IMPERIAL

MEng Individual Project
Interim Report

Department of Computing

Imperial College of Science, Technology and Medicine

Wisent++: A C++ Library for Composability-Enabled Data File Formats

Author:
Amor Zhao (CID: 02019680)

Supervisor:
Dr. Holger Pirk

Second Marker:
Pending

January 23, 2025

Contents

Chapter 1

Introduction

1.1 Project Motivation

File formats differ greatly in modern data management systems. Various data structures, file formats and compression algorithms available for managing and exchanging data makes it a growing challenge for data systems to efficiently handle diverse data models. They face the challenge of either continuously investing resources to support new file formats or restricting users to a limited selection, which may negatively impact performance and compatibility. Additionally, file formats are tightly coupled with the data they represent, meaning that any change to the data model or compression method often necessitates modifying the file format itself.

When exchanging between flat data and nested data, such discrepancy leads to inefficiencies when converting between the two, as it requires additional steps to flatten nested data or reconstruct hierarchies from flat formats, and has become a major challenge to the interprocess communication efficiency.

With the approach of storing all data in a single file, we are able to simplify the problem of formats integration, however, it also increases processing costs. This is because parsers must treat every node as if it has children, even though many nodes may not. In this way, we change our focus to how we can efficiently store and exchange leafy trees.

The Wisent library [1] provides a new exchange file format which supports CPU-efficient encoding and decoding of these leafy trees. It guarantees efficient serialisation, deserialisation and in-place processing, and supports selective access to specific parts of data in the file, which allows manipulating various data structures without being constrained by specific file format designs.

While Wisent shows better runtime performance and lower memory usage than other popular file formats like Json[2], Bson[3], and RapidJson[4], it does not support

any data compression techniques. In comparison, other columnar data storage formats, like Parquet[5], support various data compression schemas that allow users to easily create efficient storage and retrieval.

With this motivation, we aim to design and build an advanced version of the Wisent C++ library, which adapts effective data compressing algorithms, while keeping Wisent's advantage of efficient selective loading and random access.

1.2 Design Objectives

Wisent++ is designed to extend the state-of-the-art file formats to provide a flexible and efficient solution for managing and exchanging data between diverse data models, while supporting various compression schemas in modern data systems. The library should extend the five objectives of Wisent, which, as stated in the Wisent paper, includes: Nested data representation; Minimizing storage overhead; Efficient serialisation usages; Selective access and being simple to implement parser using the popular programming languages.

In our project, we add a few more features in addition to these properties. The library should also support a variety of data compression algorithms, from the simpler algorithms like RLE, Delta encoding and LZW, to the modern (and more complicated) ones like BZip2, prediction by partial matching, etc.[6] So, we added the following objectives in this project:

- We aim to design the library with encapsulated methods such that these algorithms could be applied efficiently and easily.
- Another necessary goal is to make Wisent++ adapt to current industrial need. It has been designed to build under a lowest requirement of C++ version 14, in order to best support most projects while dismissing a reasonable amount of C++ features.
- Although it is users' freedom to choose the compression algorithm in real applications, we aim to optimise the worst case scenario during the design.

Chapter 2

Background

In this chapter, we provide an overview of some existing solutions and their compression algorithms.

We will then take a look into the basic design principles and implementations of the Wisent file format, and explore our approaches to extend the Wisent library to support some of the popular modern compression algorithms.

2.1 Relavant Data File Types

2.1.1 Flat Data

Flat data indicates the type of data without an embedded structure. Because of its simplicity and compatibility, files with such data formats are commonly used in databases and spreadsheets. `csv`, `tsv` and `txt` files [7] are some of the most common examples to store flat data.

As illustrated in Figure. 2.1, the `csv` file is organised in a tabular format where each row corresponds to a record and each column corresponds to a field.

2.1.2 Nested Data

Nested data, on the other hand, is the type of data that contains an embedded structure, some typical examples includes `JSON`, `XML`, and `YAML` files[2]. These files are commonly used to represent hierarchical data structures, such as trees, graphs, and other complex data models.

