Working with Dataset-JSON using SAS©

Lex Jansen, CDISC

ABSTRACT

The Operational Data Model (ODM) is a vendor neutral, platform independent data exchange format, intended primarily for interchange and archival of clinical study data pertaining to individual subjects, aggregated collections of subjects, and integrated research studies. ODM provides the foundation for most CDISC Data Exchange Standards, such as Define-XML.

CDISC is in the late-stage development of the much-anticipated ODM v2.0 update. ODM v2.0 will include the specification of Dataset-JSON, an efficient and modern exchange format for data which addresses many of the limitations of SAS v5 XPT files.

JSON representations for exchange standards are widely used in today's architectures. In RESTful web services, JSON is often the preferred format for the service response, due to its compactness and ease of use in mobile applications. Other standards used in healthcare, such as HL7-FHIR support JSON as well as XML, together with other formats such as RDF.

This paper will show how SAS can work with Dataset-JSON, both reading and writing. We will discuss the native SAS JSON engine, but also the use of PROC LUA.

Keywords: CDISC, Operational Data Model, Dataset-JSON, Define-XML, define.xml, metadata

Note: 2023-07-24: This paper is an update to the paper that was presented at PharmaSUG 2022.

INTRODUCTION

In the United States, the approval process for regulated human and animal health products requires the submission of data from clinical trials and other studies as expressed in the Code of Federal Regulations (CFR). The FDA established the regulatory basis for wholly electronic submission of data in 1997 with the publication of regulations on the use of electronic records in place of paper records (21 CFR Part 11). In 1999, the FDA standardized the submission of clinical and non-clinical data using the SAS Version 5 XPORT Transport Format and the submission of metadata using Portable Document Format (PDF), respectively [1]. In 2005, the Study Data Specifications published by the FDA included the recommendation that data definitions (metadata) be provided as a Define-XML file.

The SAS V5 Transport format dates from 1989 and was first available as part of SAS version 5. Since that time, there have been many changes in the industry with respect to the process for submissions and the approaches to data curation and manipulation – but the data submission is still the same. Although SAS V5 Transport format is an open specification it is owned and developed by SAS, and it is not an extensible modern technology.

It has been recognized that the ASCII-based SAS Version 5 XPORT transport format has some limitations and issues [2]:

Technical limitations

- Limited variable types: the current data formats supported are limited to US ASCII (for Character formats) and IBM INTEGER and DOUBLE (for Numeric formats).
- Only supports US ASCII Character encoding. No multibyte characters are possible; this requires translation and/or transcription from the source data. [1]

---

[1] In previous versions of the SAS Version 5 XPORT transport format SAS mentioned "All character data are stored in ASCII, regardless of the operating system". In the October 2021 edition of the specification this was updated to "All character data is stored in the Windows encoding that is compatible with the SAS session encoding that is used to create the file". Although technically character data can be

- Variable names are restricted in terms of width and format. Variable names must be alphanumeric, Variable names are limited to a maximum length of 8 characters, Variable labels are restricted to a maximum length of 40 characters
- Character variable data widths are limited to 200 characters

Storage limitations

- The SAS Version 5 XPORT transport format has a highly inefficient use of storage space. There is often empty space for columns allocated, but not used by data and this can lead up to 70% wasted space. This inefficiency forces sponsors to re-size the lengths of character variables to be compliant with FDA rule. [3]
- The inability to compress datasets leads to significant file logistical issues, due to the requirement that the maximum size of the files is 5 Gigabytes or smaller

Structural limitations

- Two-dimensional "flat" data structure for hierarchical/multi-relational "round" data
- Lack of robust information model
- The SAS Version 5 XPORT transport format is not an extensible modern technology

SAS created the SAS V8 transport format to address some of the issues raised as part of the FDA Public Meeting on Study Data Exchange in 2012. The macros to generate the expanded format were released in 2012 and are supported in versions of SAS 8.2 and above [4]. Some of the currently held observations by those using SAS V5 transport format have been addressed in the SAS V8 transport format, e.g. longer character fields, longer names and labels. However, the updated format does not address the other issues and concerns.

In April 2014 CDISC published the final version 1.0 of the Dataset-XML standard [5]. Dataset-XML defines an ODM-based standard format for transporting tabular dataset data in XML between any two entities. That is, in addition to supporting the transport of datasets as part of a submission to the FDA, it may also be used to facilitate other data interchange use cases. Dataset-XML addresses the limitations of the SAS V5 XPORT transport format.

In response to the development of Dataset-XML, the Center for Drug Evaluation and Research (CDER) and the Center for Biologics Evaluation and Research (CBER) of the U.S. Food and Drug Administration (FDA) released a notice on 27 November 2013 of their intent to begin a pilot project to evaluate Dataset-XML. In the notice, it was highlighted that "although SAS Transport has been a reliable exchange format for many years, it is not an extensible modern technology," and that "FDA is announcing an invitation to sponsors to participate in this pilot project to evaluate the Dataset-XML transport format."

The objective of this pilot was to test the transport functionality of Dataset-XML, which included ensuring that data integrity was maintained, and that Dataset-XML format would support longer variable names, labels.

In April 2015, the FDA published a report to communicate the Dataset-XML pilot project findings [6]. The report mentions the following conclusions:

- Dataset-XML can transport data and maintain data integrity.

- The Dataset-XML transport format can facilitate a longer variable name (>8 characters), a longer label name (>40 characters) and longer text field (>200 characters).

- Dataset-XML requires stricter encoding in data.

- Dataset-XML requires consistency between datasets and Define.xml.

---

stored in encodings other than ASCII, still there is no method of conveying encoding information other than documenting it with the delivery of the transport file.

- Based on the file size observations, Dataset-XML produced much larger file sizes than XPORT, which may impact the Electronic Submissions Gateway (ESG) and may lead to file storage issues., and text fields.

Although the pilot was successful, the observation about the Dataset-XML file size seemed an obstacle for the acceptance of this new format. To this day, sponsors are still submitting their data in regulatory submissions in SAS Version 5 XPORT transport format.

Another concern raised about Dataset-XML has been that the metadata is completely separated from the data. To be able to process a Dataset-XML file one always needs the accompanying Define-XML document.

Dataset-JSON was adapted from the Dataset-XML Version 1 specification but uses JSON format. Like Dataset-XML, each Dataset-JSON file is connected to the Define-XML document, containing detailed information about the metadata. One aim of Dataset-JSON is to address as many of the relevant requirements in the PHUSE 2017 Transport for the Next Generation paper as possible, including the efficient use of storage space. Dataset-JSON files contain basic information about dataset variables, so that it is possible to have a simple view of a dataset contents without a need of a Define-XML document. Dataset-JSON files are also much smaller in size compared to SAS Version 5 XPORT files and dataset-XML files.

JSON is very well positioned to play a role in digital transition from a file format (XPT) to an API-based communication protocol. The first wave of APIs, called Web Services back then, was based on XML (SOAP being the most notable representative). Nowadays SOAP and XML are seen as being too heavy and too clunky for API use. Most modern web services use JSON as a data exchange format due to the speed and agility it offers. This does not mean that JSON is "better" than XML for APIs. While XML is a lot bulkier due to all its tags and nodes, it does offer a clearer structure of what each value or piece of data represents. In general, for web services or APIs, it comes down to what the average data transfer looks like. JSON is great when you are sending simple data and just want to do it fast and efficiently. JSON allows APIs to represent structured data in a way that is a better fit for the conceptual universe that most developers live in today.

A SHORT INTRODUCTION TO JSON

JavaScript Object Notation (JSON) is lightweight, text-based, language-independent syntax for defining data interchange formats [7][8]. JSON defines a small set of structuring rules for the portable representation of structured data. A JSON text is a sequence of tokens formed from Unicode code points that conforms to the JSON value grammar.

The set of tokens includes:

- Structural tokens: [and] (square brackets), { and } (curly brackets), : (colon), and , (comma)
- Literal token names: `true`, `false` and `null`

Insignificant whitespace (character tabulation, line feed, carriage return, space) is allowed before or after any token, but not within any token.

