Roaring bitmap : nouveau modèle de compression bitmap

Samy Chambi*, Daniel Lemire**, Robert Godin*, Owen Kaser***

*Département d'informatique, UQAM, 201, av. Président-Kennedy
Montréal, QC, H2X 3Y7 Canada
chambi.samy@gmail.com
godin.robert@uqam.ca
**LICEF, Université du Québec, 5800 Saint-Denis, Montréal, QC, H2S 3L5 Canada
lemire@gmail.com
***Computer Science and Applied Statistics, UNB Saint John,
Saint John, NB, E2L 4L5 Canada
o.kaser@computer.org

Résumé. Les index bitmap sont très utilisés dans les entrepôts de données et moteurs de recherche. Leur capacité à exécuter efficacement des opérations binaires entre bitmaps améliore significativement les temps de réponse des requêtes. Cependant, sur des attributs de hautes cardinalités, ils consomment un espace mémoire important. Ainsi, plusieurs techniques de compression bitmap ont été introduites pour réduire l'espace mémoire occupé par ces index, et accélérer leurs temps de traitement. Ce papier introduit un nouveau modèle de compression bitmap, appelé *Roaring bitmap*. Une comparaison expérimentale, sur des données réelles et synthétiques, avec deux autres solutions de compression bitmap connues dans la littérature : WAH (*Word Aligned Hybrid compression scheme*) et Concise (*Compressed 'n' Composable integer Set*), a montré que *Roaring bitmap* n'utilise que $\approx 25\,\%$ d'espace mémoire comparé à WAH et $\approx 50\,\%$ par rapport à Concise, tout en accélérant significativement les temps de calcul des opérations logiques entre bitmaps (jusqu'à 1100 fois pour les intersections).

1 Introduction

Le volume croissant des ensembles de données scientifiques et commerciales pousse les chercheurs à adopter des techniques d'indexation efficaces, permettant d'extraire rapidement des informations intéressantes. Les index bitmap sont parmi les types d'index les plus communément utilisés (O'Neil, 1987; Su et al., 2013). Plusieurs travaux proposent des solutions faisant recours à ces index pour améliorer les performances des entrepôts de données (Bouchakri et Bellatreche, 2011; Bellatreche et al., 2007; Boukhalfa et al., 2010; Aouiche et al., 2005; Stockinger et Wu, 2008). En effet, leur capacité à exécuter efficacement des opérations logiques entre bitmaps accélère considérablement les temps de réponse des requêtes OLAP multidimmensionnelles.

Roaring bitmap

Un bitmap est un tableau de bits qui peut représenter, de façon compacte et efficace, une liste d'entiers. Avec un bitmap de n bits, le i^e bit est mis à 1, si le i^e entier dans l'intervalle $[0, n-1]$ est présent dans la liste. Sinon, le bit est mis à 0.

Sur des attributs de grandes cardinalités, les index bitmap nécessitent d'importants espaces mémoires pour indexer les valeurs distinctes des attributs. Ceci finit par réduire considérablement les performances en matière d'espace occupé et temps de traitement. Pour atténuer ce phénomène, plusieurs techniques de compression bitmap ont été introduites, telles que : VAL-WAH (Guzun et al., 2014), VLC (Corrales et al., 2011), EWAH (Lemire et al., 2010), COMPAX (Fusco et al., 2010), etc. Ces solutions se basent sur un codage hybride qui combine une compression par plages de valeurs, avec une représentation bitmap sous forme de plusieurs chaînes de bits alignées par mots CPU. Concise (Colantonio et Di Pietro, 2010) et WAH (Wu et al., 2008) sont parmi les techniques de compression bitmap les plus sollicitées dans la littérature. Avec un mot CPU de w bits, WAH divise un bitmap de n bits en $\left\lceil \frac{n}{w-1} \right\rceil$ blocs de $w-1$ bits. Un bloc de bits hétérogènes contenant une combinaison de 0 et de 1 est transformé en un mot littéral à w bits. Son bit de poids fort est mis à 0 et le reste des bits stockent les $w-1$ bits hétérogènes. Un mot propre code une séquence de blocs de bits homogènes. Son bit de poids fort est mis à 1, le bit suivant prend un 1 ou un 0 selon le sens des bits homogènes, et les $w-2$ bits restants sauvegardent la longueur de la séquence des blocs de bits homogènes. La figure 1 illustre un exemple de compression d'un bitmap avec WAH. La partie (a) donne la liste des entiers représentée par le bitmap. Ce dernier est divisé en blocs de 31 bits non compressés dans la partie (b). Le résultat de la compression des blocs de 31 bits avec WAH est montré à la partie (c). Les trois premiers blocs de 31 bits homogènes de la partie (b) représentent la plage d'entiers 0–92. Puisque aucun entier de la liste n'appartient à cette plage, tous les bits des trois blocs sont négatifs. Un mot propre encode les trois blocs homogènes. Son bit le plus significatif est à 1, le second est à 0, ce qui correspond au sens des bits homogènes, et les 30 bits de poids faible indiquent le nombre de blocs compressés, qui, dans ce cas, est égal à $(3)_{10} = (11)_2$. Le prochain bloc de la partie (b) représente la plage d'entiers 93–123. Comme la liste ne possède qu'un seul entier appartenant à cette plage qui est 95, le bloc est alors constitué de 31 bits hétérogènes qui seront stockés dans un mot littéral. Le bit de poids fort de ce dernier prend un 0, et le reste des bits stockent les 31 bits hétérogènes. Le processus de compression se poursuit de la même manière sur les blocs restants.

