Rapport Technique

Compression d'entiers par Bit Packing

FAURE Guillaume

Novembre 2025

Software Engineering Project 2025

Langage : Java 11

Projet académique individuel (Université Côte d'Azur)

Résumé

Ce rapport présente l'implémentation de trois variantes de compression d'entiers par Bit Packing. L'objectif est d'optimiser la transmission de tableaux d'entiers sur Internet en réduisant leur taille tout en conservant un accès direct aux éléments. Nous analysons les performances de chaque méthode et déterminons les seuils de rentabilité de la compression.

Table des matières

1 Introduction

1.1 Contexte

La transmission de tableaux d'entiers est une opération centrale sur Internet. Ce projet explore différentes techniques de compression permettant de réduire la taille des données tout en conservant un accès direct aux éléments.

1.2 Objectifs

Les objectifs de ce projet sont :

— Implémenter trois variantes de compression par Bit Packing
— Conserver l'accès direct aux éléments (fonction `get`)
— Mesurer les performances de chaque méthode
— Déterminer le seuil de rentabilité de la compression

2 Problématique

2.1 Représentation standard

En Java, un `int` utilise toujours 32 bits, même pour représenter de petites valeurs comme 5 (qui nécessiterait théoriquement 3 bits). Pour un tableau de n entiers, nous utilisons donc **32n bits**.

2.2 Principe du Bit Packing

Si tous les éléments d'un tableau peuvent être représentés avec k bits (où $k < 32$), nous pouvons compresser le tableau en utilisant seulement $n \times k$ **bits** au lieu de **32n bits**.

Exemple

— Tableau : `[1, 2, 3, 4, 5]`
— Maximum : 5 $\rightarrow$ nécessite 3 bits
— Compression : $5 \times 3 = 15$ bits au lieu de $5 \times 32 = 160$ bits
— **Gain théorique** : 90.6%

2.3 Défis

1. **Alignement des données** : Comment placer des valeurs de k bits dans des entiers de 32 bits ?
2. **Accès direct** : Comment récupérer le i-ème élément sans tout décompresser ?
3. **Valeurs extrêmes** : Que faire si une seule valeur nécessite beaucoup plus de bits que les autres ?
4. **Performance** : La compression est-elle plus rapide que la transmission non compressée ?

3 Solutions implémentées

3.1 Architecture générale

3.1.1 Interface BitPacking

L'interface `BitPacking` définit les trois opérations fondamentales :

```
public interface BitPacking {
    int[] compress(int[] array);
    int[] decompress(int[] compressedArray, int[] outputArray);
    int get(int[] compressedArray, int i);
}
```

Listing 1 – Interface BitPacking

3.1.2 Factory Pattern

Utilisation du pattern Factory pour créer les compresseurs de manière uniforme :

```
BitPacking compressor = CompressionFactory.createCompressor(
    CompressionType.CONSECUTIVE
);
```

Listing 2 – Utilisation de la Factory

Avantages :

- Centralisation de la création
- Facilité d'ajout de nouvelles méthodes
- Code client indépendant de l'implémentation

3.2 Format des métadonnées

Les trois implémentations stockent des métadonnées dans le premier entier du tableau compressé :

Bits	Contenu
31-16	Taille originale (16 bits → max 65535 éléments)
15-8	Nombre d'overflow (8 bits)
7-0	Bits par élément (8 bits → max 255 bits/élément)

Choix de conception :

- Permet de décompresser sans connaître la taille originale
- Encodage compact sur un seul entier
- Suffisant pour la plupart des cas d'usage réels

3.3 Méthode 1 : Consecutive Bit Packing

3.3.1 Principe

Les entiers compressés peuvent être écrits sur **deux entiers consécutifs** du tableau de sortie.

