Paper TT11

An Evaluation of Data Formats for Clinical Reporting in a Multilingual Environment

Craig Gower-Page, Novartis, London, UK

ABSTRACT

The use of non-SAS programming languages for analyzing clinical trials data is becoming increasingly common. This trend raises an important question: How can we format and store clinical data in a way that ensures interoperability without compromising on features or performance?

This paper explores the strengths and weaknesses of several potential data formats, ranging from human-readable options like CSV and JSON to more advanced binary formats such as DuckDB and Parquet. Each format is assessed based on key criteria, including the availability of language-specific libraries, performance characteristics, and support for various data types.

Particular attention is given to the Apache Parquet format, a highly efficient binary data format that stands out for its robust performance, compatibility with popular analytical programming languages, and seamless integration with many widely used tools and databases. This paper highlights Apache Parquet as the recommended choice for modern clinical data storage and analysis.

INTRODUCTION

The current standard format for storing and internally sharing CDISC-compliant clinical trial data, particularly SDTM- and ADaM-compliant datasets is the SAS ® .sas7bdat file format; This convention largely stems from the historical ubiquity of the SAS programming language in the creation of such datasets.

In recent years however, there has been a significant increase in the use of the R programming language as an alternative to SAS for developing clinical trial deliverables. In parallel, there has been a steady rise in the secondary use of clinical trial data by other teams and external researchers who employ a much broader range of data analysis tools and programming languages.

A key limitation of the .sas7bdat file format is that it is proprietary. While it can now be read universally, thanks to reverse engineering efforts such as the ReadStat C library, there remains no reliable open-source library capable of writing data in this format. For historical context, before the introduction of the {haven} R package, which made the ReadStat library easily accessible, data interoperability between R and SAS was poor. This often required SAS programmers to create separate copies of datasets in alternative formats, such as .csv, for R users to consume. With the growing use of R and other languages for creating clinical trial data, there is a risk that users will again store data in language-specific formats, such as .rds for R or .pickle for Python—both of which lack interoperability.

There is therefore a clear need for a language-agnostic data format that maximizes interoperability while maintaining key performance characteristics such as read and write speeds and file size efficiency. This paper examines several potential formats and outlines their respective strengths and weaknesses.

Disclaimer - While this paper aims to remain as objective as possible, certain topics, such as cost–benefit trade-offs between formats, are inherently subjective. Any opinions expressed are solely those of the author and do not represent the views or position of Novartis.

DATABASES

It is worth highlighting that the natural solution for this problem is to use dedicated database software such as PostgreSQL. However, doing so would require significant changes to the workflows used by many companies, as such the use of such software is outside the scope of this paper which will instead focus on solutions that can act as drop-in replacements for .sas7bdat files. That being said, organizations that can update their processes to take advantage of databases are encouraged to do so as this is the recommended solution to this problem.

TERMINOLOGY AND CORE CONCEPTS

This section provides a conceptual overview of key terms commonly used when discussing data storage formats. Its

goal is to provide an intuitive understanding of core concepts rather than a precise technical explanation.

BINARY VS PLAIN TEXT

A key differentiator between file formats is whether they are binary or plain text. While most readers will already be familiar with this distinction, it is worth revisiting the details, as they help explain the performance differences between the two.

Plain text formats store data as printable characters, typically ASCII or Unicode, that can be directly viewed in any text editor. In contrast, binary formats store data in a compact, machine-readable form. As a general rule, plain text formats have slower read and write speeds and produce larger files. Understanding why requires a closer look at how data is represented.

Consider the number 1921. In memory, this might be stored as the hexadecimal value 0x07 0x81, requiring just 2 bytes. To write this number to disk in plain text, the computer must:

- Convert the binary value to its decimal representation,
- Determine the number of digits needed, and
- Encode each digit using its ASCII or Unicode code point.

Since the decimal representation “1921” consists of 4 characters, the plain text version requires 4 bytes of storage, double the space of the binary equivalent. A binary format, on the other hand, can simply write 0x07 0x81 directly to disk without conversion.

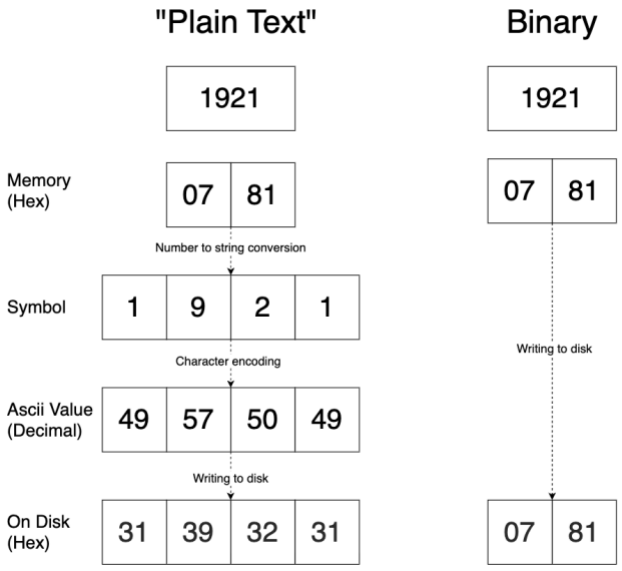


Figure 1: Comparison between how a number is stored on disk as a “plain text” format vs a binary format

In short, plain text formats incur significant overhead due to the conversion between in-memory and on-disk representations of data, while also consuming more storage space.