A JSON value can be an *object*, *array*, *number*, *string*, `true`, `false`, or `null`.
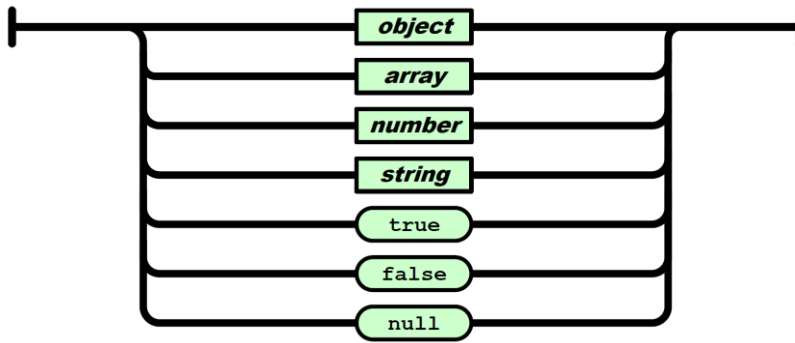
Figure 1 JSON Values

An **object** structure is represented as a pair of curly bracket tokens surrounding zero or more name/value pairs. A name is a *string*. A single colon token follows each name, separating the name from the *value*. A single comma token separates a *value* from a following name. The JSON syntax does not impose any restrictions on the *strings* used as names, does not require that name *strings* be unique, and does not assign any significance to the ordering of name/value pairs.
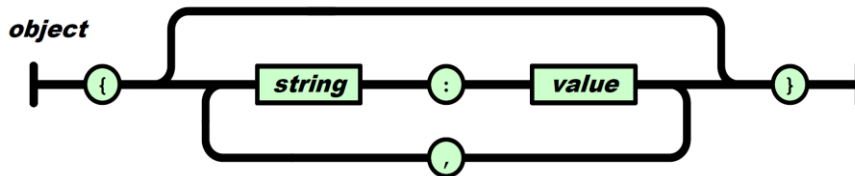

Figure 2 JSON object

Example 1: Invalid "JSON" object – no quotes around "name" and "label"

```
{
  name: "STUDYID",
  label: "Study Identifier"
}
```

Example 2: invalid "JSON" object – usage of single quotes instead of double quotes

```
{
  'name': 'STUDYID',
  'label': 'Study Identifier'
}
```

Example 3: Valid JSON object

```
{
  "name": "STUDYID",
  "label": "Study Identifier"
}
```

An **array** structure is a pair of square bracket tokens surrounding zero or more *values*. The *values* are separated by commas. The JSON syntax does not define any specific meaning to the ordering of the *values*. However, the JSON array structure is often used in situations where there is some semantics to the ordering. There is no requirement that the values in an array be of the same type.
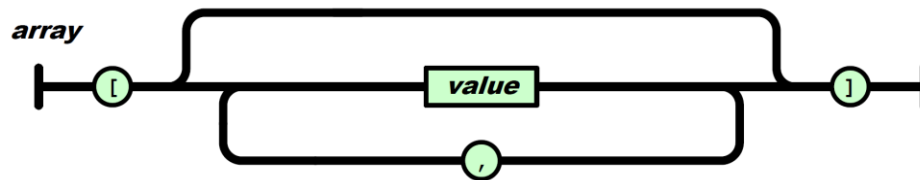

Figure 3 JSON array

Example 4: Invalid "JSON" array – missing values incorrectly represented

```
"itemData": [
    [1, "MyStudy", "001", "DM", ,    "BLACK"],
    [2, "MyStudy", "002", "DM", 26, ],
    ...
]
```

Example 5: valid JSON array – for missing values use null (numeric) or an empty string (character)

```
"itemData": [
    [1, "MyStudy", "001", "DM", null, "BLACK"],
    [2, "MyStudy", "002", "DM", 26, ""],
    ...
]
```

A **number** is a sequence of decimal digits with no superfluous leading zero. It may have a preceding minus sign. It may have a fractional part prefixed by a decimal point. It may have an exponent, prefixed by e or E and optionally + or –.
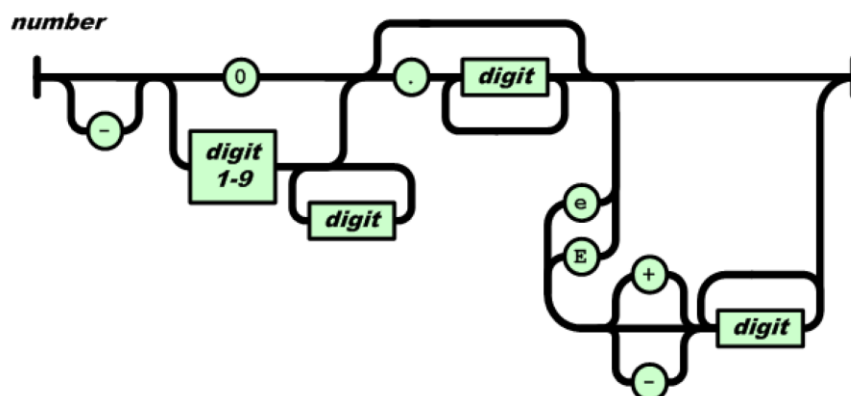

Figure 4 JSON number

A **string** is a sequence of Unicode code points wrapped with quotation marks (double quotes). All code points may be placed within the quotation marks except for the code points that must be escaped:

- Quotation mark: \"
- Backslash: \\
- Forward slash: \/
- Backspace: \b
- Form feed: \f
- Tab: \t
- New line: \n
- Carriage return:\r
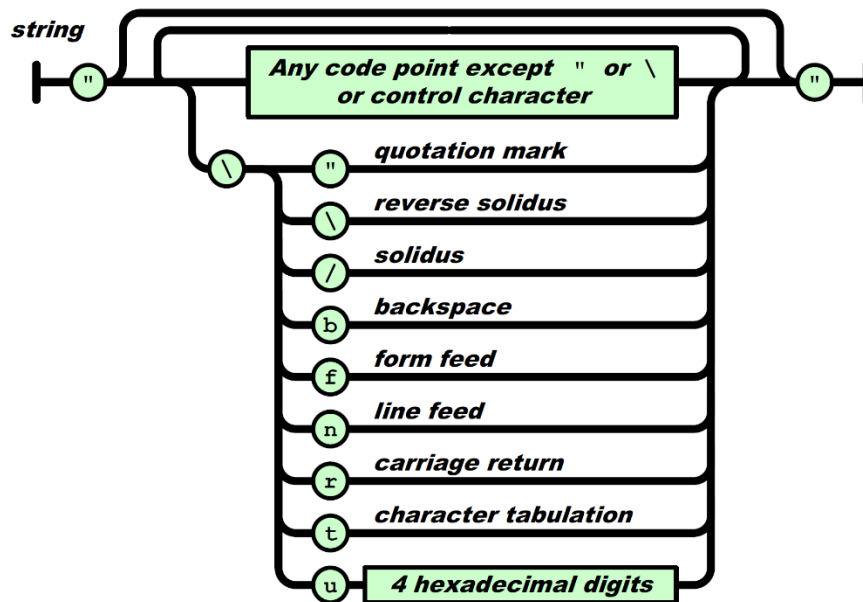- \u followed by hexadecimal characters (e.g., the smiley emoticon, \u263A

Figure 5 JSON string

DATASET-JSON DOCUMENT STRUCTURE

Dataset-JSON was adapted from the Dataset-XML Version 1 specification but uses JSON format [9].

The Dataset-JSON schema and examples can be found at CDISC's GitHub repository [15].
https://github.com/cdisc-org/DataExchange-DatasetJson

Like Dataset-XML, each Dataset-JSON file is connected to the Define-XML document, containing detailed information about the metadata. Dataset-JSON files contain basic information about dataset variables, so that it is possible to have a simple view of a dataset contents without a need of a Define-XML document.

Each Dataset-JSON file contains data for a single dataset.

