Data Compression for Speed-up Transmission with Bit Packing

Projet : Software Engineering Project 2025 Auteur : Aurelien JANTROY BUZENAC Encadrant : JC Regin Langage : Python 3.12 Date limite : 2 novembre 2025

1. Introduction

Le projet vise à accélérer la transmission de données numériques — plus précisément, de tableaux d'entiers (int32) — en appliquant une compression binaire sans perte appelée Bit Packing. Cette méthode consiste à regrouper les valeurs dans une représentation compacte, en supprimant les bits inutiles tout en conservant la capacité d'accéder directement à chaque élément, même après compression.

Dans les systèmes de communication (réseaux, IoT, stockage distribué), le temps total de transfert dépend à la fois :

de la latence réseau (temps fixe par transmission),

de la quantité de données transmises (limitée par la bande passante),

et du coût de compression/décompression.

L'enjeu est donc de déterminer si la compression est rentable : c'est-à-dire si le gain de temps dû à la réduction du volume transmis compense le surcoût de traitement CPU.

Trois méthodes de compression sont étudiées :

Crossing — les entiers peuvent déborder sur deux mots de 32 bits consécutifs.

Aligned — chaque entier est stocké sur un mot unique (aucun chevauchement).

Overflow — les valeurs typiques sont stockées en court format (k' bits), tandis que les rares valeurs "larges" sont externalisées dans une zone de débordement.

L'ensemble du projet est implémenté en Python, testé par pytest, et commandé via une interface CLI.

2. Cahier des charges et objectifs techniques

Les spécifications imposent :

Compression sans perte des entiers (int32), en mode bitstream.

Accès direct à la valeur i via get(i) sans décompresser tout le tableau.

Trois variantes fonctionnelles : crossing, aligned, overflow.

Sérialisation binaire (entête + données compactées).

Benchmarks de performance et calcul du seuil de rentabilité.

Code propre, structuré en modules (bitpack/) avec tests automatisés (tests/).

Un rapport PDF documentant conception, protocole et résultats.

Le projet doit en outre :

être développé individuellement,

être versionné sur GitHub.

Chaque méthode implémente la même interface :

compress(array) → PackedData decompress(array, PackedData) → None get(i, PackedData) → int

La fabrique (factory.py) choisit dynamiquement la bonne classe à instancier selon le paramètre --format.

3.2 Détails des trois variantes Crossing

Chaque entier est encodé sur k bits, placés à la suite les uns des autres dans un flux de 32 bits. Si un entier ne tient pas dans le mot courant, il déborde sur le suivant. Cette méthode maximise la compaction mais rend la lecture bit à bit plus coûteuse.

Aligned

Chaque entier est stocké dans le mot 32 bits suivant, même si une partie du mot reste inutilisée. Cette approche simplifie les calculs et accélère get(i), au prix d'un ratio légèrement moins bon.

Overflow

On choisit deux tailles :

k' bits pour la majorité des valeurs "petites",

k bits pour les valeurs exceptionnelles (outliers).

Chaque entier reçoit un bit de flag :

0 = valeur directe sur k' bits,

1 = index vers la zone de débordement.

Cela permet de limiter le gaspillage de bits quand seules quelques valeurs nécessitent plus de place.

3.3 Gestion de l'accès direct

L'accès get(i) repose sur un calcul de position :

position du mot 32 bits contenant le début de l'élément,

décalage à l'intérieur de ce mot,

combinaison éventuelle avec le mot suivant.

Ainsi, l'accès se fait en O(1) sans décompression globale.

4. Méthodologie expérimentale 4.1 Protocole de mesure

Pour obtenir des mesures stables et reproductibles :

Les chronomètres utilisent time.perf_counter_ns() (résolution nanoseconde).

Chaque opération (compress, decompress, get) est mesurée après :

3 tours de chauffe (warm-ups) pour éviter les effets de cache,

7 à 10 répétitions pour calculer une médiane robuste.

Les accès get() sont échantillonnés sur un sous-ensemble aléatoire de 100 000 indices.

4.2 Scénarios de données

Deux générateurs sont utilisés :

Uniform(k) : toutes les valeurs ont exactement k bits (cas simple).

Skewed(k_small, k_large, ratio) : majorité de petites valeurs, rares grandes valeurs (modèle réaliste pour l'overflow).

4.3 Équations de temps de transmission Terme Signification Unité t latence réseau (temps fixe par transmission) s B bande passante (Mb/s) Mb/s n nombre d'entiers — T_comp / T_decomp temps de compression/décompression s payload taille compressée (bits) bits

Formules :

T_no = t + (32n) / B # transmission brute T_yes = t + T_comp + (|payload| / B) + T_decomp Gain = T_no - T_yes # positif = compression rentable

5.1 Format Crossing — Scénario Uniform (n = 200 000, k = 12) Paramètre Valeur Taille brute 6 400 000 bits Taille compressée 2 400 416 bits Ratio de compression 0.3751 Temps de compression 85.9 ms Temps de décompression 78.3 ms Temps moyen d'accès get(i) 451.8 ns T_no (10 Mbps, 30 ms) 670.0 ms T_yes 434.2 ms Gain +235.8 ms (bénéfique)

Analyse : la variante crossing atteint une excellente compacité (−62 % de taille) et reste rentable dès une latence de 30 ms. Le coût CPU est modéré, ce qui prouve que l'optimisation bit-à-bit reste utile même en Python.