The main advantage of plain text formats is their human readability, that is they can be viewed in any text editor without requiring specialized software. However, this benefit is often overstated. Even moderately sized datasets (e.g., 500,000 to 1 million rows) are typically too large to load or browse effectively in most text editors. And even when such files can be opened, doing so provides little practical value, as meaningful analysis almost always requires dedicated query or processing tool.

Another common argument is that plain text formats are more likely to remain readable or parseable in the future. This too is debatable. Many plain text formats have existed for decades, with mature and stable specifications that rarely change. By contrast, many modern binary formats are newer and still evolving. Their perceived fragility stems not from them being binary but from the fact that their specifications are actively evolving. In other words, it is stability of the format, not its textual nature, that determines long-term durability.

COLUMNAR VS ROW-BASED STORAGE

When storing data on disk, particularly in binary formats, a key design choice is whether to use a row-oriented or column-oriented layout. The following image illustrates the difference between the two:

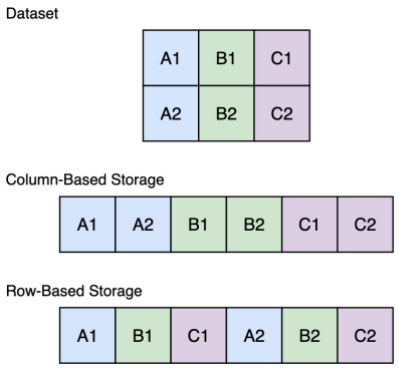


Figure 2: Example of memory layout for row-based and column-based storage

The optimal choice depends entirely on the intended use case. Row-oriented formats perform best for transactional workloads, where operations typically read or write individual records (e.g., bank transactions or inventory updates). Column-oriented formats, by contrast, are designed for analytical workloads, where operations scan or aggregate entire columns (e.g., computing statistics across many rows).

In both cases, performance gains arise from data locality, that is storing related values together so that the system can read them sequentially rather than seeking across multiple file locations.

In clinical trials, the data are almost exclusively analytical in nature; therefore, column-oriented formats generally provide superior performance.

BINARY ENCODING

One advantage of binary formats is that, since they do not need to be human-readable, they can use more efficient encoding schemes to represent data. Some common examples include run-length encoding (RLE), bit-packing and dictionary encoding.

- **Run-Length Encoding (RLE)**: There are many variations of RLE, but the basic idea is simple: instead of storing each repeated value individually, store the value once along with the number of repetitions. For example, the sequence AAABBBBAA can be stored as A3B4A2.

- **Bit-Packing**: Boolean values are conceptually just 1 or 0, yet most programming languages allocate a full byte (8 bits) for each value. For example, the sequence True, True, False might be stored as 00000001 00000001 00000000. Bit-packing compresses these values by using only the bits required, in this case all three values could fit into a single byte: 00000110.

- **Dictionary-encoding**: Instead of storing explicit values, dictionary encoding stores numeric indices along with metadata that maps each index to its corresponding value. For example, data 1121313 with dictionary {1: cat, 2: dog, 3: rabbit} represents the sequence cat, cat, dog, cat, rabbit, cat, rabbit.

These techniques make data representation significantly more space efficient. A secondary benefit is improved I/O performance: smaller files mean less data to read or write, and the CPU overhead of encoding and decoding is typically negligible compared to disk access times, which are orders of magnitude slower.

DESIRABLE CHARACTERISTICS OF A TABULAR DATA FORMAT

There is no universally optimal format for storing data. As discussed in the section on row-oriented vs. column-oriented data, different formats are suited to different use cases. It is therefore important to identify the key characteristics that should be prioritized when selecting a data format for storing clinical trial data.

For the purpose of this paper, the following criteria (listed in approximate priority order) are used. These are intended to reflect general industry requirements, though specific organizations may weigh them differently based on their use cases.

- **Availability of Implementation Libraries:** How accessible is the format across programming languages? The primary goal is to select a format that is supported by major languages such as R, SAS, Python, and Julia, thereby maximizing the accessibility of the data.

