Rapport de Projet de Software Engineering 2025

Compression d'Entiers pour la Transmission de Données

[Khourta Sofiane]

Contents

1 Introduction

La transmission efficace de grands volumes de données, tels que les tableaux d'entiers, est un enjeu majeur sur Internet. L'objectif de ce projet était de compresser ces tableaux pour réduire le nombre d'entiers à transmettre, et donc en accélérer le transfert.

L'idée principale est d'utiliser le **Bit Packing** : au lieu de stocker chaque entier sur 32 bits, seul le nombre de bits k nécessaire pour le plus grand élément du tableau est utilisé.

La contrainte fondamentale était de **ne pas perdre l'accès direct** aux éléments. Même après compression, le i-ème entier de l'array original doit pouvoir être retrouvé sans avoir à décompresser l'intégralité du tableau.

Ce rapport présente les trois solutions implémentées (NoOverlap, Overlap, et Overflow), l'architecture logicielle qui les lie, et une analyse des performances basée sur des benchmarks concrets pour déterminer quand la compression est rentable.

Lien vers le dépôt GitHub : [https://github.com/Sofiane949/BitPacking$_{SoftwareEngineering}$]

2 Architecture de la Solution

Pour garantir la maintenabilité et la modularité du projet, une architecture basée sur une interface et une Factory a été choisie.

2.1 Interface `BitPacker`

Pour traiter les différentes méthodes de compression de façon uniforme, une interface `BitPacker` a été définie. Elle garantit que chaque classe de compression implémente les mêmes méthodes de base.

```
public interface BitPacker {
    void compress(int[] input);
    void decompress(int[] output);
    int get(int i);
    int[] getCompressedData();
}
```

Listing 1: Interface BitPacker.java

2.2 Usine de Création (`BitPackerFactory`)

Conformément au sujet, une "Factory" gère la création des objets compresseurs. Le programme principal n'a qu'à demander un type de compression (via une `enum`), et la Factory lui fournit le bon objet, sans que le `Main` ait besoin de connaître les détails.

```
public class BitPackerFactory {

    public enum CompressionType {
        NO_OVERLAP,
        WITH_OVERLAP,
        OVERFLOW_NO_OVERLAP,
        OVERFLOW_WITH_OVERLAP
    }

    public static BitPacker create(CompressionType type) {
        switch (type) {
            case NO_OVERLAP:
                return new BitPackerNoOverlap();
            // ... etc.
        }
    }
}
```

Listing 2: Extrait de BitPackerFactory.java

3 Implémentation des Méthodes de Compression

Trois logiques de compression distinctes ont été implémentées.

3.1 `BitPackerNoOverlap`

C'est la version la plus simple. Elle n'autorise pas qu'un entier compressé soit écrit "à cheval" sur deux entiers de 32 bits.

Logique Le k (bits max) est calculé, ainsi que le nombre d'entiers que l'on peut stocker dans un `int` de 32 bits ($nPerInt = \lfloor 32/k \rfloor$). Chaque `int` de sortie est ensuite rempli avec $nPerInt$ entiers d'entrée, en utilisant des décalages de bits ('«').

L'accès `get(i)` est très rapide : il suffit de trouver l'index ($i/nPerInt$) et la position ($i \,\%\, nPerInt$), puis d'utiliser un décalage ('»') et un masque ('&') pour isoler la valeur.

Inconvénient Cette méthode gaspille de l'espace. Si $k = 9$, on ne peut stocker que 3 entiers ($3 \times 9 = 27$ bits), et $32 - 27 = 5$ bits sont perdus sur chaque `int` du tableau compressé.

3.2 `BitPackerWithOverlap`

Cette version est la plus efficace en termes d'espace. Elle traite le tableau compressé comme un seul et long flux de bits. Un entier peut commencer sur les derniers bits d'un `int` et se terminer sur les premiers bits du suivant.

Logique On ne raisonne plus en $nPerInt$. Une position binaire globale $bitPos = i \times k$ est suivie. L'écriture et la lecture impliquent de calculer un $index$ dans le tableau et un $shift$ (décalage) à l'intérieur de cet $index$.

Si l'entier déborde ($shift + k > 32$), la première partie doit être écrite dans $compressed[index]$ et la seconde (les bits de poids fort) dans $compressed[index + 1]$.