At the top level of the Dataset-JSON object, there are two required attributes: creationDateTme and datasetJSONVersion, and several optional attributes: fileOID, asOfDateTime, Originator, sourceSystem, sourceSystemVersion, clinicalData, and referenceData. Subject data is stored in **clinicalData** and non-subject data is stored in **referenceData**. At least one of these attributes must be provided.

creationDateTime	Required	Time of creation of the file containing the document.
datasetJSONVersion	Required	Version of Dataset-JSON standard
fileOID	Optional	A unique identifier for this file.
asOfDateTime	Optional	The date/time at which the source database was queried in order to create this document.
originator	Optional	The organization that generated the Dataset-JSON file.
sourceSystem	Optional	The computer system or database management system that is the source of the information in this file.
sourceSystemVersion	Optional	The version of the "SourceSystem" above.
clinicalData	Optional	Contains datasets for clinical data across multiple subjects.
referenceData	Optional	Contains datasets for non-subject data domains.

Example:

```
{
    "creationDateTime": "2023-03-22T11:53:27",
    "datasetJSONVersion": "1.0.0",
    "fileOID": "www.sponsor.xyz.org.project123.final",
    "asOfDateTime": "2023-02-15T10:23:15",
    "originator": "Sponsor XYZ",
    "sourceSystem": "Software ABC",
    "sourceSystemVersion": "1.0.0",
    "clinicalData": { ... },
    "referenceData": { ... }
}
```

Each of the clinicalData and referenceData attributes contain study and metadata OIDs as well as an object describing one or more item groups (datasets). The values of the **studyOID** and **metaDataVersionOID** attributes must match corresponding values in the Define-XML document.

Example:

```
"clinicalData": {
  "studyOID": "cdisc.com/CDISCPILOT01",
  "metaDataVersionOID": "MDV.MSGv2.0.SDTMIG.3.3.SDTM.1.7",
  "itemGroupData": {
      "IG.DM": { ... },
      ...
  }
}
```

ItemGroupData is an object with attributes corresponding to individual datasets. The attribute name is the OID of a described dataset, which must be the same as OID of the corresponding itemGroup in the Define-XML file.

```
"itemGroupData": {
    "IG.DM": { ... }
}
```

The dataset description contains basic information about the dataset itself and its items.

records	Required	The total number of records in a dataset
name	Required	Dataset name
label	Required	Dataset description
items	Required	Basic information about variables
itemData	Required	Dataset data

Example:

```
{
    "IG.DM": {
        "records": 18,
        "name": "DM",
        "label": "Demographics",
        "items": [ ...],
        "itemData": [ ... ]
    }
}
```

In case of a dataset without any records **itemData** needs to be set to a blank array.

Example of an empty dataset:

```
// Empty dataset
{
    "IG.CO": {
        "records": 0,
        "name": "CO",
        "label": "Comments",
        "items": [ ... ],
        "itemData": [ ... ]
    }
}
```

The **items** attribute is an array of basic information about dataset variables, so that it is possible to have a simple view of a dataset contents without having to use Define-XML The order of elements in the array must be the same as the order of variables in the described dataset. The first element always describes the Record Identifier (ITEMGROUPDATASEQ).

OID	Required	Unique identifier of a variable (must correspond to the variable OID in the Define-XML file)
name	Required	Variable name
tabel	Required	Variable description
type	Required	Data type of the variable. One of "string", "integer", "float", "double", "decimal", "boolean"
length	Optional	Variable length
displayFormat	Optional	Display format supports data visualization of numeric float and date values
keySequence	Optional	ndicates that this item is a key variable in the dataset structure. It also provides an ordering for the keys

Example:

```
"items": [
    {
        "OID": "ITEMGROUPDATASEQ",
        "name": "ITEMGROUPDATASEQ",
        "label": "Record identifier",
        "type": "integer"
    },
    {
        "OID": "IT.DM.STUDYID",
        "name": "STUDYID",
        "label": "Study Identifier",
        "type": "string",
        "length": 12
    },
    ...
]
```

Allowed values for the **type** are: "string", "integer", "float", "double", "decimal" and "boolean". Attributes **length**, **displayFormat**, and **keySequence** are optional. The main use case for the length attribute is for the "string" type.

itemData is an array of records with variables values. Each record itself is also represented as an array of variables values. The first value is a unique sequence number for each record in the dataset.

Example:

```
{
    "itemData": [
        [1, "MyStudy", "001", "DM", 56],
        [2, "MyStudy", "002", "DM", 26],
        ...
    ]
}
```

Missing values are represented by null in the case of numeric variables, and an empty string in case of character variables: `[1, "MyStudy", "", "DM", null]`

The full example of a Dataset-JSON file:

```json
{
    "creationDateTime": "2023-03-22T11:53:27",
    "datasetJSONVersion": "1.0.0",
    "fileOID": "www.sponsor.org.project123.final",
    "asOfDateTime": "2023-02-15T10:23:15",
    "originator": "Sponsor XYZ",
    "sourceSystem": "Software ABC",
    "sourceSystemVersion": "1.2.3",
    "clinicalData": {
        "studyOID": "xxx",
        "metaDataVersionOID": "xxx",
        "metaDataRef": "https://metadata.location.org/api.link",
        "itemGroupData": {
            "IG.DM": {
                "records": 600,
                "name": "DM",
                "label": "Demographics",
                "items": [
                    {"OID": "ITEMGROUPDATASEQ", "name": "ITEMGROUPDATASEQ", "label": "Record identifier", "type": "integer"},
                    {"OID": "IT.STUDYID", "name": "STUDYID", "label": "Study identifier", "type": "string", "length": 7, "keySequence": 1},
                    {"OID": "IT.USUBJID", "name": "USUBJID", "label": "Unique Subject Identifier", "type": "string", "length": 3, "keySequence": 2},
                    {"OID": "IT.DOMAIN", "name": "DOMAIN", "label": "Domain Identifier", "type": "string", "length": 2},
                    {"OID": "IT.AGE", "name": "AGE", "label": "Subject Age", "type": "integer", "length": 2}
                ],
                "itemData": [
                    [1, "MyStudy", "001", "DM", 56],
                    [2, "MyStudy", "002", "DM", 26],
                    ...
                ]
            }
        }
    }
}
```

SAS AND JSON

The original idea for this paper was to further build on the use of PROC LUA to work with JSON files. PROC LUA has been successfully used by the author, especially when consuming JSON files as part of REST API services [10][11]. The Lua JSON libraries are an efficient way to manage complex JSON files that are not too big. Using SAS is cumbersome when dealing with complex JSON files as it requires the merging of many datasets and dealing with the management of JSON MAP files to correct decisions that the SAS JSON automapper makes in terms of variable types and variable lengths. But one needs to keep in mind that PROC LUA reads the entire JSON file into a Lua table that is kept in memory. This works well with REST APIs and metadata tables but can lead to issues when working with very large datasets.

Since the Dataset-JSON format is simple and the files might get very large it was decided to use native SAS® technology for reading and writing Dataset-JSON files.

PROC LUA in SAS is still used in this paper to extract metadata from Define-XML files into SAS datasets.

Starting in SAS® 9.4, you can copy SAS data sets to JSON files with PROC JSON. Starting in SAS 9.4TS1M4, you can copy JSON files to SAS data sets with the JSON engine [12][13].

All code in this paper is at GitHub: https://github.com/lexjansen/dataset-json-sas

METADATA

Reading and writing Dataset-JSON files does not only require data as input but also metadata. For the SAS programs in this paper the metadata was defined in 3 metadata tables for (ADaM and SDTM):

- metadata_study

- metadata_tables

- metadata_columns

These metadata datasets were created from Define-XML files by the SAS program in Appendix 1. The program uses PROC LUA to parse the Define-XML file. You may have different metadata sources. Although some of the metadata may come from the SAS datasets when creating Dataset-JSON, or from the Dataset-JSON file when creating SAS datasets, this metadata can also be managed in a metadata repository.

