Wisent: An In-Memory Serialization Format for Leafy† Trees

Hubert Mohr-Daurat[1], Holger Pirk[1]

[1]Imperial College London

Abstract

Efficient data exchange is one of the key ingredients for high-performance, composable data management systems. Efficient data exchange formats exist for "flat" (i.e., relational) data. For nested data, however, users must resort to formats such as XML, JSON or their binary counterparts such as BSON or CBJSON. These formats provide the flexibility to store metadata as well as actual data but lead to costly serialization and deserialization. This makes them unfit to represent flat data, forcing users to combine flat formats (Arrow, Parquet and even CSV) for data with JSON or XML documents for metadata. This practice, known as "Data Packages", is error-prone, labor-intensive and increases system complexity.

To address this problem, we propose Wisent, a new exchange format designed to represent nested data while efficiently encoding the "flat" sections of the tree. We call such trees "leafy". To this end, Wisent serializes trees bread-first rather than the conventional depth-first serialization. We found that breadth-first serialization enables lazy decoding during tree navigation and in-place/conversion data processing of the flat sections using tight loops. We implemented a Wisent deserializer in C++, which outperforms the state-of-the-art data serialization protocols by several orders of magnitude. Wisent is not just faster to encode and decode but also much simpler to implement: to demonstrate that aspect, we implemented Wisent decoders in Swift and Python and found that it can be implemented in roughly 100 lines of code while achieving performance that is dominated only by the overhead of the respective host language.

```
{"data":{
  "startTime": "2023-06-14 07:16:47 +0100":,
  "samples":[
    {"heartrate":56}, {"heartrate":69},
    {"heartrate":60}, {"heartrate":60},
    {"heartrate":71}, {"heartrate":68},
    {"heartrate":80}, ...
]}}
```

Figure 1: A Leafy JSON Document

1. Introduction

Data exchange is a key technique enabling Data Management System composition. Particularly, when composing systems written in different languages and/or without control over the development, ABI-level communication is not an option, and efficient inter-process communication is required for transferring data between systems. Fundamentally, two kinds of data need to be exchanged: flat data, i.e., data in which data items have no designated hierarchy and nested data, i.e., data in which data items can have children (sometimes referred to as "unstructured" data). Many practical applications process some flat data (tables, streams, binary data, etc.) and some nested (metadata, query plans, ontologies, etc.). In fact, the two kinds of data are often describing the same real-world entity. The nested metadata could, e.g., describe when and where sensor data was acquired, while the flat data contains the sensor readings. This use case is prevalent enough to have merited the definition of the "Data Package Format" [1], a standard for packaging flat data and (nested) metadata in a directory.

However, implementing such a "separated" format in an application is difficult, labor-intensive and error-prone. To start, an application needs to integrate parsers for two different formats. Further, it needs to ensure that users provide the data's flat and nested parts and correctly pair the two (or more) files (usually based on their location in the directory structure). The Data Package format even allows multiple "versions" of the same flat data (e.g., different file formats, resolutions or sampling rates). In that case, an application needs to determine how they relate and which is the most appropriate to load. If one of the flat files is missing, the application needs to implement a fallback mechanism. These complications led to relatively poor adoption of the format.

The obvious alternative would be to hold all data in the same file: the flat data could be stored as leaf nodes in a (document) tree. This would lead to documents like the one illustrated in Figure 1:

† *we use the term "leafy" to say that most nodes are leafs or nodes that only have a single child which, itself, is a leaf.*

Processing such trees, however, is expensive as parsers must assume that every node has children even if most of them are merely leafs and, therefore, without children.

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW'23) — Second International Workshop on Composable Data Management Systems (CDMS'23), August 28 - September 1, 2023, Vancouver, Canada

h.mohr-daurat19@imperial.ac.uk (H. Mohr-Daurat);
hlgr@ic.ac.uk (H. Pirk)

Expensive parsers are acceptable when application performance is bound by interconnect bandwidth: when loading data through ethernet or from slow disks, e.g., a parser need not process more than hundreds of megabytes per second. However, with fast interconnects like Infiniband, NVMe or inter-process/inter-container communication becoming more commonplace, parsing is a likely bottleneck for many applications.

