Software Engineering Project 2025

Projet BitPacking - Rapport Technique

Mathis Levesque du rostu

1. Introduction

Le projet BitPacking a pour objectif d'optimiser la transmission de tableaux d'entiers en les compressant. L'idée est de réduire la taille mémoire nécessaire à la transmission tout en conservant un accès direct aux éléments, ce qui est essentiel dans les systèmes distribués ou les applications à faible latence.

Pour répondre à cette problématique, trois méthodes de compression ont été développées :

- **Méthode Split** : chaque entier est encodé dans un bloc de taille fixe (k bits) et inséré dans un tableau d'entiers de 32 bits, sans chevauchement. Cette méthode est simple et rapide, mais peut entraîner une perte d'espace si les blocs ne remplissent pas entièrement les 32 bits.
- **Méthode Overlap** : les entiers compressés peuvent chevaucher deux entiers de 32 bits, ce qui permet une utilisation plus efficace de l'espace mémoire. Cette méthode est plus complexe à implémenter, notamment pour l'accès direct aux éléments.
- **Méthode Overflow** : elle identifie les valeurs qui nécessitent un grand nombre de bits et les stocke dans une zone de débordement. Les valeurs normales sont encodées avec un bit de signal indiquant leur nature (normale ou débordement), permettant ainsi une compression plus efficace dans les cas où seules quelques valeurs sont très grandes.

Le projet a été structuré en plusieurs étapes :

1. **Analyse du problème** et définition des contraintes liées à la transmission efficace de données entières.
2. **Développement des trois méthodes de compression** en Java, chacune respectant la contrainte d'accès direct aux éléments.
3. **Intégration de deux design patterns** :
    - Le pattern Strategy pour encapsuler les différentes stratégies de compression derrière une interface commune.
    - Le pattern Factory pour instancier dynamiquement la méthode de compression choisie.
4. **Réalisation de benchmarks** sur des tableaux de tailles et de contenus variés, afin de comparer les performances (temps de compression, décompression, accès direct, ratio de compression).

2. Méthodes de compression

2.1 Méthode Split

La méthode **Split** repose sur une stratégie simple : chaque entier est encodé sur un nombre fixe de bits (bitSize), déterminé dynamiquement à partir de la valeur maximale du tableau. Le tableau compressé est constitué d'entiers de 32 bits, dans lesquels on insère les entiers compressés les uns à la suite des autres, sans chevauchement.

Implémentation :

- Le nombre d'entiers compressés pouvant tenir dans un entier de 32 bits est calculé via valuesPerInt = 32 / bitSize.
- Chaque entier est inséré dans le tableau compressé à l'aide d'un décalage (bitOffset) et d'un masque binaire.
- La méthode get(i) permet un accès direct à l'élément compressé en recalculant son index et son décalage dans le tableau compressé.
- La méthode calculateBitSize() détermine le nombre minimal de bits nécessaires pour représenter la plus grande valeur du tableau.

Avantages :

- Très rapide à exécuter.
- Accès direct très efficace.
- Implémentation simple et robuste.

Limites :

- Peut gaspiller de l'espace si les entiers ne remplissent pas complètement les 32 bits (ex. : 10 bits × 3 = 30 bits → 2 bits perdus).

2.2 Méthode Overlap

La méthode **Overlap** améliore l'efficacité de la compression en autorisant les entiers compressés à **chevaucher deux entiers de 32 bits**. Cela permet d'utiliser chaque bit disponible sans perte d'espace.

Implémentation :

- Chaque entier est inséré à la position bitPos, qui est incrémentée à chaque insertion.
- Si l'entier compressé dépasse les 32 bits restants dans l'entier courant, la partie restante est insérée dans l'entier suivant.
- La décompression et l'accès direct nécessitent de reconstruire les entiers à partir de deux blocs si un chevauchement est détecté.
- L'opérateur >>> (décalage logique à droite) est utilisé pour éviter les problèmes liés aux entiers signés.

Avantages :

- Meilleure densité de compression que Split.
- Réduction de la taille mémoire.

Limites :

- Complexité accrue dans la gestion des bits.
- Légère perte de performance sur les opérations get(i) et decompress().

2.3 Méthode Overflow

La méthode **Overflow** est conçue pour les tableaux contenant une majorité de petites valeurs et quelques valeurs très grandes. Elle permet d'éviter de fixer un bitSize trop grand pour tous les éléments.

Implémentation :

- Un bitSize est choisi pour encoder les valeurs normales.
- Une valeur spéciale (signalValue = 2^bitSize - 1) est réservée pour signaler un débordement.
- Si une valeur dépasse signalValue - 1, elle est stockée dans une liste overflow, et l'élément compressé contient signalValue suivi de l'index dans cette liste.
- Chaque élément du tableau compressé est encodé sur bitSize + 1 bits pour inclure le bit de signal.
- La méthode get(i) et decompress() reconstruisent les valeurs en vérifiant le bit de signal et accèdent à la zone de débordement si nécessaire.

Avantages :

- Très efficace pour les tableaux avec peu de valeurs extrêmes.
- Réduction significative de la taille compressée dans les cas hétérogènes.

Limites :

- Plus coûteux en temps de traitement.
- get(i) est plus lent car il nécessite une lecture conditionnelle et potentiellement un accès à la zone de débordement.

3. Design Patterns utilisés

Dans le cadre de ce projet, deux design patterns ont été utilisés pour structurer le code de manière claire, modulaire et extensible : **Strategy** et **Factory**.

