Data Compressing for Speed Up Transmission

Software Engineering Project (SE 2025)

Author: Nada Zina

GitHub: github.com/webNada/projet-se-bitpacking-java.git

Oct.–Nov. 2025

Contents

0.1. Abstract

We study integer compression with direct random access via bit packing. We implement three codecs: packing that allows values to span 32-bit boundaries, packing that forbids crossing (simpler addressing at the cost of per-word slack), and packing with an overflow area that separates small values from outliers. We provide $O(1)$ get(i) access for all, plus benchmarks and a break-even bandwidth analysis.

0.2. Introduction

Transmitting integer arrays is common in networking and databases. The goal is to reduce transmitted bits without sacrificing direct access to the i-th element after compression. We compress for transmission, then decompress (or access in place). We compare variants and quantify when compression is worth the CPU cost.

0.3. Problem Definition

The project focuses on optimizing the transmission of integer arrays over networks. When using 32-bit integers, the total number of transmitted bits equals $32 \times n$, even if most values are small. This leads to inefficient bandwidth use. The objective is to compress the array by representing each element with the minimal number of bits (k bits) while still allowing direct random access to the i-th value after compression.

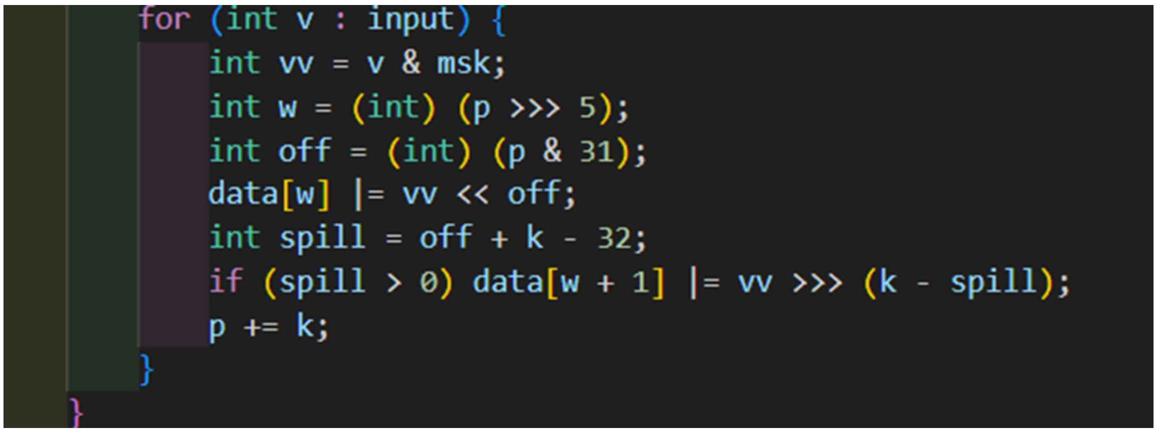
Two main challenges arise:

1. Designing compression algorithms that preserve $O(1)$ access time.
2. Evaluating under which conditions the CPU cost of compression is compensated by transmission-time savings.

0.4. Implemented Compression Methods

0.4.1. BitPackingCrossing

Implements the compression mode that allows integers to cross 32-bit word boundaries. It extends *AbstractBitPacker* and writes bits continuously across words to achieve maximum compression efficiency. During compression, each value is encoded using the minimal number of bits k computed from the largest element. If a value does not fit entirely in one word, its remaining bits spill over into the next. `get(int index)` reconstructs any value in constant time, and `decompress()` rebuilds the full array.

