Compression d’entiers par bit-packing avec accès aléatoire

Conception, implémentation et évaluation d’un mini-projet Java

Auteur : Pierre CONSTANTIN, Sous la direction de Monsieur J.C Régin

Numéro étudiant : 22116789

Date : 02/11/2025

Sommaire

1. Introduction

Ce rapport présente un projet de compression d'entiers par bit-packing réalisé en Java, avec un accent particulier sur l'accès aléatoire O(1) à la i-ème valeur après compression. L'objectif est double : diminuer l'empreinte mémoire/stockage tout en conservant une capacité de lecture directe sans décompresser l'ensemble du bloc. Le projet comprend :

- Une petite bibliothèque de compression/décompression avec trois variantes.
- Une en-tête auto-décrite (header) pour encapsuler le format.
- Des utilitaires d'E/S binaires simples.
- Une démo interactive, un micro-benchmark et une CLI de benchmark.
- Un jeu de tests unitaires et une configuration Maven minimaliste.

Le travail suit les standards Maven (src/main/java, src/test/java), se construit avec un JDK 21+ et n'embarque aucune dépendance lourde.

2. Contexte et objectifs du projet

Le besoin initial : manipuler de grands tableaux d'entiers (int32) de manière compacte, sans renoncer à l'accès direct à une valeur arbitraire. Les usages typiques incluent :

- Des index ou identifiants entiers compressés pour diminuer la mémoire vive.
- Des colonnes d'entiers dans des formats de stockage colonnes.
- Des trames/flux d'acquisition où les valeurs sont bornées ou présentent des régularités.

Objectifs concrets :

- Concevoir un format compact, accompagné d'un header auto-décrivant (magic, n, mode, k…).
- Proposer plusieurs variantes couvrant différents compromis compacité/rapidité.
- Garantir l'accès direct get(i) en O(1) pour toutes les variantes.
- Fournir une CLI de benchmark reproductible et une démo « clé en main ».

3. Vue d'ensemble de la solution

La solution repose sur une interface minimaliste (`IntCompressor`) offrant trois opérations :

- `compress(int[] input)` : produit un buffer d'entiers 32 bits (int[]) contenant un header et un payload bit-packé.
- `decompress(int[] packed)` : reconstruit le tableau d'origine.
- `get(int[] packed, int i)` : renvoie la i-ème valeur sans décompresser tout le tableau.

Trois variantes sont proposées :

- NO_CROSSING : pas de chevauchement de valeurs entre words ; simple et rapide, au prix de quelques bits « perdus » en fin de word.
- CROSSING : densité maximale quand 32 n'est pas multiple de k ; les valeurs peuvent chevaucher deux words.
- OVERFLOW : un schéma hybride qui encode la majorité des valeurs « petites » en k bits, et place les valeurs « hors bornes » (négatives ou très grandes) dans une zone dédiée.

Une en-tête (`Headers`) précède systématiquement le payload et contient les informations pour décoder : nombre d'éléments, mode, paramètre k, etc. Les opérations bit-à-bit sont factorisées dans `BitIO`.

4. Choix de conception et d'implémentation

4.1. Contrat et format

- Contrat unique (`IntCompressor`) pour uniformiser l'usage (compress/decompress/get).
- En-tête auto-décrite : permet de reconnaître la variante et ses paramètres sans contexte externe.
- Alignement 32 bits : le buffer de sortie est un `int[]`, plus pratique en Java qu'un flux de bits nu.

4.2. Convention de bits et accès aléatoire

- Convention LSB-first au sein de chaque word de 32 bits (bit 0 = LSB du word 0).

- Calcul direct de l'offset (i × k) pour `CROSSING` ; arithmétique de division/modulo pour retrouver le mot et la position.

- Pour `NO_CROSSING`, `per = floor(32/k)` détermine le nombre de valeurs par word, ce qui simplifie l'adressage et la lecture.

4.3. Variantes et compromis

- NO_CROSSING : favorise la simplicité et des accès `get(i)` très directs. Quelques bits peuvent rester inutilisés en fin de word.

- CROSSING : maximise la compacité si k ne divise pas 32, au prix de lectures/écritures qui peuvent toucher deux words.

- OVERFLOW : ajoute une zone d'overflow séparée et un bit (ou petit indicateur) pour signaler que la valeur courante est « déportée ». C'est la seule variante qui gère naturellement des valeurs négatives et des valeurs « larges ».

4.4. Usages auxiliaires et ergonomie

- `CompressorFactory` fournit une instanciation simple à partir d'un type/option.

