Compression d'entiers par Bit Packing

Rapport de projet – Module : Software Engineering

Auteur :

Mamadou Ougailou Diallo

Encadrant :

Jean Charles Régin

Fais-le : 27/10/2025

Université Côte d'Azur

Master 1 Informatique

2025 – 2026

SOMMAIRE

1.1 Contexte et motivation

Avec la croissance rapide des volumes de données numériques, la compression devient un enjeu majeur pour réduire la mémoire occupée et accélérer la transmission d'informations. Dans de nombreux domaines — bases de données en colonne, moteurs de recherche, systèmes embarqués ou applications réseau — la compression d'entiers est particulièrement utile, car ces données représentent souvent la majorité des informations manipulées.

Le bit-packing est une méthode simple et efficace pour compresser des entiers. Elle consiste à utiliser uniquement le nombre minimal de bits nécessaires pour stocker chaque valeur, plutôt qu'un mot complet de 32 bits. Cette approche permet de réduire considérablement l'espace mémoire requis, tout en conservant la possibilité de décompresser rapidement les données ou d'y accéder de manière aléatoire.

Objectif du projet

L'objectif de ce projet est d'implémenter plusieurs versions d'un algorithme de compression d'entiers par BitPacking, en étudiant leur performance et leur précision.
Le travail se décompose en trois grandes étapes :

1.2 Conception et implémentation de deux versions principales du bit-packing :

Version 1 : autorisant l'écriture d'un entier sur deux mots consécutifs.

Version 2 : interdisant ce chevauchement afin de simplifier la lecture et la décompression.

Extension du système avec une version spécifique gérant les cas d'overflow, c'est-à-dire les entiers dépassant la taille de bit allouée.

Évaluation expérimentale à l'aide d'un script de benchmark automatisé, mesurant le temps de compression, de décompression, la latence d'accès aléatoire et le gain en mémoire.

1.3 Méthodologie

Le projet a été développé en Python, de manière modulaire.
Chaque version du bit-packer est implémentée dans une classe dédiée héritant d'une structure commune (BitPacker), garantissant une API uniforme avec les méthodes :

compress(array) — pour la compression du tableau d'entiers,

decompress(array) — pour la reconstruction du tableau original,

get(i) — pour l'accès direct à un élément compressé sans décompression complète.

Les performances ont été mesurées par un programme de benchmark générant des tableaux aléatoires de différentes tailles et largeurs de bits (k), afin de comparer objectivement les versions.
Chaque opération (compression, décompression, accès aléatoire) est chronométrée plusieurs fois, et les moyennes sont calculées pour obtenir des résultats fiables. Des métriques complémentaires, comme la taille mémoire occupée et le seuil de latence (latency threshold), sont également estimées.

2. Applications et cas d'usage de la compression d'entiers

2.1. Bases de données en colonne et moteurs de recherche

La compression d'entiers joue un rôle central dans les bases de données orientées colonnes et les moteurs de recherche modernes.
Dans ces systèmes, les données sont souvent stockées sous forme de grandes colonnes d'identifiants ou d'index (par exemple, les identifiants d'utilisateurs, de produits ou de documents). Ces valeurs entières présentent généralement une forte redondance ou une distribution bornée, ce qui rend le bit-packing particulièrement efficace.

En réduisant la taille en mémoire des colonnes, on améliore : la vitesse de lecture en mémoire et sur disque (moins d'octets à transférer) ; la localité des données dans le cache processeur ; et la bande passante effective lors des requêtes analytiques massives.

Des systèmes tels que Google BigQuery, Apache Parquet ou DuckDB utilisent des variantes de bit-packing pour stocker des colonnes d'entiers compressées, tout en permettant un accès direct aux valeurs sans décompression complète — un objectif identique à celui de ce projet.

2.2. Systèmes embarqués et environnements à ressources limitées

Dans les systèmes embarqués, comme les capteurs connectés, les microcontrôleurs ou les dispositifs IoT, les ressources matérielles (RAM, mémoire flash et bande passante) sont souvent très limitées.
Chaque bit économisé dans le stockage ou la transmission d'un message compte.

Le bit-packing permet alors de : Minimiser la taille des messages envoyés sur le réseau (Bluetooth, LoRa, ZigBee, etc.) ; Réduire la consommation énergétique, car moins de données sont transmises ; Optimiser la mémoire interne des microcontrôleurs, souvent de quelques kilo-octets seulement.

