Software Engineering Project 2025

Data Compression for Integer Arrays

Author Name: Thomas Lang
Student ID: 22500855
Project Due Date: 2 Nov 2025

October 29, 2025

Abstract

This report covers the design and implementation of methods for integer array compression. We explore the Bit Packing technique in two forms: one that allows compressed integers to span across multiple output integers and one that strictly confines them (non-splitting). We also implement an advanced Adaptive Compression method using Overflow Areas to efficiently manage data outliers. The main goal is to benchmark these implementations (compression, decompression, and direct access).

Contents

1 Introduction

This project, submitted for the **Software Engineering Project 2025** course, addresses a key issue in network efficiency: the speed of integer array transmission. We implement a complete, non-lossy data compression and decompression framework using a technique called Bit Packing.

1.1 Core Concept: Bit Packing and Direct Access

Bit Packing works by determining the minimum number of bits (k) required to represent all integers in an array. It then packs multiple k-bit integers into a standard 32-bit integer word. This process significantly reduces the data size, leading to a compression factor of $32/k$. Since all original bits are preserved, the method is non-lossy.

A central requirement is to maintain direct access to the i-th element of the array. The final implementation must allow retrieval of any single packed value without needing to decompress the entire array first.

1.2 Project Components

The project is structured around the following mandatory components:

1. **Dual Compression Modes:** We implement two distinct Bit Packing versions:
   - **Spanning Mode:** Allows compressed integers to span across two consecutive output integers, maximizing space efficiency.
   - **Aligned Mode (Non-Spanning):** Enforces strict alignment, where compressed integers never cross 32-bit boundaries, simplifying random access.
2. **Adaptive Compression with Overflow Areas:** We develop a specialized scheme using an overflow area to efficiently handle outlier values (numbers requiring much more than the average number of bits k'). This maximizes overall compression efficiency when data distribution is skewed.
3. **Performance Benchmarking:** We establish a precise protocol to measure the time taken by the `compress`, `decompress`, and `get` functions to calculate the crucial network **latency threshold (t)**.

This report details the design choices, implementation protocols, and experimental results for this compression solution.

2 Design and Architecture

2.1 Directory and File Architecture

The project code follows a standard structure that separates source code, documentation, benchmarks, and testing results. The Maven directory structure is detailed below:

The main components are structured as follows:

- `src/main/java/compressor/models/`: Contains the core **abstract contract** (`BitPacker.java`) and the **Factory** mechanism (`BitPackerFactory.java`), centralizing architectural definitions.
- `src/main/java/compressor/services/`: Contains the **concrete compression implementations** (`NonSpanningBP`, `SpanningBP`, `OverflowBP`) and the application's entry point (`Main.java`).
- `src/main/java/compressor/logger/`: Dedicated package related to structured logging (`Logger`, `LogLevel`).

```
SoftwareEngineeringProject2025/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── compressor/
│   │   │       ├── logger/
│   │   │       │   ├── Logger.java
│   │   │       │   ├── LoggerFactory.java
│   │   │       │   └── LogLevel.java
│   │   │       ├── models/
│   │   │       │   ├── BitPacker.java
│   │   │       │   └── BitPackerFactory.java
│   │   │       ├── services/
│   │   │       │   ├── Spanning.java
│   │   │       │   ├── NonSpanningBP.java
│   │   │       │   └── OverflowBP.java
│   │   │       ├── timetaking/
│   │   │       │   ├── PerformanceData.java
│   │   │       │   ├── PerformanceTimer.java
│   │   │       │   └── Measurement.java
│   │   │       └── main.java
│   │   └── resources/
│   │       └── performance_data.jsonl
│   ├── test/
│   │   └── java/
│   │       └── compressor/
│   │           └── services/
│   │               ├── BitPackerTest.java
│   │               └── TestDataGenerator.java
│   └── analysis/
│       └── venv/
│           └── analyze_data.py
└── pom.xml
```

Figure 1: File architecture

- `src/main/java/compressor/timetaking/`: Contains all components for performance measurement, including `PerformanceTimer` and data entities (`Measurement`, `PerformanceData`).
- `src/test/`: Includes the unit test suite (`BitPackerTest`) and utilities for generating robust randomized test data (`TestDataGenerator`).
- `src/analysis/venv/`: Contains the Python environment and script (`analyze_data.py`) used for post-processing and visualization of the benchmark results.
- `pom.xml`: The Maven project configuration file, managing dependencies and the build lifecycle.

2.1.1 Architecture justification

The project architecture strictly adheres to the standard Maven directory layout [3], which ensures the project is easily buildable, testable, and maintainable. The modular separation is based on robust software principles:

Separation of Concerns (SoC) [4] The logic is divided into distinct packages to maximize maintainability and reduce coupling:

- The `models/` package holds abstract definitions and **contracts** (like the `BitPacker` interface and Factory).
- The `services/` package holds the **concrete, executable algorithms** (`NonSpanningBP`, etc.). This separation makes the algorithms easier to test and modify independently.

Adherence to Design Patterns Key design patterns enforce modularity and flexibility:

- The **Factory Pattern** (via `BitPackerFactory.java`) centralizes the logic for instantiating different compression strategies.
- The **Singleton Pattern** (applied to `PerformanceTimer`) ensures the global, shared benchmarking resource is managed efficiently and consistently.

Testability and Reproducibility

- A dedicated `src/test/` directory ensures clean separation of unit tests.
- The use of a separate `resources/` directory for `performance_data.jsonl` allows for **reproducible benchmarking**. The raw performance data is decoupled from the application logic for external analysis (via the Python script).

3 Design Patterns

We utilized two key design patterns to manage complexity and maximize flexibility in the project.

3.1 The Singleton Pattern: Managing Shared Resources

We chose the **Singleton Pattern** for the `PerformanceTimer` mechanism.

Why Singleton?

This pattern ensures a class has only one instance and provides a global access point to it [5]. This is crucial for our global timer utility because:

- Consistency: We must ensure that all timing data is written to a single, consistent output file (`performance_data.jsonl`). Multiple independent timer instances would corrupt the results.
- Efficiency: It prevents unnecessary resource allocation (like file handlers) by centralizing the management of the shared benchmarking file.

3.2 The Factory Method Pattern: Decoupling Creation

We employed the **Factory Method Pattern** to decouple the creation of objects from the main application code [6]. This pattern was used for both the Logger and the Bitpacker classes.

Justification

The pattern defines an interface for creating an object, but delegates the actual instantiation to a special *factory method*. This gives us two core benefits:

- **Decoupling (Flexibility):** The application code only interacts with the abstract interface (`BitPacker`), not the concrete class (`SpanningBP`). This means we can change, add, or switch compression algorithms without altering the code that uses them.
- **Open/Closed Principle:** We can introduce a new compression strategy simply by creating a new concrete class and modifying the Factory. No existing code needs to be changed.

4 Implementation: The Bit Packing Mechanisms

The **Bit Packer** is the core component that improves data efficiency by using bitwise operations to store multiple small integer values inside a single 32-bit word.

4.1 The `BitPacker` Interface and Factory

The `BitPacker` Interface The `BitPacker` interface (located in `compressor.models`) defines the strict contract for all compression strategies, adhering to the Factory Pattern (Section 3.2). This contract makes it easy to substitute algorithms. The interface defines the fundamental methods:

- `compress`, `decompress`: For bulk data transformation.
- `get`: Crucial for access to the i-th element without full decompression.

The interface also uses default methods (like `extractBits` and `insert_bits_in_result`) to share common, bit manipulation utilities across all implementations, preventing code duplication.

The `BitPackerFactory` centralizes object creation. Its static `createBitPacker` method selects and configures the correct concrete implementation (`SpanningBP`, `NonSpanningBP`, or `OverflowBP`) based on a simple input string. This maintains the decoupling benefit of the Factory Pattern.

4.2 Non-Spanning Bit Packing (`NonSpanningBP`)

This implementation is designed for simplicity and fast $O(1)$ random access. Its core principle is the Non-Spanning Rule: a compressed integer is strictly confined to a single 32-bit output word and cannot cross a word boundary.

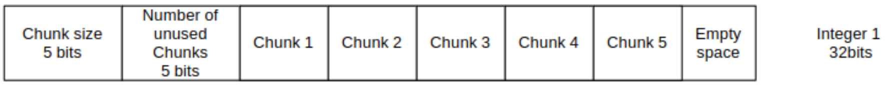

Figure 2: Non Spanning Strategy: Example of the architecture of a compressed Integer Array

Compression Logic

- Metadata: The first 10 bits of the output array store metadata: 5 bits for the required `chunk_size` (max bits per value) and 5 bits for the `unused_chunks` count. This makes the compressed array self-describing.
- Packing: The process packs values sequentially. Whenever a value would span the boundary of the current 32-bit word, the remaining bits of the current word are left empty, and the next value starts at the beginning of the next word.

The `get` Method (Random Access) This is the primary advantage of the Non-Spanning mode. The Non-Spanning Rule guarantees that the start of the i-th element is always easy to calculate. This calculation allows for constant-time $O(1)$ access, making it ideal for systems requiring fast, indexed lookups.

4.3 Analysis of Non-Spanning Performance

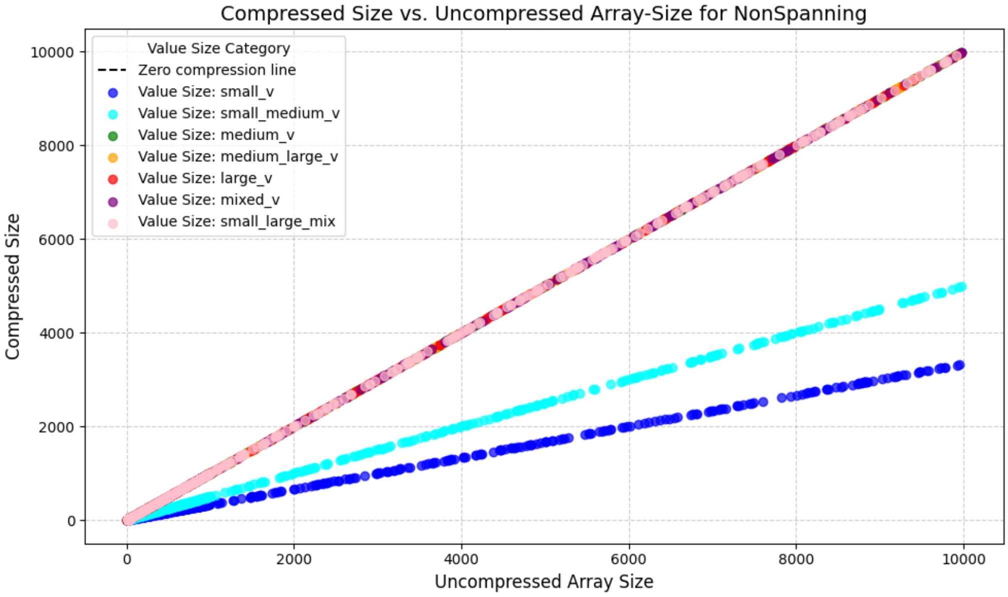

Figure 3: Compressed array size vs uncompressed array size

The compression efficiency `Analyse`: We can clearly state that the non spanning compression is only efficent for arrays that contains small values. We have a really good compression for small values (<10bits). At the Array size 10000 for example the compressed array is around 1/3 smaller than the original array. The result is still good for values up to 14 bits. But over this, the compression rate is 0 and superposes the zero compression line equally for the mixed variables arrays.