To address this problem, we propose Wisent, a new exchange format that allows the CPU-efficient encoding and, in particular, decoding of leafy trees. Wisent is designed with five objectives:

- it shall support the representation of nested data
- it shall minimize the storage overhead for leafs
- it shall allow efficient serialization, deserialization and in-place processing
- it shall support lazy/selective access to parts of the data without requiring the deserialization of the entire tree
- it shall be simple enough to not require the use of a parsing library but allow the implementation of an efficient parser in fewer than 100 lines of code in a modern programming language like Python or Swift

Note that we intentionally do not define space efficiency as an objective. As the use cases for Wisent are those with high-bandwidth communication channels (many of which even support random access), space efficiency is secondary. In addition, as lazy access is one of Wisent's objectives, space overhead potentially affects disk space and memory address space but only marginally affects physical RAM usage.

2. Data Serialization: State-of-the-Art

Flat Data Serialization

A number of storage formats have been proposed for flat data serialization. While Apache Arrow [2] only allows in-process transfer, the project integrates the Feather [3] serialization columnar format, designed for short-term inter-process communication (such as network sockets or shared memory). Apache Parquet [4] and ORC [5] also use a decomposed storage layout but are designed for longer-term storage on disk. In contrast, Apache Avro [6] uses a n-ary storage layout to serialize data to disk.

None of these formats are designed to embed metadata and other custom nested data. For this use case, they require to use side-by-side another *nested-data-oriented* serialization format. The nested data commonly embeds filenames or URLs for the related flat data resources.

LeanStore [7] uses a fragmented data layout and is implemented with intricate techniques to avoid the cost of serialization and deserialization from and to disk. However, the type of data handled by LeanStore's serialization is limited to flat data and b-trees.

Nested Data Serialization

JSON [8] is the most common text-based format for nested data serialization. The text representation makes it straightforward to integrate into any programming language but comes with a high cost for serialization and deserialization runtime performance. Binary-based formats such as BSON [9], Protocol Buffers [10] and Thrift [11] are common performance-oriented alternatives. Protocol Buffers and Thrifts efficiently serialize and deserialize custom nested data structures but require compiling the data structures into a target language's code. All these serialization formats use a depth-first tree representation, less adapted to efficient serialization and deserializing of leafy trees, such as trees embedding flat data. With Wisent, we propose a bread-first tree representation efficient for this use case.

Substrait [12] is a format specification designed to serialize trees of relational operations, and the reference serialization protocol is implemented on the top of Protocol Buffers. Flat data, such as tables, are referenced and not embedded in the nested data, as Wisent allows.

3. Design Principles

Three key design principles distinguish Wisent from state-of-the-art formats like JSON, BSON or XML: Breadth-first flattening (where existing formats perform depth-first), decomposition of data and structure (where existing formats mix the two) and a focus on simplicity of parsing where existing (binary) formats focus on compactness. Let us discuss those principles in the following.

Breadth-first Flattening. At the heart of every file format for hierarchical data (JSON, BSON, XML, ...) lies a transformation we call "flattening": the transformation of a hierarchy of "nodes" into an equivalent one-dimensional array of data items describing both the content as well as the structure of the tree. To the best of our knowledge, every serialization format for hierarchical data prescribes a depth-first flattening of the tree. While this corresponds to the "natural" way of thinking about nested data (i.e., as documents), depth-first flattening, by definition, scatters the children of a node throughout the (serialized) output buffer. This leads to two related problems: first, it requires some form of list of "pointers/indices/offsets/markers" to represent the children of a node (in xml, the closing tag plays that role). This leads to substantial storage overhead for leafy trees. Second, processing all children of a node requires resolving those