Concise adopte un mécanisme similaire pour compresser un bitmap. Toutefois, ses concepteurs introduisent un nouveau type de mot, appelé mot mixte. Lorsqu'une séquence de blocs de bits homogènes est interrompue par un seul bit (que l'on appellera *le bit pollué*) d'un bloc hétérogène, alors un mot mixte est utilisé pour coder la séquence des blocs homogènes et le bloc hétérogène dans un seul mot CPU. Ses deux bits de poids fort indiquent respectivement le type du mot (0 pour un mot mixte) et le sens des bits homogènes, les $\lceil \log_2 w \rceil$ bits suivants sont appelés *bits de position* et enregistrent la position du bit pollué dans le bloc hétérogène. Les bits restants sauvegardent la longueur de la séquence des blocs homogènes. La figure 1(d) montre le résultat de la compression de la liste d'entiers de la partie (a) avec Concise. Le deuxième mot est un mot mixte qui représente la plage d'entiers 93–247. Les bits de position indiquent la position du bit pollué dans le premier bloc de 31 bits de la séquence codée par le mot. Cette position correspond à celle de l'entier 95 dans la plage représentée par le premier bloc de la séquence. Les 25 bits de poids faible stockent la longueur de la séquence des blocs de bits homogènes. Le troisième mot est codé de façon similaire. Si la valeur des bits de position est

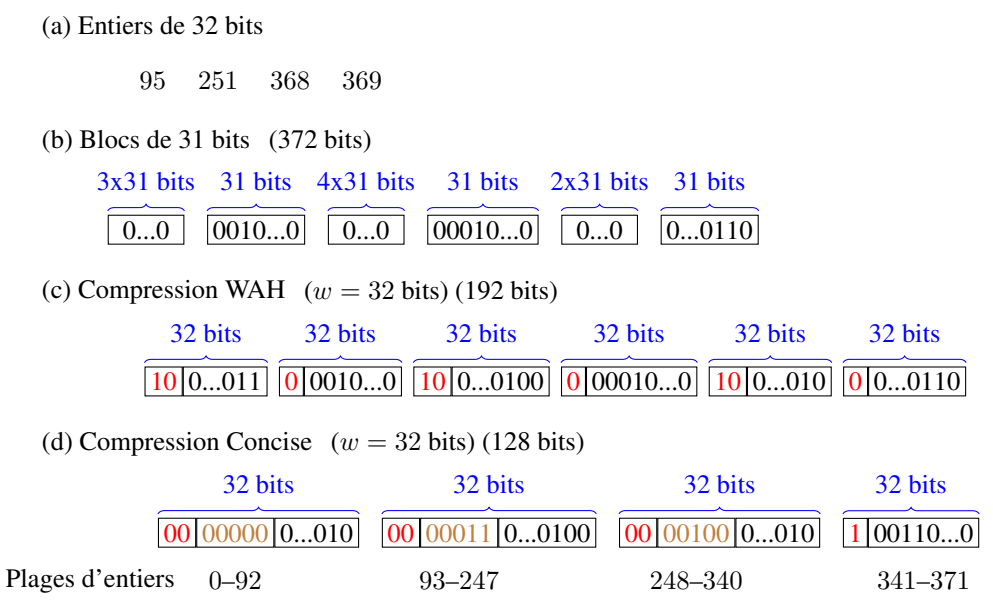


FIG. 1: Compression de la liste d'entiers $\{95, 251, 368, 369\}$ avec WAH et Concise sur un mot CPU de taille $w = 32$ bits.