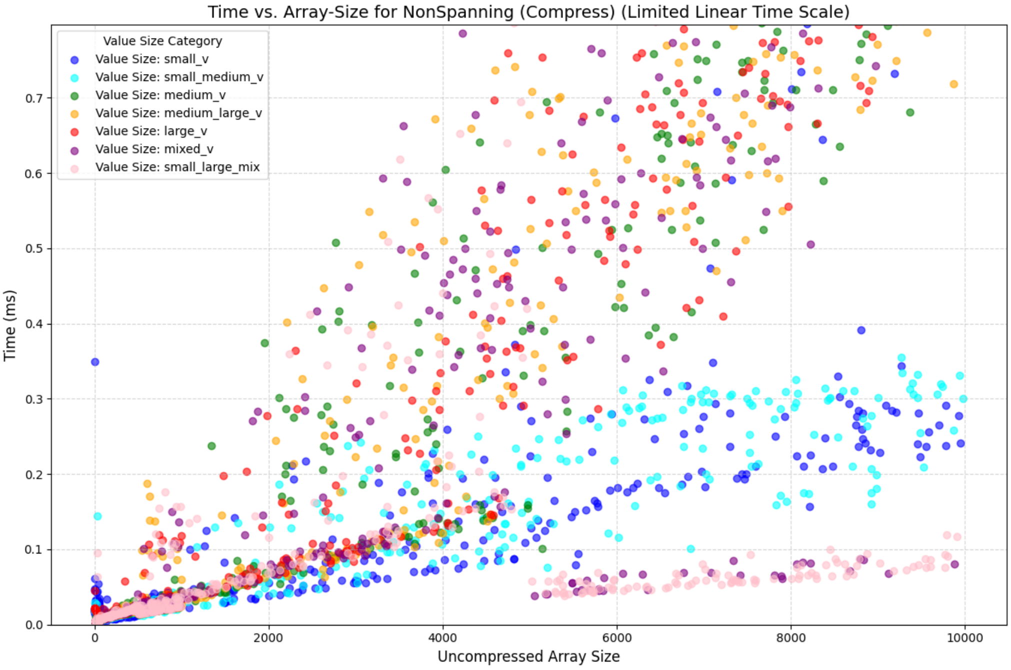

Figure 4: Non Spanning Compress method: time vs array size

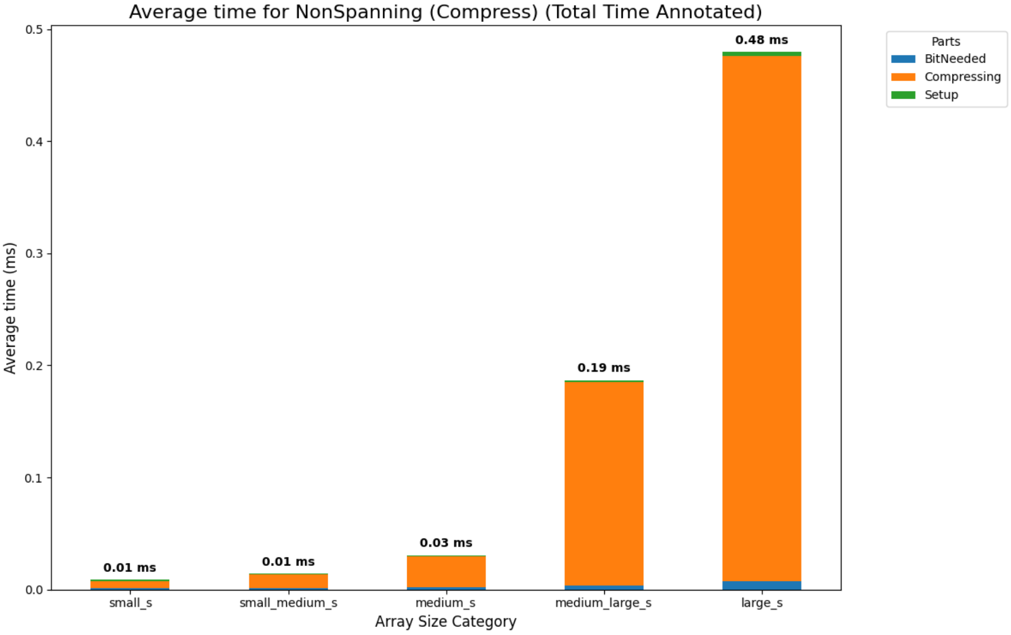

Figure 5: Non Spanning Compress method: avg time vs array size

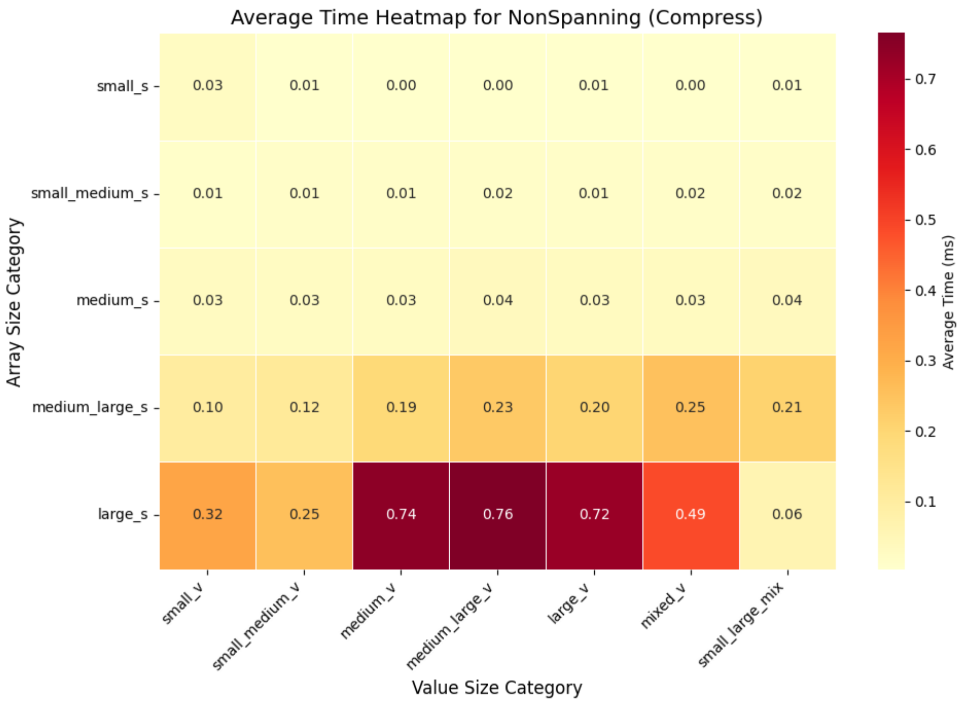

Figure 6: Non Spanning Compress method: heatmap

The compress method `Analyse`: At Figure 4 I observed a strictly increasing runtime across all tested array sizes, confirming that computational time is proportional to data volume. However for small valued arrays this increase is less steep than for the rest

At Figure 5 I grouped the array sizes and observed the time the different parts take in average to build the average runtime. Interesting is that what we already could observe in the precious figure that the time is increasing other than it may seem it isn't exponential because the possible space for small vsize is a lot less smaller than for large ones

At Figure 6 We can clearly see that the combination of large sized arrays with small values take up to 6-times longer than the other combinations.

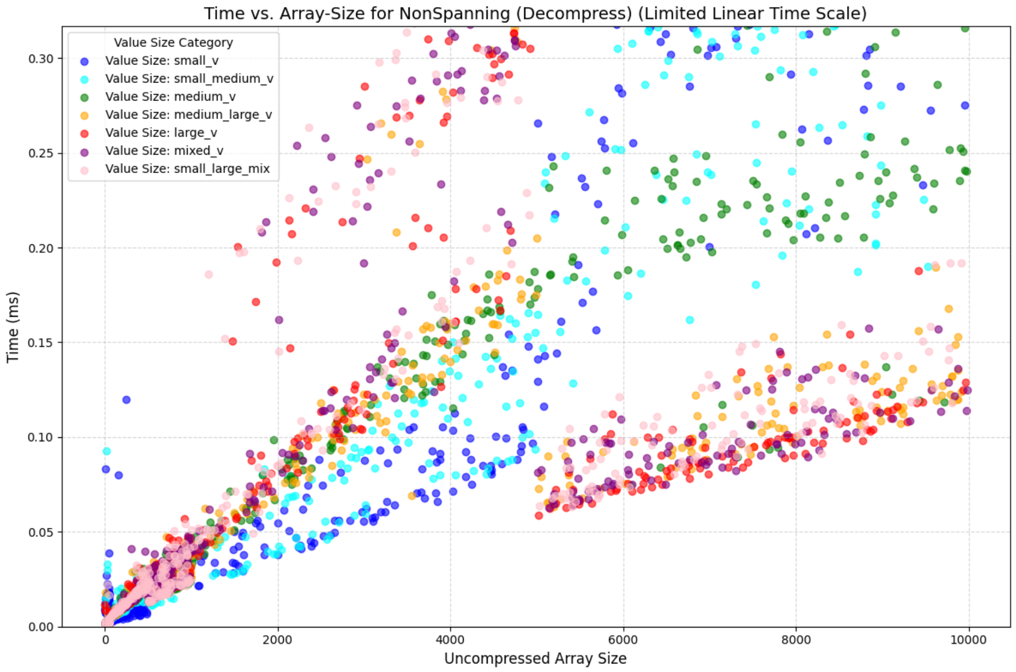

Figure 7: Non Spanning Decompress method: time vs array size

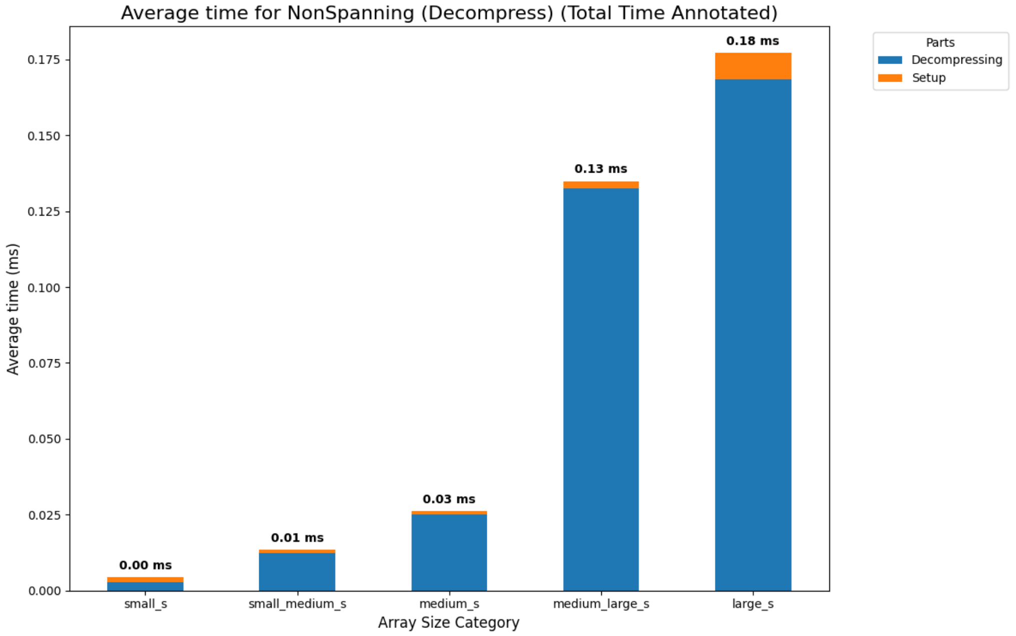

Figure 8: Non Spanning Decompress method: avg time vs array size

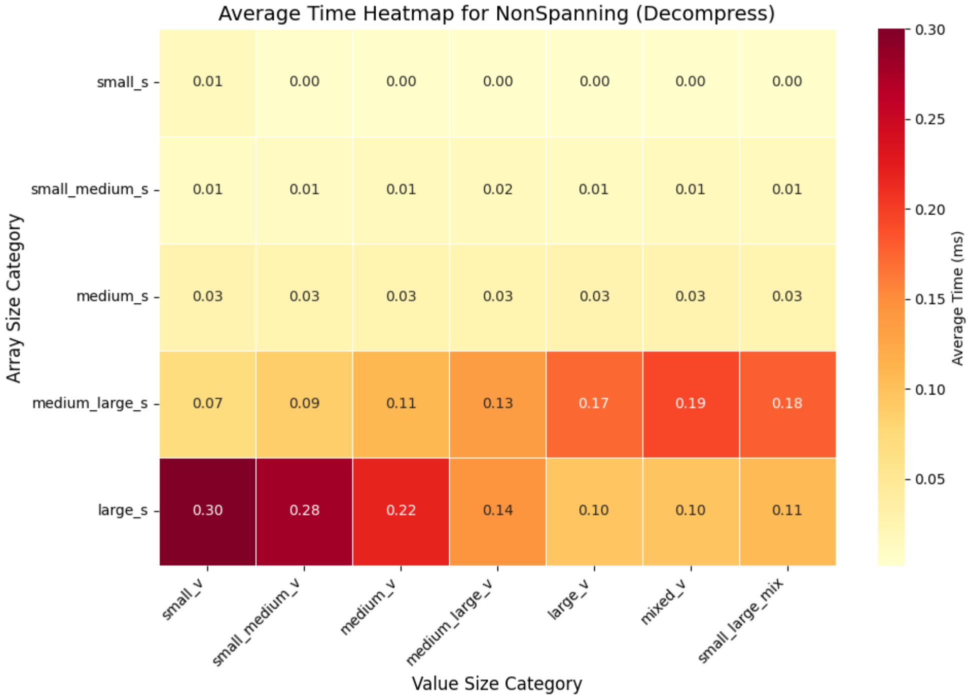

Figure 9: Non Spanning Decompress method: heatmap

The decompression method `Analyse`: At Figure 7 we see a similar evolution as in the compress method. Only we see it is generally faster und that we have for big arrays a large time needed before 5000 array sizes and it inverses afterwards. We can observe this change often at the 5000 mark. The Figure 8 shows us that we have an growth with the array size what is confirmed by the anterior figure. Interesting is that the setup for small arrays takes more than half of it's full time. The heatmap in Figure 9 jsut confirms our observations in the previous figures.