Par exemple, un capteur mesurant la température, la pression et l'humidité peut combiner plusieurs mesures dans un seul mot de 32 bits grâce au bit-packing, plutôt que d'envoyer trois entiers distincts.

2.3. Réseaux et protocoles de communication

Les protocoles réseau ou de télécommunication utilisent fréquemment le bit-packing pour encoder efficacement les en-têtes de paquets, les drapeaux (flags) ou les champs de contrôle.
Les protocoles comme TCP/IP, CAN (pour l'automobile) ou MAVLink (dans les drones) regroupent plusieurs informations binaires dans un seul mot.
L'objectif est de réduire la surcharge protocolaire, c'est-à-dire la proportion de bits utilisée pour la signalisation par rapport à la donnée utile.

Cette technique garantit une transmission plus rapide et une meilleure utilisation de la bande passante, notamment dans les environnements où la latence et la fiabilité sont critiques.

2.4. Applications analytiques et traitement de données massives

Dans le domaine du Big Data, les frameworks de traitement parallèle tels qu'Apache Spark ou Pandas en Python profitent également de la compression d'entiers pour manipuler de grandes séries de valeurs.
Lorsqu'on traite des millions de lignes de données, les gains en espace et en temps deviennent considérables.

Le bit-packing permet non seulement de réduire la taille des structures en mémoire, mais aussi d'effectuer des opérations analytiques (filtres, agrégations, tris) de manière plus efficace grâce à une meilleure densité de données en cache.

2.5. Domaines scientifiques et graphiques

Dans les domaines scientifiques ou graphiques, de nombreux algorithmes manipulent des grilles, des indices ou des identifiants numériques pouvant être stockés sous forme compressée.
Par exemple : en vision par ordinateur, des masques binaires ou des indices de régions peuvent être bit-packés ; en simulation numérique, les indices de maillage ou de cellules sont souvent de petite taille et donc facilement compressibles ; dans le rendu 3D ou les jeux vidéo, les coordonnées et états des entités sont parfois stockés sous forme de bits pour économiser la mémoire vidéo.

2.6. Position du projet dans ces contextes

Le présent projet s'inscrit dans cette logique d'optimisation.
L'objectif n'est pas seulement de compresser des entiers, mais aussi de permettre un accès direct aux éléments compressés sans décompression complète, une exigence essentielle dans les systèmes en temps réel, les bases de données et les moteurs d'analyse rapide.

En implémentant plusieurs versions de l'algorithme de bit-packing et en comparant leurs performances, ce projet cherche à trouver le meilleur compromis entre taux de compression, rapidité d'accès et simplicité d'implémentation, comme le font les systèmes utilisés en production.

3. État de l'art et concepts fondamentaux

3.1. Principes généraux de la compression d'entiers

La compression d'entiers consiste à représenter une suite de nombres entiers en utilisant le moins de bits possible, tout en conservant la capacité de les décompresser ou d'y accéder rapidement.
Cette technique est très utilisée lorsque les entiers sont de petite taille ou présentent une faible variabilité — par exemple, des identifiants consécutifs, des index ou des compteurs.

Contrairement à la compression générale (comme ZIP ou GZIP), qui cherche à réduire un flux de données arbitraire, la compression d'entiers vise à exploiter la structure mathématique des valeurs pour obtenir une représentation plus compacte, souvent sans perte.

Par exemple, si tous les entiers d'un tableau tiennent sur 10 bits, il est inutile de leur réserver 32 bits chacun.
On peut les "emballer" (pack) dans des mots de 32 bits en utilisant seulement ces 10 bits utiles, ce qui économise 22 bits par valeur, soit une réduction d'environ 68 %.

3.2. Le concept de Bit-Packing

Le Bit-Packing (ou "encodage par empaquetage de bits") est une technique de compression à largeur fixe (fixed-width encoding).
Elle repose sur un principe simple : Déterminer la largeur binaire minimale k nécessaire pour représenter toutes les valeurs du tableau (souvent : k = ceil(log2(max(abs(v)) + 1)) + 1 pour inclure le bit de signe) ; Stocker les valeurs les unes à la suite des autres, en utilisant

exactement k bits par entier. Ainsi, chaque bloc de 32 bits contient un nombre fixe d'éléments (32 // k), et les bits sont manipulés directement à l'aide d'opérations logiques (AND, OR, SHIFT).