égale à 0 (comme celle du premier mot), alors le mot mixte ne code qu'une séquence de blocs homogènes dont la longueur est égale à la valeur des 25 bits de poids faible plus un. Le dernier mot est un mot littéral qui stocke une suite de 31 bits hétérogènes. Si un entier de la liste figure à la position i dans la plage d'entiers représentée par le mot, alors le $(i)^e$ bit du mot littéral est égal à 1, sinon il est égal à 0.

Bien que ces techniques offrent de bons taux de compression, ils répondent moins efficacement aux opérations d'accès aléatoires. En effet, accéder au $i^e bit$ d'un bitmap compressé avec WAH ou Concise prendrait un temps $O(m)$ sur un bitmap compressé de m mots CPU.

On introduit un nouveau modèle de compression bitmap, nommé *Roaring bitmap*. Pour compresser la liste d'entiers représentée par un bitmap, on discrétise l'espace des entiers $[0, n)$ en des partitions de taille fixe. Cela permet de représenter différemment les plages de valeurs de fortes et faibles densités (Kaser et Lemire, 2006). Des expériences ont montré que *Roaring bitmap* utilise, en moyenne, 16 bits/entier pour compresser une liste d'entiers de 32 bits sur des faibles densités, tandis que Concise et WAH requièrent respectivement 32 bits/entier et 64 bits/entier en moyenne sur les mêmes densités. Aussi, *Roaring bitmap* a affiché des temps de calcul d'opérations logiques de 4 à 5 fois plus performants que Concise et WAH sur des distributions de données synthétiques, et jusqu'à 1100 fois meilleurs sur des ensembles de données réelles.

Le reste du papier est organisé comme suit : La section 2 introduit la structure de *Roaring bitmap*. La section 3, explique comment des accès aléatoires et des opérations logiques, ET ou OU, sont opérés sur des *Roaring bitmaps*. La section 4, présente les expériences qui ont permis d'évaluer les performances de *Roaring bitmap* sur des données réelles et synthétiques.

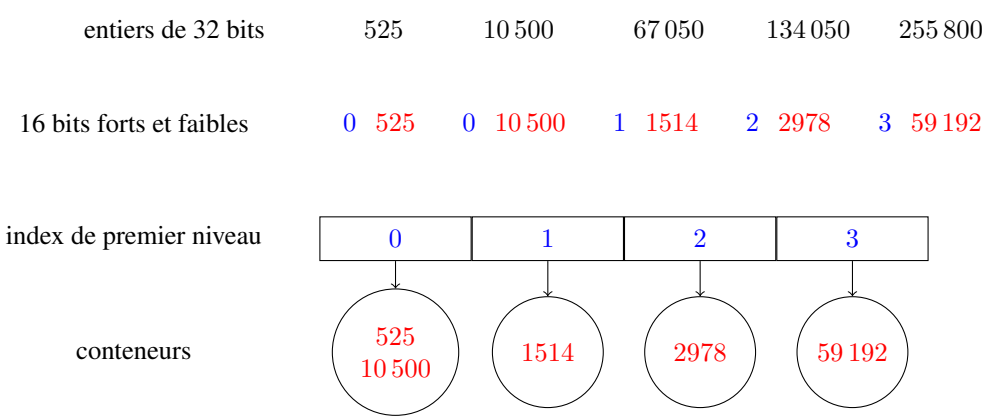


FIG. 2: Représentation de la liste d'entiers $\{525, 10\,500, 67\,050, 134\,050, 255\,800\}$ compressée avec *Roaring bitmap*.

On termine à la section 5, avec une conclusion et des travaux futurs.

2 Roaring bitmap

La structure d'index à deux niveaux de *Roaring bitmap* permet de compresser efficacement une liste d'entiers de 32 bits. Un tableau dynamique regroupe les entiers partageant les mêmes 16 bits de poids fort dans une même entrée, composée d'une clé et d'un conteneur. La clé préserve les 16 bits de poids fort du groupe, et le conteneur stocke les 16 bits de poids faible. Le tableau est trié par ordre croissant sur les valeurs des clés. Ces dernières sont utilisées tel un index de premier niveau pour accélérer les accès aléatoires et les opérations logiques.