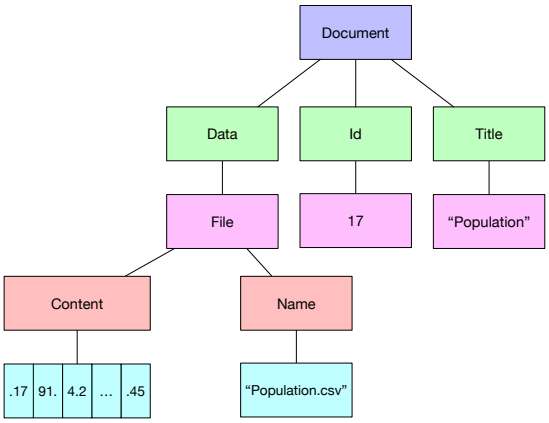

Figure 2: An Illustrative Leafy Document Tree

markers. While iterating through the children by looking for closing tags is particularly egregious, even if the children are represented as offsets in an in-memory array, the random memory accesses required to gather the leaf values are unnecessarily costly.

Breadth-first flattening, in contrast, places the children of a node in a single contiguous memory area. This not only eliminates the need for per-child pointers or markers in favor of a pair of pointers (one to the first, one to the last child). It also ensures that all children can be accessed with optimal locality.

Decomposition of Data and Structure. State-of-the-art formats for hierarchical data mix the representation of the structure of the tree (opening/closing tags, sibling offsets, end-markers, etc.) with the data itself (attribute values, text nodes, labels, etc.). This leads to a format in which the location of any data item (inner node or leaf) depends on the structure and data that appear before it in the output buffer. It is, for all practical purposes, unpredictable. This unpredictability is what requires a json/xml parser to parse the entire subtree of a node only to access its right sibling.

In contrast, we propose to decompose the structural elements from the data elements of a tree: data elements/values are flattened (breadth-first) in a buffer without any indication of their exact position in the tree. The parent-child relationships are encoded in a separate buffer. While such separation leads to worse locality for positional/navigational access to inner nodes, it allows access to the children and siblings of a node without having to parse other parts of the tree. This, in turn, enables “lazy parsing”, i.e., the navigation to a specific node without having to parse any nodes other than those along the path.

Simplicity over Size. The primary objective of most binary serialization formats is the minimization of the size of the result. This focus encourages several sophisticated techniques, such as using terminators over size fields, the dense packing of values and even data compression. Implementing this arsenal of techniques makes parsers complicated and hard to write, which is why most users use external libraries for parsing. Such external libraries often constitute optimization boundaries for the compiler: to manage the complexity of the optimization process, the compiler optimizes these libraries internally but not in the context of the code they are used in. This leads to substantial runtime overhead.

The design we propose, on the other hand, favors simplicity over output size. It is simple enough to allow the implementation of a (non-verifying) parser in fewer than 100 lines of code in a modern (i.e., reasonably succinct) programming language. This allows users to integrate the parser with the application code rather than using an external library.

4. The Wisent Data Format

To illustrate the Wisent serialization format, consider the serialization of the example tree (Figure 2) in Figure 3.

The buffer/file is divided into four sections: an *Argument Vector* that holds the breadth-first flattened representation of every node in the input tree, a *Type Bytefield* encoding the types of the values in the Argument Vector, a *Structure Vector* that reflects the structure of the tree and a *String Buffer* that holds the content of every reference types (strings, node names, etc.). Let us, now, discuss each of these components in more detail.

