L'informatique des entrepôts de données

Daniel Lemire

SEMAINE 5
Indexation des données multidimensionnelles

5.1. Presentation de la semaine

Tous les systèmes de bases de données supportent les index en arbre B et les tables de hachage. Cependant, certaines techniques d'indexation (ou de matérialisation) sont utilisées plus fréquemment dans les entrepôts de données alors qu'elles sont plus rarement utilisées dans des applications plus génériques (ou transactionnelles). Cette semaine, nous verrons certaines de ces techniques.

5.2. OLAP

Le traitement analytique en ligne (OLAP, *Online Analytical Processing*) est une approche informatique permettant d'analyser rapidement des données multidimensionnelles pour supporter la prise de décision dans

les entreprises. Contrairement au traitement transactionnel (OLTP), qui gère des opérations courantes, l'OLAP organise les données dans des structures comme des cubes multidimensionnels, facilitant des requêtes complexes telles que l'agrégation, le forage (*drill-down*), ou le découpage (*slicing/dicing*). Ces opérations permettent d'explorer les données sous différents angles, par exemple pour analyser les ventes par région, période, ou produit. Utilisé dans les entrepôts de données, l'OLAP repose sur des bases de données optimisées pour la lecture et l'analyse, offrant des performances élevées pour des rapports et des tableaux de bord, essentiels à la *business intelligence*.

5.3. Le cube de données

Le cube de données [2] est une technique de matérialisation qui permet d'accélérer substantiellement certaines requêtes multidimensionnelles. Supposons que vous souhaitiez rapidement calculer des statistiques globales de ventes. Par exemple, la requête suivante vous donnera le total des ventes par vendeur et par produit :

```
SELECT vendeur, produit, sum(montant)
FROM ventes
GROUP BY vendeur, produit
```

On pourrait aussi vouloir calculer des statistiques qui tiennent compte du temps pour mesurer par exemple l'évolution de la productivité des vendeurs :

```
SELECT vendeur, mois, sum(montant)
FROM ventes
GROUP BY vendeur, mois
```

Dans un tel scénario, le cube (ou hypercube) de données est une bonne approche. Étant donné une opération (telle que la somme) sur une mesure (telle que le montant des ventes), et un ensemble d'attributs (vendeur, mois, produit), on matérialise tous les regroupements (GROUP BY) possibles. Si nous avons les attributs, vendeur, mois, produit, alors les regroupements sont : (vendeur, mois, produit), (vendeur, mois), (mois, produit), (vendeur, produit), (vendeur), (mois), (produit). On ajoute aussi un regroupement spécial (noté $\top$) qui correspond à la requête sans regroupement sur des attributs : SELECT sum(montant) FROM ventes. Étant donné N attributs, il y a toujours 2^N regroupements possibles.

Il y a une relation de dépendance entre ces regroupements. Ainsi, on peut calculer "SELECT sum(montant) FROM ventes" plus rapidement

à partir de n'importe quel autre regroupement plutôt qu'en partant de la table originale. Dans le même esprit, on peut calculer rapidement "SELECT vendeur, sum(montant) FROM ventes GROUP BY vendeur" à partir du résultat de la requête "SELECT vendeur, mois, sum(montant) FROM ventes GROUP BY vendeur, mois". La Figure 1 illustre cette relation. Cette relation est très importante quand on doit calculer le cube de données, car elle permet d'accélérer les calculs en travaillant sur des tables de plus en plus petites. En effet, la table "SELECT vendeur, mois, sum(montant) FROM ventes GROUP BY vendeur, mois" est plus petite que la table originale (ventes).

Cette dépendance n'est valable que pour les opérations *distributives* comme la somme, le maximum ou le minimum. Ces opérations ont la propriété que si on connaît la valeur de l'opération sur tous les sous-ensembles disjoints, alors on peut calculer facilement la valeur de l'opération sur l'union de tous les sous-ensembles.

Malheureusement, on ne peut calculer la moyenne à partir d'autres moyennes. En effet, considérons cet ensemble de valeurs :

- 3,
- 4,
- 6,
- 5,
- 4.5

dont la moyenne est 4.5. On découpe cet ensemble en deux sous-ensembles dont le premier a une moyenne de 3.5

- 3,
- 4

et le second a une moyenne de ≈ 5.2

- 6,
- 5,
- 4.5.

Or, la moyenne de 5.2 et de 3.5 n'est pas 4.5. Cette observation, à savoir que la moyenne des moyennes n'est pas la moyenne, est connue sous le nom de paradoxe de Simpson.

