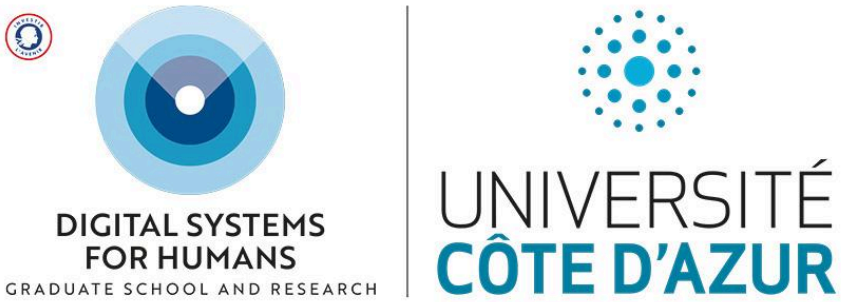

Rapport de projet - UE Software Engineering

BitPacking, Compression d'entiers

Jean-Jacques VIALE

Université Côte d'Azur - EUR DS4H
Master 1 Informatique

Encadrant : Jean-Charles Régin

02/11/2025

Introduction

La transmission de données numériques constitue l'un des enjeux majeurs des systèmes connectés modernes. Sur Internet, de nombreuses applications échangent des tableaux d'entiers, qu'il s'agisse de mesures issues de capteurs, de données compressées d'images ou encore de structures d'indexation. Ces transmissions représentent un volume de données considérable, et leur rapidité dépend directement de la taille des messages envoyés.

Dans ce contexte, la compression de données permet de réduire la quantité d'informations à transmettre. Cependant, dans certaines applications (bases de données, index inversés, traitements temps réel), il est essentiel de conserver un accès direct à chaque élément sans devoir décompresser l'ensemble du tableau. Le projet vise à répondre à ce besoin par une méthode de compression basée sur le Bit Packing, une technique qui consiste à représenter chaque entier sur le nombre minimal de bits nécessaires.

L'objectif du projet est donc double, tout d'abord concevoir et implémenter différentes versions d'un algorithme de Bit Packing permettant de compresser et décompresser efficacement des tableaux d'entiers. Ainsi qu'évaluer expérimentalement les performances de ces méthodes afin d'identifier le moment où la compression devient avantageuse selon la latence du canal de transmission.

Le projet inclut également une extension avec zone de débordement (overflow area), permettant de traiter des ensembles de données hétérogènes sans pénaliser les valeurs les plus simples. Enfin, une réflexion sera menée sur la prise en compte des nombres négatifs, qui posent des contraintes particulières dans la représentation binaire.

Ce rapport présente la démarche adoptée, les choix techniques effectués, les algorithmes développés, ainsi que les résultats expérimentaux obtenus.

1. Problématique et objectifs

1.1 Problème du volume de données dans la transmission d'entiers

La transmission de tableaux d'entiers constitue une opération fréquente dans de nombreuses applications, telles que les systèmes temps réel, les bases de données ou les réseaux de capteurs. Dans ces contextes, chaque entier est généralement représenté sur 32 bits, quel que soit sa valeur réelle, ce qui entraîne un gaspillage important de bande passante lorsque la majorité des entiers pourrait être codée sur moins de bits. Ce surplus de données peut ralentir les transmissions et dégrader les performances des applications sensibles à la latence. L'objectif du projet est donc de réduire la taille des tableaux transmis tout en conservant un accès rapide à chaque élément.

1.2 Besoin d'une compression sans perte d'accès direct

Les méthodes de compression classiques, bien qu'efficaces pour réduire la taille des données, ne permettent généralement pas un accès direct à un élément compressé. Pour obtenir la valeur du i-ème entier, il faut souvent décompresser l'ensemble du tableau. Dans certaines applications, un accès direct à chaque entier est indispensable. Dans le code fourni, cette exigence est satisfaite par la méthode get(index) de l'interface ICompressor, qui permet de récupérer un élément à partir de la représentation compressée sans avoir à décompresser tout le tableau. Cette contrainte impose une structuration spécifique du flux binaire pour que chaque entier puisse être localisé rapidement.

1.3 Principe du Bit Packing

Le Bit Packing est une technique de compression qui consiste à représenter chaque entier sur le nombre minimal de bits nécessaires pour stocker sa valeur, appelé bitWidth dans les classes implémentées. Les entiers sont ensuite packés les uns à la suite des autres dans un tableau d'entiers de 32 bits. Selon la variante choisie, plusieurs entiers peuvent occuper un même entier mémoire, ou un entier compressé peut chevaucher deux entiers consécutifs. Cette méthode permet de réduire significativement le volume de données tout en conservant un accès direct à chaque entier grâce à l'utilisation de masques et de décalages bit à bit.