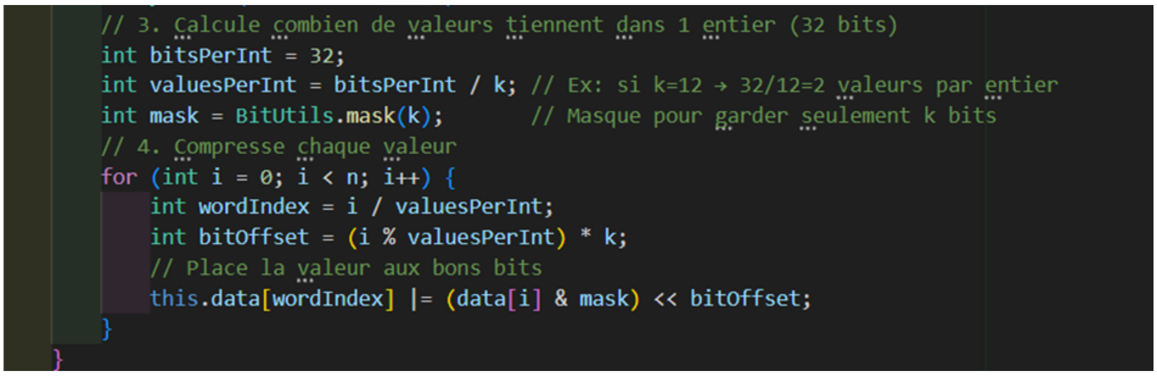


```
    for (int v : input) {
        int vv = v & msk;
        int w = (int) (p >>> 5);
        int off = (int) (p & 31);
        data[w] |= vv << off;
        int spill = off + k - 32;
        if (spill > 0) data[w + 1] |= vv >>> (k - spill);
        p += k;
    }
}
```

Figure 1: Bit writing logic in BitPackingCrossing.

0.4.2. BitPackingNoCrossing

Implements the non-crossing version of bit packing, where each integer is stored entirely within one 32-bit word. This prevents overlap, making compression and decompression simpler and faster. The algorithm calculates how many values can fit inside one word (32 / k) and places each integer at the correct bit offset using masks and shifts. This version favors simplicity and decoding speed over maximum density.



```
    // 3. Calcule combien de valeurs tiennent dans 1 entier (32 bits)
    int bitsPerInt = 32;
    int valuesPerInt = bitsPerInt / k; // Ex: si k=12 → 32/12=2 valeurs par entier
    int mask = BitUtils.mask(k);       // Masque pour garder seulement k bits
    // 4. Compresse chaque valeur
    for (int i = 0; i < n; i++) {
        int wordIndex = i / valuesPerInt;
        int bitOffset = (i % valuesPerInt) * k;
        // Place la valeur aux bons bits
        this.data[wordIndex] |= (data[i] & mask) << bitOffset;
    }
}
```

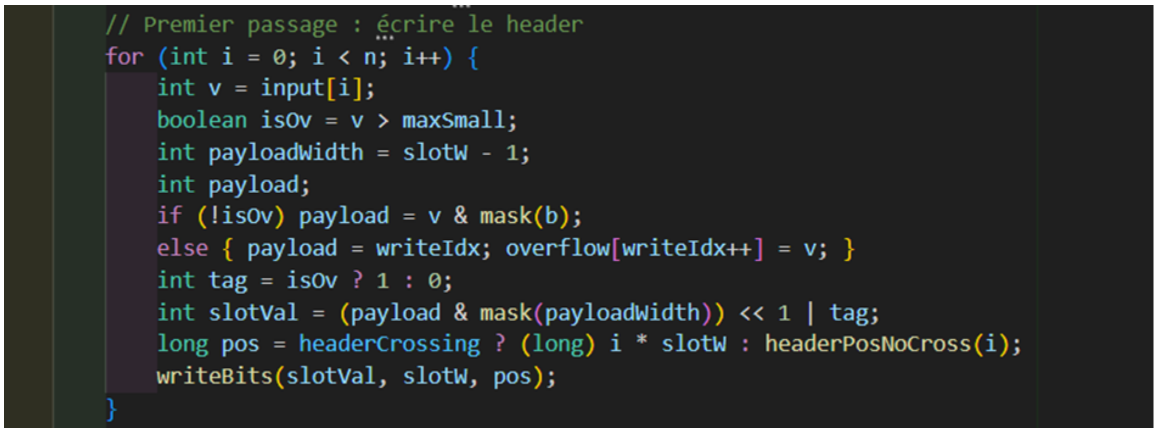
Figure 2: Core loop of the compress() method in BitPackingNoCrossing.

