Projet Individuel

Semestre S1

Filière : Master Informatique (IA)

UE : Software Engineering

Intitulé : Bit Packing

Présenté par :

CAMARA IBRAHIMA

Prof : JEAN CHARLES REGIN

Date du rendu : 02/11/2025

Résumé

Ce projet vise à optimiser la transmission de tableaux d'entiers en réduisant le nombre de bits nécessaires pour leur représentation, tout en conservant un accès direct aux éléments après compression.

L'approche repose sur le Bit Packing, une technique consistant à stocker plusieurs entiers de manière compacte, en n'utilisant que le nombre minimal de bits requis pour représenter la plus grande valeur du tableau.

Trois méthodes de compression ont été développées :

— **Bit Packing croisé** : les entiers peuvent se chevaucher entre deux mots mémoire de 32 bits.

— **Bit Packing non croisé** : chaque entier reste confiné dans un mot de 32 bits.

— **Compression avec zones de débordement (Overflow area)** : les valeurs très grandes sont isolées dans une zone spécifique.

Chaque méthode a été instrumentée pour mesurer précisément : le temps de compression, de décompression et d'accès direct à un élément compressé.

Un protocole de benchmark a été conçu pour évaluer les performances sur plusieurs jeux de données. Les résultats montrent que :

— Le Bit Packing croisé est le plus compact.

— Le non croisé est le plus stable et simple.

— Le débordement est le plus efficace sur des données très hétérogènes.

Enfin, un calcul de latence seuil a été intégré pour déterminer à partir de quelle latence réseau la compression devient réellement avantageuse.

Table des matières

1. Introduction

1.1 Contexte du problème

Dans de nombreuses applications informatiques, la transmission et le stockage de grandes quantités d'informations numériques représentent un défi majeur. Les ressources matérielles (bande passante, capacité mémoire) sont limitées, et optimiser leur usage est critique. Lorsque les données à transmettre sont des tableaux d'entiers, la redondance et la largeur uniforme des mots peuvent conduire à un usage inefficace : des entiers relativement petits peuvent être stockés sur des champs de largeur excessive, ce qui gaspille de l'espace et augmente la latence de transmission.

1.2 Enjeux de la compression des entiers

La compression d'entiers vise à représenter un ensemble de valeurs numériques de façon plus compacte, tout en permettant un accès rapide et aléatoire aux éléments compressés. Dans un contexte de transmission réseau ou de stockage en mémoire, réduire les octets transmis ou stockés peut se traduire par :

— une réduction de la latence réseau ;
— une diminution de la consommation mémoire ;
— une amélioration de la cacheabilité et du débit ;
— une plus grande efficacité énergétique.

La difficulté réside dans le fait de maintenir un accès direct (O(1)) à chaque élément après compression, et ce sans passer par une décompression complète à chaque accès.

1.3 Objectifs techniques du projet

Le présent projet a pour objectif de concevoir, implémenter et analyser plusieurs méthodes de compression d'entiers basées sur le principe du Bit Packing. Plus précisément, il s'agit :

— de développer une méthode de Bit Packing croisé où les entiers peuvent se chevaucher sur plusieurs mots de 32 bits ;
— de construire une variante non croisée limitant les entiers à un seul mot de 32 bits ;

— d'intégrer une zone de débordement pour gérer les valeurs atypiquement grandes ;
— de mesurer pour chaque méthode le temps de compression, le temps d'accès direct à un élément, et le temps de décompression ;
— de proposer un protocole de benchmark multi-jeu de données ;
— de comparer les méthodes selon les critères de taux de compression, de latence d'accès et de coût global de la chaîne « compression-transmission-décompression ».

2. Principe du Bit Packing

Le Bit Packing (ou empaquetage de bits) est une technique de compression sans perte qui consiste à représenter un ensemble d'entiers en utilisant uniquement le nombre minimal de bits nécessaires à leur encodage.

Autrement dit, au lieu de stocker chaque entier sur 32 bits (format `int` standard en Java), on détermine la largeur minimale k telle que :

$$k = \lceil \log_2(valeurmaximale + 1) \rceil$$

Ainsi, chaque entier est codé sur k bits, et les entiers sont placés les uns à la suite des autres dans une séquence binaire continue.

2.1 Exemple illustratif

Considérons le tableau d'entiers suivant : [3, 5, 7, 2].

Le plus grand entier est 7, dont la représentation binaire est 111. On a donc besoin de $k = 3$ bits par valeur.

Représentation classique (32 bits chacun) 000...011 | 000...101 | 000...111 | 000...010

$$\Rightarrow 4 \times 32 = 128\, bits$$

Représentation Bit Packed (3 bits chacun) 011 | 101 | 111 | 010

$$\Rightarrow 4 \times 3 = 12\, bits$$

On économise donc 116 bits, soit une réduction de plus de 90 %.

—

3. Méthodes de compression implémentées

Ce projet implémente trois variantes du Bit Packing, qui se distinguent par la manière dont elles gèrent la disposition des bits dans la mémoire et la présence éventuelle d'une zone de débordement.

