# L'informatique des entrepôts de données

**Daniel Lemire**

# SEMAINE 6
# Compression dans les bases de données

## 6.1. Présentation de la semaine

Nous avons déjà vu à la semaine précédente le codage par plage, et le rôle important que cette technique de compression joue dans la construction de certains index, comme les index bitmaps. Il existe cependant plusieurs techniques de compression qui sont importantes dans les entrepôts de données.

La compression procure plusieurs bénéfices. D'abord, elle permet de stocker plus d'information sur des disques. Mais ce n'est pas la fonction la plus importante en ce qui nous concerne. La compression permet aussi de passer plus de données du disque aux microprocesseurs. En effet, des données compressées se chargent souvent plus rapidement en mémoire. De plus, la compression permet de conserver plus de données en mémoire.

Nous verrons cette semaine plusieurs techniques de compression populaires.

## 6.2. Théorie de l'information

Étant donné un fichier, jusqu'à quel point pouvons-nous le compresser? Intuitivement, plus un document contient d'information, plus il sera difficile de le compresser. En effet, un fichier qui ne contiendrait qu'une seule lettre (`a`) répétée des milliers de fois ne contiendrait pas beaucoup d'information et serait donc facile à compresser (par codage par plage).

**Exemple 1.** *J'ai un texte de 3 mots "A", "B" et "C". Je choisis d'abord de représenter le premier mot sous la forme binaire "00", le second sous la forme "10" et le dernier sous la forme "11". La séquence "ABCA" devient alors "00101100" ou "00-10-11-00". Il a donc fallu 8 bits pour stocker les 4 mots, donc une moyenne de 2 bits par mot. Shannon savait qu'il était possible d'être encore plus efficace. En effet, il a représenté le premier mot avec un seul bit : "0". La séquence "ABCA" devient alors "010110" ou "0-10-11-0", donc 6 bits pour stocker 8 mots, ce qui représentait un gain substantiel ! Est il est possible de faire mieux?*

L'étude de cette mesure de la quantité d'information, en fonction de la compressibilité, fait l'objet d'une théorie générale, la théorie de l'information. On doit la théorie de l'information à Shannon [**13**]. Il fut le premier à proposer une mesure d'entropie pour l'information: c'est-à-dire une limite fondamentale à la compression des données.

La théorie de l'information, développée principalement par Claude Shannon dans les années 1940, est une discipline mathématique qui étudie la quantification, le stockage, la transmission et le traitement de l'information. Elle repose sur des concepts clés comme l'entropie, qui mesure l'incertitude ou la quantité d'information dans une source de données, et la capacité d'un canal, qui détermine la quantité maximale d'information transmissible sans erreur. Cette théorie sous-tend de nombreux domaines, notamment les télécommunications, la compression de données (comme le codage de *Huffman*), la cryptographie et l'apprentissage automatique, en fournissant des outils pour optimiser l'efficacité et la fiabilité des systèmes de communication. En analysant l'information comme une entité mesurable, elle permet de comprendre et de résoudre des problèmes liés à la transmission et à l'interprétation des données dans des environnements bruités ou limités.

Considérons une chaîne de $n$ caractères où le caractère $x$ apparaît $f(x)$ fois. La probabilité de sélectionner ce caractère, au hasard, est donc de $p(x) = f(x)/n$. La quantité d'information associée au caractère $x$ est de $-\log p(x)$[1]. L'entropie de Shannon de la chaîne de caractères est de $-\sum_x p(x) \log p(x)$ où la somme est sur l'ensemble des caractères. La quantité d'information dans le texte est de $-\sum_x np(x) \log p(x)$. Cette quantité d'information est un seuil minimal: la plupart des techniques de compression statistiques (basées sur la fréquence des caractères) ne permettent pas d'utiliser moins de $-\sum_x np(x) \log p(x)$ bits.

**Exemple 2.** *Prenons un texte de 2 mots, l'un ayant une fréquence de 500 et l'autre, de 250, pour un total de 750 ($n = 750$). Nous avons*

[1]En informatique, le logarithme est toujours en base deux.

*alors* $p(1) = 500/750 = 2/3$ *et* $p(2) = 250/750 = 1/3$. *La quantité d'information du premier mot est de* $-\log_2(2/3) \approx 0,58$ *et la quantité d'information du second mot est de* $-\log_2(1/3) \approx 1,58$. *Pour coder le texte, il faut donc au moins* $(-(2/3)\log_2(2/3) - (1/3)\log_2(1/3)) \times 750 \approx 688$ *bits, soit environ 86 octets. Vous pouvez tester ce résultat en générant un texte de seulement 2 mots qui comporte les fréquences prescrites. Vous constaterez que, quels que soient vos efforts de compression avec zip ou avec un autre outil, vous n'arriverez pas, sans tricher, à un fichier qui fait moins de 688 bits.*

## 6.3. Compression statistique

Il y a plusieurs techniques de compression statistiques qui sont utilisées dans les entrepôts de données tel que le codage de Huffman et la compression Lempel-Ziv-Welch [**18**]. Ces techniques compressent efficacement les données, surtout lorsque certains caractères, ou certains chaînes de caractères sont fréquemment répétées. Par contre, elles sont relativement lentes bien qu'il existe des stratégies permettant d'accélérer la décompression [**15**].

