Année universitaire 2025-2026

Rapport de projet
Software Engineering 2025 : Data compressing for speed up transmission

Présenté par : SANCHEZ Louise

Software Engineering - M.Regin

Master 1 Informatique - Université Côte d'Azur

Résumé

Ce projet s'inscrit dans le cadre du cours de Software Engineering du Master 1 Informatique à l'Université Côte d'Azur. Il porte sur la conception et l'évaluation de plusieurs méthodes de compression d'un tableau d'entiers afin d'en optimiser la taille et la vitesse de transmission. Trois approches ont été développées : Aligned, Overlapped et Overflow. La méthode Aligned offre une implémentation simple et rapide, tandis que Overlapped améliore le taux de compression en exploitant mieux les bits disponibles. Enfin, Overflow gère efficacement les données hétérogènes grâce à une table de débordement. Les résultats des tests montrent que chaque méthode présente des avantages spécifiques selon le type de données et les contraintes de performance. Cette étude met ainsi en évidence les compromis entre complexité, rapidité et efficacité de compression dans le développement d'outils logiciels dédiés à la manipulation de grands volumes de données numériques.

Abstract

This project was carried out as part of the Software Engineering course in the Master's program in Computer Science at Université Côte d'Azur. It focuses on the design and evaluation of several integer array compression methods to optimize data size and transmission speed. Three different approaches were implemented : Aligned, Overlapped, and Overflow. The Aligned method provides a simple and fast implementation, while Overlapped improves compression efficiency by fully utilizing available bits. The Overflow method, on the other hand, handles heterogeneous data effectively using an overflow table. Experimental results demonstrate that each technique offers specific advantages depending on data type and performance requirements. The study highlights the trade-offs between complexity, speed, and compression efficiency in the design of software tools for handling large-scale digital data.

Table des matières

1 Introduction

La compression de données est un domaine essentiel en informatique, visant à réduire la taille des données pour optimiser leur stockage et leur transmission. La problématique de ce projet est de comprendre comment compresser un tableau d'entiers tout en gardant un accès aux éléments des tableau compressés. Ce projet s'inscrit dans le cadre du cours de Software Engineering dispensé en M1 à l'Université Nice Côte d'Azur. Plus précisément, les objectifs de ce projet sont de concevoir, implémenter et comparer plusieurs méthodes de compression d'un tableau d'entiers afin d'en évaluer l'efficacité et les performances. L'enjeu est double : d'une part, réduire la taille mémoire occupée par les données sans perte d'information, et d'autre part, permettre un accès direct et rapide à chaque élément compressé sans nécessiter une décompression complète du tableau. Pour

cela, trois approches distinctes ont été développées : les méthodes Aligned, Overlapped et Overflow, chacune reposant sur des principes différents d'organisation et de gestion des bits. Ces méthodes ont ensuite été évaluées à l'aide de benchmarks portant sur le taux de compression, le temps d'exécution et la rapidité d'accès aléatoire. Ce rapport

présente successivement l'architecture logicielle mise en œuvre, les principes théoriques et les détails d'implémentation des différentes méthodes, puis une analyse comparative des performances obtenues. Enfin, une synthèse et un bilan permettront de tirer les conclusions principales et d'envisager des perspectives d'amélioration.

2 Architecture logicielle

Dans cette première partie, nous allons présenter et analyser l'architecture logicielle mise en place dans le cadre du projet de compression de données.

2.1 Vue d'ensemble de l'architecture

J'ai fait le choix d'adopter une architecture en couches. Ce modèle architectural repose sur une idée simple mais fondamentale : diviser le système en plusieurs niveaux distincts, chacun ayant une responsabilité spécifique. Dans mon architecture, j'ai deux couches, la couche de présentation et le domaine. Cette approche est peut être exagérée pour un projet de cette taille mais je trouve que séparer la logique métier de la logique applicative est essentiel au maintient d'une structure claire, évolutive et d'un code propre.

La logique métier correspond à l'ensemble des règles et des opérations qui traduisent le cœur fonctionnel du projet, c'est-à-dire tout ce qui concerne les algorithmes de compression, les manipulations binaires, les calculs de bits ou encore la gestion des structures de données. La logique applicative, quant à elle, regroupe tout ce qui concerne la coordination des opérations : la préparation des données, la sélection des méthodes de compression, la mesure des performances, l'affichage ou l'enregistrement des résultats.

Un autre choix structurant de mon architecture a été de ne pas manipuler directement les tableaux d'entiers, mais de les encapsuler dans des objets spécifiques : UnpackedData et PackedData. L'idée est d'éviter d'utiliser les types primitifs comme des conteneurs de données "bruts", ce qui conduit souvent à un code moins lisible et plus difficile à maintenir. Au lieu de cela, ces classes jouent le rôle d'abstractions qui donnent du sens aux données. UnpackedData représente les données non compressées, telles qu'elles existent en mémoire avant traitement, tandis que PackedData représente les données compressées prêtes à être stockées ou transmises. Ces classes contiennent non seulement les tableaux d'entiers eux-mêmes, mais aussi des métadonnées importantes, comme la taille originale, la taille compressée, le temps de compression ou encore le nombre de bits utilisés par valeur.