Le paradoxe de Simpson est un phénomène statistique où une tendance observée dans des sous-groupes de données peut disparaître ou s'inverser lorsqu'on analyse l'ensemble des données combinées. Découvert par Edward H. Simpson en 1951, il survient souvent dans l'analyse de données catégoriques, comme dans les études médicales ou sociales, lorsqu'une variable confondante n'est pas prise en compte. Par exemple, un traitement peut sembler bénéfique pour les hommes et les femmes

séparément, mais apparaître inefficace ou nuisible lorsqu'on agrège les données, en raison de différences dans la taille ou la composition des sous-groupes. Ce paradoxe souligne l'importance de considérer les variables confondantes et de structurer correctement les analyses statistiques pour éviter des conclusions erronées.

On peut cependant calculer la moyenne à partir de deux opérations qui sont distributives : la cardinalité et la somme. Dans un tel cas, on dit que la moyenne est une opération *algébrique*. Il suffit donc de décomposer l'opération en opérations distributives pour pouvoir calculer rapidement le cube de données. Certaines opérations ne sont ni distributives, ni algébriques. On dit qu'elles sont holistiques.

Le cube de données permet de calculer très rapidement une grande variété de requêtes. Par exemple, si on a matérialisé la vue "SELECT vendeur, mois, sum(montant) FROM ventes GROUP BY vendeur, mois", on peut rapidement calculer une requête plus précise comme "SELECT vendeur, sum(montant) FROM ventes WHERE mois='janvier' GROUP BY vendeur".

En pratique, le cube de données peut être trop volumineux pour être complètement matérialisé. On doit alors appliquer une technique de sélection des vues pour choisir les composantes du cube de données qui sont les plus utilisées. En général, les composantes moins volumineuses sont matérialisées d'abord (par exemple, "SELECT vendeur, sum(montant) FROM ventes GROUP BY vendeur").

Le cube est plus approprié aux données statiques. Dans le cas où les données sont constamment mises à jour, la mise à jour du cube peut être fastidieuse.

5.4. MOLAP

On peut représenter plusieurs tables sous forme de tableau multidimensionnel. Prenons par exemple la table suivante :

Jean	janvier	12.10
Marc	janvier	13.10
Jean	février	2.20
Marc	février	15.00
Pierre	février	22.00
Jean	mars	27.90
Marc	mars	15.00

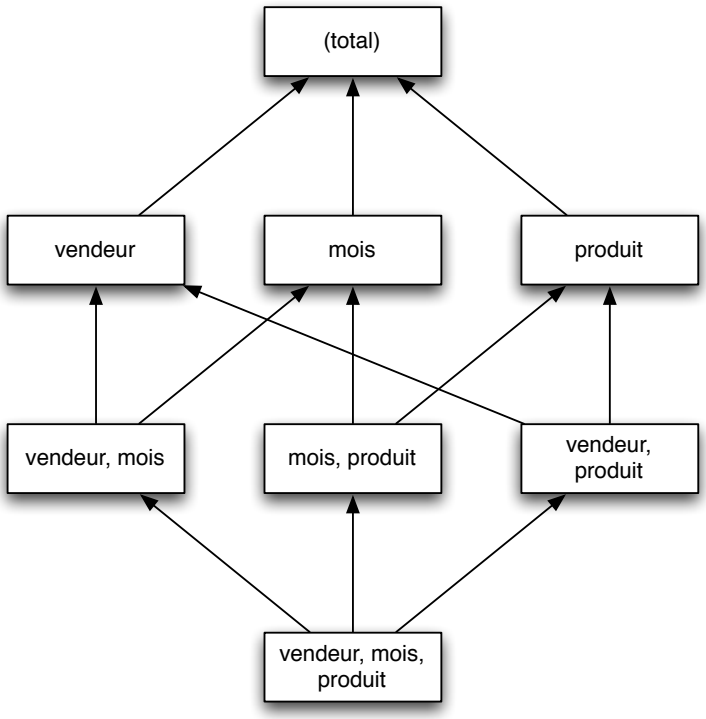


Figure 1. Lattice sur les attributs générés par la relation "peut être calculé à partir de"

Elle comporte sept enregistrements (rangées). Le tableau multidimensionnel correspondant prend cette forme :

	janvier	février	mars
Jean	12.10	2.20	27.90
Marc	13.10	15.00	15.00
Pierre	–	22.00	–

Il comporte $3 \times 3 = 9$ composantes. Dans ce cas précis, le tableau multidimensionnel est très dense : presque toutes les composantes sont associées à un enregistrement. On définit la densité par le nombre d'enregistrements divisé par le nombre composantes.