For example, the display format in the metadata_columns table can be used to attach a format to variables when reading Dataset-JSON. This is especially important for ADaM datasets that contain numeric date or datetime variables.

Figure 6 shows an example of the metadata tables.

VIEWTABLE: Metadata.Metadata_study

	fileoid	creationdatetime	studyoid	metadataversionoid
1	www.cdisc.org/StudyMSGv2/1/Define-XML_2.1.0	2021-01-24T16:59:33	cdisc.com/CDISCPILOT01	MDV.MSGv2.0.SDTMIG.3.3.SDTM.1.7

VIEWTABLE: Metadata.Metadata_tables

	oid	name	label	domain	repeating	isreferencedata	structure
1	IG.TA	TA	Trial Arms	TA	No	Yes	One record per planned Element
2	IG.TE	TE	Trial Elements	TE	No	Yes	One record per planned Element
3	IG.TI	TI	Trial Inclusion/Exclusion Criteria	TI	No	Yes	One record per I/E criterion
4	IG.TS	TS	Trial Summary	TS	No	Yes	One record per trial summary para value
5	IG.TV	TV	Trial Visits	TV	No	Yes	One record per planned Visit per
6	IG.DM	DM	Demographics	DM	No	No	One record per subject
7	IG.SE	SE	Subject Elements	SE	Yes	No	One record per actual Element pe subject
8	IG.SV	SV	Subject Visits	SV	Yes	No	One record per actual visit per sut
9	IG.CM	CM	Concomitant Medications	CM	Yes	No	One record per recorded medicati occurrence or constant-dosing int per subject
10	IG.EC	EC	Exposure as Collected	EC	Yes	No	One record per constant-dosing in

VIEWTABLE: Metadata.Metadata_columns

	dataset_name	oid	name	label	order	xml_datatype	json_dat	length	displayformat	keysequence
246	LB	IT.LB.STUDYID	STUDYID	Study Identifier	1 text	string	12			1
247	LB	IT.LB.DOMAIN	DOMAIN	Domain Abbreviation	2 text	string	2			.
248	LB	IT.LB.USUBJID	USUBJID	Unique Subject Identifier	3 text	string	8			2
249	LB	IT.LB.LBSEQ	LBSEQ	Sequence Number	4 integer	integer	3			.
250	LB	IT.LB.LBTESTCD	LBTESTCD	Lab Test or Examination Short Name	5 text	string	7			4
251	LB	IT.LB.LBTEST	LBTEST	Lab Test or Examination Name	6 text	string	39			.
252	LB	IT.LB.LBCAT	LBCAT	Category for Lab Test	7 text	string	10			3
253	LB	IT.LB.LBORRES	LBORRES	Result or Finding in Original Units	8 text	string	6			.
254	LB	IT.LB.LBORRESU	LBORRESU	Original Units	9 text	string	7			.
255	LB	IT.LB.LBORNRLO	LBORNRLO	Reference Range Lower Limit in Orig Unit	10 text	string	200			.
256	LB	IT.LB.LBORNRHI	LBORNRHI	Reference Range Upper Limit in Orig Unit	11 text	string	200			.
257	LB	IT.LB.LBSTRESC	LBSTRESC	Character Result/Finding in Std Format	12 text	string	8			.
258	LB	IT.LB.LBSTRESN	LBSTRESN	Numeric Result/Finding in Standard Units	13 float	float	8			.
259	LB	IT.LB.LBSTRESU	LBSTRESU	Standard Units	14 text	string	7			.
260	LB	IT.LB.LBSTNRLO	LBSTNRLO	Reference Range Lower Limit-Std Units	15 float	decimal	5			.
261	LB	IT.LB.LBSTNRHI	LBSTNRHI	Reference Range Upper Limit-Std Units	16 float	decimal	5			.
262	LB	IT.LB.LBNRIND	LBNRIND	Reference Range Indicator	17 text	string	8			.
263	LB	IT.LB.LBLOBXFL	LBLOBXFL	Last Observation Before Exposure Flag	18 text	string	1			.
264	LB	IT.LB.VISITNUM	VISITNUM	Visit Number	19 float	decimal	8			6
265	LB	IT.LB.VISIT	VISIT	Visit Name	20 text	string	200			.
266	LB	IT.LB.EPOCH	EPOCH	Epoch	21 text	string	9			.
267	LB	IT.LB.LBDTC	LBDTC	Date/Time of Specimen Collection	22 partialDatetime	string	.			5
268	LB	IT.LB.LBDY	LBDY	Study Day of Specimen Collection	23 integer	integer	8			.

VIEWTABLE: Metadata.Metadata_columns

	dataset_	oid	name	label	order	xml_datatype	json_datatype	length	displayformat	keysequence
126	ADLBC	IT.ADLBC.STUDYID	STUDYID	Study Identifier	1 text	string	12			.
127	ADLBC	IT.ADLBC.SUBJID	SUBJID	Subject Identifier for the Study	2 text	string	4			.
128	ADLBC	IT.ADLBC.USUBJID	USUBJID	Unique Subject Identifier	3 text	string	11			1
129	ADLBC	IT.ADLBC.TRTP	TRTP	Planned Treatment	4 text	string	20			.
130	ADLBC	IT.ADLBC.TRTPN	TRTPN	Planned Treatment (N)	5 integer	integer	8			.
131	ADLBC	IT.ADLBC.TRTA	TRTA	Actual Treatment	6 text	string	20			.
132	ADLBC	IT.ADLBC.TRTAN	TRTAN	Actual Treatment (N)	7 integer	integer	8			.
133	ADLBC	IT.ADLBC.TRTSDT	TRTSDT	Date of First Exposure to Treatment	8 integer	integer	8	DATE9.		.
134	ADLBC	IT.ADLBC.TRTEDT	TRTEDT	Date of Last Exposure to Treatment	9 integer	integer	8	DATE9.		.
135	ADLBC	IT.ADLBC.AGE	AGE	Age	10 integer	integer	8			.
136	ADLBC	IT.ADLBC.AGEGR1	AGEGR1	Pooled Age Group 1	11 text	string	5			.
137	ADLBC	IT.ADLBC.AGEGR1N	AGEGR1N	Pooled Age Group 1 (N)	12 integer	integer	8			.
138	ADLBC	IT.ADLBC.RACE	RACE	Race	13 text	string	32			.
139	ADLBC	IT.ADLBC.RACEN	RACEN	Race (N)	14 integer	integer	8			.
140	ADLBC	IT.ADLBC.SEX	SEX	Sex	15 text	string	1			.
141	ADLBC	IT.ADLBC.COMP24FL	COMP24FL	Completers of Week 24 Population Flag	16 text	string	1			.
142	ADLBC	IT.ADLBC.DSRAEFL	DSRAEFL	Discontinued due to AE?	17 text	string	1			.
143	ADLBC	IT.ADLBC.SAFFL	SAFFL	Safety Population Flag	18 text	string	1			.
144	ADLBC	IT.ADLBC.AVISIT	AVISIT	Analysis Visit	19 text	string	16			3
145	ADLBC	IT.ADLBC.AVISITN	AVISITN	Analysis Visit (N)	20 integer	integer	8			.
146	ADLBC	IT.ADLBC.ADY	ADY	Analysis Relative Day	21 integer	integer	8			.
147	ADLBC	IT.ADLBC.ADT	ADT	Analysis Date	22 integer	integer	8	DATE9.		.
148	ADLBC	IT.ADLBC.VISIT	VISIT	Visit Name	23 text	string	19			.
149	ADLBC	IT.ADLBC.VISITNUM	VISITNUM	Visit Number	24 float	decimal	8			.
150	ADLBC	IT.ADLBC.PARAM	PARAM	Parameter	25 text	string	100			.
151	ADLBC	IT.ADLBC.PARAMCD	PARAMCD	Parameter Code	26 text	string	8			2
152	ADLBC	IT.ADLBC.PARAMN	PARAMN	Parameter (N)	27 integer	integer	8			.