La figure 2 illustre un exemple de compression d'une liste d'entiers avec *Roaring bitmap*. Lors de l'insertion d'un entier de 32 bits, une recherche binaire est lancée sur le tableau pour trouver une entrée dont la clé est équivalente aux 16 bits de poids fort de l'entier à insérer. Si une telle entrée est repérée, les 16 bits de poids faible de l'entier sont ajoutés au conteneur correspondant (voir l'insertion de $10\,500$ sur la figure 2). Dans un cas échéant, une nouvelle entrée, composée d'un champ pour la clé et d'un conteneur, est créée dans le tableau. La clé reçoit les 16 bits de poids fort de l'entier inséré, et le conteneur conserve les 16 bits restants. Ainsi, *Roaring bitmap* rassemble dans une même entrée du tableau, les entiers ayant les mêmes 16 bits de poids fort. Pour mieux illustrer le principe de compression sur la figure 2, les conteneurs ont été présentés comme des entités externes du tableau, mais techniquement, une entrée du tableau est composée d'une clé et d'un conteneur.

Un conteneur est une structure de données représentée par un tableau dynamique ou un bitmap, nommés respectivement : *conteneur-tableau* et *conteneur-bitmap*. Le choix de la structure adéquate dépend de la densité du groupe d'entiers.

Un *conteneur-bitmap* est un bitmap de 2^{16} bits, pouvant représenter 2^{16} entiers compris dans l'intervalle $[0, 65\,535]$. Initialement, tous les bits du bitmap sont à zéro. Pour indiquer la présence d'un éventuel entier a, le $(a \bmod 2^{16})^e$ bit correspondant à sa position dans le bitmap est mis à 1. Cette structure de données n'utilise que 1 bit pour représenter un entier de

16 bits. Cela permet aux *conteneur-bitmaps* d'être très efficaces sur des ensembles d'entiers denses. Cependant, lorsque la densité s'affaiblit, les performances se dégradent considérablement. Revenons à l'exemple de la liste des cinq entiers compressés sur la figure 2. Avec des *conteneur-bitmaps*, le *Roaring bitmap* résultant consommera $(65\,536 + 16) \times 4$ bits, ce qui est très volumineux comparé à une représentation via un simple tableau d'entiers, qui, dans ce cas, ne nécessiterait que de 32×5 bits pour stocker un tel ensemble d'entiers. Après investigations, on a constaté que ces cas survenaient lorsque le nombre d'entiers conservés dans un conteneur est inférieur à 2^{12} (4096). Effectivement, nous avons besoin de moins de 2^{16} bits (taille statique d'un *conteneur-bitmap*) pour stocker $i \times 16$ bits, lorsque $i \in [1, 4095]$. Afin de contourner ce problème, nous utilisons des tableaux dynamiques (*conteneur-tableaux*) triés par ordre croissant, pour stocker les entiers de 16 bits d'un conteneur peu dense, ne contenant pas plus de 4096 éléments.

Chaque conteneur maintient sa cardinalité à l'aide d'un compteur, qui est mis à jour à la volée lors de modifications. Ainsi, pour connaître le nombre d'éléments distincts d'une liste d'entiers de 32 bits compris dans $[0, n)$, il suffit de calculer la cardinalité d'un *Roaring bitmap* en sommant au plus $\lceil n/2^{16} \rceil$ compteurs. Ceci permet d'exécuter efficacement des requêtes d'intervalles et de sélections.

Si l'on compressait la liste d'entiers de la figure 1 avec *Roaring bitmap*, ce dernier céerait une seule entrée sur le premier niveau contenant une clé égale à 0, et un *conteneur-tableau* stockant les entiers de la liste. La structure de données résultante consommera approximativement 16 bits/entier, pour un espace total de : $(16 + 16 \times 4)$ bits = 80 bits. Ce qui est beaucoup plus économique comparé aux 128 bits de Concise et 192 bits de WAH.

Plusieurs approches basées sur une structure de données hybride ont précédemment été proposées. Afin d'améliorer les performances de LCM, un algorithme de recherche de motifs fréquents, (Uno et al., 2005) proposent une solution qui combine trois types de structures de données : un arbre préfixe, des bitmaps et des tableaux ; chacune ayant ses avantages et inconvénients par respect à la densité des données. *RIDBIT* (O'Neil et al., 2007) est composé d'un arbre-B qui index des bitmaps, et lorsque la densité d'un bitmap est en deçà d'un seuil fixe, il est transformé en un tableau (*RID-list*). Cependant, comparé à WAH, *RIDBIT* a montré de faibles performances.

3 Operations sur des *Roaring bitmaps*

3.1 ET et OU logiques

Répondre à une requête d'interrogation nécessite l'exécution d'une série d'opérations logiques entrainant plusieurs bitmaps candidats. Cette sous-section explique comment une opération logique d'union (OR) ou d'intersection (AND) entre deux *Roaring bitmaps* est réalisée.

