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 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

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

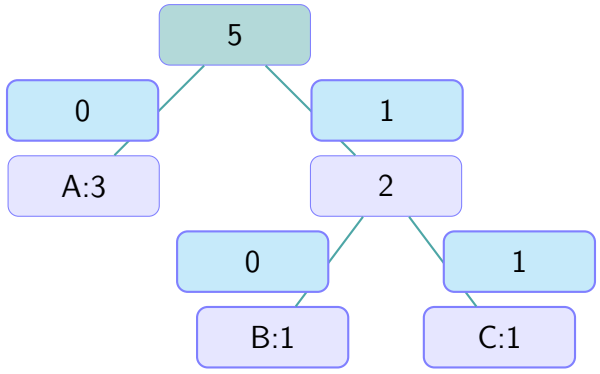
variable pour minimiser la taille des données, tandis que LZW, une méthode de compression par dictionnaire, identifie et remplace les motifs répétitifs par des références compactes. Ces algorithmes sont essentiels dans des applications comme la compression de fichiers (ZIP, GIF, PNG) et les télécommunications.

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 est une technique de compression qui repose sur le stockage des différences entre des valeurs successives plutôt que sur les valeurs elles-mêmes. Cette approche est particulièrement efficace lorsque les données sont triées ou partiellement triées (par exemple, par blocs), car les différences entre valeurs consécutives tendent à être petites. Cette méthode s'applique à divers types de données, y compris les nombres entiers, les nombres à virgule flottante [**6**, **11**, **9**, **4**], et même les structures complexes comme les index de bases de données. Elle est largement utilisée dans les bases de données orientées colonnes [**14**, **8**, **7**, **10**], où les colonnes contiennent souvent des valeurs similaires ou séquentiellement corrélées.

6.4.1. Principes de base

Formellement, le codage des différences fonctionne comme suit pour une liste d'entiers $x_1, x_2, \ldots$ dans l'ensemble $\{0, 1, 2, \ldots, N-1\}$. La première valeur, x_1, est stockée sans compression pour servir de point de référence. Ensuite, pour chaque $i \geq 1$, on calcule et stocke la différence $x_{i+1} - x_i \bmod N$. Une alternative consiste à stocker la valeur obtenue par l'opération de ou exclusif (XOR) entre valeurs consécutives, soit $x_i \oplus x_{i+1}$[2]. L'hypothèse clé est que les différences (ou les résultats XOR) sont généralement petites, ce qui permet de les coder avec moins de bits que les valeurs originales.

Pour exploiter cette propriété, les différences sont encodées à l'aide de techniques optimisées pour les petits entiers, comme le codage à octet variable [**12**, **3**, **17**, **20**]. Dans ce codage, une valeur est représentée par un ou plusieurs octets, où les 7 premiers bits de chaque octet codent une partie de la valeur, et le bit restant (le bit de continuation) indique si un octet supplémentaire est nécessaire. Par exemple, une valeur inférieure à $2^7 = 128$ est codée en un seul octet, tandis qu'une valeur plus grande peut nécessiter plusieurs octets.

[2]L'opération XOR est particulièrement utile pour les données binaires ou lorsque les différences ne sont pas nécessairement petites mais présentent des motifs répétitifs.

Exemple 3: Exemple illustratif

Considérons la liste d'entiers triés $[3, 5, 6, 8, 12]$. En appliquant le codage des différences :

- La première valeur, $x_1 = 3$, est stockée telle quelle.
- Pour $i = 1$, la différence est $x_2 - x_1 = 5 - 3 = 2$.
- Pour $i = 2$, la différence est $x_3 - x_2 = 6 - 5 = 1$.
- Pour $i = 3$, la différence est $x_4 - x_3 = 8 - 6 = 2$.
- Pour $i = 4$, la différence est $x_5 - x_4 = 12 - 8 = 4$.

La séquence compressée est donc $[3, 2, 1, 2, 4]$. Si $N = 16$ (les valeurs sont dans $\{0, 1, \ldots, 15\}$), chaque différence peut être codée avec un codage à octet variable. Par exemple, la valeur 2 est codée comme un octet `00000010` (bit de continuation à 0), ce qui est beaucoup plus compact que de stocker chaque entier complet sur 4 octets.

Pour la décompression, on commence par la première valeur ($x_1 = 3$) et on ajoute successivement les différences : $3 + 2 = 5$, $5 + 1 = 6$, $6 + 2 = 8$, $8 + 4 = 12$. La séquence originale est ainsi restaurée sans perte.