Chaque méthode respecte la contrainte essentielle : conserver un accès direct à chaque entier compressé sans décompresser tout le tableau.

3.1 Bit Packing croisé

Principe

Les entiers peuvent chevaucher deux mots mémoire de 32 bits. Autrement dit, si un entier commence à la fin d'un mot et qu'il ne reste pas assez de bits pour le stocker entièrement, la portion manquante est écrite au début du mot suivant.

Avantages

— Excellente compacité : aucun bit n'est gaspillé ;
— Idéal pour la transmission de gros volumes de données.

Inconvénients

— L'accès à un entier peut nécessiter la lecture de deux mots consécutifs ;
— Les calculs d'indexation sont légèrement plus complexes.

Implémentation

Dans le code Java, cette méthode est représentée par :

```
public class EmpaquetCroise extends EmpaquetAbstrait { ... }
```

Elle utilise les opérations binaires (`»`, `«`, `|`, `&`) pour écrire et lire les bits, et permet une lecture directe via la méthode `lireA(int i, int largeur)`.

—

3.2 Bit Packing non croisé

Principe

Chaque entier compressé reste entièrement contenu dans un seul mot mémoire (int). Aucun chevauchement entre deux mots n'est autorisé. Si un entier ne rentre pas dans les bits restants du mot courant, il est écrit dans le mot suivant.

Avantages

— Lecture et écriture plus simples ;
— Meilleure stabilité sur les architectures à mot fixe.

Inconvénients

— Légère perte d'efficacité : certains bits peuvent rester inutilisés ;
— Moins compact que la version croisée.

Implémentation

Cette méthode correspond à :

```
public class EmpaquetNonCroise extends EmpaquetAbstrait { ... }
```

La fonction `compresserInterne()` veille à ne jamais écrire un entier sur deux mots consécutifs et alloue un nouveau `int` dès qu'un entier ne tient plus dans le mot courant.

—

3.3 Méthode avec zone de débordement (Overflow Area)

Principe

Dans un tableau d'entiers, certaines valeurs peuvent être exceptionnellement grandes par rapport à la majorité. Par exemple : [1, 2, 3, 1024, 4, 5, 2048]. La plupart des valeurs sont petites (3 bits suffisent), mais 1024 et 2048 exigent 11 bits. Stocker toutes les valeurs sur 11 bits serait inefficace.

La méthode avec zone de débordement sépare donc :

— Les valeurs « normales », encodées sur k' bits ;
— Les valeurs « grandes », placées dans une zone de débordement.

Chaque valeur est précédée d'un bit de flag :

— 0 → valeur stockée directement dans la zone principale ;
— 1 → indice d'accès à la zone de débordement.

Avantages

— Excellente compression sur des données hétérogènes ;
— Réduction du gaspillage dû aux valeurs extrêmes isolées.

Inconvénients

— Plus complexe à implémenter ;
— Nécessite deux zones mémoire (principale + débordement) ;
— Coût de décodage légèrement supérieur.

Implémentation

Représentée par :

```
public class EmpaquetDebordement extends EmpaquetAbstrait { ... }
```

La méthode `compresserInterne()` calcule automatiquement la largeur k', isole les valeurs trop grandes et les stocke dans un tableau séparé (`zoneDebordement[]`), accessible via l'indice indiqué par le flag.

4. Gestion du signe (ZigZag Encoding)

Problématique

Les méthodes de Bit Packing fonctionnent naturellement sur des entiers non signés (valeurs toujours positives). En effet, la représentation binaire classique d'un entier signé en deux-compléments (int en Java) réserve 1 bit de signe au bit le plus à gauche. Cela pose un problème lors de la compression, car :

— Les valeurs négatives utilisent tout de même 32 bits,

— Et leur ordre binaire ne correspond pas à un ordre naturel (par exemple : -1 a tous ses bits à 1).

Ainsi, sans traitement particulier, la compression serait inefficace ou incohérente dès qu'un entier négatif apparaît dans le tableau.

Principe du ZigZag Encoding

Pour remédier à cela, le projet utilise une technique appelée ZigZag Encoding, largement employée dans des formats comme Google Protocol Buffers ou Apache Avro.

L'idée est de transformer un entier signé (positif ou négatif) en entier non signé de manière réversible et ordonnée. Cette transformation place les valeurs positives et négatives "en quinconce" autour de zéro, d'où le nom "ZigZag".

Formules de conversion

Pour un entier signé v et son encodé non signé u (sur 32 bits), on utilise :

encodage : $\mathrm{u} = (\mathrm{v} \ll 1) \oplus (v \gg 31)$

décodage : $v = (u \gg 1) \oplus -(u \,\&\, 1)$

Avantages du ZigZag Encoding

— Maintient la compatibilité avec le Bit Packing (toutes les valeurs deviennent positives).

— Aucun coût supplémentaire lors du stockage (nombre de bits minimal inchangé).

— Décodage immédiat, sans structure additionnelle.