Un nouveau *Roaring bitmap* est créé pour contenir le résultat, éventuellement vide, d'une opération logique entre deux *Roaring bitmaps*. Tout d'abord, les 16 bits de poids fort des deux bitmaps sont comparés en parcourant les index de premier niveau. On commence par les deux premières clés des index. Si les valeurs sont égales, une opération logique est exécutée entre les conteneurs indexés par les deux clés. Les 16 bits de poids fort communs et le nouveau conteneur retourné suite à l'opération logique, sont ajoutés au *Roaring bitmap* résultant. Les itérateurs des deux tableaux sont ensuite incrémentés d'un pas vers l'avant. Dans le cas échéant,

lorsque les valeurs des deux clés sont différentes, on avance d'une position sur le tableau de la plus petite des deux clés, en insérant, lors d'une union, la clé et une copie du conteneur qu'elle indexe, dans le nouveau *Roaring bitmap*. Pour les unions, ces itérations sont répétées jusqu'à ce que les deux index de premier niveau aient été entièrement parcourus. Dans le cas des intersections, l'opération termine dès vérification de l'un des deux index.

La comparaison de deux tableaux de premier niveau triés par valeurs de clés, lors d'une opération logique, est effectuée en un temps $O(n_1 + n_2)$, où n_1 et n_2 représentent, respectivement, le nombre d'entrées dans chaque tableau. Avec des tableaux non triés, le temps d'une même opération serait de l'ordre de $O(n_1 n_2)$. Aussi, un accès aléatoire ne consommerait qu'un temps de $O(\log_2 n)$, en appliquant une recherche binaire sur un tableau trié de n entrées, au lieu de $O(n)$ sur un tableau non ordonné.

Puisqu'un conteneur peut être représenté par deux types de structures de données : *conteneur-tableau* et *conteneur-bitmap*, une opération logique entre deux conteneurs suit l'un des trois scénarios suivants :

Bitmap vs bitmap : Au départ, 1024 OU ou ET logiques entre des blocs de 64 bits sont calculés. Le résultat est copié dans un nouveau *conteneur-bitmap*. La structure de données (bitmap ou tableau dynamique) obtenue après une intersection dépend de la cardinalité du résultat. Celle-ci est mis à jour à la volée lors des calculs logiques avec l'instruction Java *Long.bitCount*. Si l'on compte plus de 4096 entiers, on garde le *conteneur-bitmap* issu de la première série d'opérations logiques. Sinon, les bits positifs sont extraits du *conteneur-bitmap* à l'aide de l'instruction Java *Long.numberOfTrailingZeros*, et les entiers de 16 bits correspondants sont insérés dans un nouveau *conteneur-tableau*.

Bitmap vs tableau : Lorsqu'une intersection est exécutée entre un *conteneur-bitmap* et un *conteneur-tableau*, ce dernier est parcouru en premier, en vérifiant l'existence de chacun de ses éléments dans le bitmap. Tel que rapporté par (Culpepper et Moffat, 2010), cette méthode se révèle très efficace dans ce cas. Le résultat est retourné dans un nouveau *conteneur-tableau*.

Une union commence par copier le *conteneur-bitmap*, puis, y ajoute les bits positifs correspondant aux entiers du *conteneur-tableau*.

Tableau vs tableau : Lors d'une union, on essaie tout d'abord de prédire la taille de l'ensemble d'entiers résultant, en calculant la somme des cardinalités des deux conteneurs. Si celle-ci n'est pas supérieure à 4096, on fusionne les deux tableaux et le résultat est retourné dans un nouveau *conteneur-tableau*. Sinon, le résultat de la fusion des deux conteneurs est stocké dans un nouveau *conteneur-bitmap*. Ce dernier est converti en un *conteneur-tableau*, si sa cardinalité n'est pas supérieure à 4096.

Dans le cas des intersections, si la différence entre les cardinalités des deux conteneurs est inférieur à 64, une simple fusion, telle que celle utilisée par un tri-fusion, est opérée entre les deux tableaux. Sinon, on applique une intersection *galloping*
(voir Culpepper et Moffat (2010)). Le résultat est finalement ajouté dans un nouveau *conteneur-tableau*.

3.2 Accès aléatoires

Une opération d'accès aléatoire sur un *Roaring bitmap* commence par effectuer une recherche binaire sur les valeurs des clés de l'index de premier niveau. Si une entrée est trouvée,

une deuxième recherche est lancée au niveau conteneur, soit par un accès direct dans le cas d'un *conteneur-bitmap*, ou par une recherche binaire si c'est un *conteneur-tableau*.

Cette opération s'exécute en temps $O(\log_2 n)$, où n vaut au plus 2^{16} si l'on gère des entiers de 32 bits.