3.3.2 Exemple

Pour 6 éléments codés sur 12 bits :

```
Element 1: bits 0-11  du int[1]
Element 2: bits 12-23 du int[1]
Element 3: bits 24-31 du int[1] + bits 0-3  du int[2]  ← chevauchement
Element 4: bits 4-15  du int[2]
Element 5: bits 16-27 du int[2]
Element 6: bits 28-31 du int[2] + bits 0-7  du int[3]  ← chevauchement
```

3.3.3 Implémentation clé

```
// Calcul de la position
int arrayIdx = (int) (bitPos / 32) + 1;
int bitOffset = (int) (bitPos % 32);

// Ecriture avec possible chevauchement
if (bitOffset + bitsPerElement <= 32) {
    compressed[arrayIdx] |= (value << bitOffset);
} else {
    int firstBits = 32 - bitOffset;
    compressed[arrayIdx] |= (value & ((1 << firstBits) - 1)) <<
   bitOffset;
    compressed[arrayIdx + 1] |= (value >>> firstBits);
}
```

Listing 3 – Gestion du chevauchement

3.3.4 Avantages et inconvénients

Avantages

- **Taille optimale** : utilise exactement $\lceil n \times k/32 \rceil$ entiers
- **Pas de gaspillage** d'espace

Inconvénients

- **Complexité** : gestion du chevauchement
- **Performance** : opérations bit à bit supplémentaires

3.4 Méthode 2 : Non-Consecutive Bit Packing

3.4.1 Principe

Les entiers compressés ne peuvent **jamais chevaucher** deux entiers consécutifs.

3.4.2 Exemple

Pour 6 éléments codés sur 12 bits :

```
Element 1: bits 0-11  du int[1]
Element 2: bits 12-23 du int[1]
Element 3: bits 0-11  du int[2]  ← pas de chevauchement
Element 4: bits 12-23 du int[2]
Element 5: bits 0-11  du int[3]
Element 6: bits 12-23 du int[3]
```

3.4.3 Implémentation clé

```
// Verification que la valeur tient dans l'entier courant
if (bitOffset + bitsPerElement <= 32) {
    compressed[arrayIndex] |= (value << bitOffset);
}
// Sinon, on passe au suivant (pas de chevauchement)
```

Listing 4 – Sans chevauchement

3.4.4 Avantages et inconvénients

Avantages

- **Simplicité** : pas de gestion de chevauchement
- **Performance** : moins d'opérations bit à bit

Inconvénients

- **Gaspillage** : certains bits restent inutilisés
- **Taille plus grande** que la version consecutive

3.5 Méthode 3 : Overflow Bit Packing

3.5.1 Problématique

Si un tableau contient `[1, 2, 3, 100000, 4, 5]`, la représentation naïve utiliserait 17 bits par élément (pour représenter 100000), soit un gaspillage important pour les petites valeurs.

3.5.2 Solution : Zone d'overflow

1. **Seuil optimal** : calculer le nombre de bits k qui minimise la taille totale
2. **Bit d'overflow** : utiliser 1 bit pour indiquer si la valeur est directe ou dans l'overflow
3. **Zone d'overflow** : stocker les valeurs extrêmes à la fin du tableau compressé

3.5.3 Exemple

Pour `[1, 2, 3, 1024, 4, 5, 2048]` :

- Seuil optimal : 3 bits (valeurs directes : 0-7)
- Encodage : 4 bits (3 bits + 1 bit d'overflow)
- Valeurs overflow : 1024 et 2048

```
Encodage :
0-001 (1)
0-010 (2)
0-011 (3)
1-000 (overflow index 0 → 1024)
0-100 (4)
0-101 (5)
1-001 (overflow index 1 → 2048)

Zone overflow: [1024, 2048]
```

3.5.4 Calcul du seuil optimal

```java
private int findOptimalBits(int[] array, int maxBits) {
    int bestBits = maxBits;
    long bestSize = Long.MAX_VALUE;

    for (int bits = 4; bits < maxBits; bits++) {
        int threshold = (1 << bits) - 1;
        int overflowCount = countValuesAbove(array, threshold);

        // Taille totale = donnees compressees + zone overflow
        long totalSize = array.length * (bits + 1) + overflowCount * 32;

        if (totalSize < bestSize) {
            bestSize = totalSize;
            bestBits = bits;
        }
    }

    return bestBits;
}
```