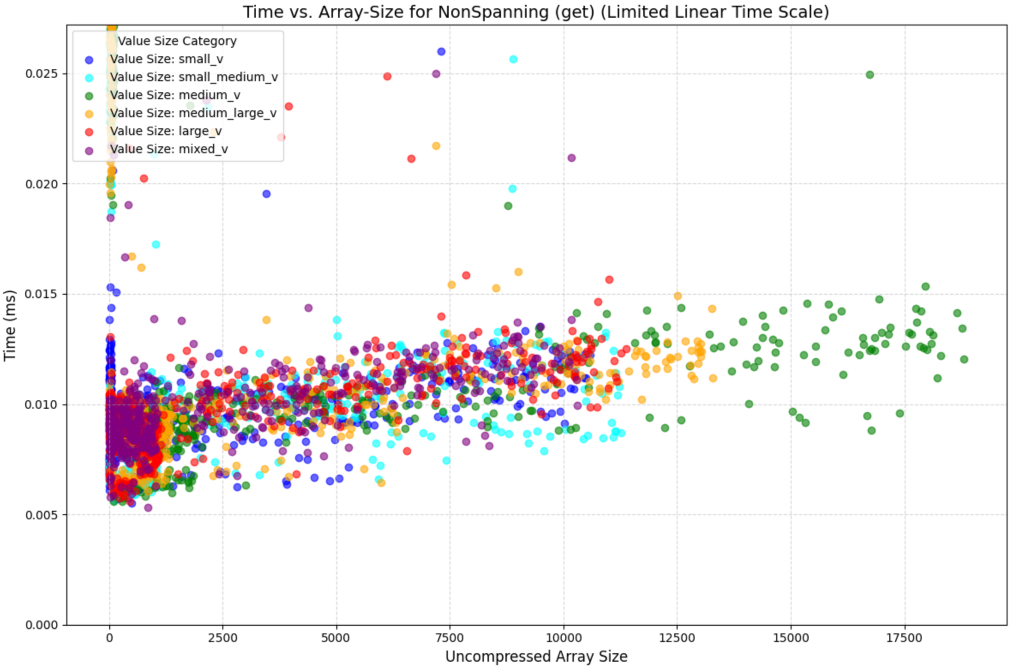

Figure 10: Non Spanning Get method: time vs array size

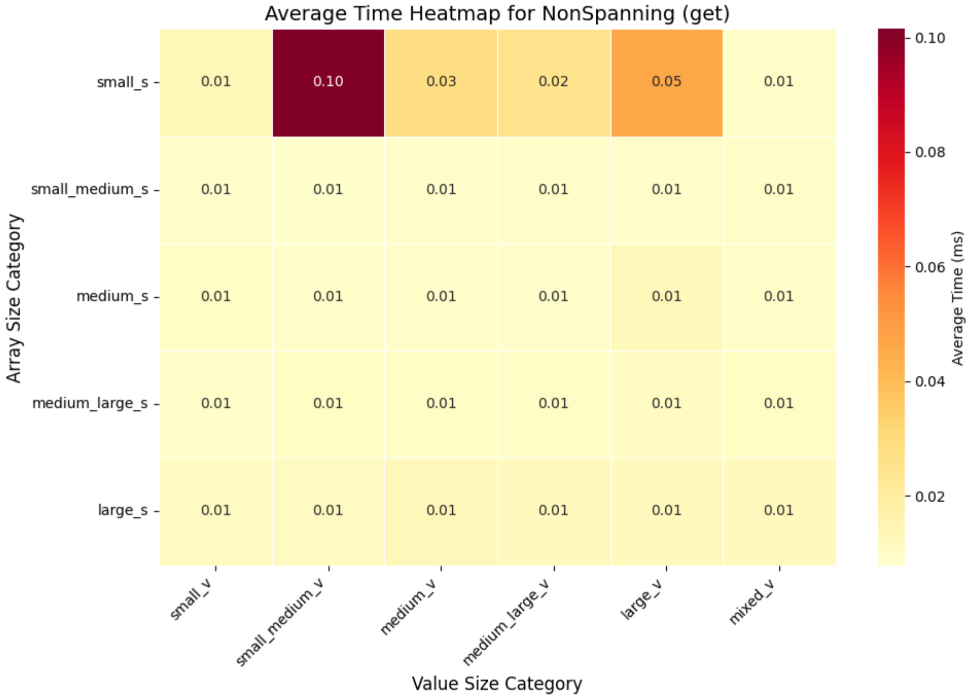

Figure 11: Non Spanning Get method: heatmap

The get method `Analyse`: For the get method it is really easy to analyze we can see clearly in Figure 10 that the time taken is independent of the array size because the average time taken doesn't change significantly this is confirmed by figure 11 where almost every get call is at 0.01ms except for the small sized arrays where it is slightly more.

4.4 Spanning Bit Packing (`SpanningBP`)

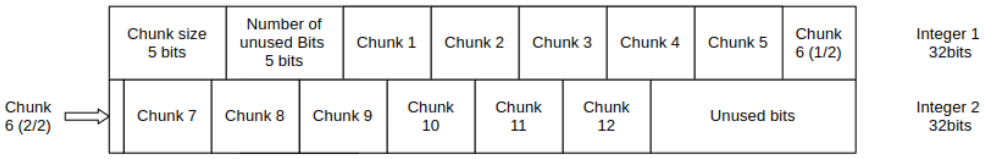

Figure 12: Spanning strategy: Example of the architecture of a compressed Integer Array

This implementation focuses purely on maximizing space efficiency. It uses the principle of a continuous bit stream, allowing compressed values to span across the boundary between two adjacent 32-bit integers.

Core Principle: Continuous Bit Stream for Maximum Density

- **Maximum Efficiency:** By eliminating all wasted bits caused by word alignment, the compressed array achieves the theoretical minimum size for the given data.
- **High Complexity:** This spatial efficiency comes at the cost of a highly complex implementation, particularly in the bit manipulation logic. Any operation (compress, decompress, or get) must anticipate and correctly handle the conditional scenario where a value is split across two adjacent 32-bit words.

Compression Logic The compression logic for `SpanningBP` is centered on updating the combined array index (`result_cursor`) and the bit offset (`bit_cursor`) after processing each value.

- Metadata and Spanning Logic: The `chunk_size` (5 bits) and the `nbr_unused_bit` (5 bits) are written in the first 10 Bits of the Output-Arrays. The compression loop then checks for boundary crossings for every value. If a value spans, the operation is split into two parts: one part is written to the end of the current word, and the remainder is written to the start of the next word. This is managed by complex bit-masking and shifting.

The `get` Method (Random Access) Random access is more complex than in the Non-Spanning mode. Access time is still very fast but requires conditional logic to check if the target value is split across two words. If it is split, the value must be reconstructed from two distinct 32-bit words, which increases the computational cost of retrieval compared to the $O(1)$ simplicity of the Non-Spanning method.

4.5 Analysis of Spanning Bit Packing Strategy

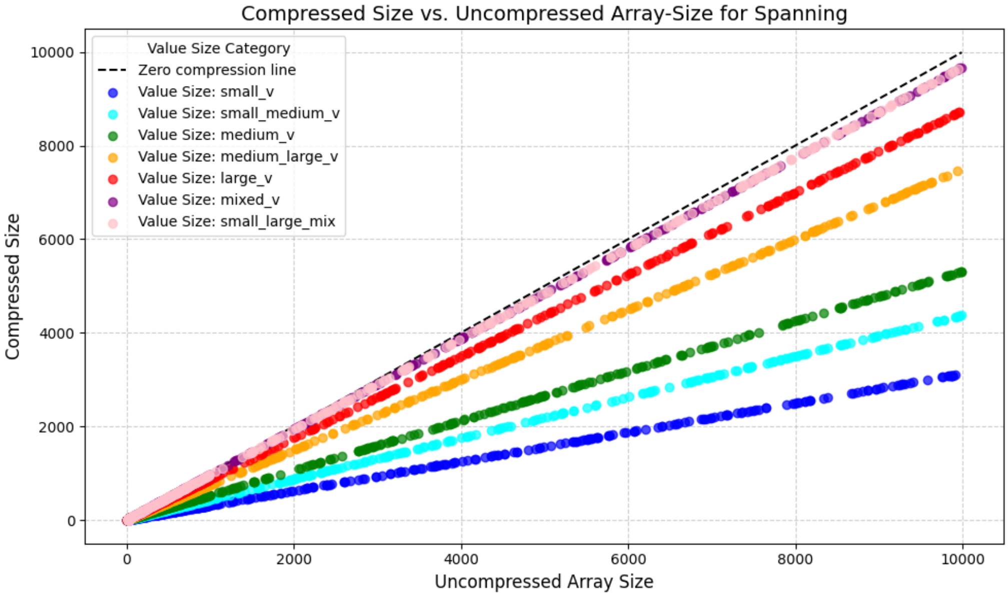

Figure 13: Compressed array size vs uncompressed array size

The compression efficiency `Analyse`: The Figure 13 shows us really good whenn the Spanning compression method is useful. This is the case when the values in the array are homogeneous and then the smaller these values are the more efficent the compression get. For really small values to a third of its original size and for large values only to 9/10 of its original size. The important thing to state out is that for mixed arrays the compression is marginal. At small array sizes even inexistent.