4 Expériences

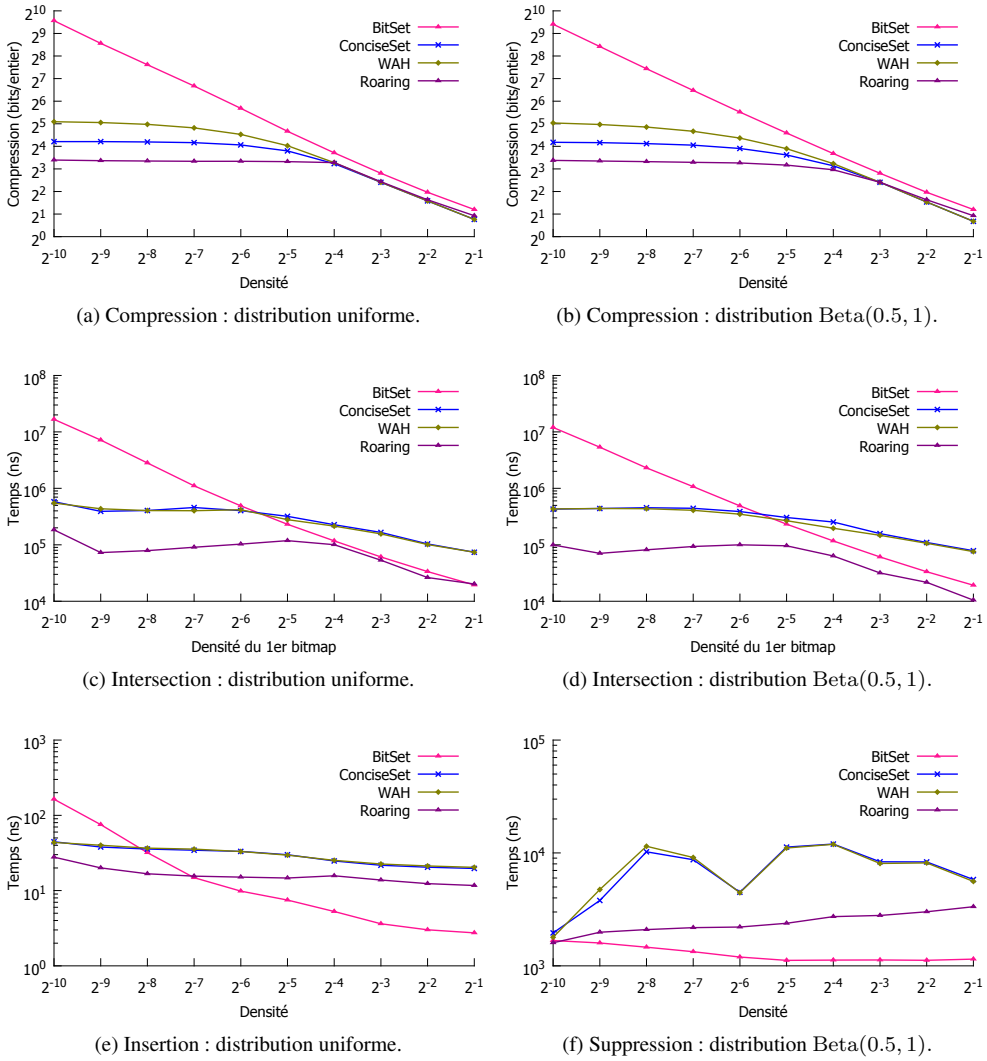
On a réalisé une série d'expériences pour comparer les performances de *Roaring bitmap* avec d'autres techniques de compression bitmap connues dans la littérature : WAH 32 bits et Concise 32 bits. Les essais ont été exécutés sur un processeur AMD FX™-8150 à Huit Cœures avec une fréquence d'horloge de 3.60 GHz et 16 GB de mémoire RAM. Pour Concise et WAH, on utilise la version 2.2 de la librairie Java CONCISE. On se sert aussi de la composante Java `BitSet` pour représenter des bitmaps non compressés. Afin de pleinement bénéficier de l'optimiseur de code de la JVM, on commence par exécuter des tests sans tenir compte des temps d'exécution. Puis, on répète chaque essai plusieurs fois et on présente la moyenne des résultats obtenus. Les temps de traitement sont donnés en nanosecondes. Nous utilisons le serveur JVM à 64 bits d'Oracle sur un système Linux Ubuntu 12.04.1 LTS. Le code source incluant les benchmarks et l'implémentation de la technique *Roaring bitmap* est librement accessible sur `http://roaringbitmap.org/`.