- **Data Type Coverage:** How expressive is the format? It should support a broad range of data types, including dates, datetimes, categorical variables (factors/dictionaries).

- **I/O Performance:** Although clinical trial datasets are typically small to moderate in size, read and write performance can substantially impact both developer productivity and the execution time of critical-path programs. It is important to note though that I/O performance is highly dependent on the optimizations and implementation quality of the underlying library, and can therefore vary significantly across programming languages. In this paper, performance comparisons are presented primarily from the perspective of the most widely used R library for each data format (for example, the {arrow} package for reading Parquet files), though results from other languages and libraries are also considered.

- **Tool Integrations:** How well is the format supported by the broader ecosystem? Integrations with tools such as data browsers, databases, and cloud platforms can greatly enhance productivity and interoperability.

- **File Size:** Although file size is often less critical for clinical trial data, larger files can still increase storage costs and impact performance over time. It remains a secondary concern but still a worthwhile consideration.

HIGH LEVEL SUMMARY OF CANDIDATE FORMATS

This section provides a high-level summary of the available formats, outlining their general strengths and weaknesses. Given the number of formats considered, only an overview is presented here to convey a surface level intuition of the main advantages and limitations of each format.

COMMA SEPARATED VALUES (CSV)

A plain-text, row-based format. CSV is often considered the lowest common denominator for tabular data storage - a format that nearly all applications can read and write. Although largely superseded in spreadsheet software by .xlsx and .ods, CSV remains widely used due to its simplicity and interoperability.

\+ **Ubiquity**: Almost every programming language provides libraries for reading and writing CSV files. Its widespread support across tools and platforms makes it a convenient lightweight exchange format.

– **Lack of standardization**: No single official specification exists. Variations in delimiters (commas, semicolons, tabs), quoting, and newline conventions often lead to interoperability problems.

– **Limited data types & no metadata**: CSV files do not store schema information such as column names, data types, or encodings. Parsers must infer types, which can cause inconsistencies (e.g., a date parsed as text in one tool but as a date in another).

– **Poor I/O performance**: As a text-based format, CSV typically results in slower I/O and larger file sizes than binary formats. However, its simplicity and tool optimization means it is often efficient for small to medium datasets.

JAVASCRIPT OBJECT NOTATION (JSON)

JSON is a lightweight, text-based format used to represent structured data as key–value pairs and ordered lists. Although derived from JavaScript syntax, it is language-independent and widely used for data interchange between systems and APIs.

\+ **Human-readable and universally supported:** JSON's simple text-based syntax is easy to read and write. Most modern programming languages include built-in or standard libraries for parsing and generating JSON, and there is a large ecosystem of tools for working with JSON data.

– **Lack of schema enforcement:** JSON does not provide built-in type or structure validation. Consumers must rely on external schema definitions (e.g., JSON Schema) to ensure consistency. There is also no standard convention for representing tabular data, such as whether to store it row-wise or column-wise.

- **Poor I/O performance:** Being a text-based and structurally verbose format, JSON is slower to parse and serialize compared to simpler text formats like CSV or binary formats more generally

- **Limited data types**: JSON supports only a small set of primitives: string, number, boolean, array, object, and null. More complex types (e.g., dates or timestamps) must be represented via one of these primitives and then inferred by the user / library.

- **Large file sizes**: JSON's plain-text and repetitive syntax result in larger file sizes compared to more compact or binary data formats.

It is worth mentioning at this point CDISC's Dataset-JSON format [1], which can be viewed as a specialized extension of JSON. As such, it inherits most of JSON's advantages and limitations though with two notable exceptions: Dataset-JSON standardizes the representation of tabular data and provides explicit support for additional logical types such date and timestamp types.

However, this comes at the cost of limited adoption, few libraries exist for parsing or generating Dataset-JSON and its use is largely restricted to the pharmaceutical and clinical research domains, making it unlikely to gain widespread support from the broader computer science and software development communities.

SAS TRANSPORT FILE (XPT)

A binary, platform-independent file format for tabular data created by SAS Institute to enable data exchange across operating systems. It was designed as a re-serialization of SAS datasets, and thus inherits many of their structural and design characteristics.

+ **Widespread Adoption in Pharma:** The XPT format is widely used within the pharmaceutical industry, particularly due to its longstanding FDA acceptance [2]. Its maturity has resulted in extensive tooling support. For example, the ReadStat C library provides XPT file bindings, which underpin many language-specific libraries (e.g., haven in R and pyreadstat in Python).

- **Limited data types:** The XPT format supports only numeric and character variables. Additional "type hints" can be provided via the SAS format system, but these are intended primarily for display purposes. As a result, there are multiple display formats for a single logical type, for example, DATE., DATE9., DDMMYY., DDMMYY10., DDMMYYC. for date variables. This creates a maintenance burden for implementation libraries, which must interpret and stay aligned with SAS's extensive and evolving list of formats.