On peut alors utiliser une technique de matérialisation appelée MOLAP[1] [12]. Au lieu de stocker les enregistrements un à la suite de l'autre, en utilisant plusieurs octets par enregistrement, on peut créer un tableau multidimensionnel qui utilisera beaucoup moins d'espace. Par exemple, dans le cas de valeurs monétaires, on peut n'utiliser que 4 octets par enregistrement quand on opte pour la forme d'un tableau multidimensionnel. L'avantage principal de cette technique n'est pas tellement qu'elle permet la compression, mais qu'elle permet un accès rapide aux données. Ainsi, dans le tableau multidimensionnel de notre exemple, il est très facile de trouver tous les enregistrements de Jean car ils sont sur une même rangée.

L'approche MOLAP décrite ici ne s'applique que si le tableau multidimensionnel est suffisamment dense. Dans le cas contraire, on peut appliquées des techniques de compression. En pratique, il est possible de combiner les tables et l'approche MOLAP. On appelle souvent cette technique HOLAP (pour *Hybrid OLAP*) [4]. Il y a plusieurs façons de s'y prendre, mais dans le cas de notre exemple, nous pourrions exclure Pierre du tableau multidimensionnel et stocker ses données dans une table.

5.5. Index de projection

L'index de projection [10] est une technique fort simple, mais parfois très efficace. Au lieu de stocker uniquement les valeurs des colonnes au sein des enregistrements, on stocke aussi les valeurs dans une liste. Ainsi, étant donné le tableau suivant :

Jean	janvier	12.10	5
Marc	janvier	13.10	4
Jean	février	2.20	3
Marc	février	15.00	2
Pierre	février	22.00	4
Jean	mars	27.90	5
Marc	mars	15.00	3

On pourra construire quatre index de projection :

[1]Bien que le nom nous rappelle qu'il s'agit d'une technique développée pour les applications OLAP, la technique est d'intérêt général.

Jean
Marc
Jean
Marc
Pierre
Jean
Marc

janvier
janvier
février
février
février
mars
mars

et ainsi de suite.

En pratique, par normalisation, on peut souvent représenter chaque valeur dans la rangée par une clé ou valeur de 32 bits ou moins. Il est ainsi possible de naviguer rapidement au sein de chaque colonne. Cette méthode est très efficace quand on doit faire des opérations arithmétiques sur des colonnes (par ex. prix $\times$ quantite $-$ rabais).

5.6. Base de données par colonne

La plupart des systèmes de bases de données relationnelles sont orientées rangées. En effet, sur un disque, les données sont stockées rangée par rangée. Par exemple, cette table

Jean	janvier
Marc	janvier
Jean	février
Marc	février

est stockée sur un disque de cette manière : Jean, janvier, Marc, janvier, Jean, février, Marc, février. Un des inconvénients de cette stratégie de stockage est que peu importe le nombre de colonnes dont nous avons besoin, nous sommes tout de même souvent contraint de lire le contenu de toutes les colonnes, puisque la lecture se fait rangée par

rangée. Évidemment, on peut accélérer le traitement en utilisant des index (dont l'index de projection) sur les colonnes. L'autre inconvénient est qu'il peut être plus difficile de compresser les données de cette manière. En effet, même si les valeurs de la colonne mois sont toutes similaires, comme elles ne sont pas stockées de manière consécutive sur un disque, on ne peut pas les compresser efficacement.

Pour de meilleurs résultats, dans les entrepôts de données, on peut stocker les données par colonnes [15, 14]. Ainsi, dans l'exemple qui nous concerne, nous stockerions d'abord les valeurs de la première colonne, Jean, Marc, Jean, Marc, puis les valeurs de la seconde colonne janvier, janvier, février, février. On peut ensuite plus facilement compresser les valeurs des colonnes. Par ailleurs, on n'a plus qu'à charger que les colonnes qui nous intéressent au lieu de tout charger.

Une des techniques de compression utilisée pour compresser les colonnes et le codage par plage (*run-length encoding*). Le *run-length encoding* (RLE) est une technique de compression de données sans perte qui réduit la taille d'une séquence en remplaçant les suites consécutives d'un même symbole par une paire indiquant le symbole et le nombre de répétitions. Par exemple, la chaîne "AAAABBCC" est encodée en "4A2B2C", où "4A" signifie quatre "A", et ainsi de suite. Cette méthode est particulièrement efficace pour les données contenant de nombreuses répétitions, comme les images *bitmap* ou les textes avec motifs répétitifs. Bien que simple et rapide, le RLE est moins performant pour des données complexes sans répétitions fréquentes, et il peut même augmenter la taille des données dans de rares cas. Il est souvent utilisé dans des applications comme la compression d'images (par exemple, dans les formats PCX ou TIFF) ou comme étape préliminaire dans d'autres algorithmes de compression. Le principe du codage par plage est le codage des longues plages de valeurs identiques par un nombre indiquant le nombre de répétitions suivi de la valeur répétée. Par exemple, la séquence 11110000 devient 4140. Dans notre exemple, la deuxième colonne peut être représenté comme étant 2× janvier et 2×février. Il n'y a donc que deux séquences de valeurs identiques (ou plages).