Figure 6 Metadata tables

WRITING DATASET-JSON FILES WITH SAS

All code in this paper is at GitHub: https://github.com/lexjansen/dataset-json-sas

PROC JSON in SAS® gives the user control over the JSON output through the utilization of options as well as the ability to control containers, write directly to the output file, and choose exactly what to include or not include in the resulting JSON file [14].

The PROC JSON syntax is as follows:

```
PROC JSON OUT=fileref | "external-file" <options>;
  EXPORT <libref.>SAS-data-set <(SAS-data-set-options)> </options>;
  WRITE VALUES value(s) </options>;
  WRITE OPEN type;
  WRITE CLOSE;
RUN;
```

The JSON procedure reads data from a SAS® data set and writes it to an external file in JSON representation as specified with a file reference or an external-file specification, followed by any options to control the output.

The EXPORT statement identifies the SAS® data set to be exported and allows the user to control the resulting output by using options that are specific to PROC JSON as well as SAS® data set options that are applied to the input SAS® data set.

In addition to exporting data sets, PROC JSON gives the user the ability to write custom information to the output file with the WRITE statement, which allows the user to write one or more literal values to the JSON output file. The value can be either a string, a number, a Boolean value (TRUE or FALSE), or NULL. The WRITE OPEN and WRITE CLOSE statements allow the user to control the containers (objects or arrays) in the JSON output file.

The WRITE VALUES statement and the WRITE OPEN/CLOSE statements allow the user to open, close, and nest containers in the JSON output file as well as write separate values to the JSON output file.

The type in the WRITE OPEN statement can be either ARRAY or OBJECT.

Options control and customize the generated output. These options include whether to:

- apply SAS formats to the resulting output (character, numeric, date, time, datetime)
- include or suppress SAS variable names in the output
- format the JSON output ("pretty print")
- include or suppress SAS metadata at the top of the JSON file
- scan and encode input strings to ensure that valid JSON output is created
- remove or retain trailing blanks from the end of character data in the JSON output

The code below shows the code that was used in this paper to create the Dataset-JSON file:

```
FILENAME jsonout "&jsonpath";

PROC JSON OUT=jsonout NOPRETTY NOSASTAGS SCAN TRIMBLANKS
                      NOFMTCHARACTER NOFMTDATETIME NOFMTNUMERIC;

  WRITE OPEN OBJECT;

    WRITE VALUES "creationDateTime" "&CurrentDateTime";
    WRITE VALUES "datasetJSONVersion" "&datasetJSONVersion";
    %if %sysevalf(%superq(_FileOID)=, boolean)=0 %then
      WRITE VALUES "fileOID" "&_FileOID";
    ;
    %if %sysevalf(%superq(_AsOfDateTime)=, boolean)=0 %then
      WRITE VALUES "asOfDateTime" "&_AsOfDateTime";
```

```
          ;
          %if %sysevalf(%superq(_Originator)=, boolean)=0 %then
            WRITE VALUES "originator" "&_Originator";
          ;
          %if %sysevalf(%superq(_SourceSystem)=, boolean)=0 %then
            WRITE VALUES "sourceSystem" "&_SourceSystem";
          ;
          %if %sysevalf(%superq(_SourceSystemVersion)=, boolean)=0 %then
            WRITE VALUES "sourceSystemVersion" "&_SourceSystemVersion";
          ;

          WRITE VALUES "&ClinicalReferenceData";
          WRITE OPEN OBJECT;
          %if %sysevalf(%superq(studyOID)=, boolean)=0 %then
            WRITE VALUES "studyOID" "&studyOID";
          ;
          %if %sysevalf(%superq(metaDataVersionOID)=, boolean)=0 %then
            WRITE VALUES "metaDataVersionOID" "&metaDataVersionOID";
          ;
          %if %sysevalf(%superq(_MetaDataRef)=, boolean)=0 %then
            WRITE VALUES "metaDataRef" "&_MetaDataRef";
          ;
            WRITE VALUE "itemGroupData";
            WRITE OPEN OBJECT;
              WRITE VALUE "&ItemGroupOID";
              WRITE OPEN OBJECT;
                WRITE VALUES "records" &records;
                WRITE VALUES "name" "%upcase(&dataset_name)";
                /* WRITE VALUES "label" %sysfunc(quote(&dataset_label)); */
                WRITE VALUES "label" "%nrbquote(&dataset_label)";

                WRITE VALUE "items";
        %* Use macro to avoid creating null values for missing attributes;
        %* Instead do not create the attribute;
                %write_json_metadata_array(work.column_metadata);
                WRITE CLOSE;

                WRITE VALUE "itemData";
                WRITE OPEN ARRAY;
                  EXPORT work.column_data / NOKEYS;
                WRITE CLOSE;
              WRITE CLOSE;
            WRITE CLOSE;
          WRITE CLOSE;
        WRITE CLOSE;
      RUN;
```

The value of the **ClinicalReferenceData** macro variable has a value "clinicalData" or "referenceData" and is calculated as follows:

```
  %if %cstutilcheckvarsexist(
        _cstDataSetName=&dataset_new,
        _cstVarList=usubjid)
    %then %let ClinicalReferenceData=clinicalData;
    %else %let ClinicalReferenceData=referenceData;
```

14

A macro was used to determine the number of records in the dataset:

```
%let records=%cstutilnobs(_cstDataSetName=&dataset_new);
```

The macro variables **studyOID**, **metaDataVersionOID**, **ItemGroupOID** and **dataset_label** can be specified or created from the metadata tables:

```
%if %substr(%upcase(&UseMetadata),1,1) eq Y %then %do;
   /* Get StudyOID and metaDataVersionOID from the metadata */
   proc sql noprint;
     %if %sysfunc(exist(&metadatalib..metadata_study)) %then %do;
       select studyOID, metaDataVersionOID into :studyOID trimmed,
:metaDataVersionOID trimmed
         from &metadatalib..metadata_study;
     %end;
   /* Get dataset label and ItemGroupOID from the metadata */
     %if %sysfunc(exist(&metadatalib..metadata_tables)) %then %do;
       select label, oid into :dataset_label, :ItemGroupOID trimmed
         from &metadatalib..metadata_tables
           where upcase(name)="%upcase(&dataset_name)";
     %end;
   quit;
%end;
```

The metadata for the variables (items) can be created from the metadata in the metadata tables or from the metadata in the datasets:

```
%if %substr(%upcase(&UseMetadata),1,1) eq Y %then %do;
/* Get column metadata: oid, label, type, length, displayformat,
   keysequence */
   data work.column_metadata(keep=OID name label type length displayFormat
                                  keySequence);
     retain OID name label type length displayFormat keySequence;
     set &metadatalib..metadata_columns(
       rename=(json_datatype=type)
       where=(upcase(dataset_name) = %upcase("&dataset_name")));
   run;
%end;
%else %do;
   /* Get column metadata from the datasets - label, type, length, format
      and derive as much as we can */
   proc contents noprint varnum data=&dataset_new
     out=work.column_metadata(
       keep=varnum name type format formatl formatd length label
       rename=(format=displayFormat type=sas_type)
     );
   run;

   proc sort data=work.column_metadata;
     by varnum;
   run;

   data work.column_metadata(drop=sas_type varnum formatl formatd);
     retain OID name label type length;
     length OID $128 type $32;
     set work.column_metadata;
```

```
        OID = cats("IT", ".", upcase("&dataset_name"), ".", upcase(name));
        if sas_type=1 then type="float";
                      else type="string";
        if formatl gt 0 then
           displayFormat=cats(displayFormat, put(formatl, best.), ".");
        if formatd gt 0 then
           displayFormat=cats(displayFormat, put(formatd, best.));
     run;

  %end;
```

The work.**column_metadata** dataset is used to create the **items** array:

```
"items": [
    {
        "OID": "ITEMGROUPDATASEQ",
        "name": "ITEMGROUPDATASEQ",
        "label": "Record identifier",
        "type": "integer"
    },
    {
        "OID": "IT.DM.STUDYID",
        "name": "STUDYID",
        "label": "Study Identifier",
        "type": "string",
        "length": 12
    },
    ...
]

    /* Create a 1-obs dataset with the same structure as the column_metadata
       dataset */
    proc sql noprint;
      create table itemgroupdataseq
        like work.column_metadata;
      insert into itemgroupdataseq
        set OID="ITEMGROUPDATASEQ", name="ITEMGROUPDATASEQ",
            label="Record Identifier", type="integer";
    quit;

    data work.column_metadata;
      set itemgroupdataseq work.column_metadata
                           (where=(upcase(name) ne "ITEMGROUPDATASEQ"));
    run;
```

The **work.column_data** dataset is the original dataset, but with the extra ITEMGROUPDATASEQ column added.

```
        data work.column_data;
          length ITEMGROUPDATASEQ 8.;
          set &dataset;
          ITEMGROUPDATASEQ = _n_;
        run;
```

READING DATASET-JSON FILES WITH SAS

The JSON LIBNAME statement in SAS® enables you to read a JSON file using the JSON engine. The engine uses a JSON map file to describe the data sets in the specified JSON file. The JSON engine can automatically generate this JSON map file for you when you assign the LIBNAME statement, or you can supply your own JSON map file in the MAP= option.