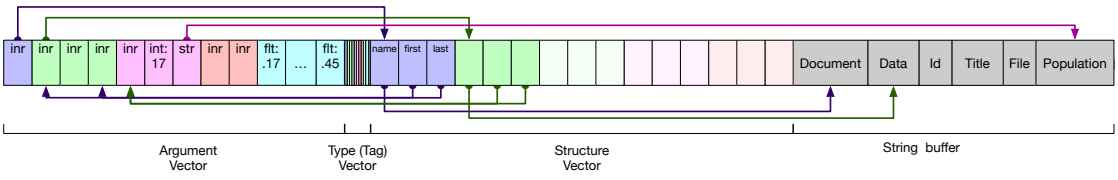

Figure 3: The Running Example Serialized using Wisent (for simplicity only some pointers are shown)

In Wisent, data is serialized breadth-first: all nodes of a given level are stored in a consecutive memory region. The color-coded levels in Figures 2 and 3 illustrate how all nodes in a level are stored in a contiguous memory region. Value nodes (ints, floats) are stored directly in the ***Argument Vector***, aligned to CPU words (we discuss sub-word alignment later in this section). Reference types (currently only strings) are represented by an offset into the String Buffer. Expressions (i.e., inner nodes) are represented as offsets in the Expression Vector.

For illustrative purposes, argument types are represented as labels in Figure 3: integers as `int`, floats as `flt` and inner nodes as `inr`. In the implementation, however, types are represented in a ***Type Vector*** that is separate from but aligned with the Argument Vector (i.e., the value at position i in the type array encodes the type of the argument at position i in the Argument Vector). To simplify alignment, we currently use entire CPU words to encode types but might reduce this to 1-byte words in the future.

The ***Structure Vector*** is an array of triples of offsets: one pointing into the string buffer indicating the label of the node, one pointing to the first child in the Argument Vector and one pointing to the last child. As all children of a node are stored in a consecutive memory region, no pointers to individual children are required.

The ***String Buffer*** is a mere sequence of (null-terminated) strings.

4.1. Opportunistic Type Run-Length-Encoding

While Wisent, as described so far, encodes leafy trees is a succinct and efficient representation, it suffers from a substantial performance hazard: it represents the type of every argument individually, i.e., without exploiting the likelihood of siblings having the same type. The array of floats in the leaf of Figure 3, e.g., would be accompanied by a Type Vector of equal size.

While the waste of space is only a minor concern outweighed by the advantages of the representation (fast type resolution and good locality of the values), the runtime cost of determining every argument type individually is a major concern. Most importantly, it prevents simple, efficient processing of the arguments (e.g., `for(int i=0; i<size; i++) sum+=input[i]`). To address this problem, we propose to opportunistically Run-Length-Encode the types as follows: if a node has five or more (adjacent) children of the same type, the type tag of the first value has its most significant bit set (a bit that is otherwise unused). Further, the type tags of the following four values are interpreted as a single 32-bit integer encoding the length of the run of identically typed arguments.

Opportunistically encoding a run of types like this does not impact the space efficiency of the format. It does, however, allow the efficient processing of the argument values without the need to determine the type of every argument individually – a major performance gain.

5. Wisent Encoding and Decoding

Like most data exchange formats, Wisent should be easy and efficient to write and, more importantly, read. We focus on two distinct scenarios for encoding and one for decoding: encoding fragmented trees (i.e., in-memory data structures made up from values and pointers), encoding from depth-first trees (i.e., converting existing JSON documents), and lazily decoding for the purpose of data analytics.

5.1. Encoding from Fragmented Trees

As in Depth-First Serialization, Breadth-First Serialization starts from the tree's root and traverses it towards the leaves. Rather than descending a single path to each of the leaves one after the other, however, Breadth-First Serialization traverses all paths simultaneously. This requires maintaining a list of references to all nodes at a level while traversing the tree. The size of this "serialization-frontier" is bounded by the breadth of the tree, i.e., the maximum number of nodes at any level (not counting leaves). While this can be large in theory, the leafy trees we focus on have many leaves but few inner nodes. This makes the size of the serialization frontier a minor concern.

For the purpose of this paper, we aimed to encode all four fragments into a single buffer. While this is not a strict requirement of the format, it simplifies memory management in a shared-memory scenario and allows serializing into a file if required. To this end, the Wisent-encoder makes two passes over the input tree: one to calculate the number of nodes and inner nodes (which determine the sizes of the Argument, Type and Expression Vectors) and one to perform the serialization itself. As Wisent is optimized for leafy trees, the first traversal is an insignificant cost factor.