- `DataIO` lit/écrit des tableaux d'entiers au format binaire `[n][v0]..[v(n-1)]`, utile pour rejouer facilement des benchmarks.

- Démos et CLI (Main, Benchmark, BenchCLI, SaveExample) : permettre d'essayer, mesurer, enregistrer/recharger.

4.5. Simplicité de build et compatibilité

- Environnement : Maven Wrapper + JDK 21+ ; aucun plugin exotique.

- Tests unitaires JUnit et micro-benchmark lean (sans frameworks lourds).

- Compatibilité out-of-the-box sur macOS/Linux/Windows, avec quelques précautions documentées.

5. Résumé des classes et des fonctions principales

Voici un aperçu synthétique (but et responsabilités) :

- `io.compress.intpack.IntCompressor` : interface commune du projet (compress, decompress, get). Garantit l'accès direct et la portabilité entre variantes.

- `io.compress.intpack.Headers` : lecture/écriture de l'en-tête (magic, n, mode, k, extras). Sert de contrat de décodage.

- `io.compress.intpack.BitIO` : primitives bas niveau pour écrire/lire k bits à partir/vers des words 32 bits, en LSB-first.

- `io.compress.intpack.BitPackingNoCrossing` : implémentation « valeurs par word », sans chevauchement ; rapide, simple, bon point de départ.

- `io.compress.intpack.BitPackingCrossing` : implémentation « bit offset continu » ; compacité améliorée quand k ∤ 32.

- `io.compress.intpack.BitPackingOverflow` : implémentation hybride avec zone d'overflow pour valeurs négatives/élevées.

- `io.compress.intpack.CompressionType` : énumération des variantes disponibles.

- `io.compress.intpack.CompressorFactory` : fabrique d'instances selon le type, utile pour la démo et les benchmarks.

- `io.compress.intpack.DataIO` : utilitaires fichiers (binaire) pour sauvegarder/charger des tableaux d'int.

- `demo.Main` : démo interactive ; génère une entrée (ou lit depuis la console), exécute les variantes, vérifie le round-trip et un exemple de `get(i)`.

- `demo.Benchmark` : micro-benchmark simple pour « prendre la température » (avec échauffement JIT/warmup).

- `demo.BenchCLI` : CLI de benchmark reproductible à partir de fichiers (TXT/CSV/BIN) et/ou de données synthétiques ; exporte ASCII + CSV.

- `demo.SaveExample` : petit exemple de sauvegarde/lecture sur disque avec `DataIO`.

- `demo.TestSuite` (et tests JUnit) : vérifications automatiques de non-régression.

6. Problèmes rencontrés et solutions

Ce projet m'a posé quelques problèmes :

- Le droit d'exécution du wrapper Maven (`./mvnw`) sur macOS/Linux : erreur « permission denied ». Solution : `chmod +x ./mvnw` (documenté dans le README).

- L'ccès réflexif aux méthodes package-private sur JDK récents : une `IllegalAccessException` est apparue dans la démo interactive lors d'appels via réflexion. Solution : activer l'accessibilité par `setAccessible(true)` pour les méthodes nécessaires dans `demo.Main`.

- L'équivalences d'arguments et quoting sous Windows (PowerShell vs CMD) : nécessité de documenter la syntaxe d'appels Maven/exec et la gestion des chemins avec espaces.

- Les avertissements Maven/JDK (accès natif jansi/hawtjni) : bénins avec JDK 21+ ; possibilité d'ajouter `MAVEN_OPTS=--enable-native-access=ALL-UNNAMED` pour les masquer.

- La découverte de la structure Maven par VS Code : le Java Language Server peut mal détecter les sources si l'espace de travail n'est pas « mavenisé ». Solution : paramètres VS Code et rechargement du projet (instructions fournies).

- La portabilité : s'assurer que le projet fonctionne de la même manière avec Java 21 et au-delà (tests et CI sur Linux).

- Le debug lié aux effets de bord de certaines fonctions d'une classe à l'autre (surtout au début du projet)

Au final, les obstacles ont été levés par une combinaison de correctifs ciblés et de documentation précise dans le README.

7. Représentation des nombres négatifs : options et impacts

Il existe deux approches principales pour encoder des entiers signés dans un schéma de bit-packing :

7.1. Variante OVERFLOW (implémentée) :

- Idée : choisir un paramètre k pour couvrir efficacement la majorité des valeurs « petites ». Pour chaque valeur, un petit indicateur signale si elle est stockée « inline » (k bits) ou bien « déportée » dans une zone d'overflow.