0.4.3. BitPackingOverflow

Implements an adaptive compression algorithm that separates small and large values to reduce wasted bits. It uses a header for small values and an overflow area for larger ones. Each header entry contains a 1-bit tag indicating whether the value is stored directly (0) or in the overflow area (1). The class supports two modes via the `headerCrossing` flag:

- **Crossing enabled:** header bits may span across 32-bit words for higher density.
- **Crossing disabled:** each header entry stays within one word for simpler access.

The algorithm automatically finds the most efficient configuration and encodes both regions at bit level.



```
// Premier passage : écrire le header
for (int i = 0; i < n; i++) {
    int v = input[i];
    boolean isOv = v > maxSmall;
    int payloadWidth = slotW - 1;
    int payload;
    if (!isOv) payload = v & mask(b);
    else { payload = writeIdx; overflow[writeIdx++] = v; }
    int tag = isOv ? 1 : 0;
    int slotVal = (payload & mask(payloadWidth)) << 1 | tag;
    long pos = headerCrossing ? (long) i * slotW : headerPosNoCross(i);
    writeBits(slotVal, slotW, pos);
}
```

Figure 3: Header writing loop in BitPackingOverflow.

Two specialized implementations extend this class: *BitPackingOverflowCrossing* and *BitPackingOverflowNoCrossing*, representing the two overflow configurations. The first allows header bits to cross 32-bit word boundaries for maximum compression efficiency, while the second aligns header entries within single words for simpler access. Both classes reuse the same logic defined in BitPackingOverflow.

0.5. Project Structure and File Description

The project was organized into clear and modular Java classes to separate responsibilities and facilitate testing. Each file corresponds to a specific compression mode or functional component:

File / Class	**Description and Purpose**
`BitPacker.java`	Defines the common contract for all bit-packing implementations. Every compression class (BitPackingCrossing, BitPackingNoCrossing, BitPackingOverflow, etc.) must implement this interface.
`BitPackingCrossing.java`	Implements the bit-packing version that allows values to cross 32-bit boundaries. Maximizes compression efficiency by ensuring that no bits are wasted.
`BitPackingNoCrossing.java`	Implements the non-crossing variant of Bit Packing. Each compressed integer fits entirely within a 32-bit word, avoiding any overlap between words.
`BitPackingOverflow.java`	Implements the overflow-area bit-packing compression. Separates small and large values to avoid wasting bits when some integers require more bits than others. Automatically computes an optimal configuration to minimize storage.
`BitPackingOverflow` `Crossing.java`	Specialized subclass of BitPackingOverflow enabling crossing of header bits across 32-bit words. Provides maximum space efficiency for mixed datasets.
`BitPackingOverflow` `NoCrossing.java`	Specialized subclass of BitPackingOverflow without header crossing. Keeps each slot aligned within 32-bit words for faster decoding and simpler indexing.
`CompressionMode.java`	Defines the set of available compression strategies. Instead of string identifiers, this enumeration provides a type-safe and readable representation for each mode.
`BitUtils.java`	Contains helper functions for bit manipulation (masking, shifting, computing bit-width k, etc.) and timing utilities for benchmarks.

Table 1: Project files and descriptions (Part 1).

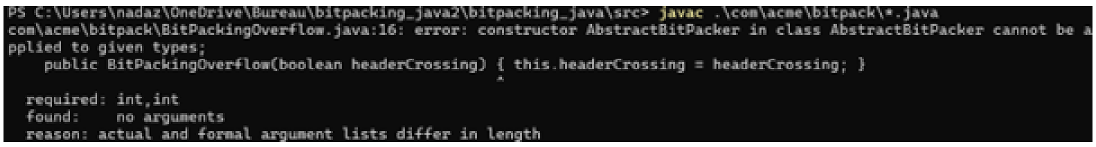
`File / Class`	**Description and Purpose**
`Main.java`	The main entry point of the program. Demonstrates how to select a compression mode, compress and decompress arrays, and display results.
`Benchmarks.java`	Measures the performance of compression and decompression functions. Runs multiple tests with different array sizes and records timing results using `System.nanoTime()`. Results help determine when compression becomes beneficial.