Ce choix de conception permet d'introduire directement les règles métier dans les classes elles-mêmes, plutôt que de les disperser dans différentes parties du code. Cela rend l'ensemble plus cohérent et réduit les risques d'erreurs. En typant explicitement ces objets, j'ai également voulu éviter ce qu'on appelle la Primitive Obsession, un anti-pattern fréquent dans les projets orientés objet. La Primitive Obsession se produit lorsqu'on utilise des types primitifs (comme int, float, ou String) pour représenter des concepts qui devraient être des entités métiers à part entière. Par exemple, manipuler un simple tableau d'entiers pour représenter des données compressées revient à ignorer tout le contexte métier associé. En encapsulant ces données dans des objets dédiés, on gagne en expressivité, en robustesse et en maintenabilité.

Listing 1 – Listing 1 : Classe PackedData

```
public final class PackedData {
    private int[] data;
    private int originalSize;
    private int compressedSize;
```

```
    private long compressionTime;
    private int bitsPerValue;

    private PackedData(int[] array) {
        this.data = Arrays.copyOf(array, array.length);
        this.originalSize = array.length;
        this.compressedSize = array.length;
        this.compressionTime = 0;
        this.bitsPerValue = 32;
    }

    public static PackedData from(int[] array) {
        if (array == null || array.length == 0) {
            throw new IllegalArgumentException("array must not be
            null or empty");
        }
        return new PackedData(array);
    }
}
```

Comme on peut le voir dans le listing 1, la classe PackedData encapsule un tableau d'entiers et fournit une méthode statique from pour créer une instance de PackedData à partir d'un tableau d'entiers. Cette méthode permet de valider l'entrée et d'assurer que le tableau n'est pas null ou vide avant de créer l'objet PackedData.

2.2 Patterns & bonnes pratiques de conception

Pour structurer davantage le code, j'ai intégré plusieurs design patterns reconnus, qui favorisent la réutilisabilité et la modularité du projet. Parmi eux, le pattern Factory occupe une place centrale. Il s'agit d'un pattern de création permet d'instancier des objets sans que le code client ait besoin de connaître leur classe concrète. Dans le cadre de ce projet, la Factory a pour rôle de gérer la création des différentes implémentations de méthodes de compression. Ainsi, la logique de création est clairement séparée de la logique métier, conformément au principe de responsabilité unique (Single Responsibility Principle). La Factory nous permet aussi de respecter le principe de ouverture/fermeture (Open/Closed Principle) en facilitant l'ajout de nouvelles implémentations de méthodes de compression. En fin avec la factory on respecte aussi le principe d'inversion de dépendance (Dependency Inversion Principle) car le code client dépend d'abstractions et non pas de classes concrètes.

J'ai également combiné la Factory avec un pattern Registry. Le Registry permet de centraliser la gestion des instances créées par la Factory, ce qui évite de dupliquer du code et facilite la maintenance. Concrètement, la Factory s'occupe de l'instanciation, tandis que le Registry gère la liste des implémentations disponibles et fournit un point d'accès unique pour les récupérer. Utiliser le registry avec la Factory m'a permis de charger dynamiquement les méthodes de compression, de les comparer facilement, et d'ajouter de nouvelles variantes sans altérer le fonctionnement global du programme. Cette méthode permet de respecter le principe open/closed. Voici une représentation schématique de mon implémentation de la Factory et du Registry.

