Rapport Projet
Compression d'Entiers par Bit Packing

Software Engineering Project 2025

Rémi Mathieu (RemiM06)

Novembre 2025

Résumé

Ce rapport présente une étude complète de la compression d'entiers utilisant la technique du Bit Packing. Trois stratégies sont implémentées et analysées : avec chevauchement (Overlap), sans chevauchement (No-Overlap) et avec zone de débordement (Overflow). L'objectif est de réduire la taille des tableaux transmis sur le réseau tout en conservant l'accès direct aux éléments. Les benchmarks et analyses montrent que le choix de la stratégie dépend fortement du contexte applicatif (CPU vs bande passante).

Table des matières

1 Introduction

1.1 Problématique

La transmission de tableaux d'entiers sur le réseau est un problème central en informatique distribuée. Dans les applications temps réel, sur réseau limité ou en streaming, la bande passante est souvent le goulot d'étranglement.

La problématique centrale de ce projet est :

— Réduire la taille des données transmises sans perte d'information
— Maintenir l'accès direct aux éléments (pas de décompression complète)
— Équilibrer le coût CPU de compression/décompression avec l'économie de bande passante

1.2 Objectifs

1. Implémenter plusieurs stratégies de compression Bit Packing
2. Mesurer et comparer les performances (temps et ratio)
3. Établir les conditions de rentabilité (break-even latency)
4. Fournir des recommandations d'utilisation selon le contexte

1.3 Contraintes

— Ne traiter que les entiers positifs (32-bit)
— Accès aléatoire O(1) au i-ème élément
— Pas de perte de données
— Code performant et facilement maintenable

2 Analyse des Stratégies

2.1 Stratégie 1 : Overlap (Avec Chevauchement)

2.1.1 Principe

Les entiers compressés peuvent s'étendre sur deux int32 consécutifs. Cette approche utilise chaque bit disponible sans gaspillage.

2.1.2 Illustration

Pour 6 entiers nécessitant 12 bits chacun :

```
int32[0] : [elem0: 12 bits | elem1: 12 bits | elem2: 8 bits]
int32[1] : [elem2: 4 bits | elem3: 12 bits | elem4: 12 bits | elem5: 4 bits]
int32[2] : [elem5: 8 bits | ...................]
```

2.1.3 Avantages

— **Meilleur ratio** : Aucun bit de remplissage (padding)
— Utilisation optimale de l'espace : $n \times k$ bits pour n éléments de k bits
— Économies maximales pour la bande passante

2.1.4 Inconvénients

— **Compression 3x plus lente** : Logique complexe de placement bit-à-bit
— Code plus difficile à maintenir et déboguer
— Accès plus complexe (deux lectures potentielles par élément)

2.1.5 Complexité

— Compression : $O(n)$ mais avec coefficient élevé
— Décompression : $O(n)$
— Accès get(i) : $O(1)$ mais avec opérations bit complexes

2.2 Stratégie 2 : No-Overlap (Sans Chevauchement)

2.2.1 Principe

Chaque entier compressé reste complètement dans un seul int32. Les bits non utilisés sont du padding.

2.2.2 Illustration

Pour 6 entiers nécessitant 12 bits chacun :

```
int32[0] : [elem0: 12 bits | elem1: 12 bits | padding: 8 bits]
int32[1] : [elem2: 12 bits | elem3: 12 bits | padding: 8 bits]
int32[2] : [elem4: 12 bits | elem5: 12 bits | padding: 8 bits]
```

2.2.3 Avantages

— **Compression 3x plus rapide** : Logique simple et directe
— Code simple et facilement testable
— Accès get(i) très direct et prévisible
— Meilleure localité mémoire (alignement des données)

2.2.4 Inconvénients

— **Ratio 15% moins bon** : Padding obligatoire
— Gaspillage de bande passante
— Non optimal pour les données avec valeurs très variables

2.2.5 Complexité

— Compression : $O(n)$ avec coefficient bas
— Décompression : $O(n)$
— Accès get(i) : $O(1)$ simple

2.3 Stratégie 3 : Overflow (Avec Zone de Débordement)

2.3.1 Principe

Séparer les données en deux zones : une zone principale pour les valeurs "normales" (90e percentile) et une zone overflow pour les outliers.