While the number of nodes in the input tree, and thereby the size of the first three sections, can be determined efficiently, determining the size of the string buffer is costly: it requires determining the length of every string node (i.e., searching for its null-terminator). While finding the null terminator is inevitable when copying strings into the buffer, it is expensive and unnecessary to calculate the length twice (once for allocation and once when filling the buffer). Instead, we use the Operating System's ability to re-allocate, i.e., increase the size of a previous memory allocation.

5.2. Encoding from Depth-First Trees

Unlike fragmented trees, depth-first flattened trees provide no direct access to entire tree elements: they are intended to be parsed by scanning the entire structure from the root to the leaves. Accessing a node child other than the first requires scanning the whole structure from the current to the target position. A naïve implementation for the serialization would require either a full copy or redundant access to the tree structure. To efficiently process the tree without this overhead, serialization is implemented instead with a two-passes approach.

The first pass is not only necessary for calculating the size to allocate the buffer but also for the starting position of the nodes at each level. During this parsing, the number of nodes at each level is calculated. Next, the per-level node count of the arguments is prefix-summed to deduce the starting position at each level.

In the second pass, the arguments are again scanned in depth-first order and scattered into their appropriate position in the Argument Vector. Equally, the expressions are stored in breadth-first order in the Expression Vector.

Both passes are computationally efficient because they are sequential scans of the input tree's memory buffer. In terms of memory usage, they require a minimal state: the first pass requires constructing a list whose size is the depth of the tree; The second pass requires two more stacks of the same size (one for the argument's next position and one for the expression child's current position) and two more values (one to keep track of the current level and one for the current position in the Expression Vector).

5.3. Lazy Decoding

Wisent documents are designed to allow lazy decoding, i.e., minimize the amount of unnecessary data that needs to be decoded to access a specific node (identified by its path from the root). Navigating to a specific node starts at the root, i.e., the first argument in the Argument Vector (virtually always an inner node). Descending to one of its children requires dereferencing its value (an offset into the Structure Vector) to get its first child. Accessing a child by its position is a simple arithmetic operation (i.e., adding the position to the index of the first child). Descending one level, thus, requires only two memory accesses (plus one to verify the node type as "inner" if required). If the child is specified by name rather than position, descending one level requires a loop over the children. However, as both the children offsets, the structure triples and the strings designating their names are located in (respective) contiguous memory regions, the data accesses within that loop have optimal locality.

Once a node is found, iterating over its children (e.g., to operate on its leaves) is as simple as running an iterator from the first to the last offset. If the types are Run-Length-Encoded (see Section 4.1), there is no need to check children's types, making the processing as simple as iterating over a plain C-array in memory.

6. Evaluation

6.1. Experimental Setup

To assess the benefits of the Wisent serialization format, we evaluate the performance of various deserializer implementations for Wisent and compare them with JSON and BSON deserializers. We also compare a C++ serializer implementation with JSON and BSON serializers.

Wisent Deserializers. We implemented three deserializer backends; C++, Swift and Python. The C++ deserializer is naturally implemented with pointers access to the buffer and casting to the underlined types. Traversing the expression's arguments is implemented with standard iterators, which makes them compatible with any std functions. The Swift deserializer implementation uses UnsafePointer<Int8> and manual pointer arithmetic to extract pointers to sections of buffer, swift-native types and structs to access elements of sections and (lazily generated) Sequences to access arguments. The Python deserializer implements the argument traversal using the generator pattern. Raw Buffer data is extracted to underlined types with the help of Python modules *struct* and *ctypes*.

Wisent Server. To demonstrate inter-process communication with the Wisent format, a Wisent server application implements the serialization of JSON files and CSV files into the Wisent format and stores the data in shared memory. The client benchmark application reads the data from shared memory and performs the deserialization.