Figure. 2.2 shows an example of a nested data file in `JSON` format.

```
name,score,grade,pass,feedback,...
Alice,100.0,A,yes,excellent
Bob,93.5,A,yes,good
Charlie,80.0,B,yes,inattentive
David,70.0,C,yes,satisfactory
Eve,61.5,D,no,poor
Frank,49.0,F,no,miserable
Grace,0.0,F,no,disastrous
Hank,75.0,C,yes,poor
Ivy,35.0,F,no,tragic
Jack,99.0,A,yes,excellent
Kate,85.0,B,yes,good
Lance,65.0,D,no,dreadful
Mandy,55.0,F,no,disastrous
Nancy,88.5,B,yes,nice
Oscar,77.0,C,yes,hopeful
...
```

Figure 2.1: Flat data example (`student-exam-feedbacks.csv`)

```
{
    "filename": "student-exam-feedbacks.csv",
    "filetype": "csv",
    "fileschema": {
        "fields": [
            {
                "name": "name",
                "type": "string"
            },
            {
                "name": "score",
                "type": "float"
            },
            {
                "name": "grade",
                "type": "string"
            }, {…
            }, {…
```

Figure 2.2: Nested data example (`student-exam-feedbacks.json`)

2.1.3 Leafy Data

In many real-world scenarios, data is stored as a combination of both flat and nested data, where a nested data file is used to represent the structure of the dataset, such as column names, data types, and other information about the flat data file, and the flat data file contains the actual tabular data. We call this type of data structure a leafy tree, where each node is either a leaf or an inner node.

In our example of Figure. 2.1 and 2.2, the flat data file contains the actual exam feedback data, while the nested data file contains the metadata about the flat data file structures. It may also be forming a bigger tree structure, where many flat data files are referenced by various nested data files like `student-exam-feedbacks.json`.

2.2 Existing Solutions

Many existing solutions for managing leafy data files are based on traditional data decomposition schemas, which store the data in a depth-first manner. We will look at some of the popular file formats here, dealing with either flat or nested data.

2.2.1 JSON

JSON is a common text-based format for nested data serialisation, easily integrated into any language but with high serialisation costs. Several libraries, including NLohmann JSON [8], RapidJSON [4], and Simd-Json[9], provide efficient JSON parsing and serialisation. However, JSON's text-based nature also makes it less efficient for large datasets, as it requires more storage space and processing power.

2.2.2 Parquet

The Apache Paquet file format [5] represents an advanced approach to data storage and transfer, with its compression properties playing a critical role in its overall efficiency. It employs a sophisticated compression algorithm that dynamically adapts to the characteristics of the data being compressed. This adaptive compression scheme is a cornerstone of the Paquet format's efficiency, as it minimizes both the space required for storage and the time needed for decompression.

Paquet allows users to choose from multiple compression algorithms, including: Snappy, Gzip, Brotli, LZO, and Zstandard (Zstd). The choice of compression algorithm can significantly impact the performance of the Paquet format, as Paquet applies different trade-offs between compression ratio and speed.

2.3 Basic Wisent File Format

We now move on to the basic design principles and implementations of the Wisent file format.

As discussed in the last section, we found that one of the major drawbacks of the traditional data decomposition schema is that it requires reading the entire data structure before accessing a specific part of data, which is particularly inefficient when only a small portion of the data is needed. As long as data is stored in a depth-first manner, the structure of leafy file is unpredictable.

Wisent, on the other hand, stores the data in a breadth-first manner, which allows efficient selective loading and random access, as we don't need to read the entire file to get the data structure.

2.3.1 Design Principles

Figure. 2.3 illustrates the Wisent structure for storing a tree node. The Wisent serialisation format is divided into four distinct sections: the Argument Vector, the Type

Bytefield, the Structure Vector, and the String Buffer. These components work together to store and represent the serialized data efficiently.