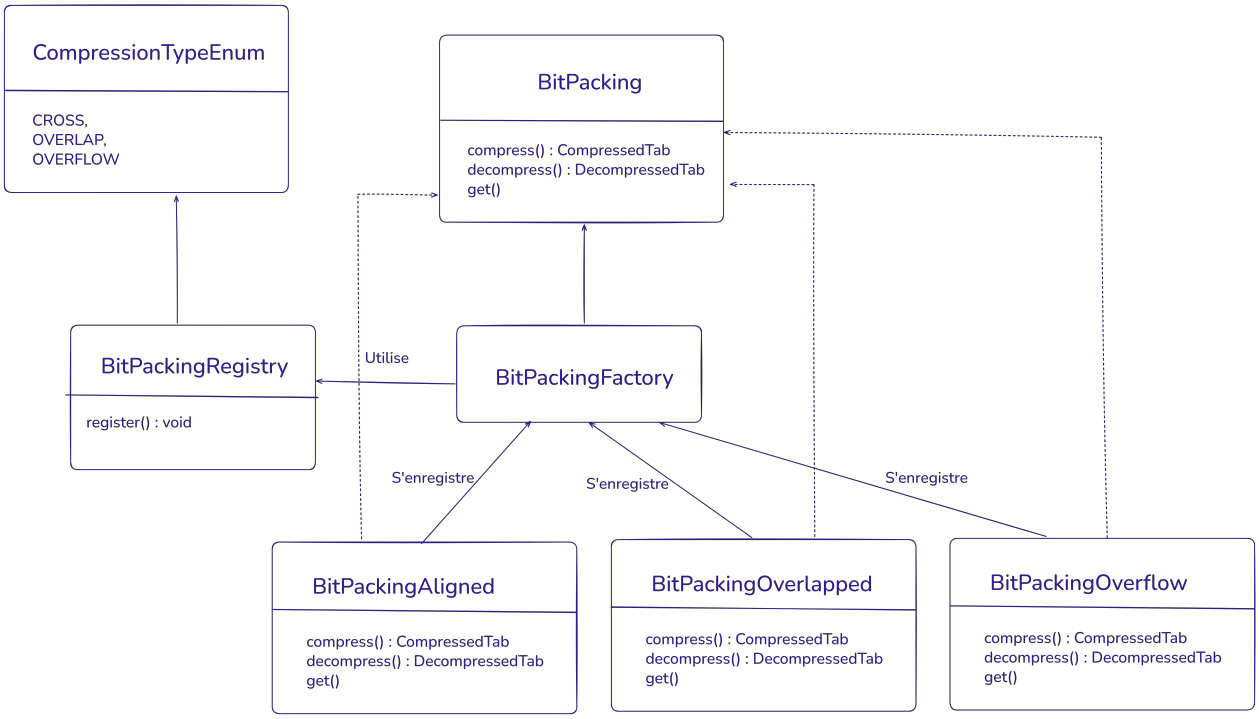

FIGURE 1 – Représentation schématique de la Factory et du Register

3 Méthodes de compression

Dans cette seconde partie on va étudier les différentes méthodes de compression implémentées dans notre projet.

3.1 Représentation UTF-32

Avant d'aborder les méthodes de compression, il est important de comprendre la représentation standard des entiers en mémoire. En Java, un entier est représenté sur 32 bits (4 octets) en utilisant le format UTF-32 (ou plus précisément, le type `int` qui est un entier signé sur 32 bits).

Par exemple, le nombre 5 est représenté en mémoire comme :

```
00000000 00000000 00000000 00000101
```

On constate que pour représenter de petits nombres, la majorité des bits sont à zéro. Le principe du bit packing consiste à exploiter cette inefficacité en ne conservant que les bits significatifs de chaque nombre.

Si nous avons un tableau d'entiers où la valeur maximale nécessite k bits, alors tous les nombres du tableau peuvent être représentés avec k bits au lieu de 32. Pour un tableau de n entiers, nous passons donc de $32n$ bits à nk bits, soit un taux de compression de $\frac{k}{32}$.

3.2 En-tête des méthodes

Toutes les méthodes de compression implémentées dans ce projet utilisent un en-tête commun pour stocker des données essentielles. Cet en-tête est crucial pour la décompression et l'accès direct aux éléments du tableau compressé. Concrètement, l'en-tête occupe les premiers 32 bits du tableau compressé et contient deux informations clés :

— La taille du tableau original (non compressé) : Cette information est nécessaire pour savoir combien d'éléments doivent être extraits lors de la décompression.

— La taille maximale des entiers dans le tableau : Cela permet de déterminer le nombre de bits nécessaires pour représenter chaque entier dans le tableau compressé.

3.3 Méthode "Aligned" (alignée)

La première méthode que j'ai implémenté dans ce projet est la méthode "Aligned" (alignée). Cette méthode est la plus simple des trois car elle consiste simplement à diviser le tableau d'entiers en blocs de taille fixe sans chevauchement ni débordement.

3.3.1 Principes théortiques

Premièrement on va essayer de comprendre les principes théoriques de cette méthode. Pour cela on va prendre pour exemple un tableau d'entiers.

```
Tab = [5, 7, 12, 1023, 511, 3];
```

On va recupérer la valeur maximale du tableau pour déterminer le nombre de bits nécessaires pour représenter chaque entier. Dans notre exemple, la valeur maximale est 1023, qui peut être représentée en 10 bits (car $2^{10} = 1024$). Ainsi, chaque entier du tableau sera encodé sur 10 bits. On va ensuite faire rentrer le plus d'entiers codé 10 bits posible dans un bloc de 32 bits.

Mot 0

Valeur	Bits utilisés	Position du mot dans le bloc
5	bit 0 → 9	mot 0
7	bit 10 → 19	mot 0
12	bit 20 → 29	mot 0
1023	bit 30 → 39 → dépasse 32 bits	transition mot 0 → mot 1

1023 ne rentre plus dans le mot 0 car il dépasse les 32 bits. On va donc le mettre dans le mot 1.

Mot 1

Valeur	Bits utilisés	Position du mot dans le bloc
1023	bit 0 → 9	mot 1
511	bit 10 → 19	mot 1
3	bit 20 → 29	mot 1

On obtient donc deux mots de 32 bits qui contiennent l'ensemble des entiers du tableau initial.

Mot	Contenu	Bits utilisés
0	[5, 7, 12]	bits 0-29
1	[1023, 511, 3]	bits 0-29

Chaque bloc de 32 bits contient donc 3 entiers codés sur 10 bits chacun, un total de 30 bits utilisés par bloc. Les 2 bits inutilisés vont être placé à la fin du mot et vont rester à zéro.