Point technique (Java) Une attention particulière a été portée à deux pièges en Java lors des manipulations de bits : 1. **Dépassement de capacité :** $i \times k$ peut dépasser la limite d'un `int`. Les calculs de position (comme $bitPos$) doivent donc être faits en `long`. 2. **Extension de signe :** Si un `int` compressé a un '1' comme premier bit, il est vu comme négatif. Un décalage ('»') propagerait des '1'. Le décalage non-signé ('»>') et un masque '& 0xFFFFFFFFL' ont dû être utilisés pour forcer Java à traiter l'entier comme s'il était non-signé.

3.3 `BitPackerOverflow`

Cette méthode répond au problème du gaspillage : si un seul "gros" nombre force $k = 30$, mais que 99% des autres nombres sont petits ($k = 3$), on gaspille énormément.

Choix de conception Plutôt que de ré-implémenter la logique, le principe de **Composition** a été utilisé. Cette classe "enveloppe" un autre compresseur (par exemple, `BitPackerOverlap`) et lui sert de "traducteur".

Logique de compression (3 passes) 1. **Analyse :** Le k de chaque nombre est calculé. Un seuil k_{normal} est choisi (ici, la médiane des k). 2. **Comptage :** Les valeurs "hors normes" (outliers) sont comptées et stockées dans un tableau séparé $compressedOverflow[]$. On calcule k_{index}, le nombre de bits pour stocker l'index de ce tableau. 3. **Traduction :** Un nouveau tableau "à compresser" est créé. Il est rempli avec des valeurs "packées" de $k_{packed} = 1 + \max(k_{normal}, k_{index})$ bits.

- Si la valeur est normale : on écrit `0-valeur`.
- Si la valeur est un outlier : on écrit `1-index`.

4. Ce nouveau tableau "traduit" est donné au compresseur interne (`BitPackerOverlap`) qui le compresse.

Logique d'accès (`get(i)`) L'accès est simple : on demande la valeur packée v au compresseur interne. On regarde le premier bit (le drapeau). Si c'est 0, on retourne le reste de v. Si c'est 1, on utilise le reste de v comme index pour aller chercher la vraie valeur dans $compressedOverflow[]$.

4 Benchmarks et Analyse de Performance

4.1 Protocole de Mesure

Pour mesurer les performances, une classe `BenchmarkRunner` a été utilisée. Le protocole est le suivant : 1. **Génération de Données :** Création de grands tableaux (1 million d'entiers) avec différents profils. 2. **Chauffe (Warm-up) :** Les fonctions sont exécutées 20 fois "à vide" avant de mesurer. C'est vital pour que la JVM (Java) ait le temps d'optimiser le code (compilation JIT). 3. **Mesure :** `System.nanoTime()` est utilisé pour une mesure précise, répétée 50 fois. 4. **Moyenne :** Le temps affiché est la moyenne de ces 50 exécutions.

4.2 Scénarios de Test

- **CAS 1 (Petites Données) :** Entiers ≤ 255 (k=8).
- **CAS 2 (Moyennes Données) :** Entiers ≤ 1000000 (k=20).
- **CAS 3 (Overflow) :** 99% de petites données, 1% de très grandes données.

4.3 Résultats des Benchmarks

Voici les résultats obtenus sur un tableau de 1 000 000 d'entiers. La taille est estimée en nombre d'entiers de 32 bits.

Table 1: Temps d'exécution (1 million d'entiers) et Taille de sortie (estimée)

Scénario	Compresseur	Taille (est.)	Tps Compress	Tps Decompress	Tps Get(i)
Cas 1 (Petites)	Original	1 000 000	0 ms	0 ms	-
(k=8)	NoOverlap	250 000	11 ms	2 ms	81 ns
	Overlap	250 000	13 ms	1 ms	127 ns
	Overflow	250 000	34 ms	6 ms	150 ns
Cas 2 (Moyennes)	Original	1 000 000	0 ms	0 ms	-
(k=20)	NoOverlap	1 000 000	17 ms	2 ms	32 ns
	Overlap	625 000	18 ms	2 ms	155 ns
	Overflow	1 000 000	100 ms	6 ms	152 ns
Cas 3 (Overflow)	Original	1 000 000	0 ms	0 ms	-
(99% k=8, 1% k=30)	NoOverlap	1 000 000	15 ms	2 ms	32 ns
	Overlap	937 500	14 ms	2 ms	32 ns
	Overflow	**478 000**	34 ms	6 ms	36 ns

Analyse des résultats

- **Temps CPU :** La méthode `Overflow` est toujours la plus lente à compresser (34-100ms) à cause de ses multiples passes d'analyse. Les méthodes `Overlap` et `NoOverlap` sont très rapides (11-18ms).