1.4 Définition des trois modes de compression mis en oeuvres

Les variantes de Bit Packing implémentées dans ce projet sont conçues pour gérer différents compromis entre densité de compression et simplicité d'accès direct. Chaque méthode repose sur le principe de représenter les entiers en utilisant le moins de bits possible, mais avec des stratégies différentes pour l'organisation des bits.

1.4.1 BitPackingNoOverlap (sans chevauchement)

Cette version stocke chaque entier compressé entièrement dans un slot de 32 bits. Elle privilégie la simplicité et la rapidité d'accès direct aux entiers.

- Déterminer le bitWidth à partir de la valeur maximale du tableau.
- Calculer combien de valeurs peuvent être stockées dans un entier de 32 bits (slotsPerInt).
- Placer chaque entier dans un slot complet, en avançant par multiples de bitWidth. Accéder directement à l'élément i via un simple calcul de bloc et d'offset.

Cette méthode est plus simple à mettre en œuvre et à déboguer. Elle peut toutefois être légèrement moins dense lorsque le bitWidth ne divise pas exactement 32, car certains bits restent inutilisés.

1.4.2 BitPackingOverlap (avec chevauchement)

Dans cette variante, un entier compressé peut être scindé sur deux entiers de 32 bits consécutifs afin de maximiser l'utilisation des bits disponibles.

- Calculer le nombre minimal de bits nécessaires pour représenter la valeur maximale du tableau (bitWidth).
- Parcourir le tableau d'entrée et placer chaque valeur dans le flux de bits global.
- Si l'espace restant dans l'entier courant est suffisant, la valeur est écrite directement.
- Sinon, diviser la valeur : une partie est placée dans l'entier courant et le reste dans l'entier suivant.
- L'accès direct à un élément compressé est assuré grâce au calcul de la position en bits et à l'application de masques et de décalages.

Cette approche permet d'obtenir une densité de compression élevée, mais rend les opérations de lecture et d'écriture un peu plus complexes.

1.4.3 BitPackingOverflow (avec zone de débordement)

Cette variante est destinée aux ensembles de données hétérogènes, où certaines valeurs sont beaucoup plus grandes que la majorité.

- Déterminer un seuil baseBits qui couvre la majorité des valeurs.
- Les valeurs supérieures à baseBits sont stockées dans une zone de débordement (overflow area).
- Chaque entier dans la zone principale est préfixé par un flag : 0 pour une valeur normale stockée sur baseBits, 1 pour un index vers l'overflow.
- Le flux de bits est packé en respectant ce flag, et l'accès direct à n'importe quel élément reste possible grâce à un calcul bit à bit.

Cette approche permet de réduire le gaspillage de bits pour les valeurs courantes tout en conservant un accès rapide aux valeurs extrêmes stockées séparément.

2. Conception et architecture logicielle

2.1 Choix du langage et justification

Le projet a été implémenté en Java, un langage robuste offrant un support natif pour les entiers 32 bits et les opérations bit à bit nécessaires au Bit Packing. Java permet également de manipuler facilement les tableaux, de gérer les fichiers CSV et de mesurer précisément les temps d'exécution avec System.nanoTime(). L'utilisation de classes et d'interfaces favorise une architecture modulaire et extensible.

2.2 Interface commune ICompressor

Pour standardiser les différentes méthodes de compression, une interface ICompressor a été définie. Elle impose trois fonctionnalités principales :

- compress(int[] array) : compresse un tableau d'entiers.
- decompress() et decompress(int[] output) : décompresse le tableau, soit dans un nouveau tableau, soit dans un tableau existant pour éviter des allocations supplémentaires.
- get(int index) : permet l'accès direct au i-ème entier compressé sans décompresser l'ensemble du tableau.

Cette interface garantit la cohérence fonctionnelle entre toutes les variantes et simplifie leur utilisation grâce à une factory.