- Avantages : gère naturellement les valeurs négatives et les valeurs très grandes, sans changer la représentation des valeurs « petites ».

- Coûts : un léger surcoût en bits (indicateur + pointeur/index vers l'overflow) et une complexité accrue dans `get(i)` (potentiellement une redirection)

7.2. Encodage ZigZag (optionnel, non imposé) :

- Idée : transformer un entier signé en entier non signé qui regroupe petits positifs et petits négatifs autour de zéro (par exemple: `0→0, -1→1, 1→2, -2→3, 2→4, ...`).

- Avantages : permet d'utiliser une variante « non overflow » (NO_CROSSING ou CROSSING) sur des données signées, avec d'excellentes densités lorsque les valeurs sont centrées.

- Coûts : un pré/post-traitement (ZigZag <→ signé), et rester vigilant sur les bornes et débordements.

Dans ce projet, `BitPackingOverflow` illustre la première approche (zone dédiée). L'encodage ZigZag reste une piste si l'on souhaite éviter une zone d'overflow, au prix d'un traitement supplémentaire.

8. Performances et méthodologie de benchmark

8.1. Méthodologie

- Données : soit synthétiques (par taille et distribution contrôlées), soit chargées depuis des fichiers texte/CSV ou binaires (`DataIO`).

- Mesures : latences médianes et étendue interquartile (IQR) pour la compression et la décompression, afin de lisser les bruits (GC, JIT, activité système).

- Échauffement JIT (warmup) : quelques itérations non mesurées pour stabiliser les timings.

- Exports : un tableau ASCII lisible en console et un CSV pour exploitation ultérieure.

8.2. Tendances observées (qualitatives)

- Compacité : `CROSSING` atteint souvent le meilleur ratio lorsque k ne divise pas 32. `NO_CROSSING` est légèrement moins compact mais très régulier. `OVERFLOW` est proche de `NO_CROSSING` tant que la part d'overflow reste faible.

- Latences : `NO_CROSSING` est généralement très rapide en `get(i)` grâce à son adressage direct. `CROSSING` peut être légèrement plus coûteux en lecture/écriture quand une valeur chevauche deux words. `OVERFLOW` ajoute un coût conditionnel en cas d'accès à une valeur « déportée ».

- Sensibilité aux données : de faibles k (valeurs fortement bornées) améliorent la compacité, mais peuvent augmenter le coût de l'overflow si la distribution est plus large que prévu.

8.3. Reproductibilité

- Les commandes Maven/Wrapper fournies dans le README permettent de rejouer aisément les benchmarks sur différentes machines.

- Les options de `BenchCLI` (nombre de runs, source des données, sorties CSV) facilitent la comparaison et l'archivage des résultats.

9. Limites, pistes d'amélioration et travaux futurs

- Sélection automatique de k : introduire un estimateur de k optimal (ou quasi optimal) à partir d'un échantillon (quantiles, entropie simple, etc.).

- Vectorisation : étudier une implémentation exploitant les intrinsics (Panama, Vector API) pour accélérer les chemins « chauds ».

- Multi-k adaptatif : partitionner le tableau en blocs avec k variables (par blocs homogènes), en conservant l'accès O(1) via un index secondaire.

- Gestion avancée des entiers signés : expérimenter un ZigZag systématique et mesurer l'impact sur la densité et les timings.

- Outillage : enrichir les tests de performance (profils CPU, profils GC) et ajouter des jeux de données publics pour la comparaison.