Listing 2 – Encodage binaire du mot 0

```
0000000101 0000000111 0000001100 00
```

Listing 3 – Encodage binaire du mot 1

```
1111111111 0111111111 0000000011 00
```

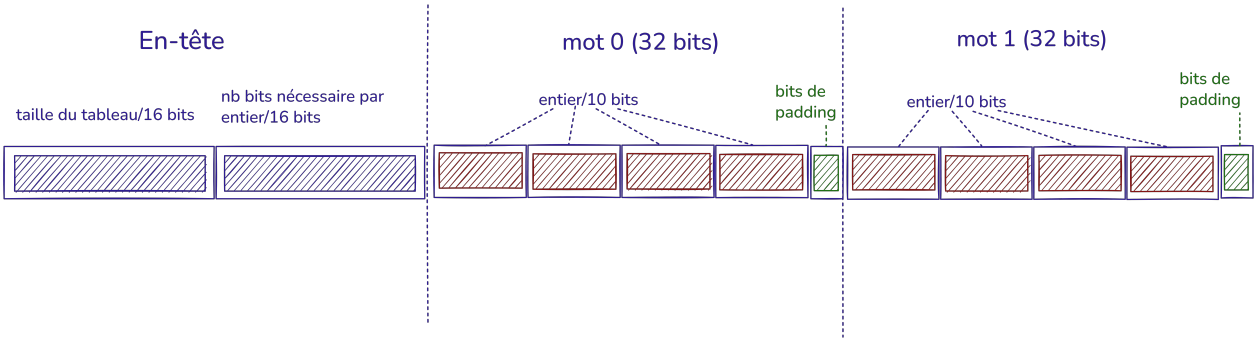

FIGURE 2 – Représentation schématique d'un tableau compressé avec la méthode "Aligned"

3.3.2 Implémentation et fonctionnement

L'implémentation de cette méthode est relativement simple. On commence par déterminer la taille maximale des entiers dans le tableau. Ensuite, on calcule le nombre de bits nécessaires pour représenter cette taille maximale. On divise ensuite le tableau en blocs de 32 bits, en insérant les entiers codés dans chaque bloc jusqu'à ce que le bloc soit plein. Lorsque le bloc est plein ou qu'un entier ne peut pas y être inséré, on passe au bloc suivant.

Algorithm 1: Compression Aligned

Input: UnpackedData fromUnpackedData
Output: PackedData toPackedData
for *each value in input array* **do**
  if $bitPosition + bitsPerValue > 32$ **then**
    Store $currentWord$ in output;
    $currentWord \leftarrow 0$;
    $bitPosition \leftarrow 0$;
    $outputIndex \leftarrow outputIndex + 1$;
  end
  $currentWord \leftarrow currentWord \mid (value \ll bitPosition)$;
  $bitPosition \leftarrow bitPosition + bitsPerValue$;
end
Store final $currentWord$ in output;

Comme on peut le voir dans l'algorithme 4, le processus de compression se déroule en plusieurs étapes bien définies :

1. On calcule la valeur maximale du tableau pour calculer le nombre de bits nécessaires (k). Par exemple, si max = 1023, alors $k = 10$ bits car $2^{10} = 1024$.
2. On calcule combien de valeurs peuvent tenir dans un mot de 32 bits. Avec $k = 10$ bits, on obtient $\lfloor 32/10 \rfloor = 3$ valeurs par mot, ce qui laisse 2 bits inutilisés par mot.
3. On alloue le tableau de sortie avec exactement le nombre de mots nécessaires : $\lceil n/valuesPerWord \rceil$ où n est le nombre d'éléments.
4. Pour chaque valeur du tableau d'entrée, on vérifie si elle peut tenir dans le mot courant. Si $bitPosition + k > 32$, cela signifie que la valeur déborderait, donc on sauvegarde le mot courant et on passe au suivant. Sinon, on insère la valeur à la position courante en utilisant un OU binaire : `currentWord |= (value « bitPosition)`.

La décompression effectue le processus inverse de manière symétrique :

1. On lit le nombre de bits par valeur (k) depuis les métadonnées du PackedData.
2. On construit un masque binaire avec k bits à 1. Par exemple, pour $k = 10$: `mask = (1 « 10) - 1 = 0x3FF = 0011111111`.
3. Pour chaque position dans le tableau de sortie, on vérifie si on doit changer de mot. L'extraction d'une valeur se fait en deux opérations : décalage à droite de $bitPosition$ bits (`»`), puis application du masque (`&`) pour ne garder que les k bits de poids faible.
4. Chaque valeur extraite est placée dans le tableau de sortie à sa position originale.

Pour l'accès direct à une valeur spécifique dans le tableau compressé on procède ainsi :

1. On détermine dans quel mot de 32 bits se trouve la valeur recherchée : `wordIndex = index / valuesPerWord`. Par exemple, avec 3 valeurs par mot, l'élément d'index 7 est dans le mot $\lfloor 7/3 \rfloor = 2$.
2. On calcule la position en bits dans ce mot : `bitOffset = (index % valuesPerWord) * k`. Pour l'index 7 : $(7 \bmod 3) \times 10 = 1 \times 10 = 10$ bits.
3. On décale à droite de $bitPosition$ bits (`»`), puis on applique le masque (`&`) pour ne garder que les k bits de poids faible.

Algorithm 2: Accès direct Aligned - get(index)

Input: int index
Output: int value
$valuesPerWord \leftarrow 32/bitsPerValue$;
$wordIndex \leftarrow index/valuesPerWord$;
$bitOffset \leftarrow (index \mod valuesPerWord) \times bitsPerValue$;
$mask \leftarrow (1 \ll bitsPerValue) - 1$;
return $(data[wordIndex] \gg bitOffset)\ \&\ mask$;

3.4 Méthode "Overlaped" (chevauchée)

La deuxième méthode que j'ai implémenté dans ce projet est la méthode "Overlaped" (chevauchée). Cette méthode est une amélioration de la méthode "Aligned" car elle permet de maximiser l'utilisation des bits disponibles dans chaque bloc de 32 bits en autorisant le chevauchement des entiers entre les blocs.