5.2 Format Aligned — Scénario Uniform (n = 200 000, k = 12) Paramètre Valeur Taille brute 6 400 000 bits Taille compressée 3 200 416 bits Ratio de compression 0.5001 Temps de compression 36.6 ms Temps de décompression 53.9 ms Temps moyen d'accès get(i) 346.0 ns T_no (10 Mbps, 30 ms) 670.0 ms T_yes 440.5 ms Gain +229.5 ms (bénéfique)

Analyse : la méthode aligned est 2 fois plus rapide à la compression et environ 1.5 fois plus rapide à l'accès. Elle sacrifie environ 12 % de compaction mais gagne en simplicité : c'est le meilleur choix pour un système à forte contrainte de latence.

5.3 Format Overflow — Scénario Skewed (n = 300 000, k_small = 6, k_large = 20, ratio = 0.001) Paramètre Valeur Taille brute 9 600 000 bits Taille compressée 3 006 240 bits Ratio de compression 0.3131 Temps de compression 539.2 ms Temps de décompression 188.6 ms Temps moyen d'accès get(i) 866.6 ns T_no (5 Mbps, 50 ms) 1970.0 ms T_yes 1379.0 ms Gain +591.0 ms (bénéfique)

Analyse : sur des données asymétriques (0.1 % d'outliers), la méthode overflow réduit la taille de 69 %. Malgré un temps CPU plus élevé, la compression reste largement bénéfique (gain de ~30 %). Cette stratégie se montre idéale pour des flux réels très inégaux (réseaux, séries temporelles).

5.4 Synthèse des résultats Format Ratio ↓ (taille) T_comp (ms) T_decomp (ms) Gain total (ms) Rendement Crossing 0.3751 85.9 78.3 +235.8 Très bon équilibre taille/CPU Aligned 0.5001 36.6 53.9 +229.5 Très rapide, ratio moyen Overflow 0.3131 539.2 188.6 +591.0 Excellent sur données hétérogènes

Les résultats chiffrés précédents proviennent directement des fichiers CSV générés par la CLI lors de l'exécution des benchmarks :

Fichier CSV	Scénario testé	Format
`bench_crossing_uniform_k12.csv`	Uniform(k=12, n=200000)	Crossing
`bench_aligned_uniform_k12.csv`	Uniform(k=12, n=200000)	Aligned
`bench_overflow_skewed.csv`	Skewed(k_small=6, k_large=20, ratio=0.001, n=300000)	Overflow

Chaque fichier contient les mesures de taille, ratio et temps d'exécution utilisées dans ce rapport.
Ces fichiers sont inclus dans le dépôt GitHub afin de permettre la **reproduction et la vérification** des résultats expérimentaux.

6. Validation de la fidélité et de l'accès direct

Pour chaque méthode :

Un tableau aléatoire est compressé.

10 000 appels get(i) sont effectués avec comparaison au tableau original.

Le tableau est ensuite décompressé et comparé bit à bit à l'original.

Exemples de résultats (validation_crossing.md) :

Vérification Résultat Mismatches get(i) 0 / 10 000 Mismatches decompress 0 / 200 000 Verdict OK Ratio (comp/brut) 0.375 T_comp / T_decomp 91 ms / 81 ms

→ Aucune perte de fidélité. La valeur extraite par get(i) correspond toujours à celle du tableau initial. Le rapport validate confirme la conformité fonctionnelle des trois formats.

7. analyse comparative Format Compacité Vitesse Cas d'usage idéal Crossing Excellente (bits bien remplis) Moyenne Transmission où la taille est critique Aligned Moyenne Excellente Systèmes temps réel, faible latence Overflow Très bonne sur données hétérogènes Moyenne Séries avec rares valeurs extrêmes

Crossing offre la meilleure compression, mais le calcul de position bit à bit coûte un peu plus en CPU.

Aligned simplifie tout : bon compromis performance / simplicité.

Overflow surpasse les deux autres lorsque les données sont asymétriques (loi de Pareto typique dans les flux réseau).

Le gain devient significatif pour des latences > 20–30 ms et des bandes passantes ≤ 10 Mb/s. Sous ces conditions, la compression réduit le temps total de transmission de 20 à 30 %.

8. Conclusion

Le projet implémente avec succès une compression Bit Packing à accès direct, déclinée en trois variantes :

crossing, aligned, overflow,

toutes testées, validées et mesurées.

Les objectifs initiaux sont atteints :

Aucune perte d’accès (get(i) = valeur d’origine),

Fidélité totale après décompression,

Compression rentable pour des conditions réalistes.

Les tests et benchmarks (pytest, validate, CSV) garantissent la reproductibilité.