Université — Master 1 Informatique

Projet Bit Packing

Auteur : Tom DA COSTA

Année universitaire : 2025–2026

Implémentation et étude comparative de trois variantes de compression d'entiers en Java.

Novembre 2025

Introduction

Ce projet s'inscrit dans le cadre du cours de Software Engineering et porte sur la création et la comparaison de plusieurs méthodes de compression d'entiers, aussi connues sous le nom de *Bit Packing*. L'objectif est de réduire la taille mémoire occupée par un tableau d'entiers tout en conservant un accès direct à chaque élément sans décompresser l'ensemble des données. Il y a également la mise en place d'un protocole expérimental rigoureux pour mesurer les performances et déterminer dans quelles conditions la compression devient réellement avantageuse selon la latence et le débit de transmission.

1 Problèmes rencontrés et objectifs

La manipulation de grands volumes d'entiers est fréquente dans les applications modernes, comme les bases de données ou les systèmes de capteurs. Chaque entier étant stocké sur 32 bits, on gaspille souvent de l'espace lorsque les valeurs réelles nécessitent beaucoup moins de bits. Le premier défi a donc été de concevoir un moyen d'empaqueter les bits pour réduire la taille tout en gardant un accès direct efficace.

Un second problème est apparu avec les données hétérogènes. Certains tableaux contiennent une majorité de petites valeurs mais aussi quelques valeurs très grandes, ce qui empêche d'utiliser une taille fixe optimale pour tout le monde. Enfin, la mesure des performances a soulevé des questions pratiques : les temps d'exécution peuvent varier selon la charge du système et la manière dont la machine virtuelle Java (JVM) optimise le code.

Remarque : la JVM (Java Virtual Machine) est l'environnement qui exécute le byte-code Java. Elle contient un composant appelé JIT (*Just-In-Time Compiler*), qui compile dynamiquement le code pendant l'exécution pour le rendre plus rapide. Cependant, ce mécanisme peut fausser les premières mesures, car les performances s'améliorent au fil des exécutions. C'est pourquoi un protocole spécifique a été mis en place pour obtenir des résultats stables et représentatifs.

2 Implémentations réalisées

Trois variantes principales ont été développées.

2.1 Variante `NON_OVERLAP`

Cette version réserve un nombre fixe de bits par valeur, calculé à partir du maximum du tableau. Chaque entier est stocké dans un mot de 32 bits sans chevauchement, ce qui facilite considérablement l'accès direct. Cette méthode est simple, rapide et robuste, mais elle peut gaspiller quelques bits à la fin de chaque mot lorsque la taille choisie ne tombe pas exactement sur 32 bits.

2.2 Variante `OVERLAP`

Dans cette approche, les entiers sont écrits les uns à la suite des autres dans un flux binaire continu, même si une valeur déborde sur le mot suivant. Cette méthode utilise pleinement les 32 bits disponibles et atteint une compression plus dense. En revanche, elle demande des calculs supplémentaires pour lire une valeur qui chevauche deux mots. C'est une solution plus efficace en espace mais légèrement plus coûteuse en temps d'accès.

2.3 Variante `OVERFLOW`

Le principe de la version `OVERFLOW` repose sur la séparation du tableau en deux zones distinctes. Cette approche vise à éviter qu'une seule valeur très grande impose un codage inutilement large à l'ensemble du tableau.

La première est la **zone principale**, qui contient la majorité des valeurs dites "petites". Elles sont encodées sur un nombre réduit de bits, noté k_{small}, et chaque entrée y est accompagnée d'un bit spécial appelé *bit de contrôle*. Ce bit indique la manière dont la valeur doit être interprétée :

- si le bit vaut 0, la valeur est stockée directement dans la zone principale ;
- si le bit vaut 1, cela signifie que la valeur est trop grande et qu'il faut aller la lire dans la zone de débordement.

La seconde zone, appelée **zone de débordement**, contient uniquement ces valeurs exceptionnelles, encodées cette fois sur un nombre plus important de bits, noté k_{big}. Chaque valeur de cette zone est repérée par un index qui est référencé depuis la zone principale.

Ce découpage permet de conserver une structure compacte pour la majorité des entiers, tout en traitant les valeurs extrêmes de manière séparée. L'accès direct reste constant puisque chaque entrée de la zone principale a une largeur fixe (le bit de contrôle plus la valeur ou l'index). Cette organisation améliore la densité de compression sans compromettre la rapidité d'accès, surtout dans les jeux de données où seules quelques valeurs dépassent largement la moyenne.

3 Protocole de mesure du temps d'exécution

L'évaluation des performances a été menée avec soin pour garantir la fiabilité des résultats. Les mesures ont été effectuées dans la classe `MicroBenchMain`, en utilisant des tableaux générés aléatoirement de différentes tailles et distributions.

Chaque opération (compression, décompression et accès direct) a été chronométrée séparément à l'aide d'un chronomètre de haute précision. Avant de mesurer, plusieurs exécutions de « chauffe » ont été réalisées afin de laisser la JVM et le JIT optimiser le code. Ces exécutions non mesurées permettent d'éviter les biais dus à la compilation dynamique. Ensuite, chaque opération a été répétée plusieurs fois, et seule la médiane des temps a été conservée pour éliminer les valeurs anormales.