3.1 Strategy Pattern

Le **Strategy Pattern** permet de définir une famille d'algorithmes, de les encapsuler dans des classes distinctes, et de les rendre interchangeables à l'exécution. Ce pattern favorise l'**ouverture à l'extension** et la **fermeture à la modification**, selon le principe SOLID.

Application dans le projet :

- Une interface BitPacking définit les méthodes communes à toutes les stratégies : compress(), decompress(), et get(i).
- Les classes BitPackingSplit, BitPackingOverlap et BitPackingOverflow implémentent cette interface, chacune avec sa propre logique de compression.
- Le code client (comme Main.java ou Benchmark.java) peut utiliser n'importe quelle stratégie sans connaître les détails de son implémentation.

Avantages :

- Permet d'ajouter facilement de nouvelles méthodes de compression sans modifier le code existant.
- Favorise la séparation des responsabilités.
- Facilite les tests unitaires et la maintenance.

3.2 Factory Pattern

Le **Factory Pattern** (ou patron de fabrique) permet de centraliser la création d'objets en fonction d'un paramètre. Il est particulièrement utile lorsqu'on souhaite déléguer la logique d'instanciation à une classe dédiée.

Application dans le projet :

- La classe BitPackingFactory contient une méthode statique create(String type) qui retourne une instance de la classe de compression appropriée.
- En fonction de la chaîne passée en paramètre ("split", "overlap", "overflow"), la factory retourne une instance de BitPackingSplit, BitPackingOverlap ou BitPackingOverflow.

Avantages :

- Simplifie la gestion des types de compression dans le code client.
- Permet de centraliser la logique de création et de la modifier sans impacter les autres composants.
- Facilite l'extension du projet (ajout d'une nouvelle méthode de compression, par exemple BitPackingDelta, sans modifier le code client).

4. Résultats des Benchmarks

Afin d'évaluer les performances des différentes méthodes de compression implémentées (Split, Overlap, Overflow), une série de benchmarks a été réalisée sur des tableaux d'entiers de tailles et de contenus variés. L'objectif était de mesurer l'efficacité de chaque méthode selon plusieurs critères : temps de compression, temps de décompression, temps d'accès direct à un élément (get(i)), ratio de compression, et seuil de latence à partir duquel la compression devient avantageuse.

4.1 Méthodologie

Les benchmarks ont été réalisés sur des tableaux générés aléatoirement avec différentes caractéristiques :

- **Taille** : 10, 100, 1 000, 10 000, 100 000 et 1 000 000 d'éléments.
- **Contenu** :
    - **Petites valeurs** : entiers entre 0 et 100.
    - **Valeurs mixtes** : entiers entre 0 et 5 000.
    - **Grandes valeurs** : entiers entre 1 000 et 1 000 000.

Pour chaque combinaison, les opérations suivantes ont été mesurées :

- compress() : temps nécessaire pour compresser le tableau.
- decompress() : temps nécessaire pour reconstruire le tableau original.
- get(i) : temps moyen pour accéder à un élément aléatoire.
- **Taille compressée** : nombre d'entiers de 32 bits utilisés.
- **Ratio de compression** : taille compressée / taille originale.
- **Seuil de latence** : estimation du temps de transmission à partir duquel la compression devient rentable.

4.2 Résultats

Méthode	Taille compressée	Temps compression (ns)	Temps décompression (ns)	get(i) (ns)	Ratio
Split	4	46000	13600	4800	0.5
Overlap	3	16700	2100	2800	0.406
Overflow	4	2138700	3300	765900	0.5

4.3 Analyse

Split offre un bon compromis entre simplicité et performance. Son temps de compression est raisonnable, et l'accès direct est très rapide. Cependant, son ratio de compression est limité par l'alignement strict sur 32 bits.

Overlap est la méthode la plus efficace en termes de ratio de compression. Elle utilise l'espace mémoire de manière optimale grâce au chevauchement, mais au prix d'une complexité accrue dans la gestion des bits.

Overflow est particulièrement utile lorsque le tableau contient une majorité de petites valeurs et quelques très grandes. Elle permet d'éviter de surdimensionner le bitSize pour tous les éléments. Toutefois, son coût en temps de compression et d'accès direct est significatif, ce qui la rend moins adaptée aux cas où la performance est critique.

5. Conclusion

Ce projet a permis d'explorer différentes stratégies de compression de tableaux d'entiers dans un contexte de transmission efficace des données. Les trois méthodes développées — Split, Overlap et Overflow — répondent à des besoins spécifiques :

- **Split** est la méthode la plus simple et la plus rapide à implémenter. Elle offre un bon compromis entre performance et simplicité, mais peut gaspiller de l'espace mémoire.
- **Overlap** améliore le taux de compression en exploitant chaque bit disponible, au prix d'une complexité accrue dans la gestion des bits.
- **Overflow** est particulièrement adaptée aux tableaux hétérogènes contenant quelques grandes valeurs. Elle permet une compression plus fine, mais au prix d'un surcoût en temps de traitement et d'un accès direct plus lent.

Les benchmarks ont confirmé ces observations : chaque méthode a ses avantages et ses inconvénients selon le type de données et les contraintes de performance. Le choix de la méthode dépend donc du contexte d'utilisation : taille du tableau, distribution des valeurs, et exigences en termes de rapidité ou de compacité.

Enfin, l'utilisation des design patterns Strategy et Factory a permis de structurer le projet de manière modulaire, facilitant l'extension, la maintenance et les tests.