterXML/smile-format-specification/blob/master/smile-specification.md

[117] ITU Telecommunication Standardization Sector. 1984. *Message handling systems: presentation transfer syntax and notation*. Standard. ITU Telecommunication Standardization Sector.

[118] ITU Telecommunication Standardization Sector. 2015. *Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Standard. ITU Telecommunication Standardization Sector.

[119] ITU Telecommunication Standardization Sector. 2015. *ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*. Standard. ITU Telecommunication Standardization Sector.

[120] ITU Telecommunication Standardization Sector. 2021. *ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. Standard. ITU Telecommunication Standardization Sector.

[121] B. Sefid-Dashti and S. M. Babamir. 2016. Toward extending apache thrift open source to alleviate SOAP service consumption. In *2016 2nd International Conference on Open Source Software Computing (OSSCOM)*. IEEE, Beirut, 1–6. https://doi.org/10.1109/OSSCOM.2016.7863681

[122] Y. Shafranovich. 2005. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC. IETF. https://doi.org/10.17487/RFC4180

[123] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[124] Z. Shelby, K. Hartke, and C. Bormann. 2014. *The Constrained Application Protocol (CoAP)*. RFC. IETF. https://doi.org/10.17487/RFC7252

[125] UNIX International Programming Languages SIG. 1993. *DWARF Debugging Information Format rev 2.0.0*. UNIX International, Waterview Corporate Center 20 Waterview Boulevard Parsippany, NJ 07054, 70.

[126] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook White Paper* 5 (2007), 8.

[127] Audie Sumaray and S Makki. 2012. A comparison of data serialization formats for optimal efficiency on a mobile platform. , 6 pages.

[128] Pavel Contoˇsˇ and Martin Svoboda. 2020. JSON Schema Inference Approaches. In *Advances in Conceptual Modeling: ER 2020 Workshops CMAI, CMLS, CMOMM4FAIR, CoMoNoS, EmpER, Vienna, Austria, November 3–6, 2020, Proceedings*. Springer Nature, Springer, Vienna, Austria, 173.

[129] Devang Swami and Bibhudatta Sahoo. 2018. Storage Size Estimation for Schemaless Big Data Applications: A JSON-Based Overview. In *Intelligent Communication and Computational Technologies*, Yu-Chen Hu, Shailesh Tiwari, Krishn K. Mishra, and Munesh C. Trivedi (Eds.). Springer Singapore, Singapore, 315–323.

[130] OASIS Web Services Business Process Execution Language (WSBPEL) TC. 2007. *Web Services Business Process Execution Language Version 2.0*. OASIS. OASIS. 1–264 pages. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[131] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2014. *ASN.1 encoding rules — Part 7: Specification of Octet Encoding Rules (OER)*. Standard. International Organization for Standardization, Geneva, CH.

[132] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2015. *Abstract Syntax Notation One (ASN.1): Specification of basic notation*. Standard. International Organization for Standardization, Geneva, CH.

[133] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2015. *ASN.1 encoding rules — Part 4: XML Encoding Rules (XER)*. Standard. International Organization for Standardization, Geneva, CH.

[134] ISO/IEC JTC 1/SC 6 Telecommunications and information exchange between systems. 2018. *ASN.1 encoding rules — Part 8: Specification of JavaScript Object Notation Encoding Rules (JER)*. Standard. International Organization for Standardization, Geneva, CH.

[135] Mark Tullsen, Lee Pike, Nathan Collins, and Aaron Tomb. 2018. Formal Verification of a Vehicle-to-Vehicle (V2V) Messaging System. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 413–429.

[136] Borey Uk, David Konam, Clément Passot, Milan Erdelj, and Enrico Natalizio. 2018. Implementing a System Architecture for Data and Multimedia Transmission in a Multi-UAV System. In *Wired/Wireless Internet Communications*, Kaushik Roy Chowdhury, Marco Di Felice, Ibrahim Matta, and Bo Sheng (Eds.). Springer International Publishing, Cham, 246–257.