Baseline Serialization Formats. We compare the Wisent serialization format with three other methods: first, reading data directly from raw JSON and CSV files (i.e., no serialization); second, embedding both metadata (from JSON) and columnar data (from CSV) into a single JSON; third, embedding metadata and columnar data into JSON and then, serializing the JSON object into the BSON format, i.e., binary format alternative to JSON.

Baseline Deserializers. Besides the serialization formats, We integrated several baselines for the implementation of the JSON deserialization: NLohmann JSON version v3.11.2 [13], RapidJSON version v1.1.0 [14] and SimdJson version v3.2.0 [15]. In addition, we used RapidCSV version v8.75 for loading data from CSV files.

Hardware. All experiments are performed on a server with two Intel Xeon Silver 4114 2.20 GHz CPUs, each with 10 physical cores, a 14 MB LLC cache and 196 GB of memory. We use Ubuntu 22.04 with Linux kernel 4.15.0-212-generic and compile all code with Clang version 14 using the compiler flags `-O2`.

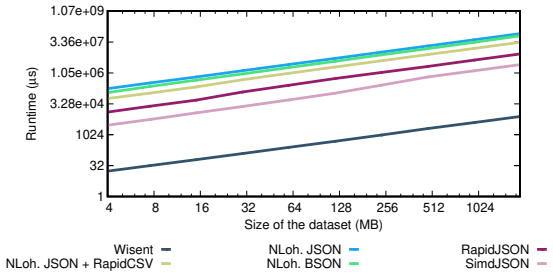

Figure 4: Runtime Performance from Deserializing and Aggregating Column Data with Various Sizes

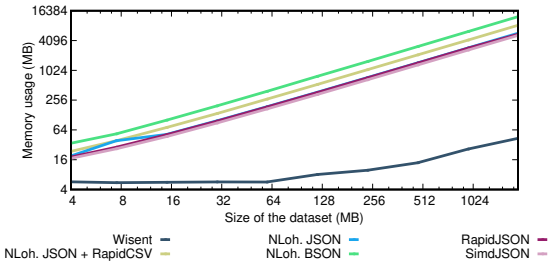

Figure 5: Memory Usage from Deserializing and Aggregating Column Data with Various Sizes

Workload. All the experiments are run using the OPDS Weather dataset [16], modified to reduce the number of columns from 256 to 100 (to assess the scalability with a significant number of rows without being limited by a large number of columns). This dataset uses the Data Package format: the main file to load is a `datapackage.json` file, which contains metadata and the path to a CSV file for loading the columnar data. We also generate a smaller dataset size (by iteratively removing every other row) and a larger one (by duplicating all the dataset rows). We, therefore, experimented with a range of data sizes from 951 KB to 8GB.

6.2. Deserialization Performance

To assess the performance of Wisent deserializer implementation, in this experiment, we vary the dataset size from scale 1/256 (i.e., 951 KB) to scale 8 (i.e., 2GB) and run an aggregation on one column of floating point values. We measure both the runtime and the memory usage.

Runtime. The result in Figure 4 shows that the Wisent deserializer outperforms the baselines with two to four orders of magnitude; the breadth-first approach in Wisent allowing contiguous data in memory and RLE optimization brings significant runtime performance benefits. SimdJson and RapidJSON have the next best performance due to ad-hoc optimizations for traversing JSON data.