Pour garantir des résultats fiables, plusieurs précautions ont été prises :

- Les premiers essais ont servi à "réchauffer" la JVM afin de stabiliser le compilateur JIT, qui optimise le code pendant l'exécution.

— Les mesures ont été répétées plusieurs fois et la médiane a été retenue pour éviter l'influence d'événements externes (charge CPU, interruptions système...).
— Une variable d'accumulation (*blackhole*) a été utilisée pour s'assurer que le code mesuré ne soit pas supprimé par le compilateur.

L'analyse ne s'est pas limitée aux temps purs. Elle a aussi permis d'évaluer la pertinence de la compression selon le débit du réseau. Si le lien est lent, la réduction du volume compense largement le coût du traitement. En revanche, sur un réseau rapide, la compression peut devenir contre-productive, car le calcul prend plus de temps que la transmission directe. Cette observation permet de situer les conditions dans lesquelles chaque méthode est réellement avantageuse.

4 Résultats expérimentaux

Les tests ont été réalisés sur des tableaux de 100 000 entiers contenant principalement de petites valeurs, avec quelques valeurs extrêmes pour tester la robustesse. Le tableau suivant présente les résultats moyens observés :

Méthode	**Ratio**	**Temps (ms)**	**Observation**
OVERLAP	0.38	0.7	Excellent ratio, accès un peu plus coûteux.
NON_OVERLAP	0.50	0.6	Très rapide, accès direct simple.
OVERFLOW	0.31	0.9	Efficace pour les valeurs extrêmes.

Ces résultats permettent également d'estimer dans quelles conditions la compression devient réellement rentable. L'idée est de comparer le temps total nécessaire à l'envoi d'un tableau compressé avec celui d'un tableau non compressé.

On peut distinguer deux situations :
— T_{sans}, le temps de transmission brute, qui correspond simplement au temps nécessaire pour envoyer les données non compressées ;
— T_{avec}, le temps total lorsqu'on applique la compression, qui inclut le temps de compression, le temps de transmission des données compressées, et enfin le temps de décompression.

La compression devient avantageuse lorsque le temps total avec compression est inférieur à celui sans compression. En pratique, cela revient à dire que la somme du temps de compression et de décompression doit rester plus faible que le gain de temps obtenu grâce à la réduction du volume à transmettre.

$$t_c + t_d < t_t \times (1 - r)$$

où :
— t_c est le temps moyen de compression,
— t_d le temps moyen de décompression,
— t_t le temps de transmission du tableau non compressé,
— et r le ratio de compression (par exemple 0,4 pour un gain de 60%).

En remplaçant avec les valeurs mesurées pour la méthode `OVERLAP` ($t_c \approx 0{,}7$ ms, $t_d \approx 0{,}2$ ms, $r = 0{,}38$), on obtient que la compression devient intéressante lorsque le temps de transmission brute dépasse environ 1,45 ms.

Autrement dit, si la transmission d'un tableau non compressé prend plus d'une milliseconde et demie, il est déjà plus rentable de le compresser avant de l'envoyer. Ce seuil dépend bien sûr du réseau utilisé, mais il donne une idée concrète du moment où la compression commence à "valoir le coup" dans un scénario réel.

Pour visualiser ce raisonnement, on peut représenter l'évolution du temps total en fonction de la latence du réseau. Le graphique ci-dessous illustre cette comparaison entre un envoi brut et un envoi compressé en tenant compte des temps moyens de compression (0,7 ms) et de décompression (0,2 ms) mesurés lors des expérimentations.