Les données triées sont généralement plus compressible. Par exemple, si vous prend $A, B, C, A, A, A, C, D, A$. Cette séquence comporte 4 valeurs distinctes: A, B, C, D. Si on tri les valeurs, on obtient la séquence $A, A, A, A, A, B, C, C, D$. On peut écrire le résultat comme étant $5 \times A$, $1 \times B$, $2 \times C$ et $1 \times D$. Il n'y a donc plus que 4 plages ou 4 séquences de valeurs identiques.

Dans le cas des tables, il est préférable d'effectuer un tri lexicographique qui commence sur les colonnes de faible cardinalité [6].

Prenons l'exemple d'une table qui a 3 valeurs distinctes dans la première colonne et 2 valeurs distinctes dans la second colonne. Si on tri la table en prenant d'abord en considération la colonne ayant 3 valeurs distinctes, nous aurons alors 3 séquences de valeurs identiques dans cette colonne, et $2 \times 3 = 6$ séquences de valeurs identiques dans la seconde colonne:

A	0
A	1
B	0
B	1
C	0
C	1

Au total, nous avons donc $3 + 6 = 9$ séquences de valeurs identiques. Que se passe-t-il si on tri en partant de la colonne de moindre cardinalité? Dans le pire des cas, nous obtenons 2 séquences de valeurs identiques sur cette colonne et $2 \times 3 = 6$ séquences de valeurs identiques dans l'autre colonne:

A	0
B	0
C	0
A	1
B	1
C	1

Le résultat net est donc de $6 + 2 = 8$ séquences de valeurs identiques.

En général, moins il y a de séquences de valeurs identiques, plus la table sera compressible.

5.7. Index bitmap

Le premier exemple d'un index bitmap dans un moteur de base de données (MODEL 204) a été commercialisé pour l'IBM 370 en 1972 [11]. On trouve maintenant les index bitmaps dans de nombreux moteurs de base de données, dont les bases de données Oracle. Le principal avantage de ce mode d'indexation est de favoriser l'accès séquentiel aux disques tant à l'écriture qu'à la lecture des données [3]. Les index bitmap sont utilisés dans les bases de données Oracle, et au sein de

nombres plateformes dédiées à l'entreposage de données comme Apache Hive, Apache Spark, Druid.io, etc.

Le principe de base d'un index bitmap est de représenter les faits de telle manière que les requêtes se traduisent par de simples opérations logiques (AND, OR).[2] Pour chaque valeur v d'un attribut a d'une dimension, un index bitmap est composé d'un tableau de booléens (ou bitmap) ayant autant de bits que de faits et dont les positions des bits mis à 1 correspondent aux faits pouvant être joints avec cet attribut. La Table 1b montre un exemple d'un index bitmap. L'index porte sur 3 dimensions. La première dimension comporte 7 rangées et 6 valeurs distinctes: nous avons 6 bitmaps correspondants (ou 6 tableaux de 7 bits). La deuxième dimension porte sur 2 valeurs distinctes (et deux bitmaps) et 7 rangées. Alors que, finalement, la dernière dimension a 3 valeurs distinctes sur 7 rangées et nous avons 3 bitmaps ayant chacun 7 bits. Par exemple, le tableau de booléens 1000010 correspond au prédicat "Ville = Montréal" ; qui n'est vrai que pour le premier et le sixième faits selon la convention que id-Ville a la valeur 1 si et seulement si Ville = Montréal. De la même manière, le prédicat "Véhicule = Autobus" donne le tableau de booléens 0100011 (aussi appelé vecteur de bits ou tout simplement bitmap) avec la convention que id-Véhicule a la valeur 2 si et seulement si Véhicule = Autobus.

À titre d'illustration, les résultats de la requête "Véhicule = Autobus AND Ville = Montréal" sont calculés en réalisant le AND logique sur les deux tableaux de booléens précédents: 1000010 AND 0100011 ou 0000010. Ainsi donc, avec une seule opération logique, nous avons que seul l'avant-dernier enregistrement correspond à la requête "Véhicule = Autobus AND Ville = Montréal".

Les index bitmaps sont rapides parce qu'au lieu de lire toutes les valeurs possibles d'un attribut donné, on peut ne lire qu'aussi peu qu'un bit par fait et par attribut, ou même moins si l'on exploite des techniques de compression.