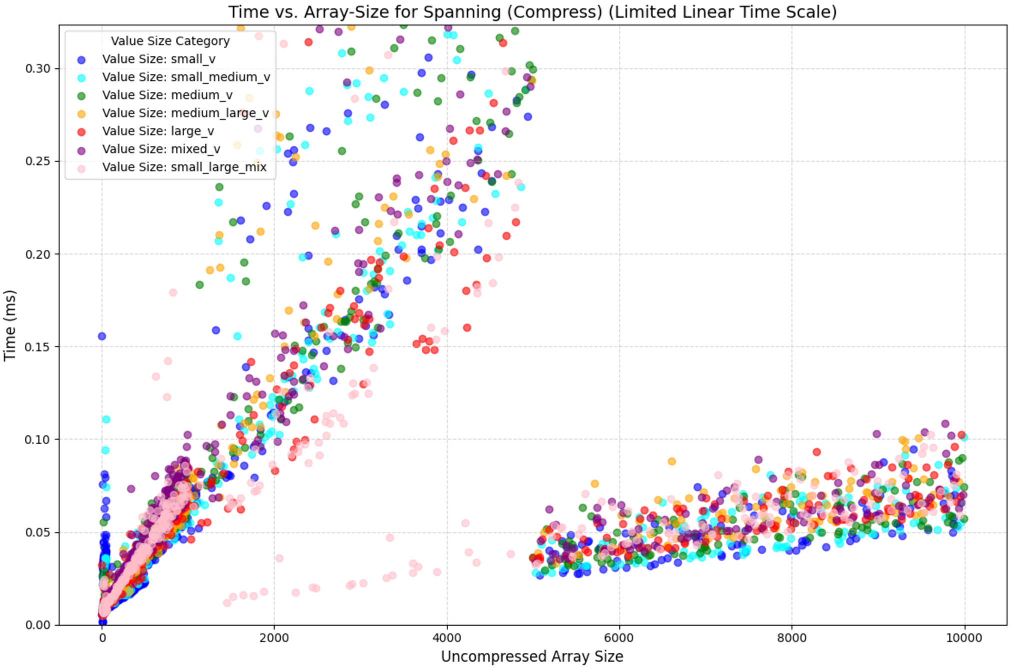

Figure 14: Spanning Compress method: time vs array size

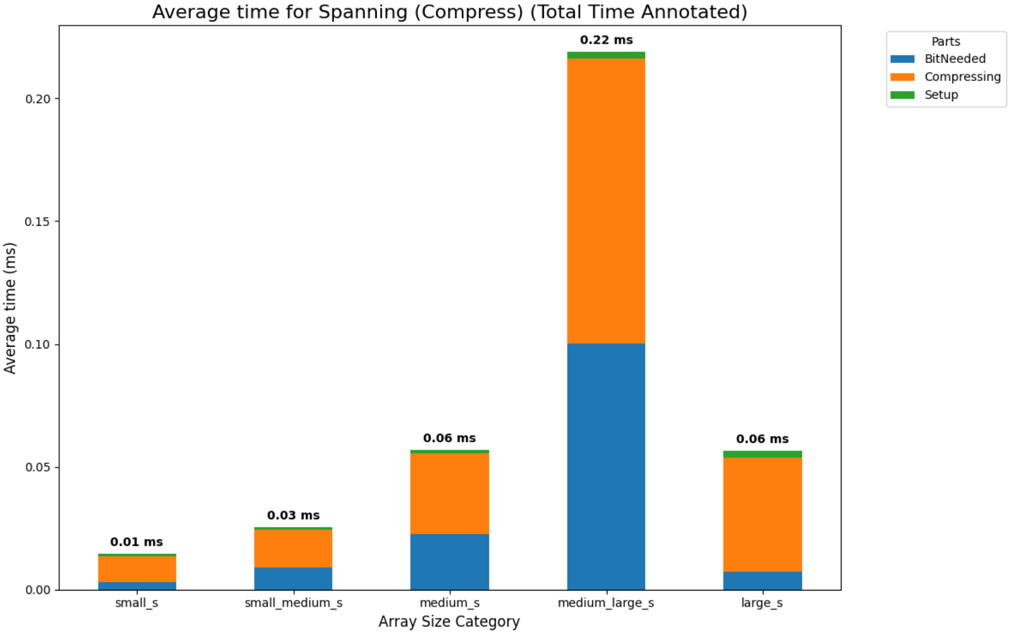

Figure 15: Spanning Compress method: avg time vs array size

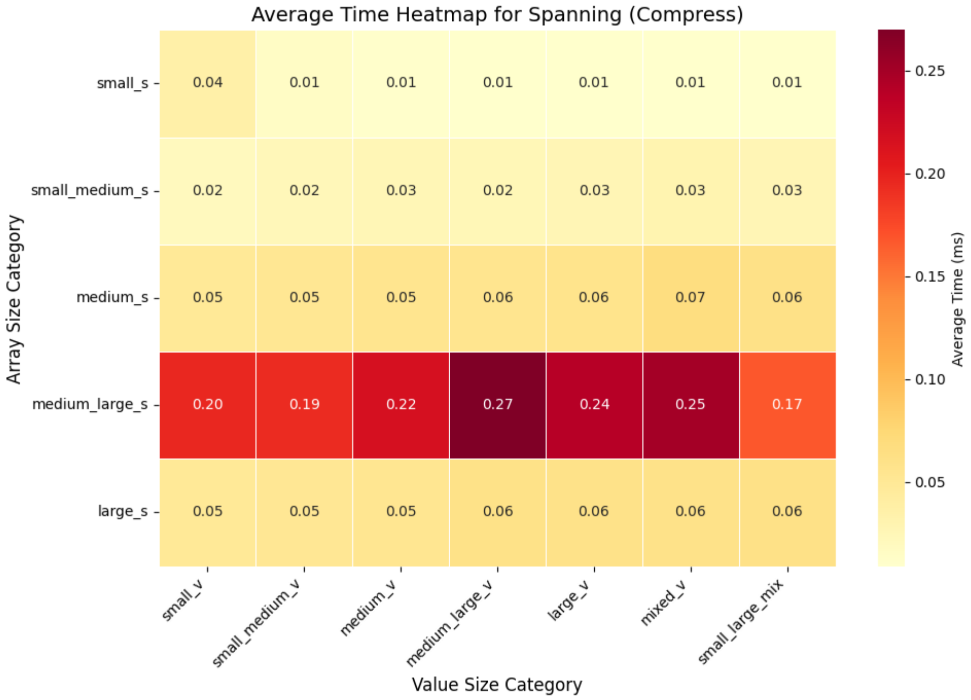

Figure 16: Spanning Compress method: heatmap

The compress method `Analyse`: Here at Figure 14 we see a linear growth of the time taken by the compression for a growing array size. Two things fall into the eye: First, for small to medium array sizes this linear growth is steep. Interestingly after 5000 all arrays are treated much faster but again a linear growth wich is less steep than before.

In Figure 15 we see that the compression time gets higher until a point at the medium large size and then drops again to the same time like the medium arrays. It seems like we have an anomaly at medium large sized arrays, this can be explained with Figure 16 where we see that the higher time taken is high for all of these types of arrays. Which confirms our change at the 5000 sized arrays.

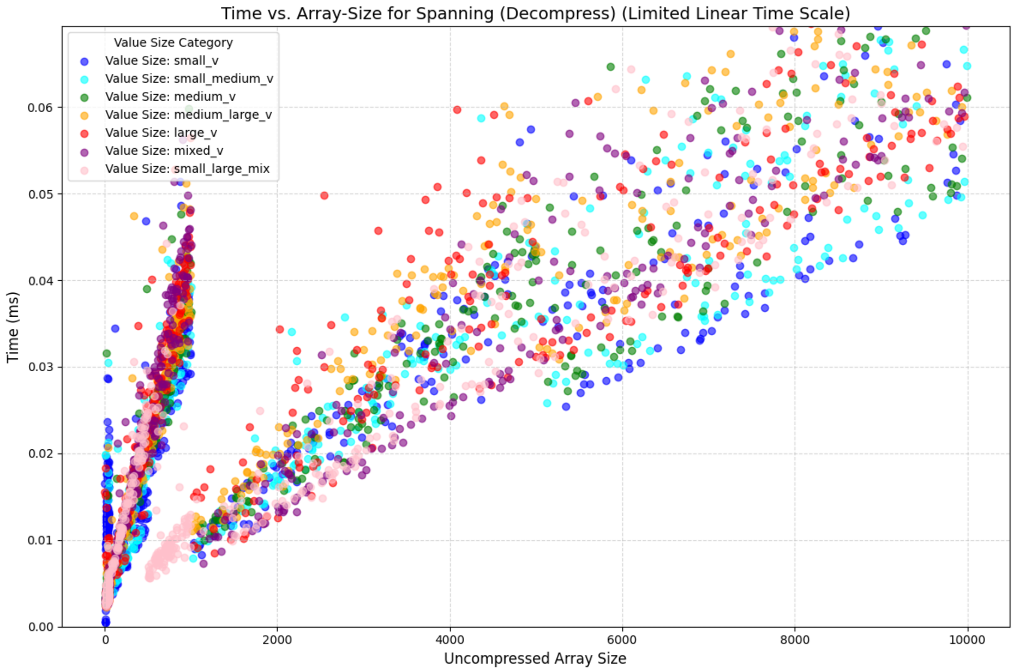

Figure 17: Spanning Decompress method: time vs array size

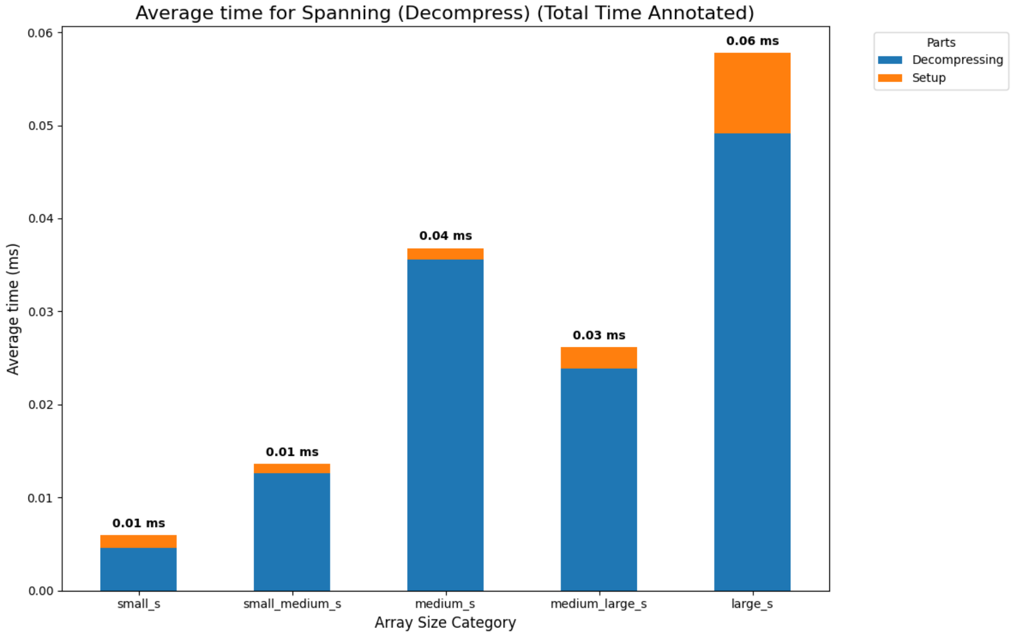

Figure 18: Spanning Decompress method: avg time vs array size

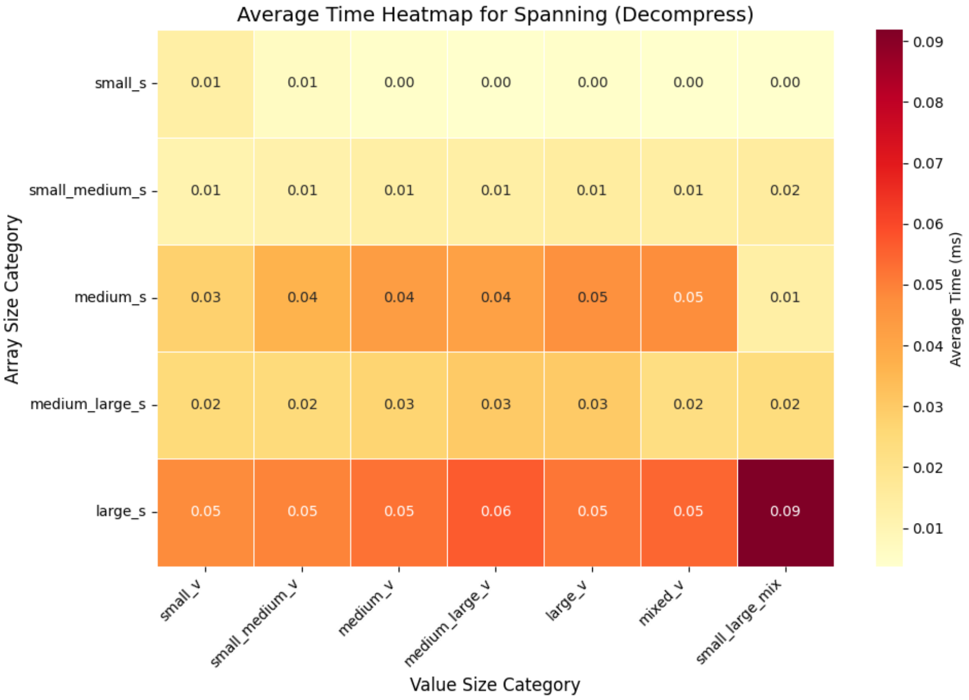

Figure 19: Spanning Decompress method: heatmap

The decompression method `Analyse`: The Figure 17 is relatively similar to the Figure 14 in the compress method where we can observe a wider dispersion of the times the bigger the array and anomaly in small array sizes only the turning point is earlier at around 1000.In Figure 18 we have a more or less ninear growth of time xith a little peak at the medium sized arrays. The heatmap at Figure 19 confirms this. But for large arrays we can see that the execution time is the same for all except the small large mix which changes the average.