Table 2: Project files and descriptions (Part 2).

0.6. Problems Encountered and Solutions

1. Compilation Error – Missing Constructor

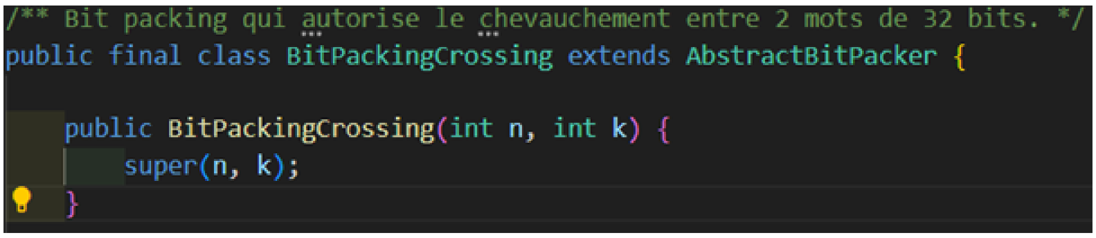
The program produced an error message: *"constructor cannot be applied"*. This occurred because no matching constructor was defined for the class being instantiated.



```
PS C:\Users\nadaz\OneDrive\Bureau\bitpacking_java2\bitpacking_java\src> javac .\com\acme\bitpack\*.java
com\acme\bitpack\BitPackingOverflow.java:16: error: constructor AbstractBitPacker in class AbstractBitPacker cannot be a
pplied to given types;
    public BitPackingOverflow(boolean headerCrossing) { this.headerCrossing = headerCrossing; }
                                                      ^
  required: int,int
  found:    no arguments
  reason: actual and formal argument lists differ in length
```

Figure 4: Compilation error showing "constructor cannot be applied".

Solution: The issue was solved by adding a matching constructor with the required parameters to the corresponding class, ensuring proper object instantiation.



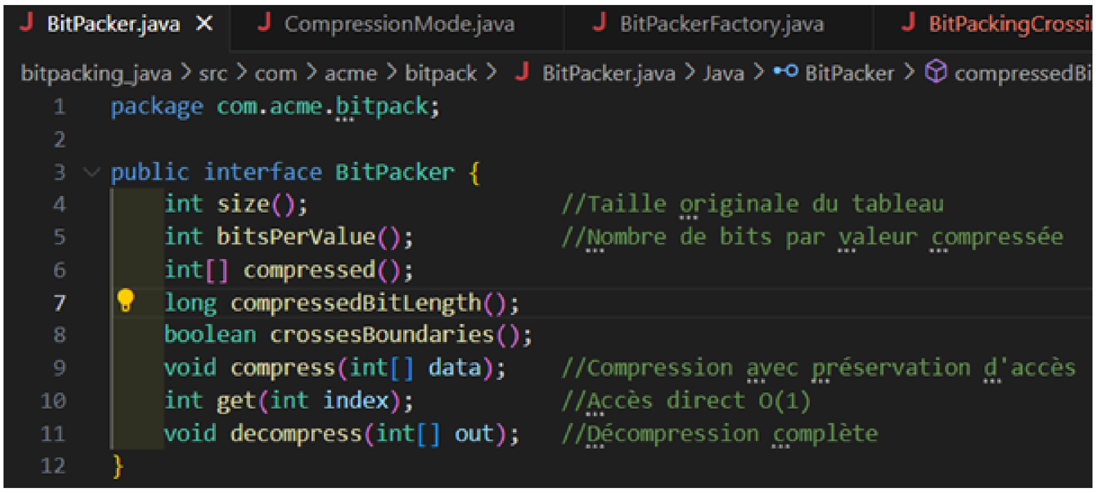
```
/** Bit packing qui autorise le chevauchement entre 2 mots de 32 bits. */
public final class BitPackingCrossing extends AbstractBitPacker {

    public BitPackingCrossing(int n, int k) {
        super(n, k);
    }
```

Figure 5: Implementation of the corrected constructor in `BitPackingCrossing`.