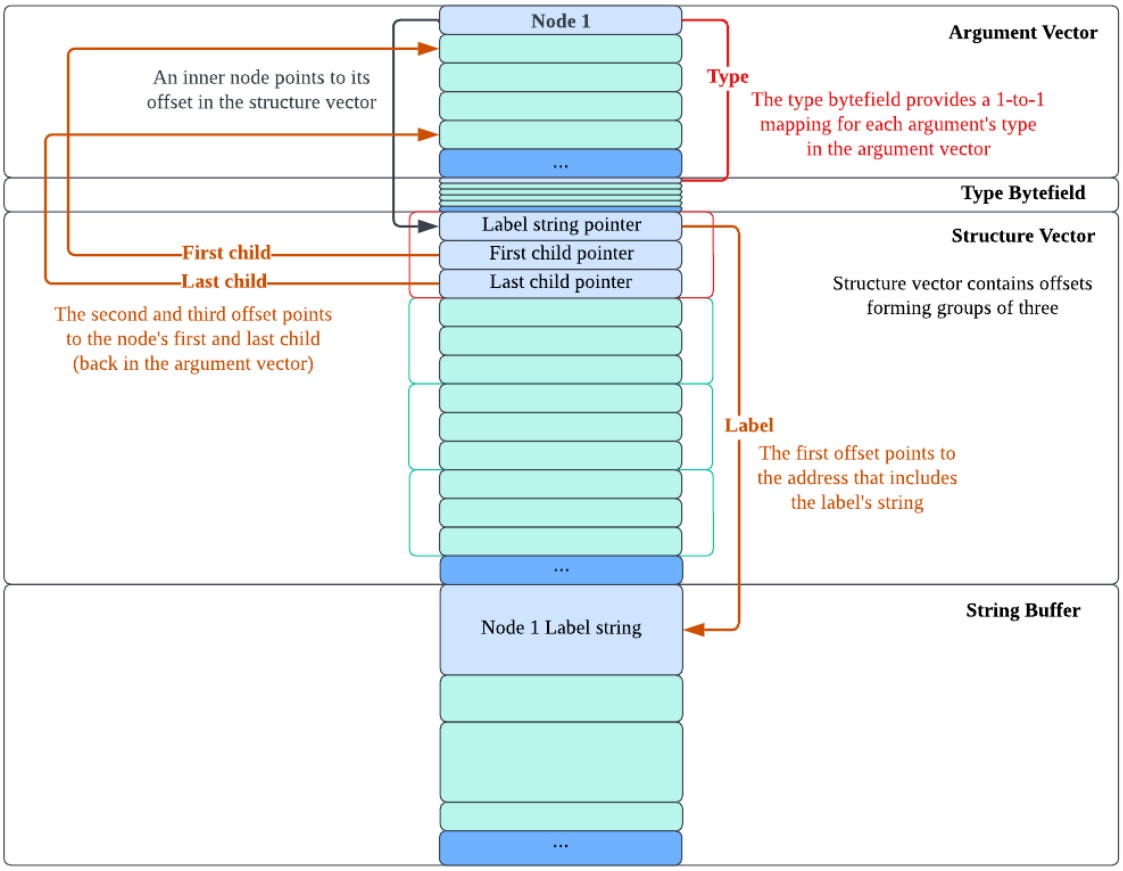

Figure 2.3: Wisent structure for storing a tree node

We take the light blue blocks as an example:

"Node 1" represents a node argument in the tree. The Type Bytefield at this index contains a 8-bit Argument Type value "5", indicating this argument in the Argument Vector to be an expression. The 3 corresponding offsets in the Structure Vector contains the information of its structual relationships. The first pointer points to a string in the String Buffer, while the second and third pointers point to the beginning and end of the following child nodes of "Node 1".

Together, they representes all the information of the node in the tree.

For non-node arguments, the argument value is directly stored in the Argument Vector, for example, numbers as `int` or `float` types, and strings as a direct offset pointing to the String Buffer section.

2.3.2 Serialisation and Deserialisation Mechanism

The Wisent library provides a simple and efficient mechanism for serializing and deserializing data. As shown in Figure2.4 and 2.5, the serialisation process converting the leafy data into a Wisent file, while the deserialisation process involves converting the Wisent file back into the original leafy data.