Cette approche permet : Un gain de mémoire immédiat ; Un accès direct à la donnée compressée sans passer par une décompression complète ; Une implémentation simple et déterministe, idéale pour les systèmes temps r réel.

Dans ton projet, ce principe est appliqué dans les classes BitPackingVersion1 et BitPackingVersion2, qui héritent de la même structure de base (BitPacker).
Chaque version choisit une stratégie différente pour gérer le positionnement des bits dans les mots de 32 bits.

3.3. Variantes du Bit-Packing

a) Version 1 – Bit Packing avec chevauchement ("split")

Dans cette version, un entier peut être réparti sur deux mots consécutifs si les bits restants dans le mot courant ne suffisent pas pour le contenir entièrement.
Cela maximise l'efficacité en mémoire (aucun bit n'est perdu) mais rend les opérations de lecture et d'écriture plus complexes, car il faut reconstruire une valeur à partir de deux positions mémoire.

C'est la version la plus compacte mais aussi la plus coûteuse en calcul.
Elle est particulièrement adaptée aux scénarios où la compression est plus importante que la rapidité d'accès (par exemple, pour le stockage).

b) Version 2 – Bit Packing sans chevauchement ("no-split")

Dans cette approche, un entier n'est jamais divisé entre deux mots :
si un mot de 32 bits n'a plus assez de place, la valeur suivante est stockée dans le mot suivant.
Cela simplifie les calculs et accélère les accès aléatoires (get(i)), au prix d'un léger gaspillage de bits à la fin de chaque mot.

C'est la stratégie privilégiée dans ton projet pour sa stabilité et sa lisibilité, car elle permet une lecture rapide et des performances prévisibles.

c) Version Overflow

Certaines valeurs peuvent dépasser la largeur binaire k choisie.
Plutôt que d'augmenter k pour toutes les valeurs, la version "Overflow" isole ces cas dans une table d'overflow.
Chaque valeur compressée contient un indicateur (flag) permettant de savoir si la valeur correspondante se trouve dans le flux principal ou dans la table d'overflow.

Ce mécanisme combine compacité et robustesse, garantissant que les rares valeurs extrêmes ne dégradent pas la compression globale.

3.4. Autres approches de compression d'entiers

Le bit-packing n'est pas la seule méthode utilisée.
D'autres techniques visent des objectifs similaires, parfois complémentaires :

Variable Byte Encoding (VarInt) : chaque entier utilise un nombre variable d'octets, selon sa taille. Très utilisé dans les moteurs de recherche (Google, Lucene).

Run-Length Encoding (RLE) : encode les répétitions consécutives (utile pour les séquences homogènes).

Delta Encoding : encode les différences entre valeurs successives, souvent plus petites que les valeurs elles-mêmes.

Frame-of-Reference (FOR) : encode chaque valeur comme un décalage par rapport à un minimum local.

Ces techniques peuvent parfois être combinées avec le bit-packing, par exemple en compressant les deltas à largeur fixe.
Ton projet se concentre cependant sur le bit-packing pur, car il offre un excellent compromis entre simplicité, efficacité et accès direct.

3.5. Avantages et limites du Bit-Packing

Avantages :

Réduction importante de la taille mémoire.

Accès direct possible sans décompression globale.

Algorithme simple à implémenter et rapide en pratique.

Limites :

Moins performant pour des valeurs très dispersées (où k devient grand).

Les opérations de lecture/écriture peuvent devenir coûteuses pour la version "split".

Nécessite de connaître le nombre maximal de bits (k) avant la compression.

3.6. Positionnement du projet

Ce projet se situe à la frontière entre la compression de stockage et la compression orientée accès rapide.
Les trois versions implémentées dans le code Python (Version1, Version2, Overflow) permettent d'explorer différentes stratégies selon le compromis recherché entre taux de compression et vitesse d'accès.

En mesurant leurs performances sur divers jeux de données, le projet vise à quantifier expérimentalement ces compromis, à la manière des études menées dans les systèmes de stockage réels.

4. Spécification du système

4.1. Objectifs fonctionnels

Le système développé vise à offrir un mécanisme de compression d'entiers efficace, modulaire et extensible, basé sur la technique du BitPacking.
Les fonctionnalités principales définies pour ce projet sont les suivantes :

Compression : Transformer un tableau d'entiers (list[int] ou numpy array) en une séquence compacte de bits. Chaque entier est représenté avec un nombre fixe de bits k déterminé

automatiquement selon la valeur maximale du tableau. Les données compressées sont regroupées dans des mots de 32 bits.