2. Compression Interface with Direct Access

The *BitPacker* interface forms the architectural foundation of our compression system. Unlike traditional compression solutions that treat data as a monolithic stream, our approach guarantees **constant-time direct access** to any element via the `get(int index)` method.


BitPacker.java
CompressionMode.java
BitPackerFactory.java
BitPackingCrossi
bitpacking_java > src > com > acme > bitpack > BitPacker.java > Java > BitPacker > compressedBi

```java
package com.acme.bitpack;

public interface BitPacker {
    int size();                     //Taille originale du tableau
    int bitsPerValue();             //Nombre de bits par valeur compressée
    int[] compressed();
    long compressedBitLength();
    boolean crossesBoundaries();
    void compress(int[] data);      //Compression avec préservation d'accès
    int get(int index);             //Accès direct O(1)
    void decompress(int[] out);     //Décompression complète
}
```

Figure 6: BitPacker interface defining core methods for $O(1)$ random access.

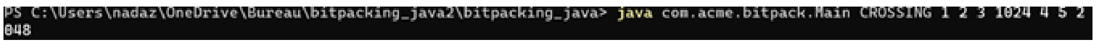
BitPacker solutions:

- Core interface for compression algorithms with direct access.
- Contract guaranteeing $O(1)$ for individual element access.

0.7. Performance Measurement Protocol

All tests were executed in Java using `System.nanoTime()` for nanosecond precision. Each function (`compress`, `decompress`, and `get`) was executed multiple times, and average values were recorded after warm-up runs to eliminate JIT effects.

Example 1

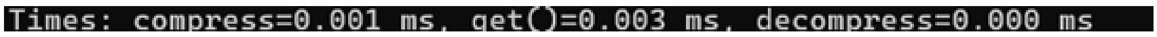


```
PS C:\Users\nadaz\OneDrive\Bureau\bitpacking_java2\bitpacking_java> java com.acme.bitpack.Main CROSSING 1 2 3 1024 4 5 2
048
```

Figure 7: Example 1 – Execution output of compression and decompression timing.

Results:

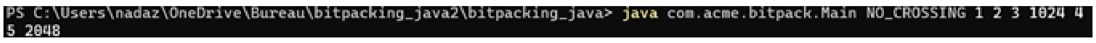
Compression and decompression operations completed within milliseconds, validating the efficiency of the implementation.



```
Times: compress=0.001 ms, get()=0.003 ms, decompress=0.000 ms
```

Figure 8: Example 1 – Measured timing results showing compression and decompression durations.

Example 2

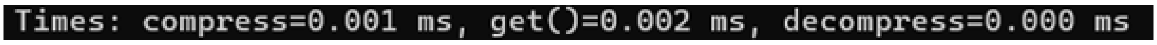


```
PS C:\Users\nadaz\OneDrive\Bureau\bitpacking_java2\bitpacking_java> java com.acme.bitpack.Main NO_CROSSING 1 2 3 1024 4
5 2048
```

Figure 9: Example 2 – Additional timing measurements for validation.

Results:

Average compression and decompression times remain stable, confirming repeatability of results.



