BitPacking

—

Gallard Nathan

Rapport de Projet : Bit Packing pour la Transmission Compacte d'Entiers

Projet : Bit Packing pour transmission d'entiers

Date : 2025-11-02

1. Introduction

1.1 Problématique

La transmission d'ensembles d'entiers sur Internet est une problématique centrale, où la réduction de la **bande passante et de la latence** est essentielle. L'idée est de compresser un tableau d'entiers pour diminuer le nombre d'éléments à transmettre, puis de les décompresser. Le défi principal est de concevoir une méthode de compression qui préserve l'**accès direct** aux éléments ($get(i)$).

1.2 Objectifs

L'objectif du projet était d'implémenter une méthode de compression basée sur le nombre de bits utilisés, appelée **Bit Packing**. Cette méthode représente chaque élément sur k bits au lieu des 32 bits conventionnels, utilisant ainsi $n \times k$ bits au lieu de $32 \times n$ bits pour n éléments.

Trois versions distinctes de la compression devaient être implémentées :

1. Une version où les valeurs peuvent chevaucher deux entiers consécutifs de 32 bits.
2. Une version où les valeurs ne doivent pas chevaucher deux entiers consécutifs de 32 bits.

3. Une version utilisant des **zones de débordement (Overflow)** pour gérer les valeurs aberrantes (outliers).

Chaque implémentation devait fournir les fonctions compress(array), decompress(array) et int get(int i) pour l'accès direct.

2. Conception et Solutions Implémentées

Le projet a été développé en Python, sans utiliser de notebooks (Jupyter Python). Le code est structuré autour d'une classe CompressionFactory qui gère la création des différents types de compresseurs.

2.1 Stratégies de Bit Packing

Trois stratégies ont été implémentées (dans bitpacking.py) :

Stratégie	Description	Implémentation
PackedCross	Les valeurs sont packées de manière la plus dense possible (sans alignement), permettant à un champ de **traverser deux mots de 32 bits**.	Classe PackedCross
PackedAligned	Les valeurs sont alignées de sorte que chaque mot contienne un nombre entier de valeurs. **Les valeurs ne traversent jamais deux mots**.	Classe PackedAligned
OverflowPacking	Utilise une zone principale avec un petit nombre de bits (small_k) et un marqueur spécial (tous les bits à 1) pour indiquer les valeurs plus grandes. Les **valeurs larges sont stockées séparément** dans une zone de débordement.	Classe OverflowPacking

2.2 Choix Techniques

- **Format d'Entier :** Les entiers sont traités comme des mots de 32 bits (constante WORD_BITS = 32). L'implémentation utilise une approche little-endian pour la manipulation des bits dans le flux, simplifiant l'extraction des bits de poids faible en premier.
- **En-tête :** Un en-tête simple est utilisé pour garder le format compact.
    - **PackedCross / PackedAligned :** Un mot de 32 bits contient k (16 bits) et la length du tableau original (16 bits).
    - **OverflowPacking :** Un mot de 32 bits contient small_k (8 bits), length (16 bits) et le nombre d'éléments de débordement (overflow_count, 8 bits).
- **Gestion des Négatifs (Bonus) :** Pour prendre en charge les nombres négatifs sans gaspiller de bits (comme un bit de signe), la technique de **codage ZigZag** est utilisée. Elle mappe les entiers signés vers des entiers non signés, conservant l'ordre relatif des valeurs absolues et est optionnelle via le paramètre use_zigzag=True.

2.3 Détails d'Implémentation Spécifiques à get(i)

La fonction $get(i)$ est cruciale car elle permet d'accéder au i-ème élément directement sans décompression complète, respectant l'exigence d'accès direct.

- **PackedCross.get(i) :** Calcule la position en bits (bitpos = idx * k), puis l'indice du mot (word_idx) et le décalage en bits (bit_off). Il récupère les bits nécessaires en combinant (ou en traversant) deux mots de 32 bits consécutifs (low et high).
- **OverflowPacking.get(i) :**
    - Si le jeton à la position $i \times small\_k$ n'est pas le marqueur d'overflow, il décode simplement la valeur.
    - S'il s'agit du marqueur d'overflow, la fonction doit parcourir tous les jetons précédents (de 0 à $i-1$) pour **compter le nombre total de marqueurs d'overflow**. Ce compte donne l'indice de la valeur réelle dans la zone de débordement.

3. Protocole de Mesure et Résultats Attendus

3.1 Protocole de Mesure

Le protocole de mesure est implémenté dans bench.py.

- **Mesure du Temps :** La fonction time.perf_counter() est utilisée pour mesurer le temps d'exécution de la compression, de la décompression et de l'accès aléatoire (get).
- **Jeux de Tests :** Des tableaux d'entiers de tailles 1k, 10k et 100k sont générés.
    - La majorité des valeurs sont petites (random.randint(0, 15)).
    - Quelques valeurs sont des *outliers* (random.randint(0, 2^{20})) sont insérées aléatoirement (environ 1 pour 1000 éléments).
- **Accès Aléatoire :** Le temps d'accès est mesuré en appelant get(i) sur un échantillon de 1% des éléments (len(values)//100).
- **Sortie :** Les résultats sont imprimés au format JSON par bench.py pour une analyse facile.

3.2 Résultats Qualitatifs Anticipés

- **Compression :** PackedCross devrait compresser légèrement mieux que PackedAligned (moins de mots car il n'y a pas de gaspillage dû à l'alignement).
- **Accès Direct (get(i)):** PackedCross devrait avoir un accès légèrement plus coûteux que PackedAligned en raison de la logique de bit-shifting potentiellement plus complexe et du masquage sur deux mots.
- **Efficacité d'Overflow :** OverflowPacking devrait réduire significativement la taille de la compression lorsque très peu d'éléments sont grands (comme dans les jeux de test conçus).

4. Limitations et Améliorations Futures

4.1 Limitations Actuelles

- **Sérialisation :** L'en-tête est minimal et le format de sérialisation n'est pas robuste (pas conçu pour un environnement de production).
- **Optimisation :** Aucune optimisation de bas niveau (comme SSE, numba) n'a été implémentée, ce qui pourrait améliorer considérablement les performances.
- **Tests Unitaires :** Les tests unitaires sont limités; il est fortement recommandé d'ajouter pytest et des tests exhaustifs pour garantir la robustesse des opérations bit à bit.

4.2 Perspectives d'Amélioration

- **Robustesse du get pour OverflowPacking :** La fonction OverflowPacking.get(i) actuelle a une complexité $O(i)$ (elle parcourt tous les éléments de 0 à $i-1$) dans le cas du débordement. Une amélioration majeure consisterait à stocker une structure de données annexe (par exemple, un tableau d'indices ou des blocs de compteurs) pour permettre un accès $O(1)$ ou $O(\log n)$ à la position de débordement.
- **Calcul du Point de Rentabilité de la Compression :** L'analyse demandée pour calculer le temps de transmission à partir duquel la compression devient avantageuse n'a pas été effectuée. Elle nécessiterait une formule et une analyse basée sur les temps mesurés et le débit de transmission.

5. Conclusion

Ce projet a permis la mise en œuvre complète de trois méthodes de Bit Packing pour la transmission d'entiers avec préservation de l'accès direct, y compris la gestion des cas avec valeurs aberrantes (OverflowPacking) et un support bonus pour les nombres négatifs via le codage ZigZag. La mise en œuvre est prête à être poussée sur GitHub, avec une structure de code modulaire grâce à la `CompressionFactory` et un outil de benchmarking fonctionnel.