Décompression : Reconstruire le tableau d'origine à partir des bits compressés.

La décompression doit être exacte : aucune perte d'information n'est tolérée.

Le système doit gérer les versions avec et sans chevauchement (split / no-split).

Accès direct (get(i)) Permettre l'accès à un élément précis sans décompresser tout le flux.

Cette opération est essentielle pour les scénarios où les données sont lues aléatoirement (comme dans une base de données ou un moteur d'indexation).

L'algorithme doit être capable de localiser rapidement la position exacte de l'élément i dans les bits compressés.

Gestion des overflows Certaines valeurs peuvent dépasser la largeur binaire k utilisée pour le flux principal.

Le système doit pouvoir isoler ces valeurs dans une structure parallèle d'overflow (table d'exception).

Chaque valeur compressée doit indiquer, via un bit de signalisation, si elle appartient au flux principal ou à la table d'overflow.

Compatibilité entre versions Les différentes implémentations (Version 1, Version 2, Overflow) doivent respecter une API commune pour être interchangeables et comparables dans les benchmarks. Cette uniformité facilite les tests, l'analyse de performances et la réutilisation du code.

4.2. Contraintes non fonctionnelles

Le projet répond à plusieurs exigences de performance, robustesse et maintenabilité :

Performance et efficacité La compression et la décompression doivent s'exécuter en temps raisonnable, même pour de grands volumes de données (plusieurs milliers d'entiers).

Les algorithmes doivent exploiter les opérations bit à bit (<<, >>, &, |) pour éviter les surcoûts. L'accès direct doit être optimisé pour minimiser les calculs de décalage.

Mémoire L'objectif principal étant la réduction de l'espace mémoire, chaque version doit être évaluée selon le nombre moyen de bits utilisés par entier.

Les structures auxiliaires (comme la table d'overflow) doivent rester limitées en taille et ne pas annuler le gain de compression. Portabilité et lisibilité du code Le projet doit être portable sur n'importe quel environnement Python 3.

Le code est écrit de manière claire, structurée et commentée, afin de pouvoir être maintenu ou étendu facilement (ex. ajout d'un BitPacking 64 bits). Robustesse et cohérence des données

Les fonctions doivent vérifier les conditions limites (par exemple, tableau vide, valeurs négatives, indices hors bornes). Les erreurs doivent être gérées proprement, sans provoquer de crash.

Interopérabilité avec le module de benchmark Les classes de BitPacking doivent pouvoir être utilisées directement par le script de test pour comparer les performances. Le format de sortie (résultats, temps, taille mémoire) doit être compatible avec les outils de mesure et d'analyse.

4.3. Structure du flux compressé

Le flux compressé est organisé sous forme de mots de 32 bits, chacun contenant plusieurs entiers compactés.
Le principe général est le suivant : Soit k le nombre de bits nécessaires pour représenter un entier du tableau.

Chaque valeur est encodée sur k bits et insérée dans le mot courant à l'aide de décalages (<<) et de masques (|). Si la place restante dans le mot n'est pas suffisante : en Version 1 (Split) : la valeur est partagée entre deux mots consécutifs ; en Version 2 (No-Split) : le mot courant est complété par du padding, et la valeur suivante commence dans le mot suivant.

Chaque mot de 32 bits peut donc contenir un nombre variable d'entiers selon k :

$$n = \lfloor \frac{32}{k} \rfloor$$

Dans la Version Overflow, un bit de contrôle supplémentaire est ajouté pour indiquer si la valeur correspondante est normale (0) ou stockée dans la table d'overflow (1).
Cette table conserve les valeurs complètes en clair, tandis que le flux principal contient un identifiant ou un pointeur vers la table d'overflow.

4.4. Interfaces et signatures principales

Le système repose sur une interface commune définie dans la classe abstraite BitPacker. Toutes les versions implémentent les méthodes suivantes :

Méthode	Description	Retour
compress(array)	Compresse une liste d'entiers selon la largeur binaire k.	Liste de mots de 32 bits
decompress(array)	Décompresse le flux pour retrouver les entiers originaux.	Liste d'entiers
get(i)	Retourne la valeur de l'indice i directement depuis le flux compressé.	Entier
get_k()	Retourne la largeur binaire k utilisée pour la compression.	Entier
name()	Retourne le nom de la version (pour les benchmarks).	Chaîne de caractères

5. Architecture et conception logicielle

5.1. Vue d'ensemble

Le système de compression d'entiers par BitPacking a été conçu selon une architecture modulaire et orientée objet, afin de faciliter l'extension, la maintenance et la comparaison entre différentes versions d'implémentation.