The syntax of the JSON LIBNAME statement is:

```
LIBNAME libref JSON <'JSON-document-path'> <options>;
```

In our example we use:

```
LIBNAME out "path to the folder where we write the datasets";

FILENAME mapfile "path where the map file will be created";

FILENAME jsonfile "path to the JSON file to read";

LIBNAME jsonfile JSON MAP=mapfile AUTOMAP=CREATE FILEREF=jsonfile
                                  ORDINALCOUNT=NONE;

PROC COPY IN=jsonfile OUT=out;

RUN;
```

The syntax means that we will let PROC JSON create the JSON map. Also, we will not create ordinal variables for the data sets.

Running this code on the the following Dataset-JSON file would create 6 datasets:

```
{
    "creationDateTime": "2023-06-28T15:38:43",
    "datasetJSONVersion": "1.0.0",
    "fileOID": "www.cdisc.org/StudyMSGv2/1/Define-XML_2.1.0/2023-06-28/dm",
    "asOfDateTime": "2023-05-31T00:00:00",
    "originator": "CDISC SDTM MSG Team",
    "sourceSystem": "Sponsor System",
    "sourceSystemVersion": "1.0",
    "clinicalData": {
      "studyOID": "cdisc.com/CDISCPILOT01",
      "metaDataVersionOID": "MDV.MSGv2.0.SDTMIG.3.3.SDTM.1.7",
      "metaDataRef": "https://metadata.location.org/CDISCPILOT01/define.xml",
      "itemGroupData": {
        "IG.DM": {
            "records": 18,
            "name": "DM",
            "label": "Demographics",
            "items": [
                {
                  "OID": "ITEMGROUPDATASEQ",
                  "name": "ITEMGROUPDATASEQ",
                  "label": "Record Identifier",
                  "type": "integer"},
                {
                  "OID": "IT.DM.STUDYID",
                  "name": "STUDYID",
                  "label": "Study Identifier",
                  "type": "string",
                  "length": 12,
                  "keySequence": 1},
```

1. alldata

2. clinicaldata

3. ig_dm_items

4. ig_dm_itemdata

5. itemgroupdata_ig_dm

6. root

Notice that the names of the datasets depend on whether we have subject data (clinicaldata) or reference data (referencedata) and on OID of a described dataset, in this case "IG.DM".

The names can be derived with the following SAS code:

```
/* Find the names of the dataset that were created */
%let _clinicalreferencedata_=;
%if %sysfunc(exist(out.clinicaldata))
  %then %let _clinicalreferencedata_=out.clinicaldata;
  %else %if %sysfunc(exist(out.referencedata))
          %then %let _clinicalreferencedata_=out.referencedata;

proc sql noprint;
  create table members
  as select upcase(memname) as name
  from dictionary.tables
  where libname="OUT" and memtype="DATA"
  ;
quit;

%let _items=;
%let _itemdata=;
%let _itemgroupdata_=;
data _null_;
  set members;
  if index(upcase(name), '_ITEMS') then
    call symputx('_items_', strip(name));
  if index(upcase(name), '_ITEMDATA') then
    call symputx('_itemdata_', strip(name));
  if index(upcase(name), 'ITEMGROUPDATA_') then
    call symputx('_itemgroupdata_', strip(name));
run;
```

Figures 7 – 10 show the datasets created by running the JSON LIBNAME statement and the PROC COPY procedure.

	studyOID	metaDataVersionOID	metaDataRef
1	cdisc.com/CDISCPILOT01	MDV.MSGv2.0.SDTMIG.3.3.SDTM.1.7	https://metadata.location.org/CDISCPILOT01/define.xml

VIEWTABLE: Work.Clinicaldata

Figure 7 - clinicaldata SAS dataset

VIEWTABLE: Work.Itemgroupdata_ig_dm

	records	name	label
1	18	DM	Demographics

Figure 8 - itemgroupdata_ig_dm SAS dataset

18

Figure 9 – ig_dm_items SAS dataset

Figure 10 – ig_dm_itemdata SAS dataset

To create our final dataset, we need to:

- get the name of the dataset to create
- label the dataset
- rename variables element1, element2, element3, ...
- label the variables
- attach a format to numeric variables when defined in the metadata. Note that the displayformat is not define in the Dataset-JSON file, but in the Define-XML document.

We can achieve this with the following code that use the metadata in the **itemgroupdata_ig_dm** dataset:

```
proc sql noprint;
  select name into :variables separated by ' '
    from out.&_items_;
  select cats("element", monotonic(), '=', name) into :rename separated
by ' '
    from out.&_items_;
  select cats(name, '=', quote(strip(label))) into :label separated by
' '
    from out.&_items_;
  select catt(name, ' $', length) into :length separated by ' '
    from out.&_items_
```

```sas
            where type="string" and (not(missing(length)));
        quit;

        proc sql noprint;
          select label, name into :dslabel, :dsname trimmed
            from out.&_itemgroupdata_
          ;
        quit;

        proc copy in=out out=dataout;
          select &_itemdata_;
        run;

/* get formats from Dataset-JSON metadata, but only when the displayformat
   variable exists */
%let format=;
%if %cstutilcheckvarsexist(_cstDataSetName=out.&_items_,
  _cstVarList=displayformat) %then %do;
  proc sql noprint;
   select catx(' ', name, strip(displayformat)) into :format separated by ' '
     from out.&_items_
     where not(missing(displayformat)) /* and (type in ('integer' 'float'
   'double' 'decimal')) */;
  quit;
%end;

proc datasets library=&dataoutlib noprint nolist nodetails;
  %if %sysfunc(exist(&dataoutlib..&dsname)) %then %do; delete &dsname; %end;
  change &_itemdata_ = &dsname;
  modify &dsname %if %sysevalf(%superq(dslabel)=, boolean)=0 %then
   %str((label = %sysfunc(quote(%nrbquote(&dslabel)))));;
    rename &rename;
    label &label;
quit;
```

For character variables we will use the lengths in the JSON variable metadata (ig_dm_items) when they are defined and longer than the lengths in the dataset created by the JSON libname engine.

```sas
/* Update lengths */
proc sql noprint;
  select catt(d.name, ' $', i.length) into :length separated by ' '
    from dictionary.columns d,
         out.&_items_ i
  where upcase(libname)="%upcase(&dataoutlib)" and
        upcase(memname)="%upcase(&dsname)" and
        d.name = i.name and
        d.type="char" and (not(missing(i.length))) and (i.length gt d.length)
 ;
quit ;

data &dataoutlib..&dsname(
    %if %sysevalf(%superq(dslabel)=, boolean)=0
      %then %str(label = %sysfunc(quote(%nrbquote(&dslabel))));
    %if %substr(%upcase(&DropSeqVar),1,1) eq Y
      %then drop=ITEMGROUPDATASEQ;
  );
```

```
   retain &variables;
   length &length;
   %if %sysevalf(%superq(format)=, boolean)=0 %then format &format;;
   set &dataoutlib..&dsname;
run;
```

VALIDATION

When reading or writing JSON files, it is important to validate the process. The published JSON specification also comes with a JSON schema that can be used to validate JSON files [15].