- **Poor I/O Performance:** Despite being binary and supported by efficient C-based parsers, XPT read/write performance is often slower than CSV. Additionally, because the format does not support compression, files are typically larger than equivalent datasets stored in modern formats.

- **Legacy Limitations:** Due to its age, the XPT format imposes several constraints, such as 8-character variable names and 40-character labels. While the Version 8 (XPT v8) format relaxes some of these limits, Version 5 (XPT v5) remains the most widely used in regulatory submissions. The format also lacks support for missing character values, and many parsers fail to handle special numeric values like NaN or Inf.

SQLITE DATABASE (SQLite)

A lightweight, self-contained, and transactional relational database engine that stores the entire database in a single cross-platform file. Unlike server-based systems (e.g., PostgreSQL or Oracle), SQLite runs entirely in-process, allowing applications to query and modify data directly without a separate database server or management process. As a result, SQLite can be treated much like a regular data file, while still providing full database functionality.

+ **Ubiquitous:** SQLite is natively supported or has mature bindings in nearly every major programming language, including R (RSQLite), Python (sqlite3), Java, C/C++, Go, and JavaScript (via WebAssembly).

+ **Ecosystem Integration:** Supported by many analytical and data-engineering tools, including Power BI, Tableau, and cloud data services (e.g., AWS Athena). SQLite files can be opened directly in browsers or desktop viewers, and libraries such as DuckDB and Arrow can query SQLite databases directly.

+ **Reasonable I/O Performance:** SQLite's read/write performance is generally faster than plain-text formats (e.g., CSV) and some binary formats (e.g., SAS XPT). However, it is typically slower than columnar analytical formats such as Parquet or Feather.

- **Large File Size:** SQLite databases are often larger than equivalent binary or compressed formats, sometimes approaching or exceeding plain-text file sizes due to page-level storage overhead and lack of

columnar compression.

– **Weak Typing Semantics**: SQLite uses a type affinity system. Columns can be declared as one of five base types (NULL, INTEGER, REAL, TEXT, BLOB), but any data type can still be inserted into any column. This system behaves more like type hints than strict enforcement. Additionally, SQLite lacks native date or timestamp types, instead storing them as TEXT or INTEGER values. This flexible typing can complicate interoperability with systems requiring strict schema validation (e.g., Pinnacle 21).

DUCKDB DATABASE (DuckDB)

An in-process, columnar analytical SQL engine that delivers high performance and strong interoperability in a lightweight, single-file format. In essence, DuckDB can be viewed as "SQLite for analytical data."

\+ **Library Availability:** Mature bindings exist for Python, R, C/C++, and Java, with growing support for Julia, Rust, and JavaScript (via WebAssembly). As of July 2025, SAS also supports reading and writing DuckDB files.

\+ **I/O Performance & File Size:** DuckDB uses a columnar storage format optimized for analytical workloads, unlike SQLite's row-oriented design for transactional use. This enables excellent read/write performance, and its compact binary format results in efficient file sizes.

\+ **Excellent Data Type Support:** Provides strong, explicit typing with native support for DATE, TIMESTAMP, INTERVAL, and categorical data types.

\+ **Tool integration:** A growing ecosystem of tools, including DBeaver, Tableau, and Power BI, now support DuckDB natively. The DuckDB engine also includes built-in integration with external data formats, such as a native Parquet and CSV reader. There are also community extensions that allow the DuckDB engine to read SAS datasets and XPT files (also via the ReadStat library).

~ **No Native Metadata Support**: DuckDB lacks a built-in mechanism for storing column labels or arbitrary metadata. However, because each DuckDB file is a self-contained database, metadata can be managed via convention, e.g., by maintaining a dedicated "metadata" table with column labels and related information.

APACHE PARQUET

A columnar, open-source binary file format designed for efficient analytical data storage and processing. It is widely used across modern data platforms and can be viewed as a standard exchange format for analytical data.

\+ **Library Availability:** Parquet is supported across nearly all major data ecosystems, with mature libraries in Python (pyarrow, fastparquet), R (arrow), Java, JavaScript, C/C++, Go, Ruby, Julia, C#, Swift and Rust.

\+ **I/O Performance & File Size:** Parquet's columnar layout enables excellent read performance for analytical queries and very compact file sizes.

\+ **Excellent Data Type Support:** Parquet enforces strong, explicit typing with native support for DATE, TIMESTAMP and complex/nested structures (lists and maps).

\+ **Tool integration:** Parquet is natively supported by nearly all modern analytics and data engineering tools, including DBeaver, Power BI, Snowflake, Spark, BigQuery, AWS Athena, Apache Drill, Apache Hive and DuckDB.