2.3.2 Illustration

Pour `[1, 2, 3, 1024, 4, 5, 2048]` :

```
Seuil (90e percentile) = 5
Valeurs normales : [1, 2, 3] → 2 bits
Outliers : [1024, 2048] → indexés par 1 bit

Zone principale : [0-1, 0-2, 0-3, 1-0, 0-4, 0-5, 1-1]
                  (flag de 1 bit + index/valeur de 1 ou 2 bits)
Zone overflow : [1024, 2048]
```

2.3.3 Avantages

— Adapté aux distributions non-uniformes (queues épaisses)
— Bonne compression même avec quelques outliers
— Flexible : percentile configurable

2.3.4 Inconvénients

— Plus complexe à implémenter
— Nécessite un tri pré-compression
— Intérêt limité si données uniformes
— Accès get(i) peut nécessiter deux niveaux d'indirection

2.3.5 Complexité

— Compression : $O(n \log n)$ (tri requis)
— Décompression : $O(n)$
— Accès get(i) : $O(1)$ mais avec vérification de flag

3 Choix Architecturaux

3.1 Pattern Factory

3.1.1 Motivation

Créer une abstraction commune pour les trois stratégies permet :
— Changer de stratégie au runtime
— Tester indépendamment chaque implémentation
— Ajouter nouvelles stratégies sans modifier le code existant

3.1.2 Implémentation

```java
public interface BitPacking {
    int[] compress(int[] array);
    int[] decompress();
    int get(int index);
    int getOriginalSize();
    int getBitsPerElement();
```

```
    int getCompressedSize();
    double getCompressionRatio();
    CompressionType getType();
}

public class BitPackingFactory {
    public static BitPacking create(CompressionType type) {
        switch(type) {
            case OVERLAP: return new BitPackingOverlap();
            case NO_OVERLAP: return new BitPackingNoOverlap();
            case OVERFLOW: return new BitPackingOverflow();
            default: throw new IllegalArgumentException();
        }
    }
}
```

3.2 Calcul des Bits Nécessaires

3.2.1 Algorithme

Pour représenter une valeur max en binaire, le nombre de bits minimum est :

$$\text{bits} = \lfloor \log_2(max) \rfloor + 1 = 32 - \text{clz}(max)$$

où `clz` est le nombre de zéros de tête (leading zeros).

```
public static int calculateBitsNeeded(int max) {
    if (max == 0) return 1;
    return 32 - Integer.numberOfLeadingZeros(max);
}
```

3.2.2 Exemple

— $max = 15 = 0b1111$: 4 bits
— $max = 255 = 0b11111111$: 8 bits
— $max = 1024 = 0b10000000000$: 11 bits

3.3 Gestion des Accès Aléatoires

Chaque stratégie offre une méthode `get(int i)` permettant de récupérer le i-ème élément sans décompression complète.

3.3.1 No-Overlap

```
public int get(int i) {
    int intIndex = i / elementsPerInt32;
    int elementPos = i % elementsPerInt32;
    int bitPos = elementPos * bitsPerElement;

    int mask = (1 << bitsPerElement) - 1;
    return (compressed[intIndex] >> bitPos) & mask;
}
```

Complexité : $O(1)$, une seule lecture int32

3.3.2 Overlap

```java
public int get(int i) {
    int bitStart = i * bitsPerElement;
    int intIndex = bitStart / 32;
    int bitPos = bitStart % 32;
    int bitsLeft = 32 - bitPos;

    if (bitsLeft >= bitsPerElement) {
        // Tout dans un int32
        return (compressed[intIndex] >> bitPos) & mask;
    } else {
        // Chevauchement sur deux int32
        int basse = compressed[intIndex] >> bitPos;
        int haute = compressed[intIndex + 1] & mask2;
        return basse | (haute << bitsLeft);
    }
}
```

Complexité : $O(1)$ mais avec branchement potentiel

4 Résultats Expérimentaux

4.1 Protocol de Benchmark

4.1.1 Méthodologie

1. Générer un dataset : 10 000 entiers - 95% entre 0-100 (distribution normale) - 5% outliers 10 000 (distribution extrême)

2. Mesurer avec `System.nanoTime()` : - Temps de compression - Temps de décompression - Temps de 1 000 accès aléatoires get()