3.4.1 Principes théortiques

Prenons le même exemple que précédemment avec le tableau d'entiers suivant :

```
Tab = [5, 7, 12, 1023, 511, 3];
```

On va de nouveau recupérer la valeur maximale du tableau (1023) pour déterminer le nombre de bits nécessaires pour représenter chaque entier. Toujours 10 bits dans notre exemple. Cette fois-ci, au lieu de diviser le tableau en blocs de 32 bits sans chevauchement, on va permettre aux entiers de chevaucher les blocs. C'est à dire que si un entier ne rentre pas entièrement dans le bloc courant, au lieu de mettre des bits de padding et de passer au bloc suivant, on va commencer à l'écrire dans ce bloc et continuer dans le bloc suivant.

Mot 0

Valeur	Bits utilisés	Position du mot dans le bloc
5	bit 0 → 9	mot 0
7	bit 10 → 19	mot 0
12	bit 20 → 29	mot 0
1023	bit 30 → 31	début dans le mot 0

Mot 1

Valeur	Bits utilisés	Position du mot dans le bloc
1023	bit 0 → 7	fin dans le mot 1
511	bit 10 → 19	mot 1
3	bit 20 → 29	mot 1

Cette méthode utilise les bits disponibles de manière plus efficace. Pour notre exemple, au lieu d'avoir 2 mots avec 4 bits inutilisés au total (2 bits par mot), nous n'avons qu'un mot et demi avec seulement 4 bits inutilisés dans le dernier mot.

Voici donc comment les entiers sont répartis dans les blocs avec la méthode "Overlaped".

Listing 4 – Encodage binaire du mot 0

```
0000000101 0000000111 0000001100 11
```

Listing 5 – Encodage binaire du mot 1

```
11111111 0111111111 0000000011 00
```

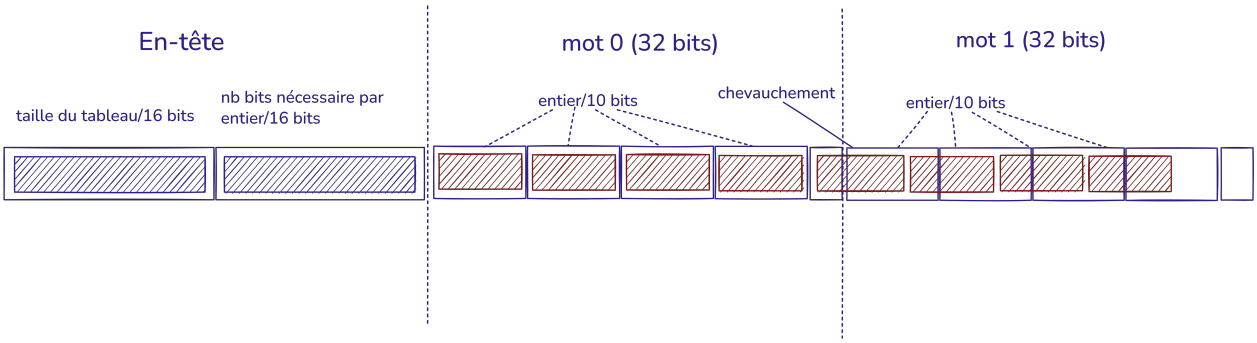

FIGURE 3 – Représentation schématique d'un tableau compressé avec la méthode "Overlapped"

3.4.2 Implémentation et fonctionnement

Concernant l'implémentation de cette méthode, elle est similaire à celle de la méthode "Aligned" avec quelques ajustements pour gérer le chevauchement.

Algorithm 3: Compression Overlapped

Input: UnpackedData fromUnpackedData
Output: PackedData toPackedData
for *each value in input array* **do**
  $wordIndex \leftarrow bitPosition/32$;
  $bitOffset \leftarrow bitPosition \mod 32$;
  $bitsRemaining \leftarrow 32 - bitOffset$;
  if $bitsRemaining \geq bitsPerValue$ **then**
    $compressed[wordIndex] \leftarrow compressed[wordIndex] \mid (value \ll bitOffset)$;
  end
  else
    $compressed[wordIndex] \leftarrow compressed[wordIndex] \mid (value \ll bitOffset)$;
    $compressed[wordIndex + 1] \leftarrow compressed[wordIndex + 1] \mid (value \gg bitsRemaining)$;
  end
  $bitPosition \leftarrow bitPosition + bitsPerValue$;
end

Algorithm 4: Accès direct Overlapped - get(index)

Input: int index
Output: int value
$bitsPerValue \leftarrow$ bits per value from metadata;
$bitPosition \leftarrow index \times bitsPerValue$;
$wordIndex \leftarrow bitPosition/32$;
$bitOffset \leftarrow bitPosition\&31$;
$mask \leftarrow (1 \ll bitsPerValue) - 1$;
if $bitOffset + bitsPerValue \leq 32$ **then**
    return $(data[wordIndex] \gg bitOffset) \wedge mask$;
end
else
    $low \leftarrow data[wordIndex] \gg bitOffset$;
    $high \leftarrow data[wordIndex + 1] \ll (32 - bitOffset)$;
    return $(low \mid high) \wedge mask$;
end

3.5 Méthode "Overflow" (débordement)

La troisième méthode que j'ai implémenté dans ce projet est la méthode "Overflow" (débordement). Cette méthode est une autre amélioration de la méthode "Aligned" car elle permet de gérer les entiers qui ne rentrent pas dans le bloc courant en utilisant un mécanisme de débordement.