[137] Y. A. Ushakov, P. N. Polezhaev, A. E. Shukhman, M. V. Ushakova, and M. V. Nadezhda. 2018. Split Neural Networks for Mobile Devices. In *2018 26th Telecommunications Forum (TELFOR)*. IEEE, Belgrade, 420–425. https://doi.org/10.1109/TELFOR.2018.8612133

[138] Jordi van Liempt. 2020. *CityJSON: does (file) size matter?* Master's thesis. Department of Urbanism, Faculty of the Built Environment & Architecture.

[139] Wouter van Oortmerssen. 2014. *FlatBuffers: Writing a Schema*. Google. https://google.github.io/flatbuffers/flatbuffers_guide_writing_schema.html

[140] Wouter van Oortmerssen. 2017. *FlexBuffers*. Google. https://google.github.io/flatbuffers/flexbuffers.html

[141] J. Vanura and P. Kriz. 2018. Perfomance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats. , 166–175 pages.

[142] Kenton Varda. 2013. *Cap'n Proto Schema Language*. Sandstorm. https://capnproto.org/language.html

[143] Santiago Vargas, Utkarsh Goel, Moritz Steiner, and Aruna Balasubramanian. 2019. Characterizing JSON Traffic Patterns on a CDN. In *Proceedings of the Internet Measurement Conference* (Amsterdam, Netherlands) *(IMC '19)*. Association for Computing Machinery, New York, NY, USA, 195–201. https://doi.org/10.1145/3355369.3355594

[144] Kunal Waghray. 2020. JSON Schema Matching: Empirical Observations. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2887–2889. https://doi.org/10.1145/3318464.3384417

[145] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc., San Diego, CA, 7675–7684. https://proceedings.neurips.cc/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf

[146] P. Wehner, C. Piberger, and D. Göhringer. 2014. Using JSON to manage communication between services in the Internet of Things. In *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, Montpellier, 1–4.

[147] M. Westhead, T. Wen, and R. Carroll. 2003. Describing data on the grid. In *Proceedings. First Latin American Web Congress*. IEEE, Phoenix, AZ, USA, 134–140. https://doi.org/10.1109/GRID.2003.1261708

[148] GLSTJ Whittaker. 2013. Improving performance of schemaless document storage in PostgreSQL using BSON.

[149] Canggih Wibowo. 2011. Evaluation of Protocol Buffers as Data Serialization Format for Microblogging Communication.

[150] Jason A Wilkins and Jaakko Järvi. 2016. drys: A Version Control System for Rapid Iteration of Serialization Protocols.

[151] Martin Wischenbart, Stefan Mitsch, Elisabeth Kapsammer, Angelika Kusel, Birgit Pröll, Werner Retschitzegger, Wieland Schwinger, Johannes Schönböck, Manuel Wimmer, and Stephan Lechner. 2012. User Profile Integration Made Easy: Model-Driven Extraction and Transformation of Social Network Schemas. In *Proceedings of the 21st International Conference on World Wide Web* (Lyon, France) *(WWW '12 Companion)*. Association for Computing Machinery, New York, NY, USA, 939–948. https://doi.org/10.1145/2187980.2188227

[152] A. Wright, H. Andrews, and B. Hutton. 2019. *JSON Schema: A Media Type for Describing JSON Documents*. Technical Report. IETF. https://tools.ietf.org/html/draft-handrews-json-schema-02

[153] Qianchuan Ye and Benjamin Delaware. 2019. A Verified Protocol Buffer Compiler. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Cascais, Portugal) *(CPP 2019)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/3293880.3294105

[154] Kamir Yusof and Mustafa Man. 2017. Efficiency of JSON for data retrieval in big data. *Indonesian Journal of Electrical Engineering and Computer Science* 7 (07 2017), 250–262. https://doi.org/10.11591/ijeecs.v7.i1.pp250-262

[155] Yaroslav Zaluzhnyi. 2016. *Serialization and deserialization of complex data structures, and applications in high performance computing*. Ph.D. Dissertation. Technische Universität Dresden.