3. Calculer métriques : - Taille compressée - Ratio de compression - Seuil de rentabilité transmission

4.1.2 Précision

— Utilisation de `nanoTime()` : résolution nanoseconde
— Conversion en microsecondes pour lisibilité
— Moyenne sur plusieurs passages (stabilisation)

4.2 Résultats : Comparaison Overlap vs No-Overlap

4.2.1 Interprétation

— **Compression** : No-Overlap est clairement plus rapide (3x) grâce à sa simplicité logique
— **Accès get()** : Performance pratiquement identique (branchement prédit avec succès dans les deux cas)
— **Ratio** : Overlap économise 635 int32 ($635 \times 4 = 2540$ octets) pour ce dataset

Métrique	Overlap	No-Overlap	Écart
Temps compression	2406 µs	835 µs	3.0x plus lent
Temps décompression	918 µs	654 µs	1.4x plus lent
Temps get() moyen	110 ns	114 ns	Similaire
Ratio compression	2.29	2.00	14.5% meilleur
Taille compressée	4365 int32	5000 int32	635 int32 savings

TABLE 1 – Comparaison des stratégies de compression

4.3 Analyse de Rentabilité : Break-Even Latency

4.3.1 Formule Théorique

Soit :
— T_c : temps de compression
— S_o : taille originale (nombre d'int32)
— S_c : taille compressée
— L : latence de transmission par int32
Le seuil de rentabilité est atteint quand :

$$T_c + S_c \times L = S_o \times L$$

Résolvant pour L :

$$L > \frac{T_c}{S_o - S_c}$$

4.3.2 Application Numérique

Pour No-Overlap :
— $T_c = 835 \times 10^3$ ns
— $S_o = 10000$ int32
— $S_c = 5000$ int32
— Économie : 5 000 int32

$$L_{\min} = \frac{835 \times 10^3}{5000} = 167 \text{ ns/int32} = 0.167 \text{ µs/int32}$$

Pour Overlap :

$$L_{\min} = \frac{2406 \times 10^3}{5635} = 427 \text{ ns/int32} = 0.427 \text{ µs/int32}$$

4.3.3 Contextes de Rentabilité

5 Justification des Choix

5.1 Pourquoi Trois Stratégies ?

Chacune répond à un besoin différent :
— **No-Overlap** : CPU-bound, applications temps réel
— **Overlap** : Bande passante critique

Contexte	Latence typique	Rentable ?
Communication intra-processus	¡10 ns	Non
Communication inter-threads	10-50 ns	Non
Boucle locale réseau (LAN)	1-10 µs	**OUI**
Réseau WAN/Internet	50-200 µs	**OUI**
Transmission satellite	¿100 ms	**OUI**

Table 2 – Rentabilité selon le contexte de transmission

— **Overflow** : Données avec distribution inégale
Plutôt qu'une solution unique, la Factory permet de choisir au runtime.

5.2 Pourquoi cette Architecture ?

1. **Interface BitPacking** : Contrat clair, facilitant les tests
2. **Factory Pattern** : Flexibilité de sélection
3. **Séparation des concerns** : Chaque classe une responsabilité unique
4. **BitPackingUtils** : Logique commune centralisée

5.3 Mesures de Performance

L'utilisation de `System.nanoTime()` est justifiée car :
— Précision suffisante pour benchmarks (nanoseconde)
— Immunisé aux ajustements d'horloge système
— Disponible sur toutes les JVM
— Permet de mesurer des opérations très rapides

6 Conclusion

6.1 Synthèse

Ce projet a validé plusieurs hypothèses clés :

1. **Trade-off CPU-Bande Passante** : No-Overlap sacrifie 15% de ratio pour 3x de vitesse
2. **Rentabilité Réelle** : Compression devient rentable dès latences réseau normales (¿1 µs)
3. **Accès Aléatoire** : Implémentable sans surcoût significatif avec les trois stratégies
4. **Flexibilité Architecturale** : Pattern Factory permet une sélection adaptée au contexte

6.2 Recommandations d'Utilisation

Contexte	**Stratégie**	**Justification**
Transmissions répétées, latence basse (¡10 µs)	No-Overlap	3x plus rapide
Transmission longue distance, bande passante limitée	Overlap	15% meilleur ratio
Données avec outliers importants	Overflow	Compression adaptative
Prototype ou test initial	No-Overlap	Plus simple à déboguer

TABLE 3 – Guide de sélection de stratégie