4.1 Données synthétiques

On a reproduit les expériences décrites dans (Colantonio et Di Pietro, 2010). Deux ensembles de 10^5 entiers sont générés lors de chaque essai avec deux types de distributions : Uniforme et $\mathrm{Beta}(0.5, 1)$ discrétisée. Les quatre techniques ont été comparées sur des données de différentes densités, variant de 2^{-10} à 0.5 (2^{-1}). Tout d'abord, un nombre réel y est généré pseudo-aléatoirement de l'intervalle [0,1). Ensuite, on ajoute l'entier obtenu de $\lfloor y \times \max \rfloor$ aux ensembles de données uniformes, où max représente le ratio entre le nombre total d'entiers à générer et la densité (d) de l'ensemble. Quant aux ensembles de données biaisées (distribution $\mathrm{Beta}(0.5, 1)$), on y ajoute $\lfloor y^2 \times \max \rfloor$, ce qui pousse les entiers à se concentrer sur des petites valeurs.

Les figures 3a et 3b montrent le nombre moyen de bits par entier, que chaque technique utilise pour stocker une liste d'entiers de 32 bits. Sur des bitmaps de faibles densités, *Roaring bitmap* ne consomme que $\approx 50\,\%$ d'espace mémoire par rapport à Concise et $\approx 25\,\%$ par rapport à WAH. Avec la croissance de la valeur de max sur les densités faibles, les entiers générés tendent à devenir de plus en plus grand, poussant `BitSet` à allouer d'importants espaces de stockage afin de représenter les larges entiers.

Les tests suivants rapportent les temps moyens consommés par chaque technique pour effectuer une intersection entre deux listes d'entiers (voir figures 3c et 3d). Les ensembles d'entiers sont représentés par deux bitmaps de densités asymétriques (l'un ayant une plus forte densité que l'autre), où la densité d_2 du deuxième bitmap est calculée à partir de la densité d du premier bitmap comme suit : $d_2 = (d-1) * x + d$; x étant un réel généré pseudo-aléatoirement de [0,1). Cette formule nous permet d'obtenir un deuxième bitmap aléatoirement plus dense. Le résultat d'une intersection est retourné dans un nouveau bitmap. Puisque `BitSet` ne supporte que des opérations *in-place*, on commence par copier le premier bitmap. Comme on peut le



(a) Compression : distribution uniforme.

(b) Compression : distribution $\mathrm{Beta}(0.5, 1)$.

(c) Intersection : distribution uniforme.

(d) Intersection : distribution $\mathrm{Beta}(0.5, 1)$.

(e) Insertion : distribution uniforme.

(f) Suppression : distribution $\mathrm{Beta}(0.5, 1)$.

FIG. 3: Compression et temps d'exécution.

constater, *Roaring bitmap* est 4 à 5 fois plus rapide que les deux techniques de compression bitmap sur toutes les densités testées. `BitSet` est 10 fois plus lent par rapport à *Roaring bitmap* sur des densités réduites. Bien que ses performances s'améliorent significativement sur des données denses, il reste toujours derrière *Roaring bitmap*.

Les mêmes tests ont été reconduits avec des unions. Les résultats n'ont cependant pas été rapportés, étant très similaires à ceux des intersections.

En se penchant sur ces observations, il serait fort probable qu'une application des *Roaring bitmaps* sur des index bitmap encodés avec un *equality-encoding* (Stockinger et Wu, 2008) ou

un *range-encoding* (Chan et Ioannidis, 1998) dans un entrepôt de données, puisse fournir de remarquables performances en matière d'espaces de stockage et temps d'exécution de requêtes OLAP complexes.

Nous avons aussi mesuré le temps moyen pris par chaque technique pour insérer un nouvel élément a dans un ensemble d'entiers S triés dans un ordre croissant, tel que : $\forall i \in S : a > i$. La figure 3e montre les résultats obtenus sur une distribution de données uniforme. Puisque WAH et Concise nécessitent de décoder séquentiellement les bitmaps compressés avant d'insérer chaque nouvel élément, ils mettent un temps linéaire par rapport à la taille des bitmaps compressés. Ce qui est beaucoup plus lent comparé à *Roaring bitmap*, qui effectue cette tâche en un temps logarithmique par rapport au nombre d'entrées de l'index de premier niveau et des *conteneur-tableaux* (dans les cas de densités faibles). L'allocation d'espaces ralentit `BitSet` sur les basses densités, mais il finit par accélérer sur des données denses, dépassant de beaucoup les autres techniques. Ceci s'explique par la diminution des taux d'allocations d'espaces, et du fait que des accès directs suffisent pour mettre à jour les bits. On n'a pas présenté les résultats obtenus sur une distribution $\mathrm{Beta}(0.5, 1)$, car des comportements similaires y ont été observés.