- **Cas 1 et 2 :** Dans les cas où les données sont "uniformes" (petites ou moyennes), `Overflow` est la moins bonne solution. Elle est lente et ne compresse pas mieux. Dans le Cas 2, `Overlap` est clairement le meilleur (taille de 625k).
- **Cas 3 :** C'est le cas crucial. Les méthodes simples (`NoOverlap`, `Overlap`) sont forcées d'utiliser un $k = 30$ pour tout le monde, à cause du 1% de "grosses" valeurs. Leur compression est donc très mauvaise (Taille $\approx$ 937k-1M). C'est là que `Overflow` brille : en payant un coût CPU (34ms), il atteint une **taille de sortie estimée à 478k entiers**, écrasant les autres méthodes.

5 Analyse de Rentabilité de la Transmission

Le sujet demande quand la compression devient rentable. Analysons les temps. Soit B la bande passante du réseau (en bits/sec) et t la latence (en sec).

Temps sans compression

$$T_{original} = \frac{\text{Taille}_{orig}}{B} + t$$

Temps avec compression

$$T_{compresse} = T_{cpu} + \frac{\text{Taille}_{comp}}{B} + t$$

(où $T_{cpu} = T_{compression} + T_{decompression}$)

Calcul du seuil de rentabilité La compression est rentable si $T_{compresse} < T_{original}$.

$$T_{cpu} + \frac{\text{Taille}_{comp}}{B} + t < \frac{\text{Taille}_{orig}}{B} + t$$

Il est à noter que la latence t s'annule des deux côtés. **La latence n'a donc aucun impact sur la rentabilité de la compression.** Le facteur déterminant est la bande passante B.

L'inéquation devient :

$$T_{cpu} < \frac{\text{Taille}_{orig} - \text{Taille}_{comp}}{B}$$

$$B < \frac{\text{Taille}_{orig} - \text{Taille}_{comp}}{T_{cpu}}$$

Ceci définit le **seuil de bande passante** B_{seuil} en dessous duquel la compression est rentable.

$$B_{seuil} = \frac{\text{Gain de Taille (bits)}}{TempsCPU(sec)}$$

Exemple concret (Cas 3 : Overflow vs Overlap) Dans le Cas 3, un compresseur "simple" (Overlap) est rapide mais produit un gros fichier, tandis que le compresseur "intelligent" (Overflow) est lent mais produit un petit fichier. Comparons-les :

- $\text{Taille}_{Overlap} \approx 937500 \times 32 = 30000000$ bits
- $\text{Taille}_{Overflow} \approx 478000 \times 32 = 15296000$ bits
- Gain de Taille = 14704000 bits
- $T_{cpu(Overlap)} = 14\text{ms} + 2\text{ms} = 16\text{ms}$
- $T_{cpu(Overflow)} = 34\text{ms} + 6\text{ms} = 40\text{ms}$
- Coût CPU supplémentaire = 40ms − 16ms = 24ms = 0.024s

Le coût CPU de 24ms est "rentable" si le gain de temps sur le réseau est supérieur.

$$B_{seuil} = \frac{\text{Gain de Taille (bits)}}{\text{Coût CPU (sec)}} = \frac{14704000 \text{ bits}}{0.024 \text{ s}}$$

$$B_{seuil} \approx 612666666 \text{ bits/s} \approx 612 \text{ Mbits/s}$$

Conclusion : Dans ce scénario, si la bande passante est inférieure à ≈ 612 Mbits/s, il est plus rapide d'utiliser la méthode `Overflow`, même si elle prend plus de temps de calcul.

6 Conclusion

Ce projet a été l'occasion d'implémenter un système de compression d'entiers modulaire et performant. Trois stratégies distinctes (NoOverlap, Overlap, Overflow) ont pu être codées, en respectant une interface commune et en utilisant une usine pour leur instanciation. Un protocole de benchmark a également été mis en place pour les évaluer.

L'analyse des performances a bien montré qu'il n'y a pas de "meilleur" compresseur universel. La méthode la plus rapide (Overlap) n'est pas toujours celle qui compresse le mieux.

Finalement, la solution `BitPackerOverflow` (utilisant `BitPackerOverlap` en interne) s'est révélée être la plus "intelligente" et la plus robuste. Elle est la seule capable de s'adapter à des données non-uniformes (le "Cas 3"), qui est un scénario très probable dans des cas d'utilisation réels. Son coût CPU supplémentaire est, comme calculé précédemment, largement compensé par le gain en taille de transmission sur la plupart des réseaux actuels.