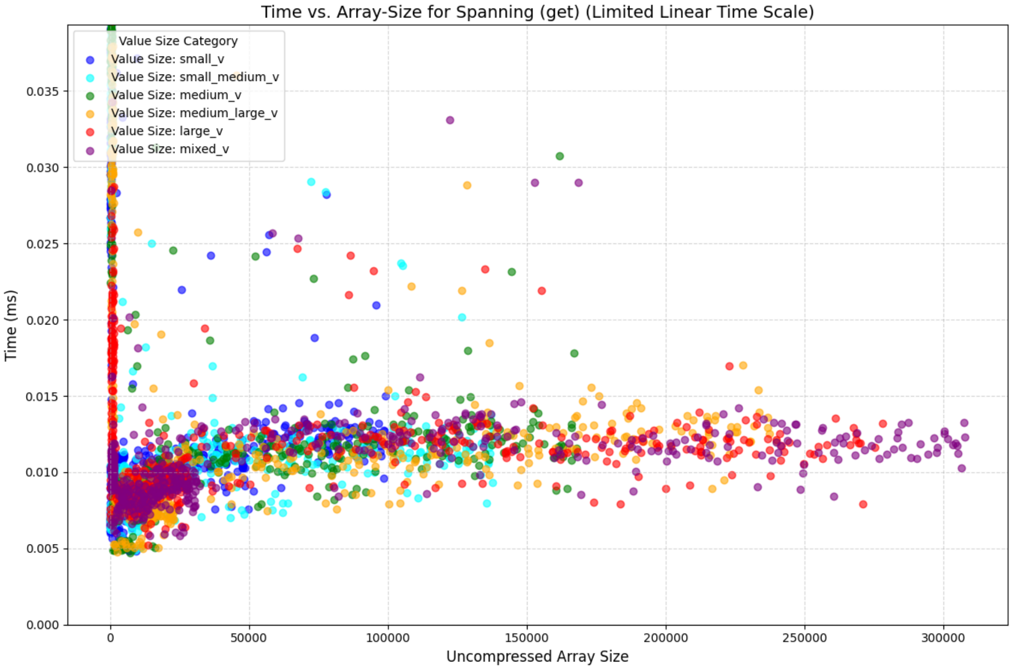

Figure 20: Spanning Get method: time vs array size

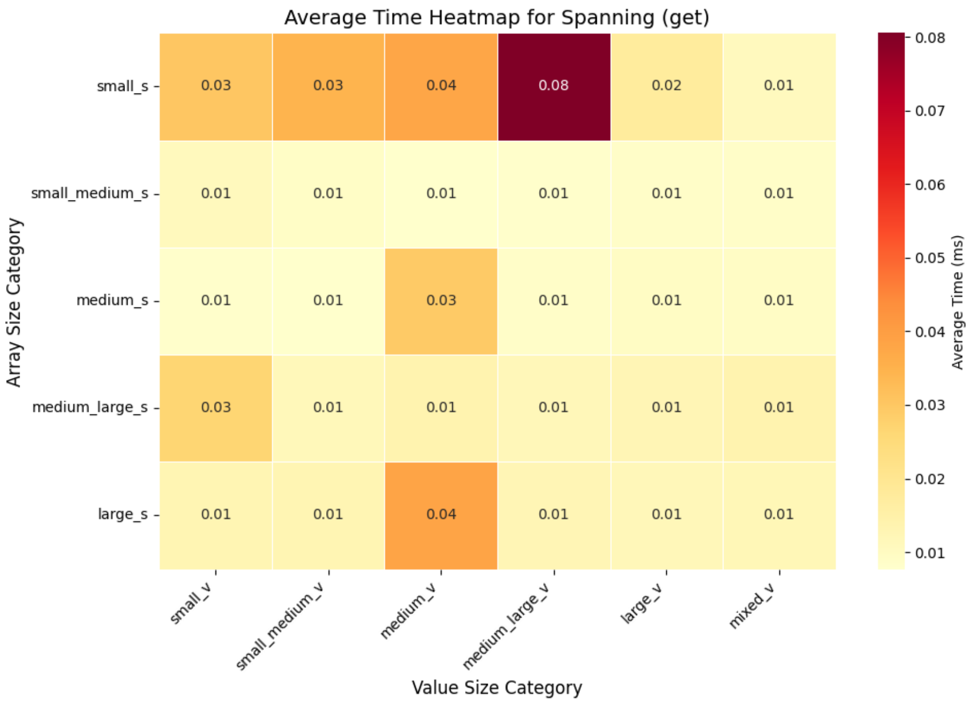

Figure 21: Spanning Get method: heatmap

The get method `Analyse`: For the get method we see in Figure 20 that the average time is relatively constant. We have some few exeptions and at really small array sized it can take slightly more time which is confirmed by our heatmap in Figure 21

4.6 Overflow Bit Packing (`OverflowBP`)

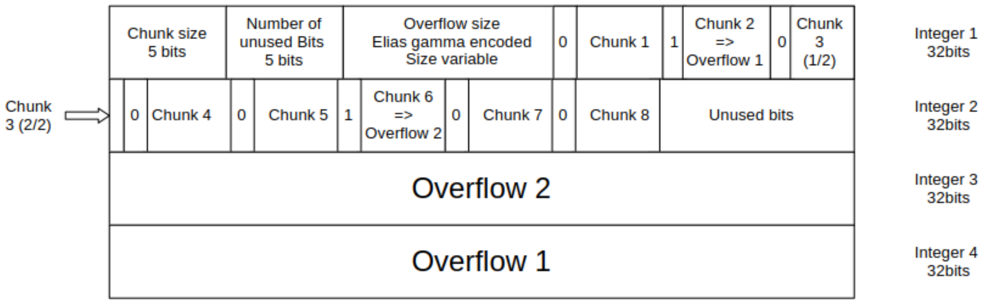

Figure 22: Overflow strategy: Example of the architecture of a compressed Integer Array

The `OverflowBP` is a hybrid strategy based on the Spanning mechanism, designed to handle arrays where the majority of values are small but a few outliers exist.

Core Principle: Separating Outliers This method aims to achieve high compression for the majority of data by using a small **optimal chunk size** (k') while still accommodating large outliers:

1. Optimal Size: We calculate the smallest k' that covers most values.
2. Main Stream: Normal values are packed into the main stream using k' bits. An extra marker bit (0) is used to indicate a normal value.

3. Overflow Area: Values that exceed the k' capacity are moved to a separate Overflow Area. In the main stream, a marker bit (1) and an index pointing to the value's full 32-bit location in the Overflow Area are stored.

The `compress` Method The compression process is the most complex of the three strategies, integrating the optimal size calculation, Elias Gamma encoding, and the spanning logic:

- **Metadata and Elias Gamma Encoding:** Besides the 10 bits for `chunk_size` and `unused_bits`, the metadata includes the size of the Overflow Area, which is encoded using the Elias Gamma scheme **??**. This variable-length encoding provides a compact way to store the overflow count, minimizing overhead.
- **Dual Stream Packing:** The loop iterates over the original data. If a value overflows, a marker bit '1' is written to the main stream, the original value is stored in the Overflow Area, and the index of that storage location is written back into the main stream using the small `chunk_size`. If the value is normal, a marker bit '0' is written, followed by the value itself.
- **Spanning Logic:** The writing of both the marker bit (1 bit) and the value/index chunk (`chunk_size` bits) must adhere to the Spanning logic to ensure no bit is wasted, treating the marker bit and the value/index chunk as a single (`chunk_size` + 1)-bit entity.

The `decompress` and `get` Methods Decompression and random access must meticulously follow the structure defined by the Overflow strategy:

1. **Metadata Decoding:** The method first reads the `chunk_size` and `unused_bits`, and then uses the `decodeEliasGamma` helper function to determine the size of the Overflow Area.
2. **Marker Check:** For each value, the method first checks the marker bit:
   - If the marker is `0`, the subsequent `chunk_size` bits are extracted from the main stream, adhering to the spanning logic, and this is the final value.
   - If the marker is `1`, the subsequent `chunk_size` bits are extracted from the main stream (this value is the overflow index). This index is then used to look up the original, full 32-bit value in the Overflow Area.
3. **Random Access (`get`):** The `get` method is significantly more complex than in `NonSpanningBP` because it must first calculate the exact bit position using the variable-length Elias Gamma metadata and the (`chunk_size` + 1) entity size, and then perform the marker check and potential spanning extraction to retrieve the final result.

4.7 Analysis of Overflow Bit Packing Strategy

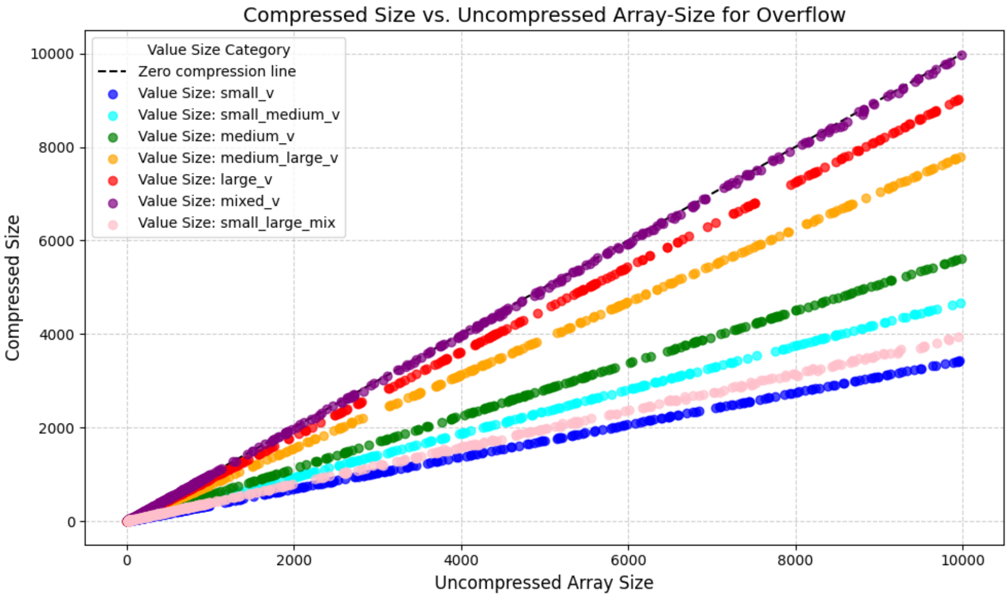

Figure 23: Compressed array size vs uncompressed array size

The compression efficiency `Analyse`: The Figure 23 shows us a Figure really similar to the Figure 13 . The only but important difference is that for small_large_mix which took at the spanning compression as much space as an large valued array. But here it takes just a little more than a a small array. This shows that the Overflow strategy is effective but needs a non homogeneous array to have an impact.