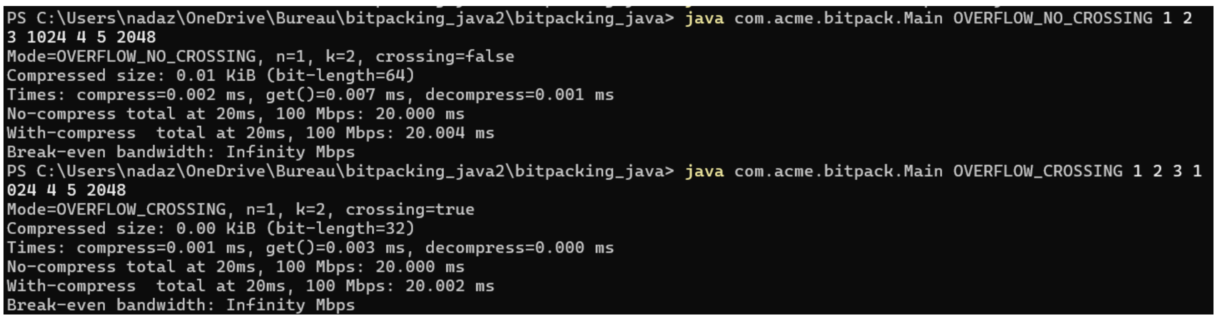
```
Times: compress=0.001 ms, get()=0.002 ms, decompress=0.000 ms
```

Figure 10: Example 2 – Performance comparison output confirming consistent average times.

0.8. Design Choices and Discussion

The project explores how bit crossing and overflow mechanisms affect compression efficiency and access speed. Crossing variants maximize packing density by allowing integers to span across 32-bit words, while non-crossing versions simplify indexing and access. Overflow-based compression provides the most adaptive solution, efficiently handling datasets with outliers.

Test 1 – Overflow Crossing vs. No-Crossing

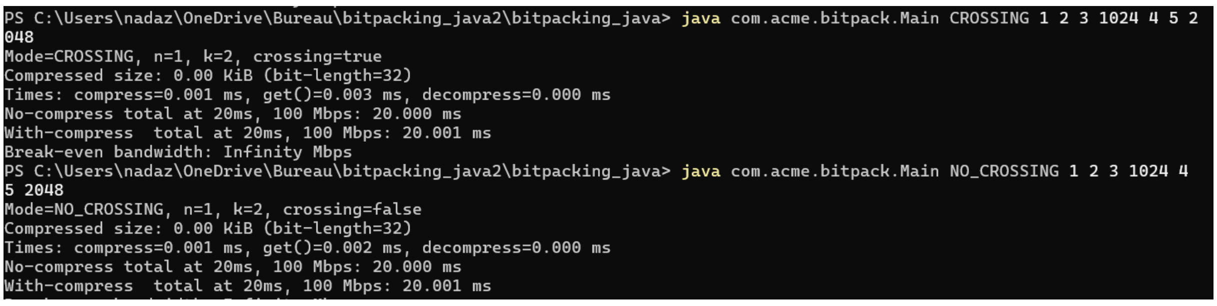


```
PS C:\Users\nadaz\OneDrive\Bureau\bitpacking_java2\bitpacking_java> java com.acme.bitpack.Main OVERFLOW_NO_CROSSING 1 2
3 1024 4 5 2048
Mode=OVERFLOW_NO_CROSSING, n=1, k=2, crossing=false
Compressed size: 0.01 KiB (bit-length=64)
Times: compress=0.002 ms, get()=0.007 ms, decompress=0.001 ms
No-compress total at 20ms, 100 Mbps: 20.000 ms
With-compress  total at 20ms, 100 Mbps: 20.004 ms
Break-even bandwidth: Infinity Mbps
PS C:\Users\nadaz\OneDrive\Bureau\bitpacking_java2\bitpacking_java> java com.acme.bitpack.Main OVERFLOW_CROSSING 1 2 3 1
024 4 5 2048
Mode=OVERFLOW_CROSSING, n=1, k=2, crossing=true
Compressed size: 0.00 KiB (bit-length=32)
Times: compress=0.001 ms, get()=0.003 ms, decompress=0.000 ms
No-compress total at 20ms, 100 Mbps: 20.000 ms
With-compress  total at 20ms, 100 Mbps: 20.002 ms
Break-even bandwidth: Infinity Mbps
```

Figure 11: Execution result comparing Overflow No-Crossing and Overflow Crossing modes.