3.5.1 Principes théortiques

Le problème avec les méthodes précédentes est qu'une seule grande valeur peut forcer l'utilisation d'un grand nombre de bits pour toutes les valeurs. Si on change d'exemple et que l'on considère :

```
Tab = [1, 2, 3, 1024, 4, 5, 2048];
```

Avec les méthodes Aligned ou Overlapped, toutes les valeurs seraient encodées sur 11 bits (car $2^{11} = 2048$), alors que les valeurs 1, 2, 3, 4, 5 pourraient être représentées sur seulement 3 bits.

La méthode Overflow résout ce problème en introduisant :

— Un **bit de flag** pour chaque valeur : 0 si la valeur est stockée directement, 1 si elle est dans la table de débordement
— Une **table de débordement** qui stocke les valeurs trop grandes
— Un **nombre de bits réduit** pour le payload, optimisé pour les valeurs typiques

Pour notre exemple, on peut décider d'utiliser 3 bits pour le payload (représentant les valeurs 0-7). Les valeurs 1, 2, 3, 4, 5 seront encodées directement avec un flag à 0. Les valeurs 1024 et 2048, trop grandes, seront placées dans la table de débordement avec un flag à 1 et un index pointant vers leur position.

Encodage avec Overflow

Valeur originale	Flag	Payload (3 bits)	Signification
1	0	001	valeur directe = 1
2	0	010	valeur directe = 2
3	0	011	valeur directe = 3
1024	1	000	index overflow = 0
4	0	100	valeur directe = 4
5	0	101	valeur directe = 5
2048	1	001	index overflow = 1

Table de débordement : [1024, 2048]

Le choix du nombre de bits pour le payload est crucial et fait l'objet d'une optimisation. L'algorithme cherche le meilleur compromis entre :

— Le coût de stockage des valeurs directes : $n \times (1 + k)$ bits où k est le nombre de bits du payload
— Le coût de la table de débordement : $m \times 32$ bits où m est le nombre de valeurs en débordement

3.5.2 Implémentation et fonctionnement

L'implémentation de cette méthode est plus complexe car elle nécessite une phase d'analyse pour déterminer le nombre optimal de bits.

Algorithm 5: Compression Overflow

Input: UnpackedData fromUnpackedData
Output: PackedData toPackedData
`// Phase 1: Déterminer le nombre optimal de bits`
$bitLengths \leftarrow$ array of bit lengths for each value;
$payloadBits \leftarrow$ findBestBitSize($bitLengths$);
$elementBits \leftarrow payloadBits + 1$ `// +1 pour le flag`
$maxPayloadValue \leftarrow 2^{payloadBits} - 1$;
`// Phase 2: Construire la table overflow`
$overflowTable \leftarrow$ empty list;
for *each value in input* **do**
  if $value > maxPayloadValue$ **then**
    Add value to $overflowTable$;
  end
end
`// Phase 3: Encoder les valeurs`
$bitPosition \leftarrow 0$;
for *each value in input* **do**
  if $value \leq maxPayloadValue$ **then**
    $encoded \leftarrow (0 \ll payloadBits) \mid value$ `// Flag = 0`
  else
    $index \leftarrow$ position of value in $overflowTable$;
    $encoded \leftarrow (1 \ll payloadBits) \mid index$ `// Flag = 1`
  end
  Write $encoded$ at $bitPosition$ (with overlap if needed);
  $bitPosition \leftarrow bitPosition + elementBits$;
end
`// Phase 4: Créer le header`
Store $payloadBits$, $originalSize$, $overflowTable$ in header;
Concatenate header and compressed data;

Algorithm 6: Optimisation du nombre de bits - findBestBitSize

Input: int[] bitLengths
Output: int optimal bit size
Sort *bitLengths* in ascending order;
$bestBitSize \leftarrow$ max bit length;
$smallestTotalSize \leftarrow \infty$;
for *each unique bit length k in sorted array* **do**
  $overflowCount \leftarrow$ count of values needing $> k$ bits;
  `// Vérifier si les index peuvent tenir`
  if $overflowCount > 0$ *and* $overflowCount > 2^k$ **then**
    Continue to next k;
  end
  $baseSize \leftarrow n \times (1 + k)$ `// Coût des valeurs encodées`
  $overflowSize \leftarrow overflowCount \times 32$ `// Coût table overflow`
  $totalSize \leftarrow baseSize + overflowSize$;
  if $totalSize < smallestTotalSize$ **then**
    $smallestTotalSize \leftarrow totalSize$;
    $bestBitSize \leftarrow k$;
  end
end
return *bestBitSize*;

4 Optimisation des performances

Dans ce projet, il y a plusieurs aspects que j'ai optimisé pour améliorer les performances globales de l'application de compression de données.