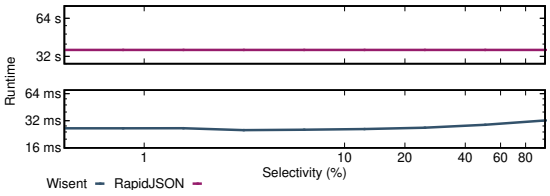

Figure 6: Runtime Performance from Deserializing, Filtering and Aggregating Column Data with Various Selectivity

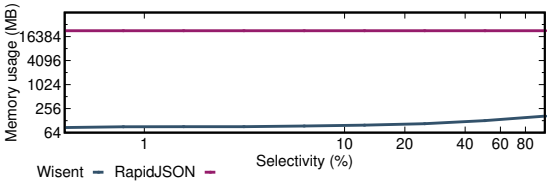

Figure 7: Memory Usage from Deserializing, Filtering and Aggregating Column Data with Various Selectivity

Memory Usage. Figure 5 shows that Wisent deserializer takes four to ten times less memory than the baselines for the smallest datasets. This gap drastically increases with larger datasets, up to a thousand times more compact for Wisent. At first glance, the result might seem surprising since Wisent is not implemented for compactness. However, Wisent support for lazy decoding (and in-place deserialization) explains why the physical memory usage is negligible even when shared memory usage is not. In contrast, other implementations require storing all data in temporary memory for reading, causing higher memory usage.

6.3. Benefits of Lazy Decoding

To assess the impact of partial data reading on the performance of the Wisent deserializer implementation, in this experiment, the aggregated column is filtered with a predicate on another column's data. We vary the selectivity from 0.4% to 100%. To avoid side effects from column data fitting the cpu cache, the dataset is fixed to the largest size at scale 32 (i.e., 8GB). We measure both the runtime and the memory usage. For this experiment, we choose as the baseline RapidJSON, which is the fastest implementation supporting the largest dataset (SimdJSON does not support such large JSON documents).

Runtime. The result in Figure 6 shows that Wisent benefits from columnar data stored contiguously in the buffer to improve data access with 20% performance benefit on the lowest selectivity compared with the maximum selectivity. Due to their different memory access pattern, other implementations do not gain from low selectivity.

	C++	Swift	Python
Line Count	153	93	135

Table 1
Lines of code for implementing Wisent deserialization

Memory Usage. The result in Figure 7 shows that memory usage with Wisent is lower for lower selectivity due to lazy decoding. RapidJSON requires all data to be copied in temporary memory; the memory usage is two orders of magnitude higher regardless of the selectivity.

6.4. Deserialization Implementation with High-Level Languages

To assess the ease of implementing a Wisent deserializer in various languages and benefit from built-in abstractions, we implemented a deserializer in Swift and Python in addition to the C++ implementation. We also measured their performance compared to the C++ implementation.

Ease of implementation. As shown in Table 1, only 93 and 135 lines of code are required to implement a deserializer, respectively in Swift and Python. This shows no obstacle in implementing a Wisent deserializer in high-level languages that are not primarily intended for low-level data manipulation code.

Reusability of built-ins language abstractions. The required helpers to implement such a deserializer are standard and built-in into the three language implementations, as described in Section 6.1. It shows that, despite the code simplicity, implementing a Wisent deserializer is straightforward to benefit from the language abstractions, making the implementation composable and replaceable.

Runtime Performance. Figure 8 shows the results of executing the aggregation query on the dataset at scale 1 (i.e., 250 MB). Swift is five times slower, and Python is 1400x slower than the C++ implementation. For Swift, this is due to miss-opportunities to compile as efficient code as C++ (e.g., vectorization) and Swift's current implementation of LazySequence, adding unnecessary virtual function calls. For Python, this is due to the runtime overhead of interpretation and dynamic typing.

6.5. Serialization Performance

To assess the performance of Wisent serialization, in this experiment, we run the serialization of the OPSD Weather dataset at scale 8 (i.e., 2GB). We compare Wisent with the serialization of CSV data into a JSON file and the serialization of the JSON file into BSON (which includes the serialization from CSV to JSON as well). We also show the overhead of loading the CSV file separately since this cost is factored into all the implementations.