With a simple Python program, the JSON file can be validated against the schema (Appendix 2).

Also, the read/write process can be validated by doing a roundtrip and comparing SAS datasets or JSON-files:

- Dataset-JSON → SAS dataset → Dataset-JSON
- SAS dataset → Dataset-JSON → SAS dataset

SAS datasets can be compared with PROC COMPARE. It can be expected that there will differences for numeric floating-point variables in the order of machine precision.

Dataset-JSON file can be compared with a utility like WinMerge (https://winmerge.org/) that allows you to compare JSON files. It can be expected that there are some differences, as SAS may output null values in objects, or the original Dataset-JSON file may not have some lengths defined. An example of this compare can be seen in the picture below
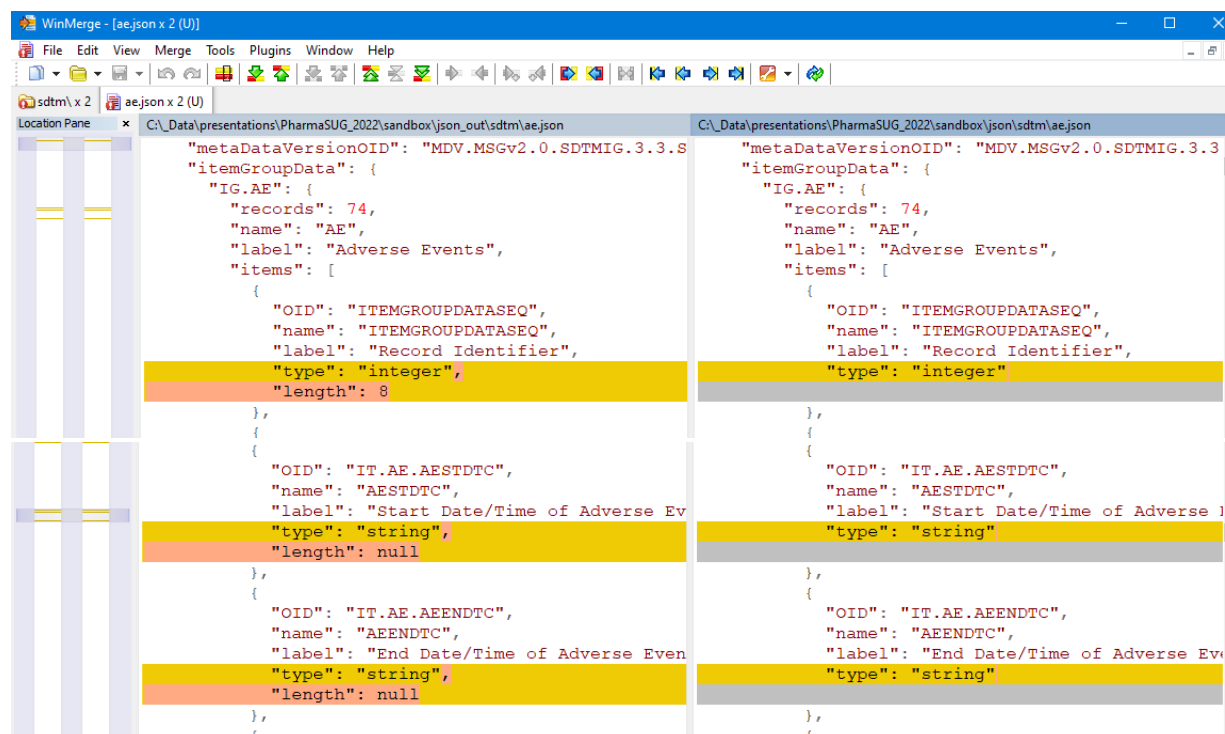

Figure 11 Example output from comparing 2 Dataset-JSON files with WinMerge (compare as "Pretty JSON")

CONCLUSION

SAS fully supports reading and writing Dataset-JSON files in an efficient way.

REFERENCES

1. SAS Technical Paper: Record Layout of a SAS® Version 5 or 6 Data Set in SAS® Transport (XPORT) Format, October 2021 (https://support.sas.com/content/dam/SAS/support/en/technical-papers/record-layout-of-a-sas-version-5-or-6-data-set-in-sas-transport-xport-format.pdf)
2. PhUSE Emerging Trends & Technologies – Transport for the Next Generation, Version 1.1 Created 20 May 2017 (https://phuse.s3.eu-central-1.amazonaws.com/Deliverables/Emerging+Trends+%26+Technologies/Alternative+Transport+%E2%80%93+Transport+for+the+Next+Generation.pdf)
3. U.S. Department of Health and Human Services Food and Drug Administration Center for Drug Evaluation and Research (CDER), Center for Biologics Evaluation and Research (CBER). Study Data Technical Conformance Guide, Version 4.9, March 2022 (https://www.fda.gov/industry/fda-data-standards-advisory-board/study-data-standards-resources)
4. SAS Technical Paper: Record Layout of a SAS® Version 8 or 9 Data Set in SAS® Transport Format, October 2021 (https://support.sas.com/content/dam/SAS/support/en/technical-papers/record-layout-of-a-sas-version-8-or-9-data-set-in-sas-transport-format.pdf)
5. CDISC Dataset-XML Specification Version 1.0, April 22, 2014 (https://www.cdisc.org/standards/data-exchange/dataset-xml)
6. Test Report for DS-XML Pilot. Center for Drug Evaluation and Research (CDER), Center for Biologics Evaluation and Research (CBER), April 8, 2015.
7. The JavaScript Object Notation (JSON) Data Interchange Format, RFC 8259 T. Bray, Ed., Internet Engineering Task Force (IETF), December 2017 https://datatracker.ietf.org/doc/html/rfc8259
8. The JSON Data Interchange Format, Standard ECMA-404 Ecma International, 2nd edition, December 2017 https://www.ecma-international.org/publications-and-standards/standards/ecma-404/
9. CDISC Operational Data Model (ODM), Version 2.0 – Dataset-JSON (Draft) (https://wiki.cdisc.org/display/ODM2/Dataset-JSON)
10. Extracting Data Standards Metadata and Controlled Terminology from the CDISC Library using SAS with PROC LUA Lex Jansen, Pharmaceutical SAS Users Group 2021 https://www.lexjansen.com/pharmasug/2021/AD/PharmaSUG-2021-AD-168.pdf
11. Parsing JSON Files in SAS© Using PROC LUA Lex Jansen, Pharmaceutical Users Software Exchange 2021 https://www.lexjansen.com/phuse/2021/ad/PRE_AD06.pdf
12. SAS® Institute Inc . 2017. "JSON Procedure" In Base SAS® 9.4 Procedures Guide, Seventh Edition. Cary, NC: SAS® Institute Inc. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/p0ie4bw6967jg6n1iu629d40f0by.htm
13. SAS® Institute Inc . 2017. "LIBNAME Statement: JSON Engine" SAS® 9.4 Global Statements: Reference, Seventh Edition. Cary, NC: SAS® Institute Inc. Available at https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.2/lestmtsglobal/n1jfdetszx99ban1rl4zll6tej7j.htm
14. Creating and Controlling JSON Output with PROC JSON Adam Linker, SAS Global Forum 2019 https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3506-2019.pdf
15. Dataset-JSON GitHub repository: https://github.com/cdisc-org/DataExchange-DatasetJson

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Lex Jansen
Sr Director, Data Science Development, CDISC (contract through Lex Jansen Consulting LLC)
Email: lexjansen@gmail.com or ljansen@cdisc.org