4.1 Recherche du nombre de bits

Pour les deux premières méthodes (Aligned et Overlapped), on clacule le nombre de bits nécessaires pour représenter la valeur maximale. Pour cela on doit premièrement trouber la valeur maximale dans le tableau. C'est cet algorithm de maximum que j'ai d'abord voulu optimiser. Je me suis prmièrement retournée vers un algorithm a complexité linéaire O(n) qui consiste à parcourir le tableau une seule fois pour trouver la valeur maximale. J'ai ensuite cherché à optimiser cet algorithm en utilisant une approche diviser pour régner (divide and conquer). Mais finalement, j'ai trouvé une méthode java qui utilise du parallélisme pour trouver le maximum dans un tableau d'entiers. Cette méthode est `Arrays.stream(array).summaryStatistics().getMax()`

Listing 6 – Recherche du maximum optimisée

```
public int getMaxValue() {
    IntSummaryStatistics stat = Arrays.stream(this.getData())
        .summaryStatistics();
    return stat.getMax();
}
```

4.2 Opérations bit à bit

Une des autres optimisations que j'ai mis en place dans ce projet est l'utilisation d'opérations bit à bit pour manipuler les données compressées à la place d'opérations arythmétiques. Toutes les opérations de manipulation de bits utilisent des opérateurs bit à bit natifs du processeur plutôt que des opérations arithmétiques :

— Décalage gauche : `value « shift` plutôt que `value * (1 « shift)`
— Décalage droite : `value »> shift` (décalage logique sans signe)
— ET logique : `value & mask` pour extraire des bits
— OU logique : `word | value` pour combiner des valeurs

4.3 Éviter les allocations inutiles

Dans les méthodes de compression et décompression, les tableaux de sortie sont alloués avec la taille exacte nécessaire, calculée à l'avance :

Listing 7 – Allocation optimale

```
// Aligned
int valuesPerWord = 32 / maxBitsNeeded;
int requiredWords = (originalArrayLength + valuesPerWord - 1)
                    / valuesPerWord;
int[] compressedData = new int[requiredWords];
```

```
// Overlapped
int totalBits = originalArrayLength * maxBitsNeeded;
int requiredWords = (totalBits + 31) / 32;
int[] compressedData = new int[requiredWords];
```

Cela évite les réallocations dynamiques et la fragmentation mémoire.

5 Résultats et analyse des benchmarks

5.1 Analyse comparative des performances

Les benchmarks ont été réalisés sur trois tailles de datasets : 1 000, 10 000 et 100 000 valeurs. Pour chaque taille, trois scénarios ont été testés :
— une distribution uniforme (valeurs entre 0 et 100),
— la présence de 2 % de valeurs outliers (valeurs très élevées),
— une plage de valeurs très étendue (0 à 100 000).

Ces expériences permettent d'évaluer les performances de chaque méthode en termes de vitesse de compression, de décompression, de temps d'accès direct et de taux de réduction.

5.1.1 Distribution uniforme (0–100)

Sur des données homogènes, les trois méthodes obtiennent des taux de réduction élevés. La méthode **Overlapped** atteint le meilleur taux avec environ 78 % de réduction, légèrement supérieur à **Aligned** (75 %). Cependant, cette meilleure compression s'accompagne d'un temps de traitement un peu plus long (+30 % environ). La méthode **Overflow**, quant à elle, garde une bonne réduction (75 %) mais reste beaucoup plus lente à la compression.

TABLE 1 – Résultats pour une distribution uniforme (0–100)

Méthode	Compression (µs)	Décompression (µs)	Accès (ns)	Réduction (%)
Aligned	885,68	185,87	38,13	75,0
Overlapped	1256,87	376,24	39,53	78,1
Overflow	4379,54	441,99	18347,70	75,0

En termes de rapidité, la méthode **Aligned** reste la plus efficace, surtout pour l'accès direct. Elle permet de lire une valeur en moins de 40 ns, ce qui en fait une solution très performante pour les systèmes où la vitesse prime sur le taux de compression.

5.1.2 Présence de valeurs outliers (2 %)

Avec des valeurs extrêmes, les différences entre les méthodes deviennent plus nettes. **Aligned** perd tout avantage de compression, car une seule valeur très grande impose d'utiliser plus de bits pour toutes les autres. **Overlapped** résiste un peu mieux grâce à son découpage en blocs, mais son taux de réduction tombe autour de 37 %. C'est dans ce cas que **Overflow** montre toute son utilité : en isolant les valeurs extrêmes dans une table de débordement, elle maintient une bonne compression autour de 60 à 70 %, même avec une distribution très irrégulière.

On remarque ici que la méthode **Overflow** est bien plus lente, aussi bien en compression qu'en accès direct. Son intérêt réside donc principalement dans les situations où le gain d'espace est prioritaire, par exemple lors d'un archivage ou d'un envoi ponctuel de données.

TABLE 2 – Résultats avec 2 % de valeurs outliers

Méthode	Compression (µs)	Décompression (µs)	Accès (ns)	Réduction (%)
Aligned	735,79	198,35	44,83	0,0
Overlapped	895,74	255,62	50,40	37,5
Overflow	3904,37	467,12	35223,75	60,5

5.1.3 Plage de valeurs étendue (0–100 000)

Quand les valeurs du dataset couvrent une grande plage, la méthode **Aligned** devient totalement inefficace, car les entiers nécessitent presque tous 32 bits. **Overlapped** conserve une compression d'environ 47 % grâce à une meilleure utilisation des bits disponibles. **Overflow** reste légèrement en dessous avec 43–44 % de réduction, mais ses temps de traitement augmentent fortement.