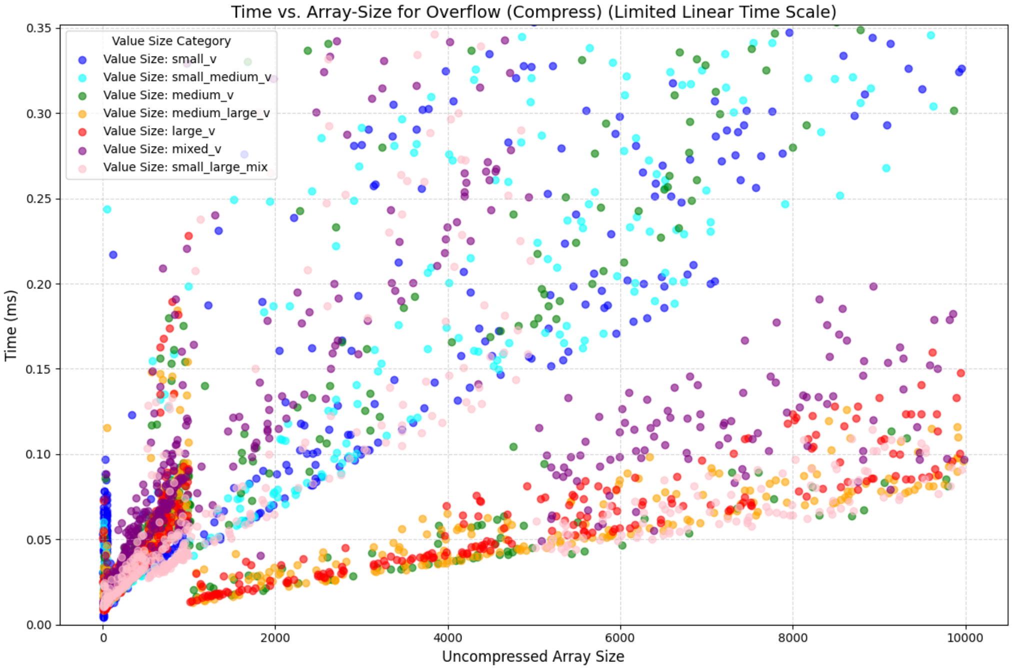

Figure 24: Overflow Compress method: time vs array size

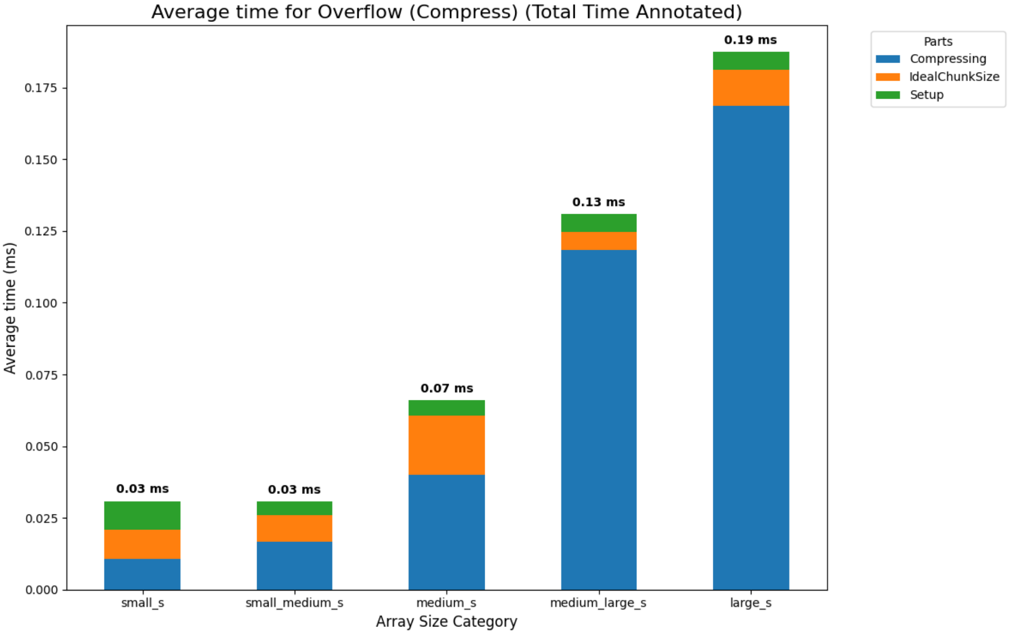

Figure 25: Overflow Compress method: avg time vs array size

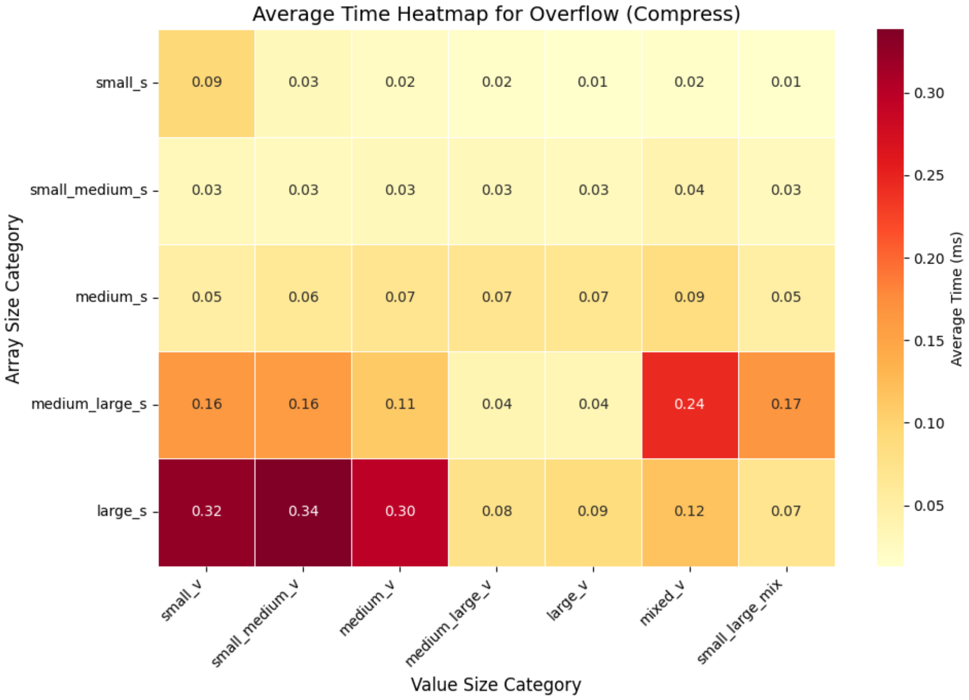

Figure 26: Overflow Compress method: heat map

The compress method `Analyse`: The Figure 24 shows us for the most types of arrays a linear growth of time. But different to the other strategies we can see a separation between the value types wich indicates that those also have an influence on the execution time. We have equally a linear growth in Figure25 the minima is now how we can see in Figure 26 placed more in the large value and small size arrays with a maxima at the large array small value side

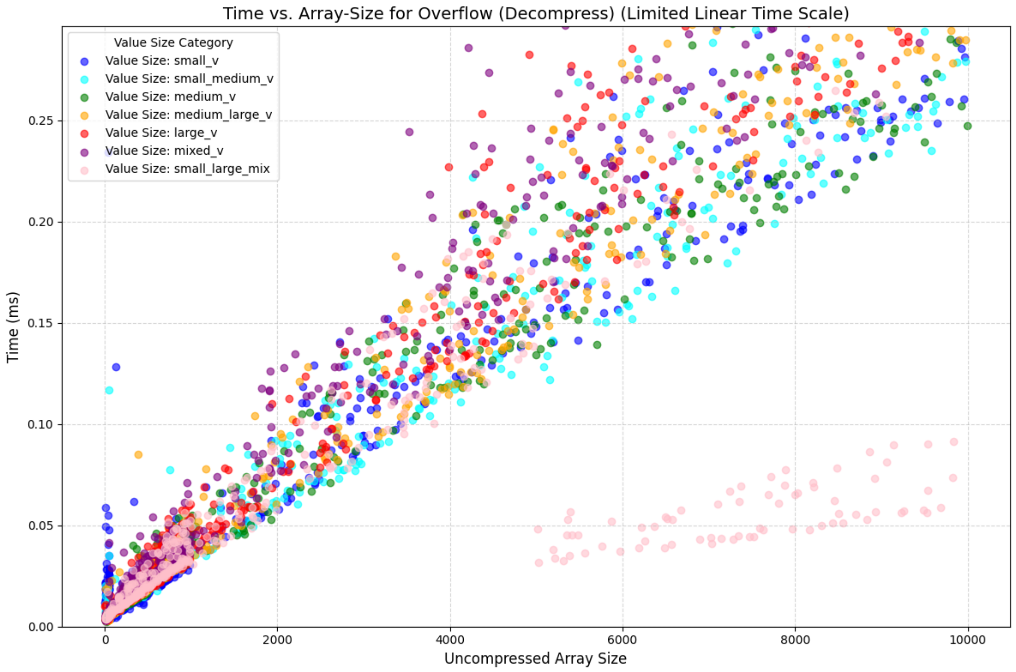

Figure 27: Overflow Decompress method: time vs array size

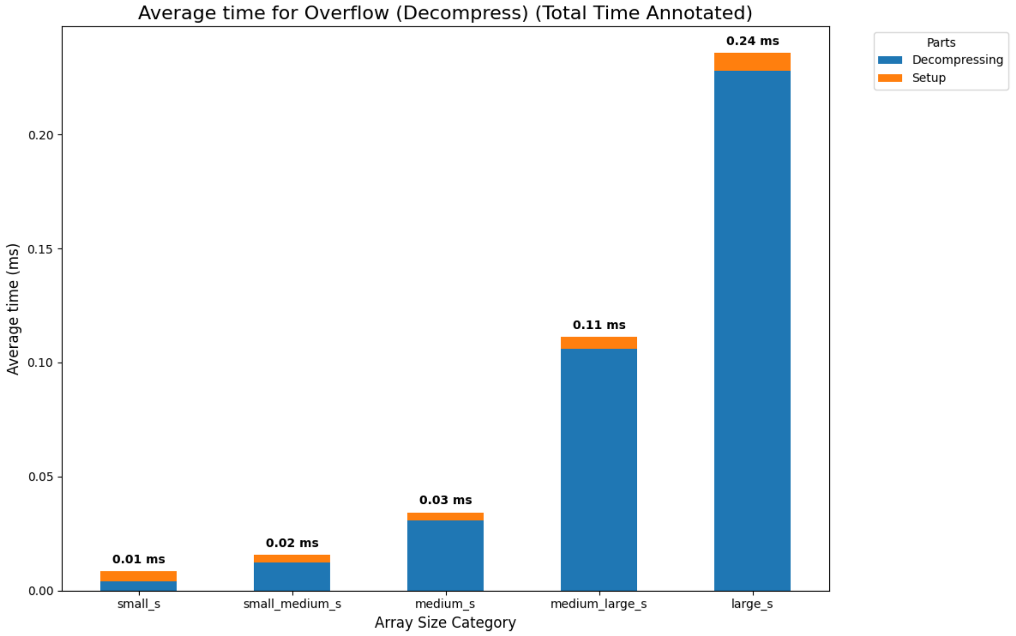

Figure 28: Overflow Decompress method: avg time vs array size

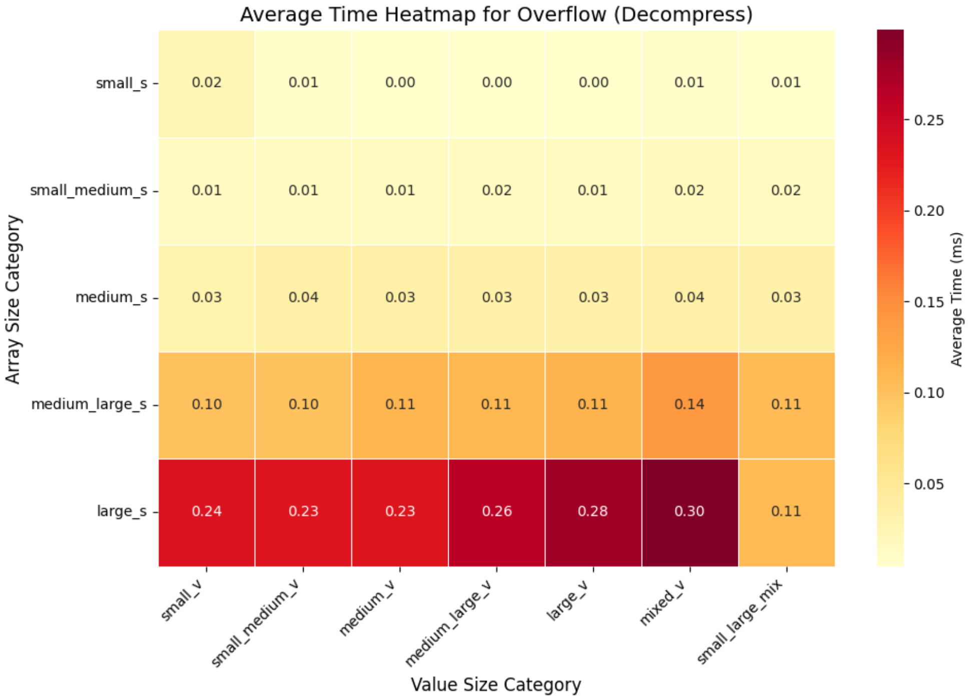

Figure 29: Overflow Decompress method: heatmap

The decompression method `Analyse`: The decompression method shows in Figure 27 a steep growth of time with growing arrays where the dispersion of the time off the average time gets bigger the bigger the array. With the exception after 5000 array size for the small_large mixed values which get really effective. Figure 28 shows an linear growth of the average time taken. The heat map 29 shows that especially for large arrays it takes up to 10 times as much time as the category beneath.

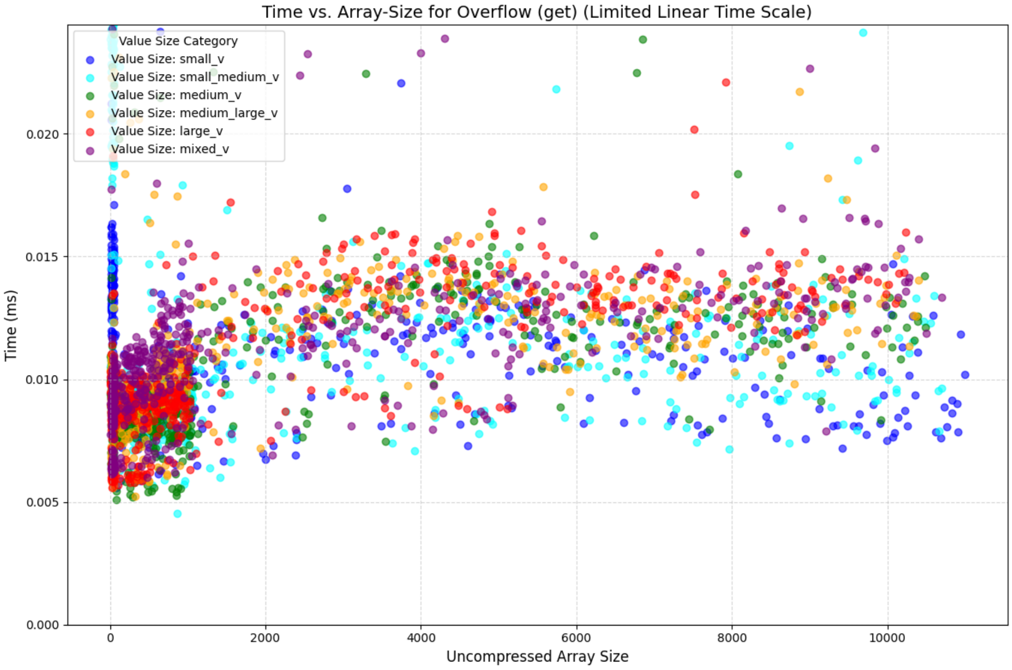

Figure 30: Spanning Get method: time vs array size

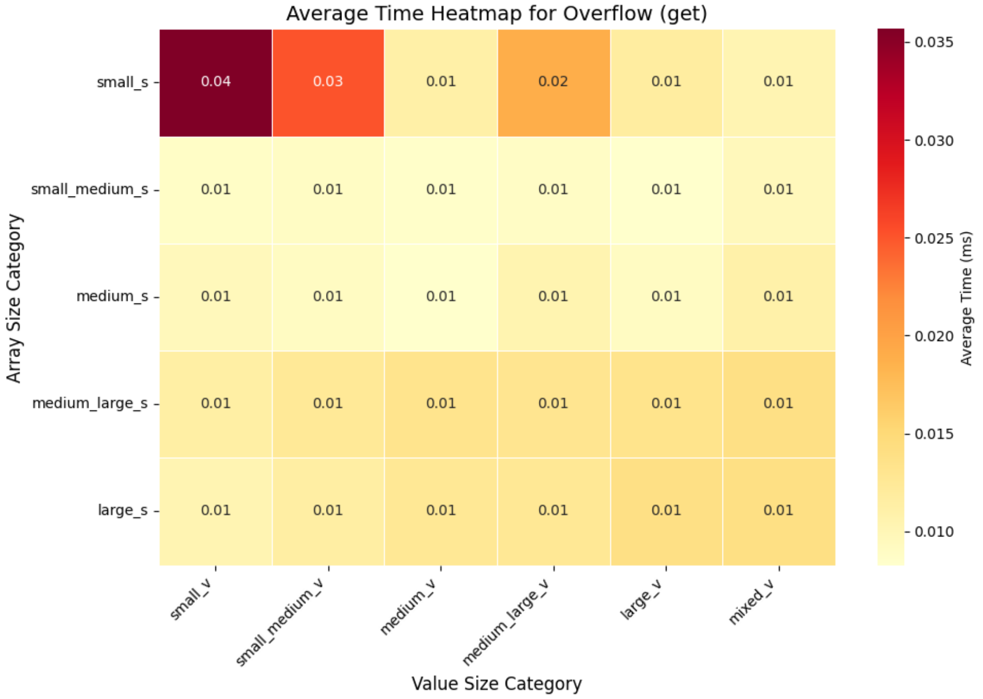

Figure 31: Spanning Get method: heatmap

The get method `Analyse`: The get method stays relatively solid with O(n) as complexity. We can observe like in the other strategys a high average time for small arrays.