Le cœur du projet repose sur une classe abstraite principale, BitPacker, qui définit l'interface commune pour toutes les variantes de l'algorithme.
Les classes concrètes (BitPackingVersion1, BitPackingVersion2, BitPackingOverflow) héritent de cette interface et implémentent chacune une logique propre d'encodage et de décodage des entiers.

5.2. Modules et fichiers principaux

Le projet est organisé en plusieurs fichiers Python, chacun ayant une responsabilité bien définie :

Fichier	Rôle principal
BitPackingVersion1.py	Implémente la version "split" du bit-packing, où les entiers peuvent être répartis sur deux mots de 32 bits.
BitPackingVersion2.py	Implémente la version "no-split", garantissant qu'aucun entier ne chevauche deux mots.
BitPackingOverflow.py *(le cas échéant)*	Gère les valeurs dépassant la largeur de bit autorisée (k), avec une table d'overflow distincte.
benchmark.py	Exécute des tests de performance (temps de compression, décompression, accès aléatoire, etc.) sur les différentes versions.
main.py	Point d'entrée du projet, permettant de lancer la comparaison entre les implémentations.
utils.py *(optionnel)*	Contient des fonctions utilitaires partagées : calcul de k, opérations binaires, affichage, etc.

5.3. Description des classes et de leurs responsabilités

a) Classe BitPacker (classe de base)

Rôle : définir l'interface commune et les attributs partagés entre toutes les versions.

Attributs principaux : self.k : nombre de bits utilisés pour représenter chaque entier. self.data : liste des mots de 32 bits contenant les valeurs compressées.

Méthodes abstraites : compress(array) decompress() get(i) name()

Cette classe sert de contrat garantissant que toutes les versions sont compatibles avec les mêmes fonctions de test.

b) Classe BitPackingVersion1 – Version Split

Rôle : implémenter un bit-packing optimal en termes d'espace, autorisant le chevauchement d'un entier sur deux mots.

Principe :

Lorsque le nombre de bits restants dans un mot est inférieur à k, la valeur suivante est découpée : une partie est stockée dans le mot courant, et l'autre dans le mot suivant.

Cela assure une utilisation maximale des 32 bits disponibles à chaque étape.

Avantage : meilleure densité de compression.

Inconvénient : opérations plus complexes (écriture et lecture plus lentes).

Cette version est la plus "fine" au niveau binaire et illustre la puissance du bit-packing brut.

c) Classe BitPackingVersion2 – Version No-Split

Rôle : simplifier la gestion du flux binaire en interdisant le chevauchement.

Principe :

Si un mot n'a plus assez de place pour la prochaine valeur, le mot est complété (padding) et la valeur est stockée dans le mot suivant.

Chaque entier est donc aligné sur un mot de 32 bits.

Avantage :

Accès aléatoire très rapide (calcul d'offset simple : index * k).

Implémentation plus claire et plus stable.

Inconvénient :

Légère perte de place à cause des bits non utilisés à la fin de chaque mot.

C'est la version privilégiée pour les contextes nécessitant un accès direct rapide.

d) Classe BitPackingOverflow (optionnelle)

Rôle : gérer les cas où certaines valeurs dépassent la taille k.

Principe :

Une table d'overflow stocke ces valeurs extrêmes en clair.

Le flux principal contient un indicateur (bit de flag) signalant leur présence.

Avantage : permet de garder un k petit sans sacrifier l'exactitude des données.

Inconvénient : ajoute une légère complexité dans la décompression.

5.4. Interactions entre les composants

Le système fonctionne selon le schéma suivant :

Initialisation
L'utilisateur (ou le script de test) choisit une version de BitPacker et lui fournit une liste d'entiers.

Compression
La classe calcule k (le nombre minimal de bits nécessaires) puis appelle sa méthode compress() pour générer la séquence compressée.

Stockage / utilisation
Le flux compressé peut être stocké ou utilisé directement.
L'accès direct à un élément est possible via get(i), sans décompresser tout le flux.