Le codage de Huffman et le codage Lempel-Ziv-Welch (LZW) sont deux techniques de compression de données largement utilisées pour réduire la taille des fichiers sans perte d'information. Le codage de Huffman, basé sur la fréquence des symboles, utilise un codage à longueur variable pour minimiser la taille des données, tandis que LZW, une méthode de compression par dictionnaire, identifie et remplace les motifs répétitifs par des références compactes. Ces algorithmes sont essentiels dans des applications comme la compression de fichiers (ZIP, GIF, PNG) et les télécommunications.

Le codage de Huffman, développé par David A. Huffman en 1952, est une méthode de compression sans perte qui attribue des codes binaires de longueur variable aux symboles en fonction de leur fréquence d'apparition. Les symboles les plus fréquents reçoivent des codes plus courts, réduisant ainsi la taille globale des données. L'algorithme construit un arbre binaire (arbre de Huffman) où les feuilles représentent les symboles, et les chemins vers ces feuilles donnent les codes.

(a) **Calcul des fréquences** : Compter la fréquence de chaque symbole dans les données.

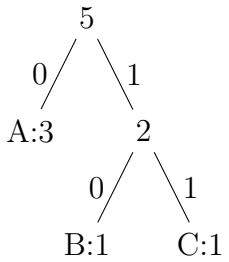
(b) **Construction de l'arbre** : Créer une file de priorité avec les symboles (nœuds) triés par fréquence. À chaque étape, extraire les deux nœuds de plus faible fréquence, les combiner en un nœud parent (dont la fréquence est la somme des deux), et

réinsérer ce nœud dans la file. Répéter jusqu'à obtenir un seul nœud (la racine).

(c) **Génération des codes** : Parcourir l'arbre pour assigner un bit "0" aux branches gauches et "1" aux branches droites, formant ainsi les codes des symboles.

Considérons la chaîne "AABAC" avec les fréquences : A (3), B (1), C (1).

- Fréquences : A (3), B (1), C (1).
- Construction de l'arbre : Combiner B et C (fréquence 2), puis combiner ce nœud avec A (fréquence 5).
- Codes : A = 0, B = 10, C = 11.




La chaîne "AABAC" est encodée en "001011" (A:0, A:0, B:10, A:0, C:11), réduisant la taille par rapport à un codage fixe (par exemple, 10 bits avec 2 bits par symbole).

Le codage Lempel-Ziv-Welch (LZW), développé par Abraham Lempel, Jacob Ziv, et Terry Welch dans les années 1970-1980, est une méthode de compression par dictionnaire qui remplace les sous-chaînes répétitives par des codes numériques. LZW construit dynamiquement un dictionnaire de motifs au fur et à mesure qu'il parcourt les données, ce qui le rend adaptatif et efficace pour des données avec des répétitions.

(a) **Initialisation** : Créer un dictionnaire contenant tous les symboles de base (par exemple, les caractères ASCII) avec des codes numériques.

(b) **Parcours des données** : Lire la séquence caractère par caractère, en construisant des sous-chaînes. Lorsqu'une sous-chaîne n'est pas dans le dictionnaire, émettre le code de la sous-chaîne précédente, ajouter la nouvelle sous-chaîne au dictionnaire avec un nouveau code, et continuer.

(c) **Décompression** : Reconstruire le dictionnaire à partir des codes reçus, en suivant le même processus d'ajout des motifs.

Considérons la chaîne "ABABABA" avec un dictionnaire initial {A:1, B:2}.

- Étape 1 : Lire "A", émettre 1, continuer.

- Étape 2 : Lire “AB”, ajouter “AB:3” au dictionnaire, émettre 2 (pour “B”), continuer.
- Étape 3 : Lire “ABA”, ajouter “BA:4”, émettre 3 (pour “AB”), continuer.
- Étape 4 : Lire “ABA”, ajouter “ABA:5”, émettre 3 (pour “AB”), continuer.

Sortie : [1, 2, 3, 3]. La décompression reconstruit la chaîne en utilisant les codes et le dictionnaire dynamique.