All code used in this paper can be found at GitHub:

https://github.com/lexjansen/dataset-json-sas

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

Appendix 1 Creating Metadata tables from Define-XML

read_definexml.lua

```lua
    local fileutils = require "fileutils"
    local tableutils = require "tableutils"

    -- this is a very rough mapping, it does not take decimal into account
    local datatype_mapping = {
            text = "string",
            date = "string",
            datetime = "string",
            time = "string",
            URI = "string",
            partialDate = "string",
            partialTime = "string",
            partialDatetime = "string",
            durationDatetime = "string",
            intervalDatetime = "string",
            incompleteDatetime = "string",
            incompleteDate = "string",
            incompleteTime = "string",
            integer = "integer",
            float = "float"
        }

    sas.filename('define', sas.symget("definexml"))

    local metadatalib = sas.symget("metadatalib")


    local define_string = fileutils.read('define')
    local define = sas.xml_parse(define_string)
    sas.symput('studyOID',define.Study["@OID"])
    sas.symput('metaDataVersionOID', define.Study.MetaDataVersion["@OID"])

    sas.submit[[
       %create_template(type=STUDY, out=@metadatalib@.metadata_study);
    ]]
    dsid_s = sas.open(metadatalib..'.metadata_study', "u")
    sas.append(dsid_s)
    sas.put_value(dsid_s, "fileoid", define["@FileOID"])
    sas.put_value(dsid_s, "creationdatetime", define["@CreationDateTime"])
    if define["@AsOfDateTime"] then sas.put_value(dsid_s, "asofdatetime",
define["@AsOfDateTime"]) end
    if define["@Originator"] then sas.put_value(dsid_s, "originator",
define["@Originator"]) end
```

```lua
    if define["@SourceSystem"] then sas.put_value(dsid_s, "sourcesystem",
define["@SourceSystem"]) end
    if define["@SourceSystemVersion"] then sas.put_value(dsid_s,
"sourcesystemversion", define["@SourceSystemVersion"]) end
    sas.put_value(dsid_s, "studyoid", define.Study["@OID"])
    sas.put_value(dsid_s, "metadataversionoid",
define.Study.MetaDataVersion["@OID"])
    sas.update(dsid_s)
    sas.close(dsid_s)

    local itemtbl = {}
    for i, it in ipairs(define.Study.MetaDataVersion.ItemDef) do
      items = {}
      items["Name"] = it['@Name']
      if it.Description then items["Description"] =
it.Description.TranslatedText[1] end
      items["DataType"] = it['@DataType']
      items["Length"] = tonumber(it['@Length'])
      items["DisplayFormat"] = it['@DisplayFormat']
      itemtbl[it['@OID']] = items
    end

    -- print(tableutils.tprint(itemtbl))
    -- print(tableutils.tprint(define.Study.MetaDataVersion.ItemGroupDef))

    sas.submit[[
       %create_template(type=TABLES, out=@metadatalib@.metadata_tables);
    ]]
    dsid_t = sas.open(metadatalib..'.metadata_tables', "u")
    sas.submit[[
       %create_template(type=COLUMNS, out=@metadatalib@.metadata_columns);
    ]]
    dsid_c = sas.open(metadatalib..'.metadata_columns', "u")
    local tbl = {}
    for i, itgd in ipairs(define.Study.MetaDataVersion.ItemGroupDef) do
      sas.append(dsid_t)
      sas.put_value(dsid_t, "OID", itgd['@OID'])
      sas.put_value(dsid_t, "name", itgd['@Name'])
      if itgd.Description then sas.put_value(dsid_t, "label",
itgd.Description.TranslatedText[1]) end
      sas.put_value(dsid_t, "domain", itgd['@Domain'])
      sas.put_value(dsid_t, "repeating", itgd['@Repeating'])
      sas.put_value(dsid_t, "isreferencedata", itgd['@IsReferenceData'])
      sas.put_value(dsid_t, "structure", itgd['@Structure'])
      sas.update(dsid_t)
```

```lua
    itemref = itgd.ItemRef
    for j, it in ipairs(itemref) do
      sas.append(dsid_c)
      sas.put_value(dsid_c, "dataset_name", itgd['@Name'])
      sas.put_value(dsid_c, "OID", it['@ItemOID'])
      sas.put_value(dsid_c, "name", itemtbl[it['@ItemOID']].Name)
      sas.put_value(dsid_c, "label", itemtbl[it['@ItemOID']].Description)
      sas.put_value(dsid_c, "xml_datatype", itemtbl[it['@ItemOID']].DataType)
      sas.put_value(dsid_c, "order", tonumber(it['@OrderNumber']))
      if tonumber(itemtbl[it['@ItemOID']].Length) ~= nil then
sas.put_value(dsid_c, "length", itemtbl[it['@ItemOID']].Length) end
      if itemtbl[it['@ItemOID']].DisplayFormat ~= nil then
sas.put_value(dsid_c, "DisplayFormat", itemtbl[it['@ItemOID']].DisplayFormat) end
      if it['@KeySequence'] ~= nil then sas.put_value(dsid_c, "keySequence",
tonumber(it['@KeySequence'])) end
      sas.put_value(dsid_c, "json_datatype",
datatype_mapping[itemtbl[it['@ItemOID']].DataType])
      sas.update(dsid_c)
    end

  end
  sas.close(dsid_c)
  sas.close(dsid_t)

sas.filename('define')
```

```sas
%* update this location to your own location;
%let root=<your project folder>;
%include "&root/programs/config.sas";

%macro CreateMetadataFromDefineXML(definexml=, metadatalib=);
  proc lua infile='read_definexml';
  run;
%mend CreateMetadataFromDefineXML;


/* Create metadata from Define-XML for ADaM */
%CreateMetadataFromDefineXML(
    definexml=&root/json/adam/define.xml,
    metadatalib=metaadam
    );

/* Some manual data type updates */
data metaadam.metadata_columns;
  set metaadam.metadata_columns;
  if xml_datatype='float' then do;
```

```sas
    if index(name, 'VISIT') then json_datatype='decimal';
  end;
run;

/* Create metadata from Define-XML for SDTM */
%CreateMetadataFromDefineXML(
   definexml=&root/json/sdtm/define.xml,
   metadatalib=metasdtm
   );

/* Some manual data type updates */
data metasdtm.metadata_columns;
  set metasdtm.metadata_columns;
  if xml_datatype='float' then do;
    if name ne 'LBSTRESN' then json_datatype='decimal';
  end;
run;
```

Appendix 2 Validating a Dataset-JSON file against the Dataset-JSON schema

```python
import argparse
from os import listdir, getenv
from os.path import isfile, join
import json
import jsonschema as JSD

def validate_json(json_data, schema_file):
    """
    Validates a Dataset-JSON file against a defined json schema, given a
schema_file

    Arguments:
        json_data: The resulting Dataset-JSON file to validate
        schema_file: Path to a schema file defining dataset-JSON schema
    """
    try:
        with open(schema_file) as f:
            schema = json.load(f)
        JSD.validate(json_data, schema=schema)
        print("  Ok!")
        return True
    except Exception as e:
        print(f"Error encountered while validating json schema: {e}")
        return False

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument( "-d", "--directory", help="Directory location for
.json files", dest="directory", required=True)
    parser.add_argument( "-s", "--schemafile", help="Location for .json
schema", dest="schemafile", required=True)
    return parser.parse_args()

if __name__ == "__main__":

    args = parse_args()

    schemafile = args.schemafile

    files = [join(args.directory,f) for f in listdir(args.directory) if
isfile(join(args.directory, f))]
    print(f"Validating against schemafile {schemafile}")
    for f in files:
```

```
        isJson = f.find(".json")
        if isJson > 0:
          print(f"Validating {f}")
          validate_json(json.load(open(f, encoding="utf8")), schemafile)
```

Result：

```
Error encountered while validating json schema: 'label' is a required
property
...
On instance['clinicalData']['itemGroupData']['ADAE']['items'][1]:
    {'OID': 'ADAE.STUDYID',
     'length': 12,
     'name': 'STUDYID',
     'type': 'string'}
```

In this case the STUDYID variable in the ADAE Dataset-JSON file did not have a required label attribute.