2.3 Classes de compression

2.3.1 BitPackingNoOverlap

Cette classe implémente la compression sans chevauchement. Dans le code :

- Le tableau compressé est stocké dans `compressedData`, un tableau d'entiers 32 bits.
- La largeur en bits de chaque valeur (`bitWidth`) est calculée à partir de la valeur maximale du tableau.
- Le nombre de valeurs pouvant tenir dans un entier 32 bits (`slotsPerInt`) est déterminé par `32 / bitWidth`.
- La méthode `compress` parcourt le tableau d'entrée et utilise des décalages (`offset`) et des masques binaires pour placer chaque entier dans le slot approprié de `compressedData`.
- La décompression est réalisée dans `decompress`, qui peut soit créer un nouveau tableau, soit remplir un tableau existant, en utilisant les mêmes calculs de bloc et d'offset.

- L'accès direct à un élément via `get(index)` est immédiat grâce au calcul du bloc et du décalage correspondant dans `compressedData`.

2.3.2 BitPackingOverlap

Cette classe gère la compression avec chevauchement sur deux entiers de 32 bits. Dans l'implémentation :

- Le flux binaire complet est suivi par `bitPos`, indiquant la position en bits courante.
- La largeur en bits (`bitWidth`) est calculée à partir de la valeur maximale.
- La méthode `compress` écrit directement dans `compressedData` si l'espace restant dans l'entier courant est suffisant, sinon elle divise la valeur en deux parties, réparties sur l'entier courant et le suivant.
- La décompression et la récupération directe (`decompress` et `get`) reconstruisent les entiers en appliquant des masques et des décalages similaires à ceux utilisés lors de l'écriture.
- Le calcul de `blockIndex` et `bitOffset` permet de localiser rapidement chaque entier dans le flux compressé.

2.3.3 BitPackingOverflow

Cette classe est destinée aux données hétérogènes, avec gestion des valeurs dépassant la largeur standard (`baseBits`) :

- Les petites valeurs sont stockées dans `compressedData` avec un flag d'indication (0 = valeur normale, 1 = valeur dans overflow).
- Les grandes valeurs sont placées dans `overflowData`, séparément du flux principal.
- `compress` identifie les valeurs dépassant `baseBits`, construit `overflowData`, et packe chaque entier bit par bit avec son flag.
- `decompress` reconstruit les entiers en vérifiant le flag et en récupérant la valeur dans `compressedData` ou `overflowData` selon le cas.
- L'accès direct via `get` parcourt le flux en suivant le même schéma pour retrouver la valeur correcte, qu'elle se trouve dans la zone principale ou dans l'overflow.

Chaque classe respecte l'interface commune `ICompressor`, ce qui garantit une uniformité des méthodes `compress`, `decompress` et `get` et facilite l'utilisation via `BitPackingFactory`.

2.4 Factory BitPackingFactory

La classe BitPackingFactory permet de créer dynamiquement la variante de compression souhaitée à partir d'un paramètre :

- "overlap" crée un BitPackingOverlap
- "nooverlap" crée un BitPackingNoOverlap
- "overflow" crée un BitPackingOverflow

Cette organisation respecte le design pattern Factory et facilite l'extension du projet à de nouvelles méthodes de compression.

3. Évaluation et validation expérimentale

3.1 Tests unitaires

Des tests unitaires JUnit 5 ont été réalisés pour vérifier la fiabilité, la réversibilité et la cohérence des trois implémentations de compression :
`BitPackingNoOverlap`, `BitPackingOverlap` et `BitPackingOverflow`.

Pour BitPackingNoOverlap, trois tests valident :

- la réversibilité (compression → décompression identique),
- l'accès direct via `get(i)`,
- et l'efficacité (taille compressée < taille d'origine).

Pour BitPackingOverlap, les tests confirment :

- la fidélité après décompression malgré le chevauchement des bits,
- la validité de l'accès direct,
- et un gain d'espace par rapport à l'entrée.

Enfin, pour BitPackingOverflow, cette version ajoute une zone de débordement pour les valeurs trop grandes. Les tests vérifient :

- la bonne restitution des valeurs normales et overflow,
  la présence effective de la zone `overflow`,
- et la robustesse sur des données mélangées.

3.2 Benchmarks de performance

Afin d'évaluer la rapidité et l'efficacité des trois implémentations, une campagne de benchmarks automatisés a été réalisée à l'aide de deux outils internes : `BenchmarkGenerator` et `Benchmark`.

Le générateur de données créé plusieurs jeux d'essai représentatifs :

- Uniformes, pour des valeurs réparties régulièrement
- Boltzmann, simulant une distribution exponentielle avec beaucoup de petites valeurs et quelques grandes
- EdgeCase32bit, pour tester les cas extrêmes aux limites de la plage des entiers 32 bits

Chaque jeu de données est décliné en plusieurs tailles (de 1 000 à 10 millions d'éléments) et sauvegardé au format CSV.