6.4.2. Avantages de la méthode

Le codage des différences présente plusieurs atouts :

- **Efficacité pour les données corrélées :** Lorsque les valeurs consécutives sont proches (comme dans des listes triées ou des séries temporelles), les différences sont petites et peuvent être codées avec un minimum de bits.
- **Flexibilité :** La méthode s'applique à divers types de données, y compris les nombres à virgule flottante (en codant les différences entre les parties significatives) et les index de bases de données.
- **Compatibilité avec d'autres techniques :** Le codage des différences peut être combiné avec des méthodes comme le codage de Huffman ou le codage par plages pour optimiser davantage la compression [19, 16].
- **Vitesse :** Les opérations de calcul des différences et de décompression sont rapides, ce qui est crucial pour les bases de données où les accès fréquents sont nécessaires.

6.4.3. Limites et considérations

Malgré ses avantages, le codage des différences a certaines contraintes :

- **Dépendance à l'ordre des données :** L'efficacité dépend fortement de la proximité des valeurs consécutives. Si les données ne sont pas triées ou présentent des variations importantes, les différences peuvent être grandes, réduisant l'efficacité du codage.
- **Sensibilité aux erreurs :** Une erreur dans une différence peut corrompre toutes les valeurs suivantes lors de la décompression, ce qui nécessite des mécanismes de vérification.
- **Surcharge pour la première valeur :** La première valeur non compressée peut représenter une surcharge non négligeable si la séquence est courte.

Exemple 4: Exemple complémentaire

Prenons une colonne d'une base de données orientée colonnes contenant les timestamps d'événements : $[1000, 1005, 1010, 1012]$. En codant les différences :

- Première valeur : 1000 (stockée telle quelle).
- Différences : $1005 - 1000 = 5$, $1010 - 1005 = 5$, $1012 - 1010 = 2$.

La séquence compressée est $[1000, 5, 5, 2]$. En utilisant un codage à octet variable, les différences 5, 5 et 2 peuvent être codées avec un seul octet chacune, tandis que 1000 nécessite deux octets. Comparé au stockage de quatre entiers sur 4 octets chacun (16 octets au total), la séquence compressée peut être stockée en seulement 5 octets, soit une réduction significative.

6.4.4. Applications pratiques

Le codage des différences est largement utilisé dans plusieurs domaines :

- **Bases de données orientées colonnes :** Des systèmes comme Vertica ou MonetDB exploitent cette technique pour compresser les colonnes contenant des valeurs séquentiellement corrélées, comme les identifiants ou les timestamps [14, 7].
- **Séries temporelles :** Dans les applications IoT ou les bases de données de capteurs, les données temporelles (comme les températures ou les mesures de pression) présentent souvent des variations faibles, idéales pour le codage des différences.
- **Compression d'index :** Les index inversés dans les moteurs de recherche utilisent le codage des différences pour compresser

les listes de documents, où les identifiants de documents sont souvent triés et proches les uns des autres [19, 16].

- **Traitement des nombres à virgule flottante :** En traitement d'images ou en simulation scientifique, les différences entre valeurs flottantes consécutives peuvent être codées pour réduire l'espace de stockage [9].

6.4.5. Optimisations avancées

Pour améliorer l'efficacité du codage des différences, plusieurs optimisations sont possibles :

- **Pré-tri des données :** Si les données ne sont pas triées, un pré-tri par blocs peut réduire les différences, bien que cela augmente la complexité initiale.
- **Codage entropique :** Les différences peuvent être codées avec des méthodes comme le codage de Huffman ou le codage arithmétique pour exploiter leur distribution statistique.
- **Codage par plages :** Si les différences sont concentrées dans certaines plages, un codage par plages peut réduire davantage la taille des données [1].