~ **Arbitrary Metadata Support**: Whilst Parquet stores structural schema metadata (names, types, and encodings), it does not explicitly store semantic metadata such as labels or derivations. However, the format supports arbitrary key-value metadata, allowing semantic information to be stored and retrieved by convention if required.

It is also worth highlighting here the Feather format. Many Parquet parsing libraries load data into memory in the Apache Arrow format. In fact, the C++ Parquet implementation is maintained by the Apache Arrow project and serves as the foundation for libraries such as the R {arrow} package and Python's pyarrow library. Feather essentially represents this in-memory Arrow layout serialized directly to disk. Compared with Parquet, it typically offers faster read and write performance but produces significantly larger files. It should be noted that the Feather format has two major versions V1 & V2; unless stated otherwise, all references to the Feather format in this paper should be interpreted as the V2 format also known as the Arrow IPC file format.

APACHE ORC

A columnar, open-source binary storage format optimized for high-performance analytical and big data workloads. Originally developed for Apache Hive, ORC is designed to provide efficient compression, fast predicate pushdown, and optimized read performance. Although not explicitly tested during the research for this paper, available online benchmarks [3] suggest that ORC often performs slightly better than Parquet in terms of I/O efficiency and file size reduction. In general, its functionality and use cases largely overlap with Parquet; however, Parquet enjoys broader library support and adoption while ORC remains more prevalent within Hadoop-based environments.

+ **I/O Performance & File Size:** ORC's columnar design, built-in compression enable excellent query performance and compact file sizes often outperforming Parquet in selective read workloads.

+ **Excellent Data Type Support:** Provides strong, explicit typing with native DATE, TIMESTAMP and complex/nested types (lists, maps, structs).

~ **Library Availability:** Has libraries available for Java, C++, and Python (pyorc, pyarrow). Notably though there is no standalone library for R with access to the file only being available via Spark (i.e. needing a separate process / application to handle the I/O).

~ **Tool integration:** Widely supported within the Hadoop ecosystem and some other general database tools such as AWS Athena.

APACHE AVRO

A row-oriented, open-source binary format designed for data serialization and schema evolution. Originally developed for Apache Hadoop, Avro is optimized for streaming, messaging, and long-term data storage, rather than analytical querying.

+ **I/O Performance & File Size:** Avro delivers high performance for transactional and streaming workloads, though it is generally slower than Parquet or ORC for analytical queries.

+ **Excellent Data Type Support:** Provides explicit data typing with broad support for complex and logical types, including Dates, Timestamps, Enums, Arrays, and Maps.

~ **Library Availability:** Official libraries are available for Java, Python, C/C++, and C#, with community libraries for other languages. R support is limited and requires Apache Spark.

~ **Tool integration:** Widely supported in Hadoop, Spark, Kafka, Hive, and AWS Glue, though direct BI tool integration (e.g., Tableau, Power BI) remains limited.

OTHER FORMATS OF NOTE

This section outlines additional data formats that were considered but deemed unsuitable for the outlined requirements, specifically, being a drop-in replacements for SAS dataset files requiring minimal changes to existing workflows for storing heterogeneous relational data. This section is included solely for the reader's interest

- **HDF5**: A binary file format designed for storing and managing hierarchical, structured, and large-scale array data. It represents data as multidimensional, homogeneously typed arrays organized within a hierarchy of groups, each containing rich metadata describing content and relationships. HDF5 supports efficient I/O, compression, and complex data relationships, enabling flexible organization and retrieval of related datasets. It is widely used in domains such as sensor analysis, genomics, and image processing.

- **Apache Iceberg & Delta Lake**: These formats function as metadata and transaction management layers for large numbers of analytical datasets. They provide database-like features, including ACID transactions, schema evolution, versioning, and snapshot isolation, by managing metadata on top of underlying storage formats such as Parquet or ORC.

- **Protocol Buffers**: A binary, strongly typed, and schema-enforced serialization format that can be viewed as a more efficient and structured alternative to JSON that has been optimized for inter-process and network communication.

RECCOMENDATION

Across all currently available file formats, the clear standouts for storing and analyzing clinical trial data, while also serving as drop-in replacements for existing SAS dataset files, are Parquet and DuckDB. Although ORC offers slightly better compression and read/write performance, its more limited language support (particularly in R) makes it

less suitable than these two formats. In terms of the high-level requirements, as outlined at the start of this section, there is little difference between Parquet and DuckDB, and either would be a suitable choice for most organizations. However, due to Parquet's better performance, smaller file sizes, and more familiar API (i.e., access to a single file rather than querying tables through an SQL interface), the author's preferred choice is Parquet.

APACHE PARQUET

Given the above recommendation for Apache Parquet, this section aims to provide additional details about the format and general considerations that anyone wanting to adopt it should be aware of.