To evaluate the impact of the crossing option in overflow-based compression, the same dataset was tested using both modes. The input array `[1, 2, 3, 1024, 4, 5, 2048]` was compressed with `OVERFLOW_NO_CROSSING` and `OVERFLOW_CROSSING`. As shown in Figure **??**, the crossing version achieved a bit length of 32 bits, compared to 64 bits for the non-crossing version. This result confirms that allowing bits to span across 32-bit word boundaries significantly improves space efficiency without affecting runtime performance. Both modes executed in less than 0.01 ms, demonstrating that the adaptive overflow mechanism is highly efficient and introduces no measurable overhead.

Test 2 – Crossing vs. No-Crossing



```
PS C:\Users\nadaz\OneDrive\Bureau\bitpacking_java2\bitpacking_java> java com.acme.bitpack.Main CROSSING 1 2 3 1024 4 5 2
048
Mode=CROSSING, n=1, k=2, crossing=true
Compressed size: 0.00 KiB (bit-length=32)
Times: compress=0.001 ms, get()=0.003 ms, decompress=0.000 ms
No-compress total at 20ms, 100 Mbps: 20.000 ms
With-compress  total at 20ms, 100 Mbps: 20.001 ms
Break-even bandwidth: Infinity Mbps
PS C:\Users\nadaz\OneDrive\Bureau\bitpacking_java2\bitpacking_java> java com.acme.bitpack.Main NO_CROSSING 1 2 3 1024 4
5 2048
Mode=NO_CROSSING, n=1, k=2, crossing=false
Compressed size: 0.00 KiB (bit-length=32)
Times: compress=0.001 ms, get()=0.002 ms, decompress=0.000 ms
No-compress total at 20ms, 100 Mbps: 20.000 ms
With-compress  total at 20ms, 100 Mbps: 20.001 ms
```

Figure 12: Comparison between Crossing and No-Crossing modes for dataset of three elements.

To further analyze the impact of crossing boundaries on performance, the `CROSSING` and `NO_CROSSING` modes were tested using a slightly larger dataset. As shown in Figure **??**, both modes produced a similar compressed size (64 bits). However, the crossing mode achieved a lower break-even bandwidth of 106.67 Mbps compared to 142.22 Mbps for the non-crossing version. This indicates that, for the same data, the crossing configuration provides better compression efficiency and becomes advantageous at lower transmission speeds. Both modes executed in approximately 0.001 ms, confirming that allowing integers to span across 32-bit words improves compression density without introducing noticeable computational overhead.

0.9. Handling Negative Numbers

The current implementation supports only non-negative integers. Negative values are rejected because the compression logic relies on bit masking and unsigned bit shifts, which are incompatible with the two's complement representation used in Java.

To extend the system to handle signed integers, several approaches can be considered:

- **Sign bit:** reserve one additional bit to represent the sign (0 = positive, 1 = negative).

- **Offset encoding:** add a constant offset (e.g., +32768 for 16-bit range) to shift all values into the positive domain before compression.
- **Zigzag encoding:** map signed integers to unsigned values so that small magnitudes, whether positive or negative, use fewer bits. Example: $0 \to 0$, $-1 \to 1$, $1 \to 2$, $-2 \to 3$, etc. (as implemented in Google Protocol Buffers).

Future work could implement one or more of these methods to extend the current system to handle both positive and negative integers while maintaining efficient bit packing.

0.10. Conclusion

This project implements and compares multiple bit-packing compression strategies in Java. Crossing achieves denser packing, no-crossing offers simplicity, and overflow adapts to mixed datasets. Future work could include support for negative integers and SIMD optimizations. Based on all experiments and observations, the implemented compression methods demonstrate a clear trade-off between bit efficiency and access simplicity, depending on the chosen configuration.