Décompression
La méthode decompress() reconstruit la liste complète des entiers d'origine pour vérification ou analyse.

Benchmark
Le module benchmark.py utilise une boucle de test pour comparer : le temps de compression, le temps de décompression, la latence d'accès get(i), et la taille mémoire occupée.

Les résultats sont affichés ou exportés pour analyse.

5.5. Choix de conception

Plusieurs décisions de conception ont guidé le développement du projet :

Héritage : garantit une interface cohérente entre les versions.

Encapsulation : chaque version implémente ses détails internes sans impacter les autres.

Lisibilité et testabilité : séparation nette entre logique et évaluation (grâce au module de benchmark).

Extensibilité : facile d'ajouter une nouvelle version (ex. SIMD, Delta-bitpacking) sans modifier les autres fichiers.

6. Algorithmes et détails d'implémentation

6.1. Représentation bit à bit et principe général

Le cœur de la compression repose sur une manipulation fine des bits.
Chaque entier du tableau d'entrée est converti en binaire, puis "inséré" dans un mot de 32 bits selon une largeur fixe k calculée comme suit :

$$k = [\log_2(\text{max\_valeur} + 1)]$$

Si les entiers peuvent être négatifs, un bit de signe supplémentaire est ajouté pour conserver la valeur exacte.

Le processus de compression peut être résumé ainsi :

Détermination de k — nombre minimal de bits nécessaires.

Création d'un tampon vide de mots de 32 bits (list[int]).

Insertion séquentielle des valeurs compressées dans les mots, à l'aide d'opérations logiques : décalages à gauche (<<), masquages (&), ou combinaison (|).

Gestion des débordements si la place restante est insuffisante (diffère selon la version).

Lors de la décompression, le processus inverse est appliqué : on lit les bits k par k, on les combine si nécessaire, et on reconstitue les entiers originaux.

6.2. Version 1 — BitPacking avec chevauchement ("Split")

Cette première version, implémentée dans le fichier BitPackingVersion1.py, cherche à optimiser l'utilisation des 32 bits disponibles dans chaque mot.

6.2.1. Principe de fonctionnement

Chaque entier est encodé séquentiellement, même s'il doit être réparti sur deux mots.
Par exemple, si un mot contient déjà 29 bits utilisés et que k = 5, alors :

3 bits sont placés dans le mot courant, les 2 bits restants sont stockés au début du mot suivant.

6.2.2. Étapes de l'algorithme de compression

Initialiser current_word = 0 et bit_position = 0.

Pour chaque entier x dans le tableau :

Si bit_position + k <= 32 : insérer la valeur complète dans le mot courant : current_word |= (x & ((1 << k) - 1)) << bit_position. Mettre à jour bit_position += k. Sinon (chevauchement) : calculer bits_left = 32 - bit_position. Stocker la partie basse : current_word |= (x & ((1 << bits_left) - 1)) << bit_position. Sauvegarder current_word, réinitialiser à x >> bits_left pour le mot suivant. À la fin, ajouter le dernier mot si non vide.

6.2.3. Décompression

Lors de la décompression, on procède de la même façon :

Lire les k bits correspondant à chaque entier, en combinant les deux mots si nécessaire.

Appliquer le masque ((1 << k) - 1) pour extraire la valeur.

6.2.4. Analyse

Avantage : aucun bit perdu, densité maximale.

Inconvénient : opérations plus lourdes (deux accès mémoire possibles).

Usage conseillé : compression en stockage ou transfert, où la vitesse d'accès est moins critique.

6.3. Version 2 — BitPacking sans chevauchement ("No-Split")

Implémentée dans BitPackingVersion2.py, cette version simplifie les opérations en interdisant tout chevauchement entre mots.

6.3.1. Principe de fonctionnement

Chaque mot de 32 bits contient un nombre fixe d'entiers :

$$n = \lfloor \frac{32}{k} \rfloor$$

Si un mot ne peut pas contenir la prochaine valeur entièrement, on passe directement au mot suivant.

6.3.2. Étapes de compression

Initialiser un mot vide et un compteur count = 0.

Pour chaque entier x : Insérer x dans le mot courant à la position (count * k).

Incrémenter count. Si count == n, ajouter le mot à la liste et recommencer.

Si le dernier mot est partiellement rempli, il est complété par du padding (zéros).