TABLE 3 – Résultats pour une plage de valeurs étendue (0–100 000)

Méthode	Compression (µs)	Décompression (µs)	Accès (ns)	Réduction (%)
Aligned	566,34	175,11	36,55	0,0
Overlapped	1247,93	420,48	1482,03	46,9
Overflow	6568,84	789,35	52107,93	43,7

Ce scénario montre bien que le choix d'une méthode dépend directement du type de données. Lorsque les valeurs sont dispersées sur une large plage, seules les approches plus flexibles comme **Overlapped** ou **Overflow** restent efficaces.

5.2 Comparaison globale des méthodes

Le tableau suivant résume les points forts et les limites de chaque méthode, ainsi que leurs cas d'utilisation privilégiés.

TABLE 4 – Comparaison des méthodes de compression

Méthode	Points forts	Limites	Cas d'usage idéal
Aligned	Très rapide en compression, décompression et accès direct. Structure simple et stable.	Inefficace avec des valeurs très variables ou outliers.	Données homogènes, traitements temps réel, accès fréquents.
Overlapped	Bon compromis entre taux de compression et performance. Exploite mieux les bits disponibles.	Légère perte de vitesse, complexité plus élevée.	Données numériques uniformes, stockage optimisé, lectures occasionnelles.
Overflow	Très bon taux de compression pour données hétérogènes. Gère bien les valeurs extrêmes.	Lente en compression et accès, structure complexe.	Archivage ou transfert ponctuel, compression maximale souhaitée.

5.3 Synthèse

L'analyse des résultats montre que chaque méthode a son propre domaine d'efficacité :

— **Aligned** est la plus rapide, mais uniquement utile quand les valeurs sont proches les unes des autres.

— **Overlapped** représente le meilleur compromis global : elle garde un bon taux de compression tout en restant performante.

— **Overflow** est la plus efficace sur le plan de la taille, mais trop lente pour un usage fréquent.

Ainsi, le choix de la méthode dépend du contexte : pour un traitement rapide, *Aligned* reste la plus adaptée ; pour un stockage efficace sans sacrifier la performance, *Overlapped* est préférable ; et pour une compression maximale sur des données très variées, *Overflow* reste la plus performante malgré sa lenteur.

6 Bilan & Conclusion

Pour conclure, trois méthodes de compression ont été étudiées et comparées : Aligned, Overlapped et Overflow. Les résultats du benchmark montrent que chacune d'elles présente des avantages et des limites selon le type de données traité :

Aligned se distingue par sa rapidité et sa simplicité d'implémentation. Elle est particulièrement efficace pour des données homogènes où les valeurs sont peu dispersées. Overlapped offre un excellent compromis entre efficacité de compression et performance.

En maximisant l'utilisation des bits disponibles, elle améliore le taux de réduction sans dégrader fortement les temps de traitement. Overflow, enfin, se démarque par sa ca-

pacité à gérer les valeurs extrêmes et les distributions très hétérogènes, mais au prix d'une complexité et d'une latence plus importantes. D'un point de vue global, la mé-

thode Overlapped apparaît comme la solution la plus équilibrée : elle combine de bons résultats en compression et une vitesse satisfaisante, tout en restant relativement simple à maintenir. La méthode Aligned reste néanmoins préférable pour des contextes où la vitesse d'accès est critique, tandis que Overflow s'impose dans les cas où la priorité est donnée à la réduction maximale de la taille, comme pour l'archivage ou la transmission à faible bande passante. Enfin, ce projet a permis de mettre en pratique de nombreux

concepts de génie logiciel : architecture modulaire, encapsulation, patterns de conception, et optimisation des performances. Il a également offert une réflexion concrète sur les compromis entre vitesse, espace mémoire et complexité algorithmique — des enjeux centraux dans le domaine de l'ingénierie logicielle et du traitement de données

Table des figures

Références

[Cob23] Cobo3. *BitPacking - GitHub Repository*. Consulté en 2025. 2023. URL : https://github.com/Cobo3/BitPacking?tab=readme-ov-file.

[Fle24] Luke Fleed. *Compressed FixedVec and the Builder Pattern*. Consulté en 2025. 2024. URL : https://lukefleed.xyz/posts/compressed-fixedvec/#builder-pattern.

[Med20] Data Science Medium. *Smart Way of Storing Data*. Consulté en 2025. 2020. URL : https://medium.com/data-science/smart-way-of-storing-data-d22dd5077340.

[OSS24] Quickwit OSS. *Bitpacking Source Code*. Consulté en 2025. 2024. URL : https://github.com/quickwit-oss/bitpacking/tree/eec95b24af3ffd0795606ab9ebe
src.

[Ove23] Stack Overflow. *Bin Packing with Overflow*. Consulté en 2025. 2023. URL : https://stackoverflow.com/questions/75423263/bin-packing-with-overflow.

[Tec16] Kinematic Soup Technologies. *Data Compression : Bit Packing 101*. Consulté en 2025. 2016. URL : https://kinematicsoup.com/news/2016/9/6/data-compression-bit-packing-101.

[Uni24] Cornell University. *Bit Packing Lecture Notes - CS3410 Computer Organization*. Consulté en 2025. 2024. URL : https://www.cs.cornell.edu/courses/cs3410/2024fa/notes/bitpack.html.