Alors qu'on considère parfois que les index bitmaps sont surtout appropriés aux attributs de petite cardinalité, comme le sexe ou le statut marital, Wu et al. ont montré qu'ils sont également efficaces lorsque les cardinalités des données sont très élevées [**13, 17**]. Par ailleurs, les index bitmaps permettent facilement d'indexer plusieurs dimensions, alors que les performances des index multidimensionnels en arbre, tels que

[2]Rappelons que la proposition x AND y est vraie si et seulement si les deux intrants (x, y) sont vrais. La proposition x OU y est fausse si et seulement si un des deux intrants (x, y) sont faux.

Table 1. Une table et son index bitmap

(a) Table

id-Ville	id-Véhicule	id-Couleur
1	1	1
2	2	1
3	1	2
4	1	1
5	1	3
1	2	1
6	2	1

(b) Index bitmap simple

100000	10	100
010000	01	100
001000	10	010
000100	10	100
000010	10	001
100000	01	100
000001	01	100

les arbres R, se dégradent rapidement lorsque le nombre de dimensions augmente [16].

Les index bitmaps sont souvent compressés par des techniques basées sur le codage par plage. En effet, comme il est fréquent que les bitmaps comprennent un grand nombre de 0 ou de 1 sur de longues plages, la compression par plage s'avère particulièrement appropriée. Cependant, il existe une autre raison pour laquelle cette compression est efficace : elle rend plus rapide l'exécution des opérations logiques. Considérons deux bitmaps B_1 et B_2 comportant chacun $|B_1|$ et $|B_2|$ valeurs vraies. Des algorithmes simples, comme l'Algorithme 1, permettent de calculer les opérations $B_1 \wedge B_2$ et $B_1 \vee B_2$ en temps $O(|B_1| + |B_2|)$. En utilisant des variantes de cet algorithme, nous obtenons le lemme suivant.

Lemma 5.1. *Étant donné deux bitmaps B_1 et B_2 compressés par plage et comportant $|B_1|$ et $|B_2|$ valeurs vraies, on peut calculer les opérations logiques $B_1 \wedge B_2$, $B_1 \vee B_2$, $B_1 \oplus B_2$, $B_1 \wedge not(B_2)$, $B_1 \oplus not(B_2)$ en un temps $O(|B_1| + |B_2|)$.*

Algorithm 1 Calcul du AND logique de deux bitmaps

INPUT: deux bitmaps B_1 et B_2
$i \leftarrow$ itérateur sur les valeurs vraies de B_1
$j \leftarrow$ itérateur sur les valeurs vraies de B_2
S ensemble représentant les valeurs vraies de $B_1 \wedge B_2$ (initialement vide)
while i ou j n'est pas arrivé à la fin du bitmap **do**
  $a \leftarrow \text{position}(i) < \text{position}(j)$
  $b \leftarrow \text{position}(i) > \text{position}(j)$
  while $\text{position}(i) \neq \text{position}(j)$ **do**
    if $(\text{position}(i) > \text{position}(j))$ **then**
      incrémenter l'itérateur i si possible, sinon sortir de la boucle
    else
      incrémenter l'itérateur j si possible, sinon sortir de la boucle
    end if
  end while
  if $\text{position}(i) = \text{position}(j)$ **then**
    ajouter $\text{position}(i)$ à S
  end if
end while

Observons que le nombre de séquences de valeurs vraies doit être égal (plus ou moins un) au nombre de séquences de valeurs fausses.

L'inconvénient de l'Algorithme 1 est que les opérations logiques se font bit-par-bit alors que le microprocesseur opère sur des mots. Au lieu de manipuler chaque bit un à un, il peut-être plus judicieux de manipuler des mots. Nous distinguons deux types de mots : les mots propres ne comportant que des bits vrais (1x11) ou des bits faux (0x00), et les autres, appelés mots impropres. Nous pouvons coder toute séquence de mots propres de même nature par un entier suivi d'un bit pour distinguer les deux types de mots propres (1x11 et 0x00), alors que les séquences de mots impropres peuvent être codées par un entier déterminant la longueur de la séquence, suivi par les mots impropres représentés *in extenso*.

Wu et al. [17] appliquent cette idée sur l'encodage WAH, mais plutôt sur des mots de 31 bits. Ils représentent les séquences de mots propres du même type par un mot dont le premier bit est faux, le deuxième bit indique le type du mot propre (0x00 ou 1x11) et les 30 bits suivants sont utilisés pour stocker la longueur de la séquence. Les mots impropres de 31 bits sont stockés *in extenso*, tout en mettant le premier bit à la valeur vraie pour indiquer qu'il s'agit d'un mot impropre.

Avec la compression alignée sur les mots, pour effectuer une opération logique comme AND, il suffit de lire à la fois les mots propres de type 1x11 et impropres.