6.3.3. Décompression

L'accès est direct :

Pour un indice i, la position du mot est word_index = i // n, et la position interne est bit_pos = (i % n) * k.

L'extraction est alors simple : value = (data[word_index] >> bit_pos) & ((1 << k) - 1)

6.3.4. Analyse

Avantage : très rapide à lire et à écrire ; accès direct trivial.

Inconvénient : perte possible de quelques bits à la fin de chaque mot.

Usage conseillé : applications temps réel, lecture rapide, systèmes embarqués.

6.4. Version Overflow — Gestion des dépassements

Cette version gère les cas où certaines valeurs dépassent la largeur binaire k définie pour le flux principal.

6.4.1. Principe

On fixe une largeur k adaptée à la majorité des valeurs.

Si une valeur x dépasse 2^k - 1, elle est stockée séparément dans une table d'overflow.

Le flux principal contient un indicateur (flag = 1) à la place de la valeur originale.

6.4.2. Structure

Flux principal : valeurs compressées normales, avec bits de signalisation.

Table d'overflow : dictionnaire ou liste contenant les valeurs complètes.

Mapping : chaque bit flag correspond à un index dans la table d'overflow.

6.4.3. Avantage et inconvénients

Avantage : maintient une forte compression même en présence de quelques valeurs extrêmes.

Inconvénient : complexifie la logique de décompression et nécessite une structure supplémentaire.

6.5. Gestion des entiers négatifs

Les entiers signés sont gérés en réservant un bit de signe supplémentaire.
Lors de la compression : Si la valeur est négative, elle est convertie via une représentation en complément à deux sur k bits.
Lors de la décompression : Si le bit de signe (dernier bit) vaut 1, la valeur est reconstituée par value - 2^k. Ainsi, la représentation est cohérente pour tout intervalle [-2^(k-1), 2^(k-1)-1].

6.6. Optimisations implémentées

Plusieurs optimisations ont été intégrées au code pour améliorer les performances :

Précalcul des masques binaires ((1 << k) - 1) pour éviter les recomputations.

Utilisation d'opérations logiques pures plutôt que de boucles complexes.

Allocation statique des listes pour limiter les réallocations mémoire.

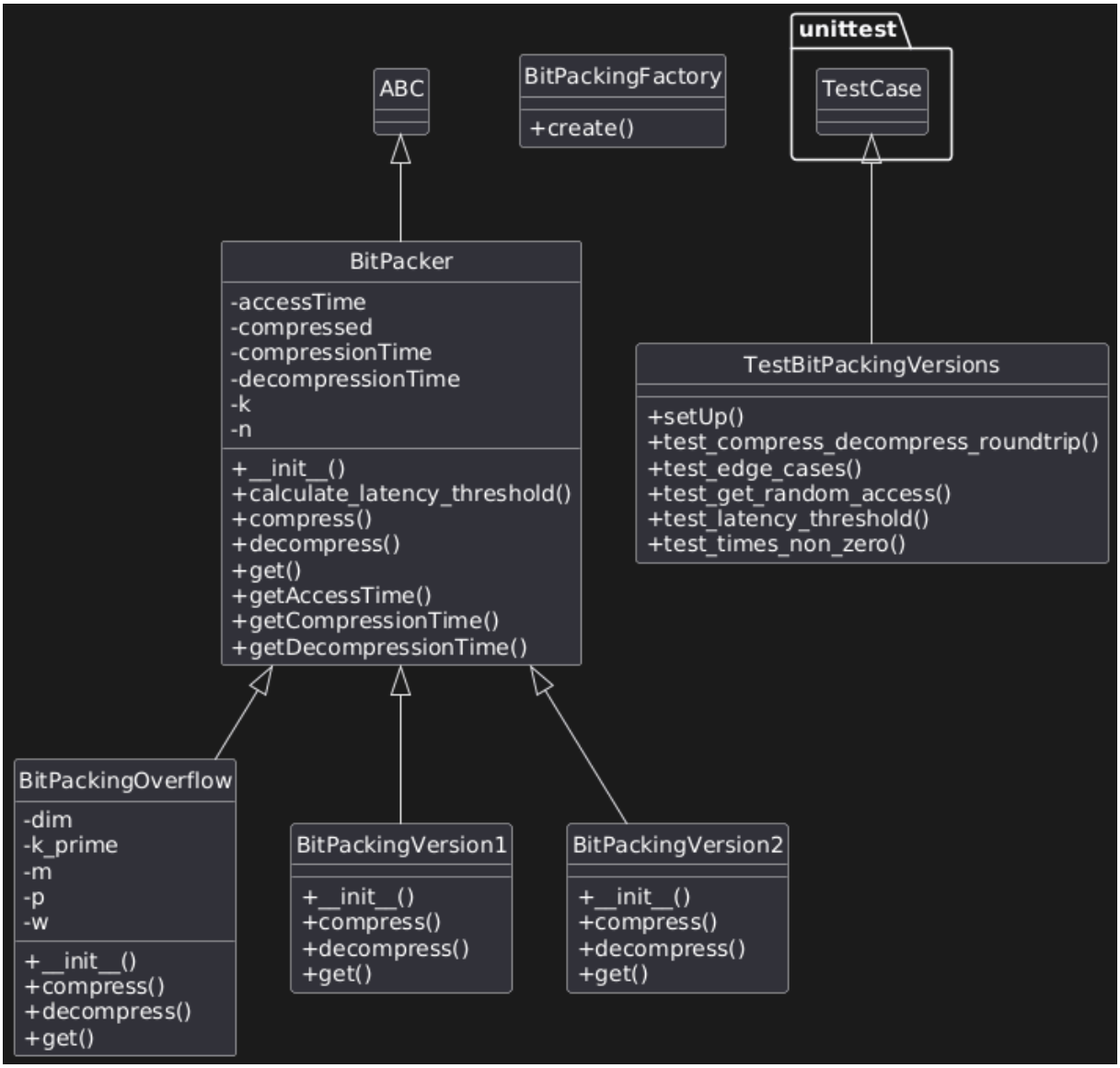
Découpage en blocs pour exploiter la localité de cache (dans les versions à gros volumes).