10. Conclusion

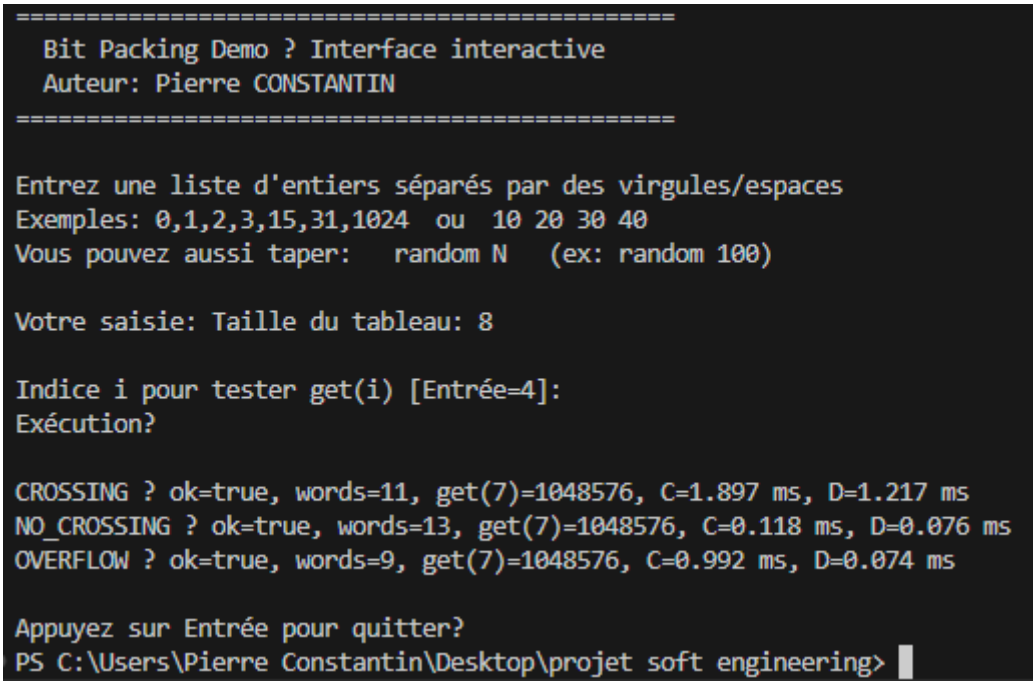
Le projet démontre qu'un schéma de bit-packing soigné permet d'obtenir des gains notables de compacité tout en conservant un accès aléatoire O(1). Les trois variantes proposées couvrent un bon éventail de compromis : simplicité et vitesse (`NO_CROSSING`), compacité (`CROSSING`), et support natif des valeurs négatives (« grandes ») via une zone dédiée (`OVERFLOW`).

L'ergonomie (header auto-décrit, utilitaires `DataIO`, CLI de benchmark) et la simplicité de build (Maven Wrapper, JDK 21+) rendent l'ensemble facile à essayer et à intégrer. Les quelques difficultés rencontrées (droits d'exécution, accès réflexif, quoting Windows) ont été résolues et documentées.

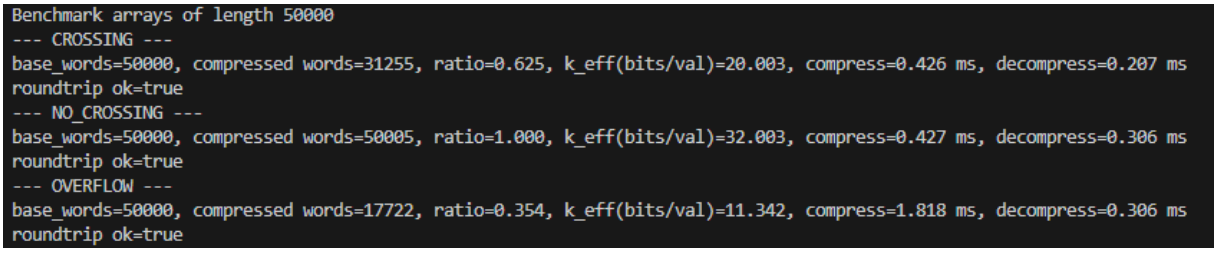
À court terme, l'amélioration la plus rentable serait l'estimation automatique de k et une évaluation plus poussée sur des jeux de données variés. À moyen terme, des pistes de vectorisation et de multi-k par blocs pourraient encore améliorer le compromis compacité/latence.

Annexe :

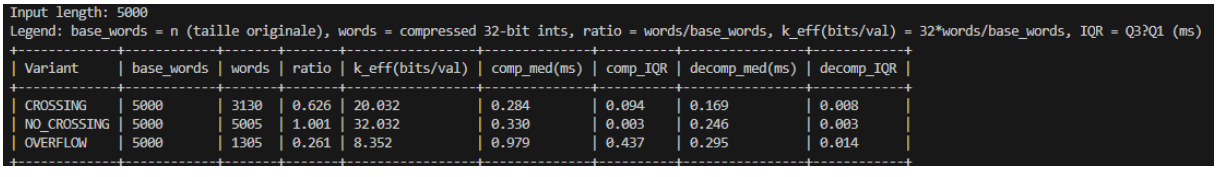
Voici quelques captures du terminal lors de l'utilisation du programme :



Document 1 – Interface du main



Document 2 – Interface du benchmark (automatique, compare les 3 méthodes)



Document 3 – Interface du benchmark CLI (répétable, prend un tableau en entrée, produit un tableau lisible en sortie)