APACHE PARQUET VS APACHE ARROW

As mentioned in the high-level summary, the Apache Parquet and Apache Arrow projects are closely related. Apache Parquet is an on-disk columnar storage format, while Apache Arrow is an in-memory columnar data format. Although they are separate projects, the primary C++ implementation for reading and writing Parquet files is developed and maintained within the Apache Arrow project.

This C++ implementation underpins many other libraries, including the R {arrow} and Python pyarrow packages. As a result, when you read Parquet data into memory using these libraries, the data is automatically converted into the Arrow in-memory format.

Because of this tight integration, the two projects are often conflated. However, it is important to understand that Parquet and Arrow serve different purposes within the data processing workflow—Parquet optimizes storage on disk, while Arrow optimizes data representation and processing in memory.

DATA TYPE MAPPING

A common source of confusion when users begin working with language-agnostic data storage formats is how data types are represented and converted. This confusion arises because language-specific formats (such as SAS7BDAT, RDS, and Pickle) serialize a language's data structures directly, with no need to explicitly manage type mappings.

When using a language-agnostic format, however, there is rarely an exact one-to-one correspondence between a programming language's types and those supported by the storage format. This issue is particularly relevant for Apache Parquet, as there are two layers of type mapping:

1. Between Parquet and Apache Arrow (as described in the previous section), and
2. Between Arrow and the client programming language.

As an example Figure-3 outlines the data mappings between Apache Arrow data types (light blue boxes) to R data types (black boxes) as performed by the {arrow} R package

In practice, Parquet's broad type support allows most language client libraries to handle these mappings automatically, making the process largely transparent to users. However, there are several important considerations:

- Parquet supports data types not natively available in R, such as Unions and float16.
- SAS, with its more limited type system, cannot represent complex data structures. For example, factor (dictionary) variables are converted to character strings, losing the underlying numeric index.
- Parquet differentiates between integer subtypes (e.g., int16, int32, int64), but when read into R, all of these map to the same Integer type. As a result, performing a round trip (Parquet → R → Parquet) leads to loss of type fidelity, since all integers will be stored as int64; that is unless the users code explicitly manages the type specification.
- Implicit data conversions often occur during Parquet read/write operations. For instance, R stores timestamps as seconds, whereas Parquet stores them as milliseconds, so the client library must perform automatic conversions on the fly when reading and writing data.

METADATA & COLUMN LABELS

As mentioned in the High-Level Summary section, Parquet supports storing arbitrary metadata as key–value pairs in the file header. This flexibility is both a strength and a limitation. On the positive side, it allows inclusion of valuable metadata such as labels and semantic information. However, because there is no standardized structure for this metadata, each language implementation is free to define its own conventions, which may not be compatible with others.

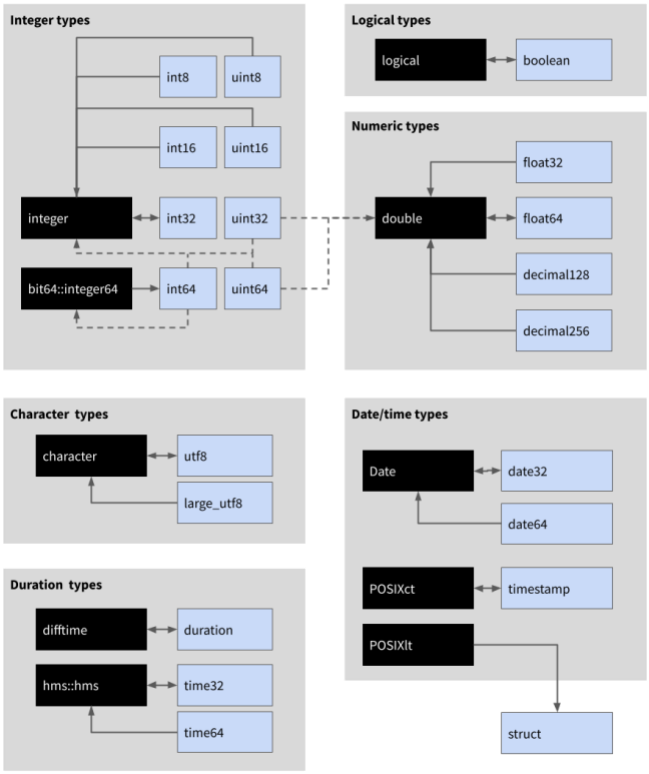


Figure 3: Data conversion mappings between Apache Parquet and R objects
Image sourced from the Apache Arrow R documentation (https://arrow.apache.org/docs/r/articles/data_types.html) © The Apache Software Foundation, licensed under the Apache License, Version 2.0.