Mechanism: Wisent Serialiser

```
Input: Leafy source data
Output: Serialised Wisent file

1. Iterate through the
   datapackage to count
   number of arguments
2. Allocate memory for
      Argument Vector,
      Type Bytefield,
      Structure Vector,
      and String Buffer
3. For each node in the
   datapackage,
   serialise the node
```

Figure 2.4: Wisent Serialiser

Mechanism: Wisent Deserialiser

```
Input: Wisent file
Output: Deserialised data /
        selected columnar data

1. Load the Wisent data file
2. Split the file into
      Argument Vector,
      Type Bytefield,
      Structure Vector,
      and String Buffer
3. For each node in the
      Structure Vector,
      reconstruct the tree
   Or lazy loads the
      selected columnar data
```

Figure 2.5: Wisent Deserialiser

We can test the performance of basic Wisent library through a series of benchmarks. The results show that Wisent outperforms other popular serialisation formats, such as JSON, BSON, and RapidJSON, in terms of both runtime performance and memory usage. The benchmarks demonstrate that Wisent is capable of serializing and deserializing data structures efficiently, making it a suitable choice for applications that require high-performance data processing.

2.3.3 Approaches of Extending Wisent for Compression

Since the Wisent library does not currently support data compression, in this project, we aim to extend the Wisent library to support various data compression algorithms.

Algorithm: Delta Encoding

```
Input: data[]
Output: encoded[]

1. Initialize encoded[]
   with the same size as data[]
2. Set encoded[0] = data[0]
3. For i from 1 to length of
   data[] - 1 do
4.    encoded[i] =
         data[i] - data[i - 1]
5. End For
6. Return encoded[]
```

Figure 2.6: Delta Encoding

Algorithm: Run-Length Encoding

```
Input: data[]
Output: encoded[]

1. Initialize encoded[]
   with an empty list
2. Set count = 1
3. For i from 1 to length of
   data[] - 1 do
4.    Count for the same
      element, increment count
5.    encoded[] is appended
      with (count, data[i - 1])
6. End For
7. Return encoded[]
```

Figure 2.7: Run-Length Encoding

Figure. 2.6 and 2.7 presents the pseudocode for two simple and popular compression algorithms: Delta Encoding and Run-Length Encoding[10]. These algorithms are easy to implement and can provide significant compression ratios for certain types of data.

The Wisent library implements run-length encoding as a built-in feature for its Type Bytefield, which can be enabled by setting a threshold in the Wisent helper library. It was designed in a runtime performance perspective, in order to reduce the time of referring to the argument type, rather than to save the storage space, though.

Some other modern compression algorithms, including BZip2, LZW, and Prediction by Partial Matching [6], are more complex and require additional considerations for integration with the Wisent data processing mechanism. We may apply them in separate approaches in the string buffer and argument vectors.

Chapter 3

Project Plan

3.1 Timeline

The project plan (as shown in Table 3.1 and 3.2) outlines the timeline and key milestones. The "planned work" column shows the detailed tasks we focus on during each time period, and the "key milestones" column states the deliverables or objectives to be achieved by the end of that period.

Timeline	Week	Planned Work	Key Milestones
Beginning of Nov.	1	Meet supervisor.	N/A
11th Nov. - 8th Dec.	2 - 5	Develop understanding of the basic Wisent design. Study the Wisent test code. Reproduce test results of the benchmarks.	Set up a test codebase with basic Wisent features by the beginning of Dec.
9th Dec. - 15th Dec.	6	Autumn term final exams	
16th Dec. - 19th Jan.	7 - 11	Study the state-of-the-art modern data compression algorithms and applications. Explore possible ways of compression implementation.	Complete interim report by 23rd Jan.
20th Jan. - 9th Feb.	12 - 13	Review the project plan and evaluation plan. Make necessary changes.	Review project with second marker by 6th Feb.