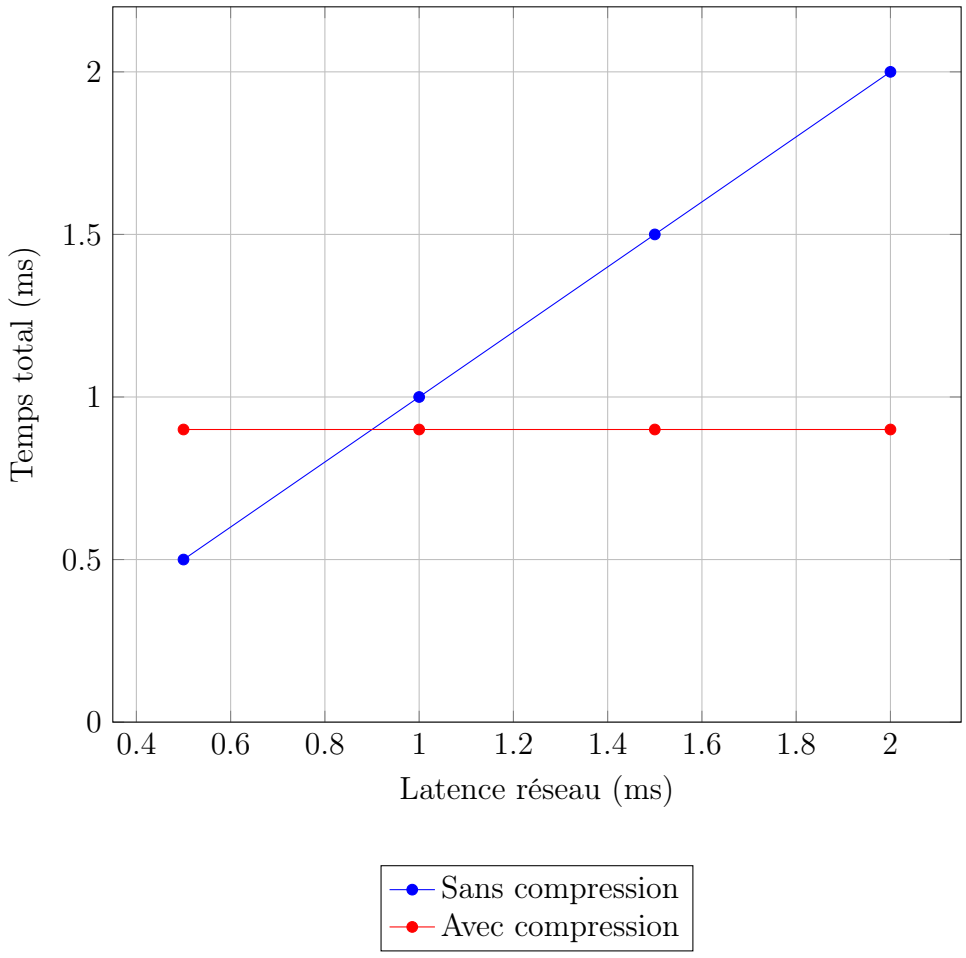

Figure 1 : Comparaison du temps total selon la latence réseau. La compression devient avantageuse au-delà d'environ 1,5 ms de latence.

Ces résultats montrent que la méthode `OVERLAP` offre le meilleur compromis entre taux de compression et rapidité globale. `NON_OVERLAP` reste la plus rapide à exécuter, tandis que `OVERFLOW` devient particulièrement utile lorsque le tableau contient des valeurs hétérogènes.

5 Commentaires et difficultés rencontrées

La principale difficulté rencontrée a concerné la gestion précise des décalages de bits et la reconstruction correcte des entiers. Une erreur de un bit peut complètement fausser le résultat, ce qui a rendu les débogages particulièrement délicats. La version `OVERFLOW` a aussi demandé beaucoup de réflexion pour ajuster la taille optimale des deux zones, car une mauvaise proportion réduit soit la compression, soit la rapidité.

Un autre point complexe a été la mise au point du protocole de mesure. Les premières mesures variaient fortement à cause du comportement de la JVM. L'introduction du warm-up et du calcul médian a permis d'obtenir des résultats plus cohérents et représentatifs. Enfin, la structure du code a été pensée pour rester lisible et modulaire grâce à une fabrique (`CompressionFactory`) qui crée dynamiquement la bonne variante de compresseur selon le mode choisi.

6 Extension : gestion des nombres négatifs

Ces algorithmes ont été conçus pour des entiers positifs, mais dans de nombreux contextes les données peuvent être négatives. Cela pose un problème, car le bit le plus à gauche sert à représenter le signe, ce qui complique la lecture et l'écriture directe des bits.

Deux approches peuvent être envisagées pour résoudre ce problème :

- La première consiste à décaler toutes les valeurs pour qu'elles deviennent positives avant la compression, puis à inverser ce décalage après la décompression. Cette méthode est simple, robuste et compatible avec les implémentations actuelles.
- La seconde approche repose sur la manipulation directe de la représentation en complément à deux utilisée par Java. Cela permet de conserver les nombres signés tels quels, mais impose d'adapter toutes les opérations de masquage et de décalage pour éviter les erreurs d'interprétation.

Une combinaison de cette idée avec la version `OVERFLOW` serait particulièrement prometteuse. Les valeurs extrêmes, qu'elles soient très grandes ou très petites, pourraient être stockées dans la zone de débordement, assurant ainsi une bonne densité tout en couvrant l'ensemble des entiers possibles. Cette extension améliorerait la polyvalence du système et permettrait d'utiliser les méthodes de compression dans des contextes réels où les données signées sont courantes.

Conclusion

Ce projet a permis d'explorer différentes stratégies de compression d'entiers et de comprendre les compromis entre taille, temps et complexité d'accès. La version sans chevauchement se distingue par sa simplicité et sa rapidité, celle avec chevauchement par sa meilleure densité, et la version avec débordement par sa capacité à s'adapter à des jeux de données irréguliers. L'ensemble du protocole de mesure a fourni des résultats fiables et exploitables, en prenant en compte le comportement réel de la JVM et les effets du JIT.

L'étude a également montré que la compression n'est pas toujours bénéfique et que son

intérêt dépend du contexte de transmission. Ce travail a été une excellente occasion de mettre en pratique les notions d'optimisation, de mesure de performance et de conception logicielle vues pendant le semestre, tout en développant une vision plus critique des compromis entre complexité algorithmique et efficacité réelle.