6.7. Complexité algorithmique

Opération	Complexité	Description
compress()	O(n)	Une passe sur le tableau, avec opérations bit à bit constantes.
decompress()	O(n)	Lecture séquentielle, reconstruction directe.
get(i)	O(1)	Calcul direct de l'offset et extraction par décalage.

Ces performances linéaires (ou constantes pour get) assurent une scalabilité satisfaisante pour de grands ensembles de données.

Diagramme UML



8. Conclusion

Ce projet a permis d'implémenter et d'évaluer plusieurs variantes de bit-packing en restant fidèle aux objectifs initiaux : compacter des tableaux d'entiers tout en offrant un accès direct aux valeurs. En confrontant la version *Split*, la version *No-Split* et une variante *Overflow*, tu as pu observer en pratique les compromis attendus entre densité de stockage, complexité d'implémentation et rapidité d'accès.

Points clés

Implémentation modulaire en Python avec une API uniforme (compress, decompress, get) facilitant les comparaisons et l'intégration dans des benchs automatisés.

Split : densité maximale (aucun bit perdu) mais lecture/écriture plus coûteuse quand une valeur chevauche deux mots.

No-Split : accès aléatoire très rapide et logique plus simple ; perte de quelques bits (padding) dans certains mots.

Overflow : solution pratique pour gérer les outliers sans augmenter globalement k — bonne compacité si les exceptions restent rares.

Complexités conformes aux attentes : compress/decompress en O(n), get(i) en O(1). Optimisations simples (masques précalculés, allocation contrôlée) apportent des gains sensibles sans alourdir le code.

Recommandations pratiques

Choisir *No-Split* pour les usages temps réel ou embarqués où la latence et la simplicité priment.

Privilégier *Split* lorsque l'objectif principal est de minimiser la taille (archivage, transport batch).

Activer la stratégie *Overflow* si le jeu de données contient quelques valeurs très grandes — elle préserve la compaction globale sans pénaliser la majorité des valeurs.

Mesurer toujours sur des jeux de données représentatifs : la meilleure option dépend fortement de la distribution des valeurs et du profil d'accès.

Perspectives d'amélioration

Vectorisation / SIMD pour accélérer les opérations bit-wise sur de gros blocs.

Extension au bit-packing 64 bits pour environnements gourmands ou architectures 64-bits.

Combinaisons avec d'autres techniques (delta encoding, RLE, VarInt) pour tirer parti des caractéristiques des données.

Benchmarks sur workloads réels (colonnes de BD, séries temporelles, IoT) pour valider les choix en production.