Table 3.1: Project plan

Timeline	Week	Planned Work	Key Milestones
10th Feb. - 23rd Mar.	14 - 19	Develop and integrate advanced compression algorithms. Continuously perform unit testing and performance evaluations throughout the development process.	The expected performance metrics should be well-defined by this time.
24th Mar. - 13th Apr.	20 - 22	Test performance on various datasets, analyse the approaches.	Our initial evaluation results should meet the expectations.
14th Apr. - 27th Apr.	23 - 24	Identify the performance bottle-neck, explore possible walk throughs for the performance restrictions.	We aim to finish the algorithm implementation by the start of summer term.
28th Apr. - 11th May.	25 - 26	Summer term final exams	
12th May. - 31st May.	27 - 29	Finish evaluation and write final report. Wrap up project and clean documentations.	All necessary design objectives should be met by the start of Jun.
1st Jun. - 13th Jun.	30 - 31	Complete final report.	
14th Jun. - 24th Jun.	32	Prepare presentation materials.	

Table 3.2: Project plan (continued)

3.2 Progress to Date

At the time of writing, the project has progressed according to the initial plan. The test environment has been successfully set up, the basic Wisent serialisation and deserialisation features have been implemented and the performance test results have been reproduced. The interim report has been completed, and the evaluation plan has been outlined. The implementation and tests so far have been kept in a github repository[1] for version control and source code management.

[1]Github repository access available upon request at `https://github.com/AmorZhao/WisentCpp/`. Accessed: 2025-01-23, permission required.

The next steps involve carrying on extending the library to support the compression algorithms as we discussed, followed by regular testing and evaluation. In the upcoming few days, we will also be reviewing the project and evaluation plan with the second marker, necessary changes should be made based on the feedback.

3.3 Fallback Positions

For the overall completion of the project, the following fallback plans can be taken:

- If implementation of complex compression algorithms proves too challenging, we would focus on simpler algorithms like RLE and Delta encoding while maintaining the core Wisent functionality.
- While certain compression algorithms may be incompatible with the properties of the Wisent format, we may find alternatives for such algorithms and focus on optimal overall performance.
- If meeting all planned compression algorithm implementations becomes unfeasible, we would focus on implementing and thoroughly testing a smaller subset of algorithms.

3.4 Possible Extensions

If time and resources permit, the following extensions could be explored:

- Implement parallel compression and decompression to improve performance on multi-core systems.
- Create bindings and parsing support as built in functions.
- Apply partial support for C++17 features, such as adapting constant expressions and using C++ standard optional library instead of the experimental or boost libraries[11].

Chapter 4

Evaluation Plan

4.1 Benchmarks

To evaluate the performance of Wisent++, we will use a set of benchmarks that measure various aspects of the library, including compression ratio, compression and decompression speed, and memory usage. The benchmarks will be conducted using a variety of datasets to ensure comprehensive evaluation. We will compare the performance of Wisent++ against baseline libraries such as JSON, BSON and Parquet to establish a performance baseline.

4.1.1 Datasets

The outcome of a compression algorithm is dependent on the characteristic of test dataset. Thus a wide variety of test data should be used to evaluate the performance of Wisent++ compression, including both best and worst case scenarios. We will use the following datasets for benchmarking:

- **Synthetic Data**: Generated data with known properties to test specific scenarios.
- **Real-World Data**: Datasets from various domains such as JSON files, CSV files, and log files.
- **Large-Scale Data**: Datasets with large volumes to test the scalability of the library.

At the moment of writing this report, we used the Open Power System Data packages [12] for the real-world data, and the example synthetic data was generated by a Python script.

4.1.2 Metrics

The following metrics will be used to evaluate the performance of Wisent++:

- **Compression Ratio** [13]: The ratio of the size of the compressed data to the size of the original data.
- **Compression Speed**: The time taken to compress the data.
- **Decompression Speed**: The time taken to decompress the data.
- **Memory Usage**: The amount of memory used during compression and decompression.