- **Huffman** : Optimal pour les données avec des fréquences de symboles inégales, mais nécessite une passe initiale pour calculer les fréquences. Moins efficace pour les motifs répétitifs complexes.
- **LZW** : Adaptatif, ne nécessite pas de pré-analyse, et excelle pour les données avec des motifs répétitifs. Plus complexe à implémenter, mais largement utilisé dans des formats comme GIF et TIFF.

## 6.4. Codage des différences

Le codage des différences compresse les données en stockant les différences entre les valeurs successives plutôt que les valeurs elles-mêmes. Si les valeurs sont triées, ou partiellement triées (par bloc), ce mode de compression peut être très efficace. Elle s'applique à plusieurs types de données, y compris les nombres à virgule flottante [**6, 11, 9, 4**]. On l'utilise dans les bases de données orientées colonnes [**14, 8, 7, 10**].

Formellement, le code fonctionne comme suit. Nous voulons coder une liste d'entiers $x_1, x_2, \ldots$ dans $\{0, 1, 2, \ldots, N-1\}$. La première valeur est toujours stockée sans compression. Ensuite, on stocke les valeurs $x_{i+1} - x_i \bmod N^2$ pour $i = 1, 2, \ldots$. On s'attend à ce que la plupart du temps, cette différence soit petite. Il se trouve qu'on peut stocker en mémoire de petits entiers en utilisant peu de bits [**17, 19, 16, 1**]. La technique la plus simple est le codage à octet variable [**12, 3, 17, 20**] : on stocke les premiers 7 bits de la valeur, puis on utilise un bit supplémentaire pour indiquer si un octet supplémentaire sera nécessaire pour le stockage, et ainsi de suite.

```
1
2 public class VariableByte {
3
4     private static byte extract7bits(int i, long val) {
```

[2]On peut aussi stocker la valeur $x_i \oplus x_{i+1}$ où $\oplus$ est le ou exclusif.

```
5         return (byte) ((val >> (7 * i)) & ((1 << 7) -
              1));
6     }
7
8     private static byte extract7bitsmaskless(int i, long
          val) {
9         return (byte) ((val >> (7 * i)));
10    }
11
12    public int compress(int[] in, int inlength, byte[]
          out) {
13        if (inlength == 0)
14            return 0;
15        int outpostmp = 0;
16        for (int k = 0; k < inlength; ++k) {
17            final long val = in[k] & 0xFFFFFFFFL;
18            if (val < (1 << 7)) {
19                out[outpostmp++] = (byte) (val | (1 <<
                      7));
20            } else if (val < (1 << 14)) {
21                out[outpostmp++] = (byte)
                      extract7bits(0, val);
22                out[outpostmp++] = (byte)
                      (extract7bitsmaskless(1, (val)) | (1
                      << 7));
23            } else if (val < (1 << 21)) {
24                out[outpostmp++] = (byte)
                      extract7bits(0, val);
25                out[outpostmp++] = (byte)
                      extract7bits(1, val);
26                out[outpostmp++] = (byte)
                      (extract7bitsmaskless(2, (val)) | (1
                      << 7));
27            } else if (val < (1 << 28)) {
28                out[outpostmp++] = (byte)
                      extract7bits(0, val);
29                out[outpostmp++] = (byte)
                      extract7bits(1, val);
30                out[outpostmp++] = (byte)
                      extract7bits(2, val);
31                out[outpostmp++] = (byte)
                      (extract7bitsmaskless(3, (val)) | (1
                      << 7));
32            } else {
33                out[outpostmp++] = (byte)
                      extract7bits(0, val);
34                out[outpostmp++] = (byte)
                      extract7bits(1, val);
```

```
35                  out[outpostmp++] = (byte)
                        extract7bits(2, val);
36                  out[outpostmp++] = (byte)
                        extract7bits(3, val);
37                  out[outpostmp++] = (byte)
                        (extract7bitsmaskless(4, (val)) | (1
                        << 7));
38              }
39          }
40          return outpostmp;
41      }
42
43      public int uncompress(byte[] in, int inlength,
44              int[] out) {
45          int p = 0;
46          int finalp = inlength;
47          int tmpoutpos = 0;
48          for (int v = 0; p < finalp; out[tmpoutpos++] =
                v) {
49              v = in[p] & 0x7F;
50              if (in[p] < 0) {
51                  p += 1;
52                  continue;
53              }
54              v = ((in[p + 1] & 0x7F) << 7) | v;
55              if (in[p + 1] < 0) {
56                  p += 2;
57                  continue;
58              }
59              v = ((in[p + 2] & 0x7F) << 14) | v;
60              if (in[p + 2] < 0 ) {
61                  p += 3;
62                  continue;
63              }
64              v = ((in[p + 3] & 0x7F) << 21) | v;
65              if (in[p + 3] < 0) {
66                  p += 4;
67                  continue;
68              }
69              v = ((in[p + 4] & 0x7F) << 28) | v;
70              p += 5;
71          }
72          return tmpoutpos;
73      }
74
75  }
```