Lemma 5.2. *Étant donnés deux bitmaps B_1 et B_2 compressés avec alignement sur les mots et comportant $|B_1|$ et $|B_2|$ mots non nuls, incluant 1x11 et tout mot impropre, on peut alors calculer les opérations logiques $B_1 \wedge B_2$, $B_1 \vee B_2$, $B_1 \oplus B_2$, $B_1 \wedge not(B_2)$, $B_1 \oplus not(B_2)$ en un temps $O(|B_1| + |B_2|)$.*

L'espace occupé par un bitmap B compressé avec alignement sur les mots est dans $O(|B|)$ et $O(|B|) = O(\text{vrai}(B))$ où $\text{vrai}(B)$ est le nombre de valeurs vraies dans un index bitmap. De plus, lorsque les index bitmaps sont suffisamment creux, c'est-à-dire qu'ils comportent beaucoup plus de valeurs fausses que de valeurs vraies, la plupart des mots propres n'ont que des bits faux (0x00) : le coût des opérations est dans $O(\text{impropre}(B_1) + \text{impropre}(B_2))$.

Le tri de la table améliore grandement la compression [7]. En effet, le tri a pour résultat de favoriser de longues séquences de valeurs identiques. Le résultat net est que non seulement les index sont plus petits, mais ils sont aussi plus rapides.

L'inconvénient du codage par plage est qu'il est nécessite souvent l'accès en séquence de quasiment toutes les données. Il existe des solutions de rechange, notamment les index Roaring: http://roaringbitmap.org.

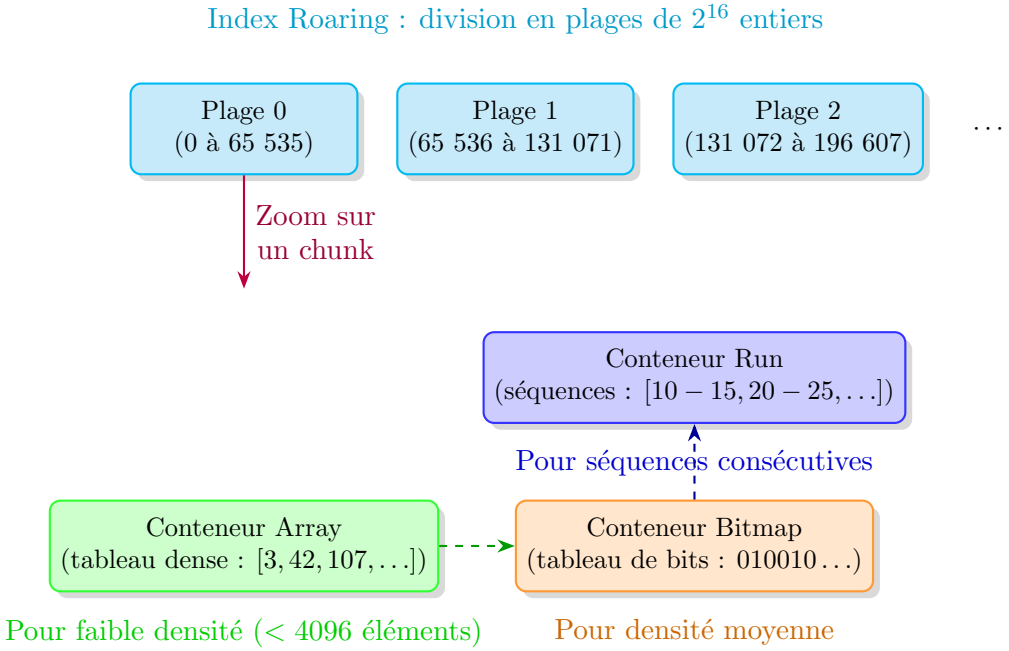
Les index Roaring, ou *Roaring Bitmaps*, constituent une solution de rechange avancée aux index bitmaps traditionnels, conçue pour optimiser l'efficacité en termes d'espace et de performance pour les opérations logiques [1, 8, 9]. Développés pour répondre aux limitations des bitmaps classiques, notamment leur consommation de mémoire dans des ensembles de données à forte cardinalité, les index Roaring combinent plusieurs techniques de compression et de partitionnement. Ils sont particulièrement adaptés aux bases de données modernes, aux moteurs de recherche et aux systèmes d'analyse de données comme Apache Lucene, ClickHouse ou encore Elasticsearch.

Le principe des index Roaring repose sur une structure hybride qui divise les entiers (représentant les positions des bits à 1 dans un bitmap) en conteneurs distincts. Chaque conteneur couvre un intervalle de 65 536 valeurs (correspondant à des entiers de 16 bits) et utilise l'une des trois représentations suivantes : un tableau trié pour les ensembles denses, une liste d'entiers pour les ensembles clairsemés, ou un bitmap pour les ensembles intermédiaires. Cette approche permet d'adapter la méthode de stockage à la densité des données dans chaque conteneur,

réduisant ainsi l'espace mémoire requis. Par exemple, pour un conteneur clairsemé avec peu de valeurs, une liste d'entiers est plus efficace qu'un bitmap complet, tandis qu'un conteneur dense bénéficie d'un bitmap compressé.