4.1.3 Benchmarking Procedure

The benchmarking procedure will involve the following steps:

1. **Data Preparation**: Prepare the datasets by ensuring they are in the correct format and size.
2. **Compression**: Compress the datasets using Wisent++ and measure the compression ratio, compression speed, and memory usage.
3. **Decompression**: Decompress the datasets and measure the decompression speed and memory usage.
4. **Comparison**: Compare the results with other popular compression libraries such as zlib, LZ4, and Brotli.

4.1.4 Tools and Environment

The evaluation will be conducted with the following specifications:

Operating System: Ubuntu 22.04 with a virtualised linux environment version 5.15.167.4-microsoft-standard-WSL2.

Compiler: Clang 14.0.0 would be used for the evaluation.

Hardware: The experiments mentioned in this report were conducted on a 4-core Intel Core i5-8265U 1.6 GHz CPU with 6 MB cache and 32 GB of RAM. For evaluation of larger datasets, we would use a similar server with extended memory and storage capacity.

4.1.5 Expected Outcomes

The benchmarks are expected to demonstrate the following:

- Wisent++ should achieve competitive compression ratios compared to other libraries.
- Wisent++ should provide fast compression and decompression speeds.
- Wisent++ has efficient memory usage during compression and decompression.

The results of the benchmarks will be used to identify areas for improvement and to validate the design objectives of Wisent++.

4.2 Other Qualitative Measurements

As a C++ library, Wisent++ should also be evaluated on its readability, maintainability and usability.

The development of the library should keep a consistent coding style [14], provide comprehensive documentation, and allow easy integration into existing projects. The library should also be designed to be extensible and robust, with clear error handling mechanisms. These qualitative aspects could be assessed through code reviews, usability testing, and potentially user feedback.

Chapter 5

Ethical Considerations

This project does not involve human participants or sensitive data. The Wisent++ library is designed to be free of any data collection.

While the developers of the library cannot control the possible applications of it, it should be clearly restricted for responsible and beneficial uses only.

Bibliography

1. Mohr-Daurat H and Pirk H. Wisent: An in-memory serialization format for leafy trees. CEUR Workshop Proceedings 2023. Available from: https://ceur-ws.org/Vol-3462/CDMS9.pdf
2. JSON. Accessed: 2025-01-23. Available from: https://www.json.org/json-en.html
3. MongoDB I. BSON Specification. Accessed: 2025-01-23. Available from: http://bsonspec.org/
4. RapidJSON: A fast JSON parser/generator for C++ with SAX/DOM style API. Accessed: 2025-01-23. Available from: https://rapidjson.org/
5. Apache Parquet. Accessed: 2023-01-23. Available from: https://parquet.apache.org/
6. Gupta A, Bansal A, and Khanduja V. Modern lossless compression techniques: Review, comparison and analysis. IEEE, 2017 Mar. Available from: https://ieeexplore.ieee.org/document/8117850
7. Comma Separated Value (CSV) File Format. Accessed: 2023-01-23. Available from: https://www.creativyst.com/Doc/Articles/CSV/CSV01.shtml
8. Lohmann N. JSON for Modern C++. Accessed: 2025-01-23. Available from: https://json.nlohmann.me/
9. Langdale G and Lemire D. Parsing gigabytes of JSON per second. The VLDB Journal 2019; 28:941–60. Available from: http://arxiv.org/abs/1902.08318
10. Gopinath A and Ravisankar M. Comparison of lossless data compression techniques. IEEE Xplore 2020. Available from: https://ieeexplore.ieee.org/abstract/document/9112516
11. cppreference.com. std::optional - C++ Reference. Accessed: 2025-01-23. Available from: https://en.cppreference.com/w/cpp/utility/optional#
12. Open Power System Data. Accessed: 2023-01-23. Available from: https://open-power-system-data.org/
13. Wikipedia. Data compression ratio. Accessed: 2023-01-23. Available from: https://en.wikipedia.org/wiki/Data_compression_ratio

14. Google. Google C++ Style Guide. Accessed: 2025-01-23. Available from: https://google.github.io/styleguide/cppguide.html