En conclusion, le codage des différences est une technique de compression puissante et polyvalente, particulièrement adaptée aux données séquentiellement corrélées. Son efficacité dépend de la nature des données et de l'encodage des différences, mais sa simplicité et sa compatibilité avec d'autres méthodes en font un outil essentiel dans les systèmes de gestion de données modernes.

```java
/**
 * Classe mettant en oeuvre la compression Variable Byte pour les entiers.
 * Cette technique encode les entiers en un nombre variable d'octets.
 */
public class VariableByte {

    /**
     * Extrait 7 bits d'une valeur longue.
     * @param i Position (multiple de 7 bits).
     * @param val Valeur.
     * @return Les 7 bits sous forme d'octet.
     */
    private static byte extract7bits(int i, long val) {
        return (byte) ((val >> (7 * i)) & ((1 << 7) - 1));
    }
```

```
16
17      /**
18       * Extrait 7 bits sans masque, pour le dernier octet.
19       * @param i Position (multiple de 7 bits).
20       * @param val Valeur.
21       * @return Les 7 bits sous forme d'octet.
22       */
23      private static byte extract7bitsmaskless(int i, long
            val) {
24          return (byte) (val >> (7 * i));
25      }
26
27      /**
28       * Compresse un tableau d'entiers en format Variable
             Byte.
29       * @param in Tableau des entiers.
30       * @param inlength Longueur du tableau.
31       * @param out Tableau de sortie pour les octets.
32       * @return Nombre d'octets dans out.
33       */
34      public int compress(int[] in, int inlength, byte[]
            out) {
35          if (inlength == 0) {
36              return 0;
37          }
38          int posOut = 0; // Position dans le tableau de
                sortie
39          for (int k = 0; k < inlength; ++k) {
40              final long val = in[k] & 0xFFFFFFFFL;
41              if (val < (1L << 7)) { // 1 octet
42                  out[posOut++] = (byte) (val | (1 << 7));
43              } else if (val < (1L << 14)) { // 2 octets
44                  out[posOut++] = extract7bits(0, val);
45                  out[posOut++] = (byte)
                        (extract7bitsmaskless(1, val)
46                              | (1 << 7));
47              } else if (val < (1L << 21)) { // 3 octets
48                  out[posOut++] = extract7bits(0, val);
49                  out[posOut++] = extract7bits(1, val);
50                  out[posOut++] = (byte)
                        (extract7bitsmaskless(2, val)
51                              | (1 << 7));
52              } else if (val < (1L << 28)) { // 4 octets
53                  out[posOut++] = extract7bits(0, val);
54                  out[posOut++] = extract7bits(1, val);
55                  out[posOut++] = extract7bits(2, val);
56                  out[posOut++] = (byte)
                        (extract7bitsmaskless(3, val)
```

```
57                          | (1 << 7));
58              } else { // 5 octets (max pour int)
59                  out[posOut++] = extract7bits(0, val);
60                  out[posOut++] = extract7bits(1, val);
61                  out[posOut++] = extract7bits(2, val);
62                  out[posOut++] = extract7bits(3, val);
63                  out[posOut++] = (byte)
                        (extract7bitsmaskless(4, val)
64                          | (1 << 7));
65              }
66          }
67          return posOut;
68      }
69
70      /**
71       * Retrouve le tableau d'octets en entiers originaux.
72       * @param in Tableau des octets.
73       * @param inlength Longueur du tableau.
74       * @param out Tableau de sortie pour les entiers.
75       * @return Nombre d'entiers dans out.
76       */
77      public int uncompress(byte[] in, int inlength, int[]
            out) {
78          int posIn = 0; // Position dans le tableau
79          int fin = inlength; // Fin du tableau
80          int posOut = 0; // Position dans le tableau de
                sortie
81          while (posIn < fin) {
82              int v = in[posIn] & 0x7F; // Premier octet
83              if (in[posIn] < 0) { // Bit de continuation
84                  out[posOut++] = v;
85                  posIn += 1;
86                  continue;
87              }
88              v |= (in[posIn + 1] & 0x7F) << 7; // 2e octet
89              if (in[posIn + 1] < 0) {
90                  out[posOut++] = v;
91                  posIn += 2;
92                  continue;
93              }
94              v |= (in[posIn + 2] & 0x7F) << 14; // 3e
                    octet
95              if (in[posIn + 2] < 0) {
96                  out[posOut++] = v;
97                  posIn += 3;
98                  continue;
99              }
```

```
100             v |= (in[posIn + 3] & 0x7F) << 21; // 4e
                    octet
101             if (in[posIn + 3] < 0) {
102                 out[posOut++] = v;
103                 posIn += 4;
104                 continue;
105             }
106             // 5e octet
107             v |= (in[posIn + 4] & 0x7F) << 28;
108             out[posOut++] = v;
109             posIn += 5;
110         }
111         return posOut;
112     }
113 }
```

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 des systèmes de grande envergure tels que Google [5] et 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'une occurrence est détectée dans une table, on la remplace par un identifiant correspondant à la chaîne en question.

6.5.1. Processus détaillé

Le processus de codage des chaînes fréquentes peut être décomposé en plusieurs étapes :

(a) **Analyse des données :** On parcourt la base de données pour identifier les sous-chaînes qui apparaissent fréquemment. Cette étape peut utiliser des algorithmes comme le suffix tree ou l'algorithme de Knuth-Morris-Pratt pour détecter efficacement les répétitions.

(b) **Construction du dictionnaire :** Les chaînes fréquentes sont stockées dans un dictionnaire, où chaque chaîne est associée à un identifiant unique, généralement un entier ou un code compact. Le dictionnaire est conçu pour minimiser l'espace de stockage tout en permettant un accès rapide.

(c) **Substitution :** Chaque occurrence d'une chaîne fréquente dans les données est remplacée par son identifiant. Cette étape réduit la taille des données, surtout si les chaînes répétées sont longues et fréquentes.