Les opérations logiques (AND, OR, XOR, NOT) sur les index Roaring sont exécutées en exploitant la structure des conteneurs. Lorsqu'une opération est effectuée, seuls les conteneurs correspondants (ceux ayant des clés communes) sont traités, ce qui réduit considérablement le coût computationnel. De plus, les conteneurs sont optimisés pour des accès mémoire alignés, ce qui améliore la performance sur les architectures modernes. Par exemple, pour calculer $B_1 \wedge B_2$, les conteneurs de B_1 et B_2 sont comparés clé par clé, et l'opération logique est appliquée uniquement aux données pertinentes. Cette approche garantit un temps d'exécution proportionnel au nombre de conteneurs non vides, souvent bien inférieur à celui des bitmaps traditionnels.

En pratique, les index Roaring sont largement utilisés dans les systèmes nécessitant des requêtes rapides sur de grands ensembles de données. Par exemple, dans les bases de données analytiques comme ClickHouse, ils permettent d'accélérer les requêtes de filtrage sur des colonnes avec des valeurs de cardinalité élevée, comme les identifiants d'utilisateurs ou les timestamps. Dans les moteurs de recherche comme Elasticsearch, les index Roaring optimisent les opérations de recherche booléenne, où des combinaisons complexes de filtres doivent être résolues rapidement. Leur faible empreinte mémoire les rend également idéaux pour les environnements contraints, tels que les applications embarquées ou les systèmes distribués où la scalabilité est cruciale.



5.8. Activité pratique

Pour approfondir votre compréhension des index bitmaps, nous vous proposons de réaliser l'activité suivante. Rendez vous à l'adresse suivante: https://github.com/lemire/roaring_demo. Suivez-y les consignes. Vous y trouverez un exemple d'implémentation des index Roaring en Java.

5.9. Indexation des données spatiales

En ce qui concerne les données spatiales et géographiques, on utilise fréquemment les arbres R et leurs variantes, les arbres R* et les arbres R+. Il s'agit de structures efficaces en autant que le nombre de dimensions ne soit pas trop élevé.

Les arbres R (*Rectangle trees*) sont des structures de données arborescentes utilisées pour l'indexation spatiale, permettant d'organiser et de rechercher efficacement des objets dans un espace multidimensionnel, comme des points, des rectangles ou des polygones. Introduits par Antonin Guttman en 1984, les arbres R regroupent les objets dans des rectangles englobants (*bounding boxes*) hiérarchiquement organisés, où chaque nœud de l'arbre représente un rectangle contenant des sous-rectangles ou des objets. Cette structure optimise les requêtes spatiales, comme la recherche de proximité ("trouver tous les points dans un

rayon donné") ou les intersections, en réduisant le nombre de nœuds à explorer grâce à la hiérarchie. Les arbres R sont largement utilisés dans les bases de données géographiques, les systèmes d'information géographique (SIG), et les applications de cartographie, bien que leur performance puisse dépendre de la distribution des données et des stratégies d'équilibrage.

5.10. Accélération

La plupart de ces techniques de compression peuvent être accélérées en utilisant les instructions avancées sur les processeurs modernes. Par exemple, les processeurs Intel et AMD offrent des instructions SIMD (Single Instruction Multiple Data) qui permettent de traiter plusieurs données en parallèle. Lemire et Boytsov [5] ont montré que les instructions SIMD peuvent être utilisées pour accélérer le décodage des entiers encodés pour atteindre des taux dépassant le milliard d'entiers décodés par seconde.

Même sans utiliser les instructions SIMD, il est possible d'accélérer les opérations de compression et de décompression en utilisant du code conventionnel optimisé. Par exemple, la librairie JavaFastPFOR offre d'excellentes performances.

À consulter : `https://github.com/fast-pack/JavaFastPFOR`

Nous vous invitons à la mettre à l'essai. Récupérer le projet JavaFastPFOR sur GitHub, puis exécuter les tests de performance avec Maven. Pour ce faire, vous devez d'abord télécharger Maven et l'installer, ce que vous avez sans doute déjà fait. Ensuite, vous pouvez exécuter les commandes suivantes dans le répertoire du projet JavaFastPFOR:

```
mvn compile
mvn exec:java
```

Vous devriez voir les résultats des tests de performance s'afficher dans la console. Les résultats devraient indiquer que la librairie JavaFastPFOR est capable de décoder plus d'un milliard d'entiers par seconde sur un ordinateur moderne. Vous pouvez aussi consulter le projet sur GitHub pour en savoir plus sur son fonctionnement et ses performances.