For example, the R {arrow} library stores column attributes, often used for column labels, within the file metadata by serializing all attributes into a single binary blob under the "R" key. Other implementations, such as Python's pyarrow library, cannot interpret this format and therefore do not restore any of this metadata when reading the file. Similarly, the SAS LIBNAME engine for Parquet ignores column labels entirely, neither reading nor writing them.

The key takeaway is that while the industry could define a common convention for storing such metadata, it is unlikely to be natively supported by existing libraries. This would require maintaining custom, domain-specific libraries or wrappers, reducing interoperability and undermining Parquet's cross-language compatibility, similar to the challenges discussed for Dataset-JSON.

Therefore, since embedding semantic metadata directly in Parquet files is a industry-specific requirement rather than a general use case, the Novartis internal recommendation is to store such metadata in separate JSON files rather than within the raw Parquet data.

SAS INTEGRATION

Another key consideration is that the SAS LIBNAME engine [4] for reading and writing Parquet files is currently implemented with some questionable design decisions. Whether or not these are actual problems depends on the specific use case; however, they are listed here for transparency:

- By default, SAS reads all character values into 42 KB–length character variables. This results in excessively large datasets and severely degraded performance. For example, a dataset that loads in under one second in R can take up to six minutes in SAS. This issue can be partially mitigated using the CHAR_COLUMN_LIMIT option; however, this setting applies globally and will silently truncate data if set too low.
- The LIBNAME Parquet integration is available only in SAS Analytics Pro and SAS Viya, meaning users of SAS 9.4 cannot use it.
- When writing Parquet files, SAS determines the Parquet data type based on the variable's informat rather than its format. For instance, a numeric column with a DATE9. format will still be written as a numeric type

rather than a Parquet date type. Because informats are rarely specified during processing, this behavior can lead to data type loss when writing Parquet files.
- As noted previously, the SAS LIBNAME engine does not save nor restore column labels.
- Earlier versions of SAS Analytics Pro / SAS Viya contain critical bugs. For example, timestamp variables were incorrectly converted between milliseconds and seconds, and missing numeric values were, under certain scenarios, written as zeros. It should be noted though these issues have been fixed and are not present in the current releases.

Given these limitations, it is recommended not to use the SAS LIBNAME engine for Parquet files. Instead, organizations are encouraged to develop wrapper macros to work around these shortcomings. For example:

- a macro that uses R or Python to convert Parquet files to XPT format on the fly for SAS to read.
- Alternatively, a call to a Python script to cast character columns to fixed-width binary columns, which SAS can read without issue.
- Another option is to use the SAS DuckDB integration, which includes its own Parquet reader and writer and avoids many of the above issues. Please note that this approach was not evaluated as part of the research for this paper but may offer better interoperability.

While these workarounds may reduce ease of use for SAS users, the trade-off is considered worthwhile to maintain language-agnostic data access for the broader analytics community. Since SAS continues active development, it is reasonable to expect improvements in its Parquet interfaces over time in response to industry feedback.

CONCLUSION

In this paper, the need for a language-independent data storage format has been highlighted to avoid issues with data lock-in as the industry becomes increasingly multilingual. Several formats were discussed, ranging from popular plain-text formats (CSV, JSON) to modern binary alternatives (Parquet, Avro, ORC).

Among the currently available options, Parquet and DuckDB were identified as the best candidates for fulfilling the requirements of a drop-in replacement for the existing industry standard of SAS datasets. Parquet, in particular, was found to be the most suitable for direct replacement, with additional details provided on specific format-related considerations.

It is worth emphasizing that if organizations can update their workflows, it is recommended to adopt dedicated database software such as PostgreSQL. However, in lieu of such options, Parquet remains the recommended file format for organizations facing this challenge.

REFERENCES

[1] CDISC, Dataset-JSON, https://www.cdisc.org/standards/data-exchange/dataset-json (accessed Oct 2025)
[2] FDA, “Specifications for File Format Types Using eCTD Specifications”, https://www.fda.gov/media/85816/download (accessed Oct 2025)
[3] Rahul Sarabu, “Performance, Cost Implications: Parquet, Avro, Orc” https://dzone.com/articles/performance-and-cost-implications-parquet-avro-orc (accessed Oct 2025)
[4] SAS, “LIBNAME Statement for Parquet”, https://documentation.sas.com/doc/en/workbenchcdc/v_001/vwbacdata/n091pc8v4twa0rn197odmyxtoqjf.htm (accessed Oct 2025)

ACKNOWLEDGMNTS

This paper was written with the support of ChatGPT-5 (OpenAI, 2025). Each paragraph was written by the author before being given to the AI with various prompts of the form “please re-write the following if needed to ensure it is as concise as possible without sacrificing clarity and ensuring technical accuracy”. The output was then reviewed and adjusted by the author as needed to ensure key points were not lost and that the language still resembled the author’s style of writing

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:
Author Name: Craig Gower-Page
Company: Novartis Pharmaceuticals UK Limited
Address: The WestWorks, White City Place, 195 Wood Lane, London, W12 7FQ
Email: craig.gower-page@novartis.com

Brand and product names are trademarks of their respective companies.