VariableByte.java

## 6.5. Codage des chaînes fréquentes

Il existe une stratégie de compression remarquablement simple qui est souvent très efficace. Le codage des longues chaînes répétées [**2**] est utilisé par Google [**5**] et dans le système de bases de données IBM DB2. Le principe est fort simple : on traverse l'ensemble de la base de données, et on cherche de longues chaînes de caractères répétées. On construit alors un dictionnaire de ces chaînes, et chaque fois qu'on en rencontre une dans une table, on la remplace par un identifiant correspondant à la chaînes en question.

**Exemple 3.** *Prenons la chaîne de caractères* `aaabbbacdbacdbacdbabbbb`. *On découvre la chaîne de caractères* `bacd`. *On lui associe donc un identifiant, par exemple l'entier 1, et on substitue cet identifiant dans la chaîne originale:* `aaabb111babbbb`.

## 6.6. Compression des préfixes

Il se produit souvent que, dans une séquence de valeurs, plusieurs préfixes sont récurrents. Prenons par exemple la séquence AAABB, AAABCC, AAACCC. La compression des préfixes calcule d'abord une chaîne de caractères qui permettra d'identifier des préfixes. Dans ce cas particulier, prenons la chaîne AAABCC. On prend alors chaque chaîne de caractères dans notre séquence et on la compare avec cette chaîne de référence. D'abord on constate que la chaîne AAABB réutilise les 4 premiers caractères de la chaîne de référence: on peut donc la stocker comme étant 4B. La chaîne AAABCC est indentique à la chaîne de référence: il n'y a donc rien à stocker. La dernière chaîne, AAACCC, peut être codée comme étant `3ccc`.

A consulter : `http://msdn.microsoft.com/en-us/library/cc280464.aspx`

## 6.7. Questions d'approfondissement

(a) Le site Twitter permet aux utilisateurs de publier des messages de 140 caractères ou moins. Est-ce que la compression Lempel-Ziv-Welch est appropriée dans ce cas?
(b) Soit les nombres de 0 à $2^{32}-1$. En utilisant 32 bits par nombre, combien faut-il d'octets pour les stocker? En utilisant le codage des différences, quel est le taux de compression?

## 6.8. Réponses suggérées

(a) La compression Lempel-Ziv-Welch ne compresse que les chaînes de caractères suffisamment longues. Malheureusement, elle risque d'être inefficace sur des chaînes de 140 caractères.

(b) Il faut $2^{32} \times 4 = 2^{34}$ octets pour stocker la liste sans compression. Avec le codage des différences, chaque différence est de 1, ce qui ne requiert qu'un seul octet. Le taux de compression est donc de 1:4.

# BIBLIOGRAPHIE


1. V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.
2. J. Bentley and D. McIlroy. Data compression with long repeated strings. 135(1-2):1–11, 2001.
3. B. Bhattacharjee, L. Lim, T. Malkemus, G. Mihaila, K. Ross, S. Lau, C. McArthur, Z. Toth, and R. Sherkat. Efficient index compression in DB2 LUW. 2(2):1462–1473, 2009.
4. M. Burtscher and P. Ratanaworabhan. High throughput compression of double-precision floating-point data. pages 293–302, 2007.
5. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.
6. V. Engelson, D. Fritzson, and P. Fritzson. Lossless compression of high-volume numerical data from simulations. pages 574–586, 2000.
7. A. L. Holloway and D. J. DeWitt. Read-optimized databases, in depth. *Proceedings of the VLDB Endowment*, 1(1):502–513, 2008.
8. A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: Compression and bandwidth trade offs for database scans. pages 389–400, New York, NY, USA, 2007. ACM.
9. P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, 2006.
10. V. Raman and G. Swart. How to wring a table dry: entropy compression of relations and querying of compressed relations. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, pages 858–869. VLDB Endowment, 2006.

11. P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. pages 133–142, 2006.
12. F. Scholer, H. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. pages 222–229, New York, NY, USA, 2002. ACM.
13. C. E. Shannon. A mathematical theory of communications. *Bell Syst. Tech. J*, 1948.
14. M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB'05*, pages 553–564, 2005.
15. T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.
16. H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 401–410, New York, NY, USA, 2009. ACM.
17. J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, pages 387–396, New York, NY, USA, 2008. ACM.
18. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. 24(5):530–536, 1978.
19. J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2):6, 2006.
20. M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 59–71, Washington, DC, USA, 2006. IEEE Computer Society.