Le programme `Benchmark` lit ensuite les fichiers générés et mesure, pour chaque méthode de compression, le temps moyen de compression et de décompression (en millisecondes), le temps d'accès direct moyen via `get(i)` (en nanosecondes), le taux de compression (rapport entre les tailles compressée et originale), ainsi qu'une latence critique estimant le point d'équilibre entre le gain de taille et le coût en calcul. Chaque mesure est répétée cinquante fois afin de lisser les fluctuations et de stabiliser le compilateur JIT (*Java Just-In-Time*). Les résultats sont finalement enregistrés dans des fichiers CSV distincts pour chaque catégorie de données, facilitant ainsi l'analyse et la comparaison des performances.

3.3 Analyse des résultats

Les résultats des benchmarks montrent des tendances nettes entre les trois stratégies de compression. Les données d'entrées et de sortie des benchmarks sont disponible

Pour BitPackingNoOverlap, les temps de compression et de décompression sont faibles et stables, avec un accès direct rapide (≈20 ns). Cependant, le ratio reste souvent égal à 1 pour les jeux de données *Boltzmann* et *EdgeCase32bit*, indiquant une absence réelle de compression : la méthode est simple et rapide, mais inefficace en termes de gain d'espace. Sur les données *Uniform*, le ratio tombe à 0,5, démontrant que la méthode reste adaptée lorsque les valeurs occupent peu de bits.

BitPackingOverlap offre un compromis intéressant : les ratios de compression sont significativement meilleurs (≈0,4 à 0,53 selon la distribution), tout en conservant des temps de traitement faibles et un accès direct performant (autour de 20–30 ns). Cette méthode montre une montée en charge linéaire et une bonne stabilité des temps, ce qui en fait la plus équilibrée entre performance et efficacité.

Enfin, BitPackingOverflow se distingue par une gestion correcte des valeurs extrêmes, mais au prix de coûts importants. Les temps de compression explosent dès 100 000 éléments (plusieurs millisecondes à secondes), et les accès directs deviennent extrêmement coûteux (jusqu'à plusieurs microsecondes). Le ratio reste toutefois avantageux sur certaines distributions (*Boltzmann* ≈0,47–0,69), mais cette méthode n'est pas compétitive en pratique à cause de son surcoût en calcul.

En résumé, BitPackingOverlap s'impose comme la solution la plus performante et équilibrée : elle combine une compression efficace, un accès direct rapide et une scalabilité correcte. NoOverlap reste utile pour des cas simples ou des environnements contraints, tandis que *Overflow* est une solution de secours réservée aux jeux de données contenant de rares valeurs extrêmes.

3.4 Analyse et discussion des performances

La synthèse des trois méthodes de compression permet de mettre en évidence leurs forces et leurs limites respectives. La méthode BitPackingNoOverlap se distingue par sa rapidité en compression et décompression et par la simplicité de l'accès direct aux éléments, mais elle est légèrement moins efficace en densité lorsque le bitWidth ne divise pas exactement 32, avec des ratios de compression souvent proches de 1 pour des données hétérogènes. La méthode BitPackingOverlap améliore la densité grâce au chevauchement des bits, offrant un ratio de compression inférieur à 1 pour la plupart des distributions, tout en conservant un accès direct correct, mais légèrement plus coûteux que NoOverlap. Enfin, BitPackingOverflow optimise la compression pour les ensembles de données comportant des valeurs extrêmes, réduisant le gaspillage de bits pour les petites valeurs. Cependant, le coût en temps augmente avec la taille de la zone d'overflow, notamment pour les accès directs et pour les grandes tailles de données. Cette comparaison montre qu'il est possible de choisir la méthode adaptée selon l'usage : rapidité brute pour NoOverlap, densité maximale pour Overlap, ou adaptation aux outliers pour Overflow.

Les performances observées dépendent fortement de la distribution des données. Pour des ensembles uniformes, NoOverlap et Overlap présentent des performances similaires, tandis que Overflow n'apporte pas de gain significatif. En revanche, pour des distributions de type Boltzmann ou contenant des valeurs extrêmes, Overflow réduit le ratio de compression et permet de conserver un accès direct correct, mais au prix d'un coût en temps beaucoup plus élevé, notamment sur `get(i)` et pour de grandes tailles de données. Les temps de compression et de décompression augmentent de manière linéaire avec la taille des tableaux, ce qui confirme la scalabilité des algorithmes. L'analyse de la latence critique (`tCrit`) montre que la méthode Overflow ne devient rentable que pour des volumes très élevés de données, où la réduction de taille compense le coût supplémentaire en calcul.