(d) **Stockage et décompression :** Le dictionnaire est stocké avec les données compressées. Lors de la décompression, chaque identifiant est remplacé par la chaîne correspondante en consultant le dictionnaire.

Exemple 5: Exemple illustratif

Prenons la chaîne de caractères `aaabbbacdbacdbacdbabbbb`. En analysant les répétitions, on découvre que la sous-chaîne `bacd` apparaît trois fois. On lui associe un identifiant, par exemple l'entier 1, et on substitue cet identifiant dans la chaîne originale : `aaabb111babbbb`. Si une autre sous-chaîne, par exemple `bbb`, est également fréquente, on peut lui associer l'identifiant 2, ce qui donnerait `aa2b112a2`. Cette compression réduit la longueur de la chaîne de 22 caractères à 10, soit une réduction significative.

6.5.2. Avantages de la méthode

Le codage des chaînes fréquentes présente plusieurs avantages :

- **Efficacité :** La méthode est particulièrement performante pour les bases de données contenant des motifs répétitifs, comme des logs système, des séquences ADN, ou des textes structurés.
- **Simplicité :** L'algorithme est intuitif et peut être implémenté avec des structures de données standards comme des tables de hachage pour le dictionnaire.
- **Compatibilité :** Cette technique peut être intégrée à d'autres méthodes de compression, comme le codage de Huffman ou la compression Lempel-Ziv, pour améliorer les performances globales.

6.5.3. Limites et considérations

Malgré ses atouts, cette méthode a certaines limites :

- **Taille du dictionnaire :** Si les chaînes fréquentes sont nombreuses mais peu répétées, le dictionnaire peut devenir volumineux, réduisant l'efficacité de la compression.

- **Complexité initiale :** L'analyse des répétitions peut être coûteuse en temps de calcul, surtout pour de très grandes bases de données.
- **Dépendance au contexte :** La méthode est moins efficace pour les données sans motifs répétitifs clairs, comme des images compressées ou des données aléatoires.

Exemple 6: Exemple complémentaire

Considérons une base de données textuelle contenant les phrases suivantes : `bonjour le monde`, `bonjour le soleil`, `bonjour le ciel`. La sous-chaîne `bonjour le ` apparaît dans chaque phrase. On associe à cette sous-chaîne l'identifiant 1, ce qui donne : `1monde`, `1soleil`, `1ciel`. Le dictionnaire contient une seule entrée : {1: `bonjour le ` }. Cette compression réduit l'espace requis tout en conservant la possibilité de restaurer les phrases originales.

6.5.4. Applications pratiques

Cette technique est largement utilisée dans des contextes où les données textuelles ou semi-structurées présentent des répétitions fréquentes. Par exemple :

- **Moteurs de recherche :** Google utilise des variantes de cette méthode pour compresser les index de pages web, où les URL ou les balises HTML contiennent des motifs répétitifs.
- **Bases de données :** IBM DB2 applique ce codage pour optimiser le stockage des tables contenant des données textuelles redondantes.
- **Bioinformatique :** Le codage des chaînes fréquentes est employé pour compresser des séquences ADN, où certains motifs génétiques se répètent fréquemment.

En conclusion, le codage des chaînes fréquentes est une méthode puissante et flexible pour la compression de données. Bien qu'elle nécessite une analyse initiale et un dictionnaire, son efficacité dans des contextes riches en répétitions en fait un outil incontournable dans de nombreux systèmes modernes.

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`. Ainsi, la séquence initiale AAABB, AAABCC, AAACCC peut être représentée de manière compressée comme suit : 4B, 0, 3CCC, où 0 indique que la chaîne est identique à la référence. Cette méthode permet de réduire la taille des données en éliminant les redondances dans les préfixes partagés.

Pour décompresser, on utilise la chaîne de référence AAABCC et les codes stockés. Pour 4B, on prend les 4 premiers caractères de la référence (AAAB) et on ajoute B, ce qui donne AAABB. Pour 0, on récupère directement la chaîne de référence AAABCC. Enfin, pour 3CCC, on prend les 3 premiers caractères de la référence (AAA) et on ajoute CCC, ce qui donne AAACCC.

Cette technique de compression est particulièrement efficace lorsque les séquences partagent de longs préfixes communs, comme dans les bases de données textuelles ou les structures d'indexation, où elle permet de minimiser l'espace de stockage tout en conservant l'intégrité des données.

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

6.7. Questions d'approfondissement

(a) Le site Twitter permettait 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.

(q) 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.

Votre avis compte !

Chers étudiants,
Si vous repérez une erreur dans ce document ou si vous avez une suggestion pour l'améliorer, nous vous invitons à remplir notre **formulaire anonyme**.
Votre contribution est précieuse pour nous !

Cliquez ici pour accéder au formulaire

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.