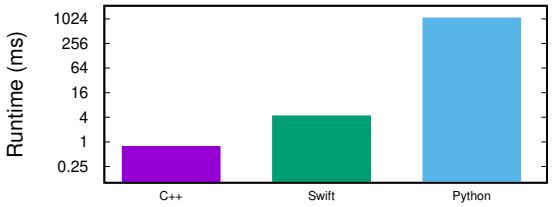

Figure 8: Runtime Performance for Wisent Deserialization with Alternative Backends

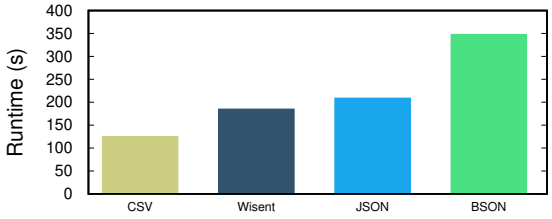

Figure 9: Serialization Runtime Performance

As shown in Figure 9, Wisent is faster to serialize than JSON and BSON. The cost for Wisent serialization is primarily due to the CSV parsing. Without including the CSV parsing cost, Wisent outperforms JSON and BSON with a factor of 1.4x and 3.7x, respectively. This result shows no prohibitive cost to serialize into Wisent format.

7. Conclusion and Future Work

We introduce a novel approach to the serialization of trees, specifically focusing on the efficient exchange of "leafy" trees through high-throughput channels like shared memory, RDMA or non-volatile memory. We propose to serialize such trees breadth-first rather than depth-first serialization, which is the de-facto standard. We demonstrate that, when combined with decomposed storage of data, structure and variable-sized datatypes, breadth-first serialization enables lazy decoding during tree navigation and in-place data processing without the need for parsing or conversion.

While we outline the key ideas of Wisent in this paper, we still consider this effort a work in progress. Moving forward, we plan to address some technical issues with Wisent, most importantly by supporting the sub-word alignment of types (which is required to, e.g., efficiently process single-precision floating point values). Further, we will study Wisent in a wider context, e.g., a distributed processing scenario. Finally, we plan to integrate Wisent into existing systems. As virtually all high-performance systems store in-memory data in C-arrays, this integration will be possible without conversion overhead.

References

[1] Frictionless Standards, Data Package, 2023. URL: https://specs.frictionlessdata.io/data-package/.

[2] Apache Arrow, 2023. URL: https://arrow.apache.org.

[3] Apache Arrow, Feather File Format, 2023. URL: https://arrow.apache.org/docs/python/feather.html.

[4] Apache Parquet, 2023. URL: https://parquet.apache.org/.

[5] Apache ORC, 2023. URL: https://orc.apache.org/specification/ORCv1/.

[6] Apache Avro, 2023. URL: https://avro.apache.org/.

[7] V. Leis, M. Haubenschild, A. Kemper, T. Neumann, LeanStore: In-Memory Data Management beyond Main Memory, in: 2018 IEEE 34th International Conference on Data Engineering (ICDE), IEEE, Paris, 2018, pp. 185–196. URL: https://ieeexplore.ieee.org/document/8509247/. doi:10.1109/ICDE.2018.00026.

[8] JSON, 2023. URL: https://www.json.org/json-en.html.

[9] MongoDB, BSON Format, 2023. URL: https://www.mongodb.com/basics/bson.

[10] Google, 2023. URL: https://protobuf.dev/overview/.

[11] M. Slee, A. Agarwal, M. Kwiatkowski, Thrift: Scalable Cross-Language Services Implementation (2004).

[12] Substrait, 2023. URL: https://substrait.io.

[13] N. Lohmann, JSON, 2023. URL: https://json.nlohmann.me.

[14] RapidJSON, 2023. URL: https://rapidjson.org.

[15] G. Langdale, D. Lemire, Parsing Gigabytes of JSON per Second, The VLDB Journal 28 (2019) 941–960. URL: http://arxiv.org/abs/1902.08318. doi:10.1007/s00778-019-00578-5. arXiv:1902.08318.

[16] S. Pfenninger, I. Staffell, Weather Data, 2019. URL: https://data.open-power-system-data.org/weather_data/2019-04-09. doi:10.25832/WEATHER_DATA/2019-04-09.