5.11. Questions d'approfondissement

(a) Est-ce que la médiane est une opération algébrique?

(b) Étant donné un tableau ayant trois colonnes, où la dernière colonne est une mesure (total des ventes) et où la cardinalité de la première colonne est 200 et la cardinalité de la seconde est 12. Quelle sera la taille (en octet) de la représentation MOLAP correspondante? Combien devrait-il y avoir d'enregistrements, en supposant un coût de stockage de 12 octets par enregistrement, pour que la représentation MOLAP soit efficace? Quelle est la correspondante?

(c) Considérons une table dont la première colonne comporte 12 valeurs distinctes, la seconde 450 et la troisième 4. Que pouvez-vous dire sur l'efficacité de l'encodage par plage si vous utilisez une base de données orientée colonne? Indice: calculez le nombre de séquences de valeurs identiques dans chaque colonne.

(d) Un ingénieur veut calculer une opération logique AND entre deux bitmaps (non compressés) comportant 1024 valeurs booléennes. Combien d'opérations le microprocessseur devrait-il effectuer?

(e) Étant donné une table comportant une colonne ayant 3 valeurs distinctes, combien y aura-t-il de bitmaps (ou vecteur de bits) dans l'index bitmap correspondant?

5.12. Réponses suggérées

(a) Il s'agit d'une opération holistique pour laquelle on n'utilisera sans doute pas la technique du cube de données.

(b) Il faut donc au moins $9\,600/12 = 800$ enregistrements pour que l'approche MOLAP puisse être considérée efficace sur le plan du stockage si on suppose douze octets par enregistrement. La densité critique est de $800/(200 \times 12) = 33\%$.

Si on fait abstraction du stockage des valeurs et que l'on suppose que la mesure peut être stockée en n'utilisant que 4 octets, alors le total sera de $200 \times 12 \times 4 = 9\,600$ octets.

Quand on utilise un index bitmap simple, chaque valeur distinct génère un bitmap. Il y aura donc 3 bitmaps. (e)

Sur un microprocesseur à 64 bits, il suffira de 16 opérations AND entre des pairs de mots de 64 bits. (p)

Dans le pire des cas, un tri lexicographique débutant sur la troisième colonne, se poursuivant sur la première colonne et se terminant sur la deuxième colonne va générer 4 séquences de valeurs identifiques (sur la troisième colonne),4×12 séquences dans la première et $4 \times 12 \times 450$ sur la deuxième colonne. (On place les colonnes dans cet ordre pour minimiser le nombre total de séquences à coder.) Donc, la colonne la moins compressible aura 21 600 séquences de valeurs identiques dans le pire des cas. Ainsi donc, si la table a beaucoup plus que 20 000 enregistrements, la table sera compressée efficacement par un encodage par plage. (c)

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. S. Chambi, D. Lemire, O. Kaser, and R. Godin. Better bitmap performance with roaring bitmaps. *Software: practice and experience*, 46(5):709–719, 2016.
2. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE '96*, pages 152–159, 1996.
3. M. Jurgens and H. J. Lenz. Tree based indexes versus bitmap indexes: A performance study. *International Journal of Cooperative Information Systems*, 10(3):355–376, 2001.
4. O. Kaser and D. Lemire. Attribute value reordering for efficient hybrid OLAP. *Information Sciences*, 176(16):2304–2336, 2006.
5. D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, 2015.
6. D. Lemire and O. Kaser. Reordering columns for smaller indexes. *Information Sciences*, 181:2550–2570, June 2011.
7. D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.
8. D. Lemire, O. Kaser, N. Kurz, L. Deri, C. O'Hara, F. Saint-Jacques, and G. Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.
9. D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, 46(11):1547–1569, 2016.
10. P. O'Neil and D. Quass. Improved query performance with variant indexes. In *SIGMOD '97*, pages 38–49, 1997.

11. P. E. O'Neil. Model 204 architecture and performance. In *2nd International Workshop on High Performance Transaction Systems*, pages 40–59, 1989.
12. R. T. Rimi and K. Azharul Hasan. Efficient key-value encoding for molap query processing. In *Proceedings of International Conference on Communication and Computational Technologies: ICCCT 2022*, pages 105–114. Springer, 2022.
13. V. Sharma. Bitmap index vs. b-tree index: Which and when? En ligne : http://www.oracle.com/technology/pub/articles/sharma_indexes.html, mars 2005. Dernier accès le 22 avril 2008.
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. M. J. Turner, R. Hammond, and P. Cotton. A DBMS for large statistical databases. In *VLDB'79*, pages 319–327, 1979.
16. R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB '98*, pages 194–205, 1998.
17. K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.