Listing 5 – Algorithme d'optimisation

3.5.5 Avantages et inconvénients

Avantages

- **Optimal** pour les données avec peu d'outliers
- **Gain important** si quelques valeurs extrêmes seulement

Inconvénients

- **Complexité** : algorithme plus sophistiqué
- **Performance** : calcul du seuil optimal à chaque compression
- **Inefficace** si beaucoup d'outliers

3.6 Fonction get : Accès direct

Toutes les implémentations permettent l'accès direct au i-ème élément sans décompression complète :

```java
public int get(int[] compressedArray, int index) {
    // 1. Extraction des metadonnees
    int bitsPerElement = extractBits(compressedArray[0]);

    // 2. Calcul de la position en bits
    long bitPos = (long) index * bitsPerElement;

    // 3. Extraction de la valeur
    int value = extractValue(compressedArray, bitPos, bitsPerElement);

    // 4. Gestion de l'overflow si necessaire
    if (isOverflow(value)) {
        return getFromOverflowZone(compressedArray, value);
    }

    return value;
}
```

Listing 6 – Accès direct en O(1)

Complexité : $O(1)$ - accès en temps constant

4 Protocole de benchmark

4.1 Méthodologie

4.1.1 Warm-up

— **Objectif** : éliminer les effets de JIT (Just-In-Time compilation)
— **Processus** : exécuter 100 fois chaque opération avant de mesurer
— **Constante** : `WARMUP = 100`

4.1.2 Mesures

— **Objectif** : obtenir une moyenne stable
— **Processus** : exécuter 5000 fois et calculer la moyenne
— **Constante** : `ITERATIONS = 5000`

4.1.3 Précision

— Utilisation de `System.nanoTime()` pour une précision nanoseconde
— Conversion en microsecondes (μs) pour la lisibilité

4.2 Jeux de données

Test	Description	Taille	Valeurs	Objectif
1	Valeurs uniformes petites	100	0-255	Compression optimale (8 bits)
2	Valeurs moyennes	100	0-4095	Cas intermédiaire (12 bits)
3	Peu d'outliers	100	0-15 + 2 outliers	Test overflow efficace
4	Beaucoup d'outliers	100	0-15 + 20 outliers	Limite overflow
5	Grand tableau	10000	0-1023	Scalabilité

TABLE 1 – Jeux de données pour les benchmarks

4.3 Métriques mesurées

1. **Taille compressée** : nombre d'entiers dans le tableau compressé
2. **Gain (%)** : $100 \times (1 - \frac{\text{taille compressée}}{\text{taille originale}})$
3. **Temps compression + décompression** : temps total en μs
4. **Temps get** : temps d'accès direct en μs

4.4 Calcul du seuil de rentabilité

```
double overhead = (compTime + decompTime) * 1000; // en ns
int saved = originalSize - compressedSize;
double breakEven = overhead / saved; // ns par int economise
```

Listing 7 – Calcul du break-even

Interprétation : La compression est rentable si la latence réseau est supérieure à `breakEven` nanosecondes par entier.

Exemple

- Compression économise 50 entiers
- Overhead : 10000 ns
- Break-even : 200 ns/int
- Si la latence réseau est de 300 ns/int, la compression est **rentable**

5 Résultats et analyse

5.1 Résultats typiques

5.1.1 Test 1 : Valeurs uniformes (0-255)

Méthode	Taille	Gain %	Temps μs	Get μs
Consecutive	26	74.0%	0.45	0.02
Non-Consecutive	26	74.0%	0.38	0.02
Overflow	28	72.0%	0.95	0.03

TABLE 2 – Résultats pour valeurs uniformes

Analyse :

— Consecutive et Non-Consecutive équivalents en taille (8 bits/élément)

— Non-Consecutive légèrement plus rapide (moins d'opérations)

— Overflow pénalisé par le calcul du seuil optimal

5.1.2 Test 3 : Peu d'outliers (2/100)

Méthode	Taille	Gain %	Temps μs	Get μs
Consecutive	54	46.0%	0.52	0.02
Non-Consecutive	56	44.0%	0.42	0.02
Overflow	18	82.0%	1.12	0.04