APPENDIX

PERFORMANCE BENCHMARKS

The following section presents the raw data and graphs for the performance metrics for selected candidate file formats tested in R and Python. These metrics were obtained using a dataset of one million rows and 84 columns (12 columns for each of the following data types: integer, float, logical, date, datetime, character, and factor/dictionary) containing randomly generated data.

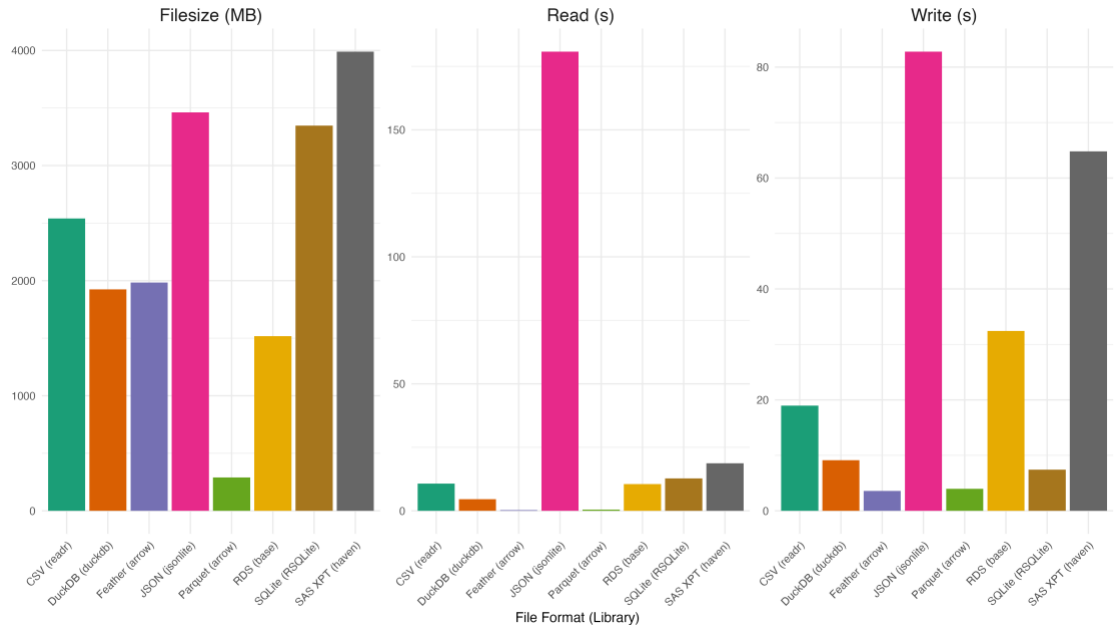
For each format, the read and write times to disk were recorded along with the resulting file size. Each operation was repeated three times, and the average values are reported below. All tests were conducted on a Lenovo P1 Gen 7 laptop equipped with an Intel Ultra 7 155H (x86_64) CPU and 32 GB LPDDR5X RAM.

The selection of formats was based on library availability; for example, ORC was excluded because it cannot currently be read in R without Apache Spark.

R

Package	Version
R (base)	4.5.1
readr	2.1.5
duckdb	1.4.0
arrow	21.0.0.1
jsonlite	2.0.0
RSQLite	2.4.3
haven	2.5.5

Format	Write (secs)	Read (secs)	File Size (MB)
CSV (readr)	18.98	10.72	2539
DuckDB (duckdb)	9.10	4.54	1923
Feather (arrow)	3.58	0.28	1983
JSON (jsonlite)	82.80	180.80	3461
Parquet (arrow)	3.98	0.45	288
RDS (base)	32.41	10.53	1517
SQLite (RSQLite)	7.40	12.71	3346
SAS XPT (haven)	64.80	18.75	3990



PYTHON

Module	Version
Python	3.12.8
pandas	2.3.3
duckdb	1.4.1
pyarrow	21.0.0
pyreadstat	1.3.1

Format	Write (secs)	Read (secs)	File Size (MB)
CSV (pandas)	118.90	14.84	2677
DuckDB (duckdb)	8.53	3.13	1922
Feather (pyarrow)	2.38	1.11	1984
JSON (json)	60.64	30.49	3436
Parquet (pyarrow)	2.90	0.32	289
Pickle (pickle)	0.85	0.38	379
SQLite (sqlite3)	69.38	16.97	3630
SAS XPT (pyreadstat)	110.99	41.15	3933