Malgré leur efficacité, ces méthodes présentent certaines limites. La gestion des nombres négatifs n'est pas encore implémentée, ce qui restreint l'usage aux entiers positifs ; il serait possible d'y remédier en décalant toutes les valeurs ou en utilisant le complément à deux. La zone d'Overflow peut devenir très coûteuse en accès direct si elle contient plusieurs milliers d'éléments, et l'introduction d'une structure indexée pourrait améliorer la rapidité. Pour les très grands ensembles de données, la parallélisation de la compression et de la décompression pourrait réduire les temps moyens. Enfin, une amélioration possible consisterait à analyser automatiquement la distribution des données pour choisir dynamiquement le type de compression le plus adapté (NoOverlap, Overlap ou Overflow), optimisant ainsi simultanément le temps de calcul et le taux de compression.

4. Gestion des nombres négatifs

Dans l'état actuel, les implémentations BitPacking considèrent uniquement des entiers positifs. L'introduction de nombres négatifs pose un problème, car le calcul du bitWidth se base sur la valeur maximale du tableau, sans tenir compte des valeurs négatives. Utiliser directement des entiers signés provoquerait des erreurs lors du décalage de bits et de l'accès direct via `get(i)`, puisque le bit de signe serait interprété comme une donnée.

Pour résoudre ce problème, plusieurs stratégies peuvent être envisagées. La solution la plus simple consiste à décaler toutes les valeurs pour qu'elles deviennent positives, en ajoutant à chaque élément la valeur absolue minimale du tableau. Une autre approche consiste à utiliser une représentation en complément à deux pour chaque entier, ce qui permet de conserver la sémantique des nombres signés tout en continuant à utiliser les opérations de bitpacking. Cette dernière solution est plus élégante mais nécessite d'adapter le calcul du bitWidth et le masquage dans toutes les méthodes (`compress`, `decompress` et `get`).

Enfin, si la distribution des valeurs contient à la fois de très grandes valeurs positives et négatives, l'usage d'une zone de débordement peut également s'appliquer aux nombres négatifs, en séparant les valeurs extrêmes pour limiter le gaspillage de bits dans le flux principal. L'adoption d'une de ces méthodes permettrait d'étendre le projet pour gérer des tableaux d'entiers complets, tout en conservant l'accès direct et la densité de compression.

5. Utilisation du programme

Le programme peut être exécuté directement à partir du JAR généré par Maven. Après compilation avec `mvn clean package`, le fichier `target/BitPacking-1.0.jar` contient l'application prête à l'emploi.

Pour lancer tous les benchmarks présents dans le dossier `data/in` :

```
java-jar target/BitPacking-1.0.jar
```

Pour traiter un fichier CSV spécifique :

```
java-jar target/BitPacking-1.0.jar "chemin/vers/fichier.csv"
```

Les résultats sont affichés à l'écran et sauvegardés dans le dossier `data/out`. Les tests unitaires peuvent être exécutés avec :

```
mvn test
```

Pour plus d'informations et d'exemples d'utilisation, se référer au **`README.md`** à la racine projet.

Conclusion

Le projet a permis de concevoir et d'implémenter plusieurs algorithmes de compression d'entiers, chacun optimisé pour différents scénarios : compression stricte sans chevauchement, compression avec chevauchement pour un gain d'espace, et gestion des valeurs extrêmes via une zone de débordement. Les tests unitaires ont validé la fiabilité, la réversibilité et la cohérence des méthodes, tandis que les benchmarks ont mis en évidence les compromis entre taux de compression et temps de traitement.

Les résultats montrent que chaque approche a ses forces : le NoOverlap offre des accès directs rapides, le Overlap réduit l'espace mémoire pour des données homogènes, et le Overflow gère efficacement les valeurs rares très grandes. Le protocole de mesure, répétitif et structuré, permet de quantifier le point où la compression devient avantageuse en termes de latence.

Enfin, le projet reste extensible pour traiter les nombres négatifs et d'autres types de données, et la documentation associée fournit toutes les instructions pour exécuter et tester les programmes. Ces travaux constituent ainsi une base solide pour l'optimisation de la transmission de tableaux d'entiers dans des systèmes distribués.