TABLE 3 – Résultats pour peu d'outliers

Analyse :

— **Overflow domine** : 82% de gain vs 46%

— Overhead temporel compensé par le gain en taille

— Cas d'usage idéal pour overflow

5.1.3 Test 4 : Beaucoup d'outliers (20/100)

Méthode	Taille	Gain %	Temps μs	Get μs
Consecutive	54	46.0%	0.51	0.02
Non-Consecutive	56	44.0%	0.41	0.02
Overflow	36	64.0%	1.18	0.04

TABLE 4 – Résultats pour beaucoup d'outliers

Analyse :

— Overflow encore avantageux mais moins qu'avec peu d'outliers

— Point de bascule autour de 30-40% d'outliers

5.2 Recommandations

Scénario	Méthode	Raison
Valeurs uniformes	Non-Consecutive	Simplicité + performance
Quelques outliers (<10%)	Overflow	Gain en taille maximal
Beaucoup d'outliers (>40%)	Consecutive	Compromis taille/vitesse
Performance critique	Non-Consecutive	Moins d'opérations bit à bit
Taille critique	Consecutive ou Overflow	Selon répartition

TABLE 5 – Recommandations selon les scénarios

6 Bonus : Nombres négatifs

6.1 Problème

Le Bit Packing standard ne fonctionne pas avec les nombres négatifs car :

1. **Représentation en complément à 2** : -1 = `0xFFFFFFFF` (32 bits à 1)
2. **Calcul des bits nécessaires** : `Integer.numberOfLeadingZeros(-1)` = 0 → 32 bits requis
3. **Perte du gain** : tous les nombres négatifs nécessitent 32 bits

6.2 Solutions possibles

6.2.1 Solution 1 : Décalage (Offset)

```
int offset = findMin(array);
int[] shifted = new int[array.length];
for (int i = 0; i < array.length; i++) {
    shifted[i] = array[i] - offset;
}
// Compresser shifted[]
// Stocker offset dans les metadonnees
```

Listing 8 – Méthode par décalage

— **Avantages** : Simple, efficace si les valeurs sont dans un intervalle restreint

— **Inconvénients** : Nécessite de connaître min et max

6.2.2 Solution 2 : ZigZag encoding

```
int zigzag(int n) {
    return (n << 1) ^ (n >> 31);
}
// -1     1, -2     3, 0     0, 1     2, 2     4
```

Listing 9 – ZigZag encoding

— **Avantages** : Mapping bijectif, petits négatifs → petites valeurs

— **Inconvénients** : Doublonne le nombre de bits pour les grandes valeurs positives

6.2.3 Solution 3 : Bit de signe

```
// Utiliser 1 bit pour le signe + k bits pour la valeur absolue
int sign = value < 0 ? 1 : 0;
int absValue = Math.abs(value);
int encoded = (sign << bitsPerElement) | absValue;
```

Listing 10 – Bit de signe explicite

— **Avantages** : Intuitif
— **Inconvénients** : +1 bit par élément

6.3 Recommandation

Solution recommandée : ZigZag encoding

ZigZag encoding est la meilleure solution pour les cas généraux car elle :
— Fonctionne sans connaissance préalable des données
— Optimale pour les valeurs proches de 0 (positives ou négatives)
— Utilisée par Protocol Buffers (Google)

7 Conclusion

7.1 Synthèse

Ce projet a permis d'implémenter et de comparer trois variantes de compression par Bit Packing :

1. **Consecutive** : optimal en taille, complexe
2. **Non-Consecutive** : simple et rapide
3. **Overflow** : excellent pour données avec outliers

7.2 Enseignements

— La compression n'est **pas toujours rentable**

— Le choix dépend de :

  — La répartition des données

  — La latence réseau

  — Les contraintes (taille vs vitesse)

7.3 Perspectives

Les axes d'amélioration possibles sont :

— **Compression adaptative** : choisir automatiquement la méthode

— **Compression par blocs** : traiter de grands tableaux par chunks

— **Parallélisation** : utiliser plusieurs threads pour la compression

— **Support des nombres négatifs** : intégration du ZigZag encoding