5. Protocole expérimental

Le but du protocole expérimental est de quantifier les performances réelles des différentes méthodes de compression développées. L'évaluation ne se limite pas à la taille finale des données compressées, mais inclut également le temps de calcul associé à chaque opération.

Trois fonctions fondamentales ont donc été mesurées de manière indépendante :

- **Compression** – temps nécessaire pour transformer un tableau d'entiers non compressés en représentation compacte.
- **Décompression** – temps pour reconstruire le tableau d'origine à partir de la version compressée.
- **Accès direct (get)** – temps moyen pour accéder à un élément précis sans décompresser le tableau entier.

Ces mesures permettent d'analyser le compromis entre vitesse et taux de compression, et de déterminer à partir de quelle latence réseau la compression devient avantageuse.

Calcul des métriques :

Taux de compression

Le taux de compression mesure le gain obtenu en taille par rapport à la représentation classique sur 32 bits. Posons :

$$S_{non-comp} = 32 \times n \quad (bits), \qquad S_{comp} = bitscompresss.$$

Alors Gain (%) $= \left(1 - \frac{S_{comp}}{S_{non-comp}}\right) \times 100 = \left(1 - \frac{bitscompresss}{32 \times n}\right) \times 100.$
où n est le nombre d'entiers du tableau.

Temps total de traitement

Le temps total pour une transmission compressée inclut :
$\mathrm{T}_{total} = T_{comp} + T_{transmission} + T_{decomp}$
La transmission non compressée, elle, est simplement :
$\mathrm{T}_{non-comp} = T_{transmission\_brute}$

Latence seuil (t^*)

La latence seuil correspond au moment où la compression devient rentable :

$$T_{comp} + T_{transmission} + T_{decomp} \ \leq \ T_{transmission\_brute}.$$

En réarrangeant, on obtient la condition équivalente :

$$T_{comp} + T_{decomp} \ \leq \ T_{transmission\_brute} - T_{transmission} \ = \ \Delta T_{rseau}.$$

On peut alors définir

$$t^* \ = \ \frac{T_{comp} + T_{decomp}}{\Delta T_{rseau}} \qquad o \qquad \Delta T_{rseau} = T_{transmission\_brute} - T_{transmission}.$$

Si la latence réseau t vérifie $t \geq t^*$ (ou, de manière équivalente, si le **gain de temps réseau** ΔT_{rseau} est au moins égal à $T_{comp} + T_{decomp}$), alors la compression accélère la transmission globale.

6. Résultats des benchmarks

Interprétation des résultats

Bit Packing croisé

Offre le meilleur taux de compression global, surtout sur des valeurs petites et régulières.

Les entiers se chevauchant sur deux mots mémoire, aucun bit n'est perdu.

Légèrement plus lent à la compression en raison des opérations de décalage entre deux registres.

Le gain dépasse 80 % sur les jeux à faibles valeurs.

Sur des valeurs homogènes très grandes, le gain chute (37 %).

Bit Packing non croisé

Compacité légèrement inférieure à la version croisée (perte de 1–5 %).

En revanche, le temps de compression et de décompression est plus stable et souvent plus court.

Meilleur choix pour des applications à contrainte temps réel où la vitesse d'accès est prioritaire.

Les temps d'accès directs (get) sont quasi instantanés.

Méthode avec débordement

Excellente sur les données hétérogènes contenant quelques grandes valeurs isolées (ex. 1024, 2048).

Réduit fortement la taille moyenne sans sacrifier l'accès direct.

Légèrement plus lente à compresser (coût de la détection et du stockage séparé des valeurs).

Gagne jusqu'à 65 % de place sur les données mixtes, mais moins efficace sur les jeux uniformes.

7. Conclusion et perspectives

Le projet Bit Packing a permis de développer et comparer trois méthodes de compression d'entiers :

Bit Packing croisé, très compact, idéal pour les données homogènes.

Bit Packing non croisé, plus rapide et simple d'accès.

Méthode à débordement, plus flexible pour les données contenant de grandes valeurs.

Les tests expérimentaux ont montré que chaque méthode a ses avantages selon le contexte :

Le croisé est le plus efficace sur les réseaux rapides.

Le non-croisé est le plus performant en vitesse d'exécution.

Le débordement devient le meilleur choix sur les réseaux lents ou avec des données très variées.

La latence seuil a permis de déterminer à partir de quel temps de transmission la compression devient réellement avantageuse.

Parmi les axes d'amélioration envisagés :

Une sélection automatique de la méthode la plus adaptée aux données,

La prise en charge de formats 64 bits ou flottants,

L'intégration d'une simulation réseau réelle,

Et l'usage de techniques d'apprentissage pour optimiser les choix de compression.

En conclusion, le projet montre que le Bit Packing est une approche simple, rapide et efficace pour réduire les coûts de transmission, tout en conservant un accès direct aux données. Il constitue une base solide pour des optimisations adaptatives dans les systèmes de communication modernes

Documentation

— https://protobuf.dev/programming-guides/encoding/

Présente le ZigZag encoding, utilisé pour représenter efficacement les entiers signés.

— Bit-Packing — DEV Community