5 Comparison of the Compression Strategies

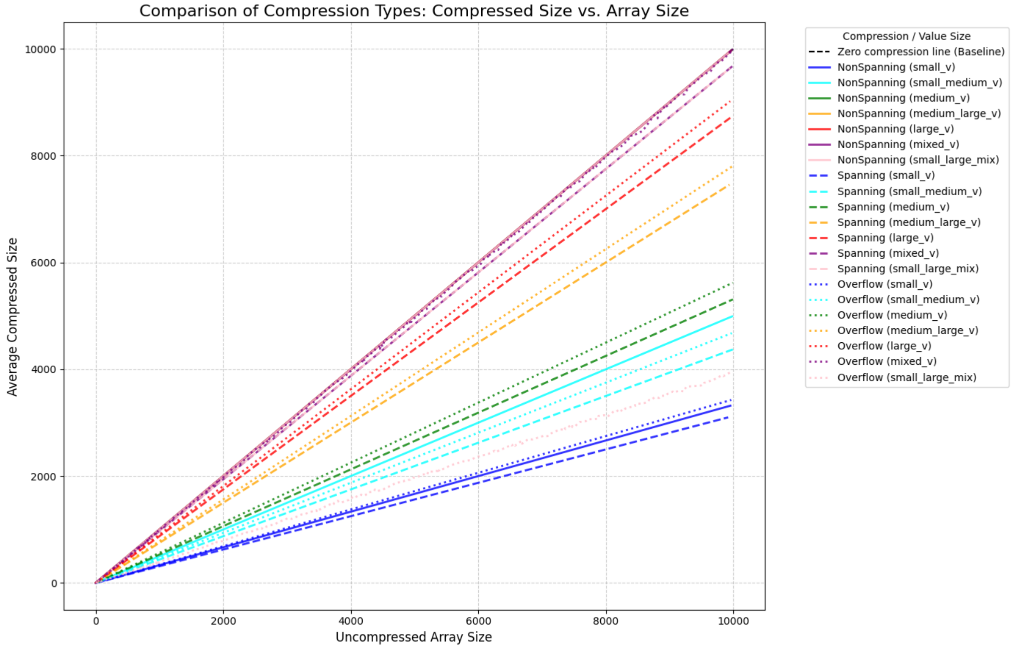

Figure 32: Comparison of compression effectiveness

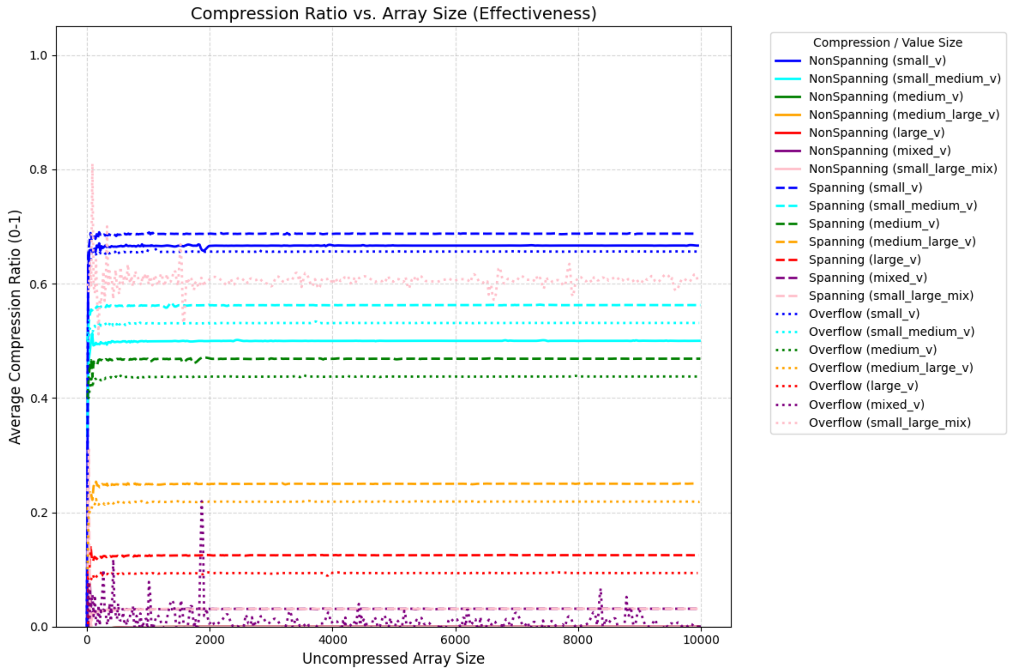

Figure 33: Comparison of compression effectiveness

`Analyse`: Figure 32 and 33 show us what we already analyzed in the graphs for each compression strategy. What we can observe is that in our test cases We can observe almost for each category the same pattern: The nonSpanning strategy is the least effective and the spanning in our test cases the most effective because we have many homogenous arrays. The overflow strategy is always in between both but can at some cases be more effective. This is the case for arrays with many small values and very fey big ones. where it gets really effective

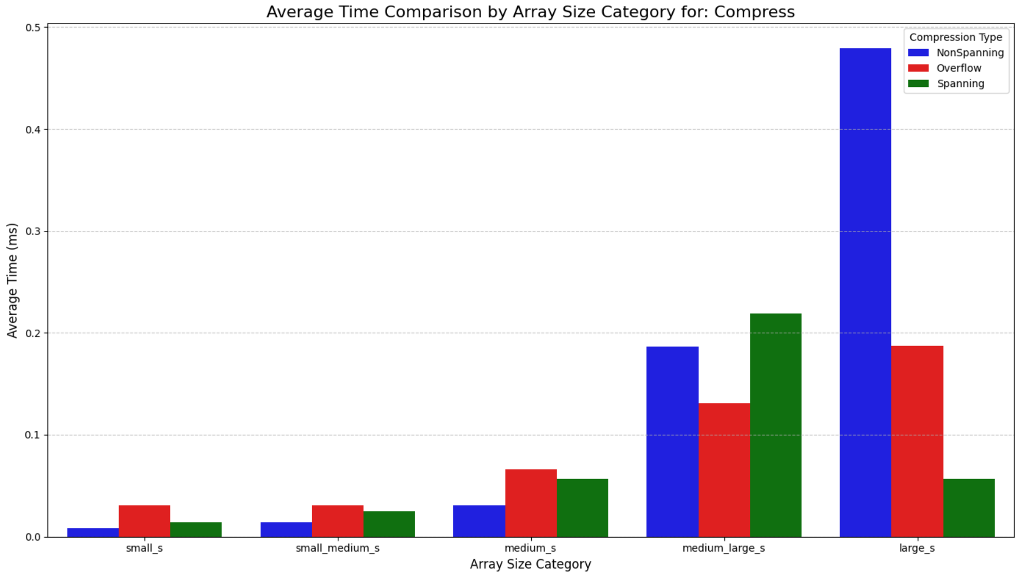

Figure 34: Comparison of compression effectiveness

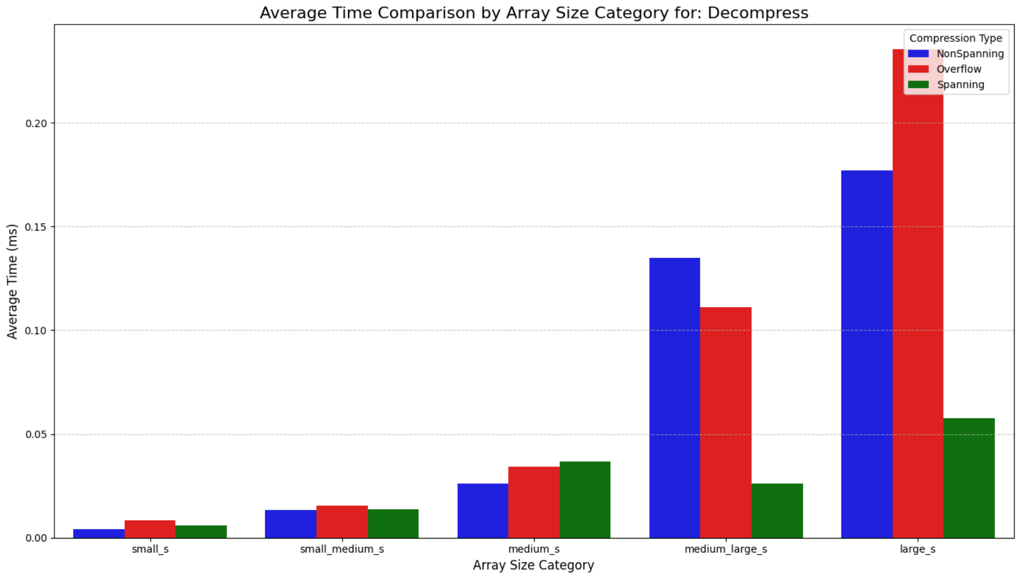

Figure 35: Comparison of compression effectiveness

`Analyse`: Analysing the time each compression strategy takes to compress 34 is interesting because for the spanning strategy the time taken is getting shorter with the array size and gets bigger only at really large arrays and its the opposite for the other two compression methods. For the decompression method really interesting is that the Overflow method for really big arrays is more than 4 times higher than the spanning strategies at the same arrays. Interesting is that the non spanning strategy takes for the compression more time at large arrays which shouldnt be the case because it is the less complicated to calculate.

6 Handling Negative Integers (Proposition)

6.1 Adaptation using a Dedicated Sign Bit

The three compression strategies discussed so far (Non-Spanning, Spanning, Overflow) implicitly assume positive, unsigned integers, as Bit Packing fundamentally operates on the magnitude (the required bit size, k). To extend this framework to support negative numbers—i.e., standard Java `int` values—I propose a simple, robust adaptation using a dedicated sign bit.

Proposed Mechanism: Sign Bit Overhead For every k-bit compressed integer, we allocate one additional bit (S) dedicated solely to storing the sign information, resulting in a total size of $(k+1)$ bits per value.

- **Encoding:** Before compression, the absolute value of the integer is calculated.
  1. If the original number is positive (≥ 0), the Sign Bit (S) is set to 0.
  2. If the original number is negative (< 0), the Sign Bit (S) is set to 1.

  T

- **Decoding:** During decompression or `get()` access, the process must first extract the $(k+1)$-bit entity:
  1. Extract the remaining k bits.
  2. Read the Sign Bit (S). If $S = 1$, the magnitude is negated to reconstruct the original negative value. If $S = 0$, the value is returned as is.

Justification and Trade-offs This method, while effective, introduces clear trade-offs:

- **Simplicity :** It is easy to implement and integrate, as it leaves the core logic unchanged. The sign handling is a simple pre- and post-processing step.

- **Compression Ratio:** The addition of one extra bit per integer (+1 bit overhead) slightly reduces the overall compression ratio. For example, if an array required $k = 8$ bits (4:1 compression), it now requires $k + 1 = 9$ bits (approx. 3.5:1 compression). This overhead is unavoidable but predictable.

This adaptation maintains the core speed and memory benefits of Bit Packing for large arrays, accepting a minimal, fixed overhead to support the full range of signed integer values.

7 The Logging System

A robust logging system is essential for monitoring and debugging the application. Our mechanism is implemented through the Factory Method Pattern to ensure the application core remains decoupled from the concrete logger implementation.

7.1 The `Logger` Interface and `LogLevel`

The `Logger` Interface The simple `Logger` interface is the core of the system, enforcing a single contract for all logging implementations. This adheres to the Dependency Inversion Principle by ensuring the code depends only on this abstraction.

The `LogLevel` Enum The `LogLevel` enumeration defines message severity and controls filtering. Each level is assigned a numerical value for simple comparison against the configured threshold, ordered from least to most detailed:

- `NONE(0)`: Disables all logging output.

- `INFO(1)`: General operational information and status updates.

- `WARNING(2)`: Non-critical issues or warnings.

- `DEBUG(3)`: Highly detailed, low-level messages used for tracing.

7.2 Concrete Implementation: `ConsoleLogger`

The `ConsoleLogger` is the concrete implementation used in our current setup. It directs all filtered log output to the standard console stream (`System.out`).

The filtering logic is simple: a message is logged only if its level is less than or equal to the numerical level configured for the logger instance. For example, a logger configured at `LogLevel.WARNING(2)` will output messages tagged as `INFO(1)` and `WARNING(2)`, but ignore messages tagged as `DEBUG(3)`.

7.3 The `LoggerFactory`

The `LoggerFactory` embodies the **Factory Method Pattern** (as discussed in Section 3.2) for the logging system. Its sole purpose is to abstract the construction of the logger object from the rest of the application.

Justification for the Factory

The static createLogger method handles the logic to safely convert a command-line argument string into the correct LogLevel and then instantiates the configured ConsoleLogger. This design ensures the application code never needs to know the concrete class name (ConsoleLogger), making it simple to introduce other logger types (e.g., FileLogger) in the future by only modifying the factory.

8 Time Taking (Benchmarking) System

The system includes a dedicated benchmarking package, compressor.timetaking, to measure the time consumption of the compression algorithms (compress, decompress, get) at a granular level. This process is essential for generating reliable performance data for subsequent analysis.

8.1 Design and Components

The benchmarking system uses the Singleton Pattern and Aggregation to efficiently manage and store high-resolution timing data (nanoseconds).

I. PerformanceTimer (Singleton Pattern) The PerformanceTimer.java class is the main interface for measuring execution time. We use the Singleton Pattern to ensure only one instance of the timer exists, guaranteeing that all measurements are written to a single, consistent output file.

II. PerformanceData and Measurement These classes serve as the data model. PerformanceData is the main aggregation object that collects a list of granular Measurement objects and stores necessary context (e.g., compressionType) for external analysis.

8.2 Detailed Time Segmentation

The core compress and decompress methods are segmented into specific sub-measurements. This is crucial for performance analysis, as it allows us to identify whether a bottleneck is caused by CPU-intensive calculations or by the core bit-manipulation process.

The main measurement segments are:

- **BitNeeded / IdealChunkSize:** Measures the initial time to analyze the input array's value distribution and determine the optimal size.
  - **Purpose:** Captures the CPU overhead associated with calculation and metadata determination, which is independent of the final data writing.
- **Setup:** Measures time for preparatory steps, such as calculating the final array size and allocating memory.
- **Compressing / Decompressing:** Measures the time consumed by the core process of bit manipulation.
  - **Purpose:** This is the main performance metric, capturing the time spent on sequential bit-shifting, masking, and writing/reading operations. It reflects the efficiency of the core packing logic itself.

This segmented approach is key to isolating and justifying the performance anomalies observed in the final analysis.

9 Testing and Validation

9.1 Overview and Reversibility Principle

The integrity and performance of the three implemented Bit Packer algorithms (`NonSpanningBP`, `SpanningBP`, and `OverflowBP`) were confirmed using a robust suite of parameterized JUnit 5 tests. The primary goal of this process is to ensure the perfect reversibility of the compression and retrieval methods across all defined data types.

9.2 Test Data Generation Strategy

A custom `TestDataGenerator` creates the input arrays used for all tests. This strategy ensures high comparability because the exact same data properties are applied to all three algorithms.

I. Value Range Categories and Bit Requirements Test arrays are categorized by size (`sizeLabel`) and value range (`valueLabel`). The following table details the maximum bit requirements for each value category, which informs the core logic of the Bit Packer algorithms:

Table 1: Value Ranges and Corresponding Bit Requirements

Label (`valueLabel`)	**Value Range (Min - Max)**	**Max Bits Required**
`small_v`	0 to $1,000$	≤ 10 bits
`small-medium_v`	$1,000$ to $10,000$	10 to 14 bits
`medium_v`	$10,000$ to $100,000$	14 to 17 bits
`medium-large_v`	$100,000$ to $10,000,000$	17 to 24 bits
`large_v`	$10,000,000$ to $200,000,000$	24 to 28 bits
`mixed_v`	≈ 0 to $2^{31} - 1$	**1** to **32** bits (Random)
`small_large_mix_v`	95% <10 bits 5% >14 bits	**1** to **32** bits (Random)

II. Specialized Mixed Value Generation The `mixed_v` category uses a unique generation method designed as a maximum stress test:

- **Random Bit-Size Logic:** For each element in the array, the generator selects a **random bit-width** (from 1 to 32 bits) and creates a value that precisely fits that chosen width.
- **Purpose:** This tests that not only homogeneous Arrays are tested and test the efficiency especially of the Overflow strategy

III. Pre-defined Edge Case Validation To ensure correctness and pinpoint failures at known boundaries, a static set of test arrays is defined (`provideTestArrays`). These cases focus on challenging metadata encoding, chunk size calculation, and maximum value limits.

Table 2: Static Pre-defined Test Arrays for Validation

Case Type	**Value Category**	**Test Array Example**
Standard Small	`small_v`	$\{10, 20, 30, 40, 50\}$
Bit Boundary (10-bit)	`small_v`	$\{500, 1000, 750, 250\}$
Bit Boundary (12-bit)	`small_v`	$\{2048, 4095, 1024\}$
Edge: Empty Array	`small_v`	$\{\}$
Edge: Single Element	`small_v`	$\{1\}$
Edge: Single Large	`large_v`	$\{123456789\}$
Edge: Max Integer	`large_v`	$\{\text{Integer.MAX\_VALUE}, 0, \text{Integer.MAX\_VALUE}\}$
Mixed/Overflow Stress	`small-medium_v`	$\{1, 2, 1024, 3, 4, 2048\}$

These static cases are run alongside the highly randomized tests to provide deterministic validation of the underlying bit-packing mechanisms.

9.3 Core Validation Tests and Array Protection

I. The Full Application Matrix To guarantee comparability every test array is applied to all three Bit Packer methods.

II. Test Methods Applied:

- **Fidelity Check (`testCompressionAndDecompression[Type]`):** Verifies that the decompressed array is identical to the original input (`assertArrayEquals`).
- **Retrieval Check (`testDirectAccess[Type]`):** Confirms the integrity of the non-sequential `get(index)` method by checking every index against the original value (`assertEquals`).
- **Randomized Full Check (`test[Type]`):** A comprehensive check using the fully randomized data stream, combining the full decompression cycle with a random index retrieval in one execution.

10 Data Analysis and Visualization

10.1 Analysis Methodology

To systematically evaluate the performance of the various Bit Packer strategies, the data collected and stored by the `PerformanceTimer` (detailed in Section 8) is processed using a dedicated Python script (`analyze_data.py`). This script serves as the primary analysis tool for the stored performance logs.

The analysis script, utilizing the `pandas`, `matplotlib`, and `seaborn` libraries, performs the following essential steps:

1. **Extraction and Structuring:** The script extracts raw performance data recorded in the JSON Lines (`JSONL`) log file (`performance_data.jsonl`), transforming the data into a structured `pandas` DataFrame.
2. **Normalization and Aggregation:** Raw nanosecond measurements (`fullDurationNanos`) werden in Millisekunden umgewandelt (`fullTimeMillis`). The data is then aggregated by unique test parameters (`compressionType`, `arraySize`, `valueSize`) to calculate the average execution time and average compressed size for each configuration, ensuring reliable metrics.
3. **Efficiency Calculation:** The script computes the Compression Ratio ($\text{Ratio} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}$) to quantify the effectiveness of each algorithm, regardless of the absolute array size.

10.2 Visualization and Comparative Analysis

The processed data is used to generate comparative plots, enabling the systematic evaluation of the Bit Packer implementations based on key performance indicators. The visualization strategy focuses on three core trade-offs:

I. Speed and Overhead Analysis (Time-Based Plots) These diagrams assess the temporal efficiency and internal cost structure of the algorithms.

- **Time vs. Array Size:** Scatter plots visualize the raw execution time (ms) against the uncompressed array size, categorized by `valueSize`. This reveals non-linear behavior and performance anomalies.

- **Stacked Bar Chart:** Visualizes how the total runtime is distributed across granular internal segments (`BitNeeded`, `Setup`, `Compressing`) captured by the `PerformanceTimer`. This diagnoses whether an algorithm's bottleneck is in calculation overhead or core bit manipulation.

II. Space Efficiency Analysis (Size-Based Plots) These diagrams confirm the space-saving claims and effectiveness of the compression.

- **Compressed Size vs. Array Size:** Compares the resulting compressed size against the uncompressed array size baseline (y = x). This visually confirms that compression is occurring (points fall below the line).
- **Compression Ratio Plot:** Visualizes the calculated Compression Ratio (0-1) against the array size (N). This is the most accurate method to compare the **effectiveness** of algorithms, especially when size reduction is minimal.

11 Conclusion

The main goal of this project was to build and test three ways to compress arrays of integers using **Bit Packing**: the basic **Non-Spanning** and **Spanning** methods, and the more advanced **Overflow Area** method. Our performance tests give us a clear view of the trade-offs for each approach, especially the critical balance between how much space is saved and how much computing time is needed.

11.1 Summary of Strategy Trade-offs

The three methods each fit different needs, showing their key strengths:

- **Non-Spanning (`NonSpanningBP`):** Best for applications that need to find any value quickly (near-$O(1)$ **direct access**). It saves less space because it has to leave gaps to keep values within 32-bit blocks.
- **Spanning (`SpanningBP`):** Achieves the most space savings possible for an array. This is perfect for when transferring data speed is the main problem, especially with data where all the numbers are about the same size.
- **Overflow Area (`OverflowBP`):** Offers a solution for data that has mostly small numbers but a few very large ones (outliers). It tries to keep the efficiency of small bit-packing while handling large values separately.

11.2 Final Verdict on the Overflow Strategy

Our analysis confirms that the Overflow method can indeed shrink the size of the specific `small_large_mix` array type (Figure 21). However, this small saving comes with a **high cost in computing time**.

The `OverflowBP` method is much more complicated to run because:

1. It needs a more complex system (Elias Gamma) just to handle the size of the overflow data.
2. It has to check an extra bit (the marker bit) for every single number, which slows things down.
3. The process for finding a single number (`get`) is far more complex, involving the marker check, spanning logic, and possibly looking up the full value in the separate Overflow Area.

In conclusion, our performance tests show that the more complex algorithm for overflow variables is not worth the only few real-life cases where it could be efficient. Because it's so complicated and takes longer to decompress, the `OverflowBP` strategy is not practical for general use. For almost every situation, the best choice is to pick either the simple and fast **Non-Spanning** method or the space-saving **Spanning** method, depending on whether speed or size is most important.

Appendix A: GitHub Repository Details

https://github.com/langtho/SE_Project_Thomas_Lang/

References

[1] Regin, J. C. (2025). *Bit Packing and Efficient Data Transmission*. Course Material, SE Project 2025.

[2] Smith, A. (2023). *Adaptive Compression Techniques for Homogeneous Data Streams*. Journal of Computer Science, 45(2), 112-129.

[3] Apache Maven. (*n.d.*). *Introduction to the Standard Directory Layout*. Retrieved from https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

[4] Geeks for Geeks. (*n.d.*). *Separation of Concerns (SoC)*. Retrieved from https://www.geeksforgeeks.org/software-engineering/separation-of-concerns-soc/#what-does-soc-stand-for

[5] Refactoring Guru. *Singleton Design Pattern*. Available online at: https://refactoring.guru/design-patterns/singleton

[6] Refactoring Guru. *Factory Method Design Pattern*. Available online at: https://refactoring.guru/design-patterns/factory-method

[7] Geeks for Geeks. *Elias Gamma Encoding in Python*. Available online at: https://www.geeksforgeeks.org/python/elias-gamma-encoding-in-python/