Dans le dernier test, on mesure le temps moyen consommé par chaque technique pour supprimer un élément sélectionné aléatoirement d'un ensemble d'entiers. Les résultats obtenus sur une distribution $\mathrm{Beta}(0.5, 1)$ sont présentés à la figure 3f. On voit clairement que *Roaring bitmap* est beaucoup plus performant comparé aux deux autres techniques de compression bitmap. Grâce à ses accès directs, `BitSet` affiche les meilleures performances sur ces essais. Des résultats similaires ont été observés sur des données de distribution uniforme.

4.2 Données réelles

Les techniques d'indexation précédentes ont été comparées à nouveau sur des ensembles de données réelles (Lemire et al., 2012). Cependant, étant trop large pour nos tests, on a utilisé de l'ensemble *Weather* que les données de septembre 1985 (Kevin et Raghu (1999) ont suivi la même approche). Aussi, l'ensemble de données de très faible densité *Census2000* a été écarté des tests, car le surplus de mémoire consommé par la structure de *Roaring bitmap* nécessitait quatre fois plus d'espace comparé à un bitmap compressé avec Concise. Toutefois, en matière de calculs logiques, *Roaring bitmap* a montré de bien meilleures performances, en exécutant des intersections 4 fois plus vite.

On garde l'ordre originel (non trié) des ensembles de données. Tout d'abord, on construit un index bitmap sur chaque ensemble. Par la suite, on sélectionne des bitmaps avec une approche similaire au *Stratified Sampling* : 150 échantillons d'attributs sont choisis par remplacement. On collecte ensuite 150 bitmaps en sélectionnant aléatoirement un bitmap de chaque attribut. Puis, on divise l'ensemble des 150 bitmaps obtenus en trois groupes de 50 bitmaps. Le tableau 1 présente les caractéristiques des bitmaps sélectionnés. Chaque test entraîne un trio de bitmaps, un de chaque groupe. Une première opération logique est exécutée entre deux bitmaps, et le résultat (renvoyé dans un nouveau bitmap) est calculé par la suite avec le bitmap restant. Dans le cas de `BitSet`, on commence par copier le premier bitmap, puis on effectue le reste des opérations avec des calculs *in-place*. Le tableau 2a montre le facteur de croissance de l'espace mémoire lorsqu'on remplace *Roaring bitmap* par `BitSet`, WAH et Concise. Les valeurs au-dessus de 1.0 indiquent de combien *Roaring bitmap* devance la technique corres-

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Lignes	4 277 807	199 523	1 178 559	1 015 367
Densité	1.2×10^{-3}	1.7×10^{-1}	1.3×10^{-3}	6.4×10^{-2}
Bits/entier	18.7	2.92	22.3	5.83

TAB. 1: Caractéristiques des bitmaps sélectionnés.

pondante. Les tableaux 2b–2c présentent les facteurs de croissance des temps de calcul des opérations logiques.

Roaring bitmap nécessite jusqu'à deux fois moins d'espace mémoire comparé à Concise et WAH, excepté pour l'ensemble *Wikileaks*, qui contient de larges plages de 1 qui sont incompressibles par *Roaring bitmap*. Pour ce qui est des temps de calcul des opérations logiques, *Roaring bitmap* a montré des accélérations significatives, allant jusqu'à 1100 fois plus vite lors des ET logiques sur les données de *Census1881*. Comparé à `BitSet`, celui-ci a montré de bons temps de traitement sur *CensusIncome* et *Weather*, mais aux dépens d'un espace de stockage beaucoup plus large.

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	2.23	1.4	0.82	1.35
WAH	2.45	1.63	0.83	1.46
BitSet	36.56	2.85	50.29	3.37

(a) facteurs de croissance d'espaces mémoires lorsque *Roaring bitmap* est remplacé par d'autres techniques.

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	1160.17	6.97	8.10	7.33
WAH	1016.28	6.22	8.04	6.42
BitSet	895.47	0.36	35.30	0.55

(b) Facteurs de croissance des temps de calcul de ET logiques si *Roaring bitmap* est remplacé par d'autres techniques.

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	54.41	4.73	2.09	5.04
WAH	47.72	4.25	2.02	4.46
BitSet	27.06	0.24	3.57	0.38

(c) Facteurs de croissance des temps de calcul de OU logiques si *Roaring bitmap* est remplacé par d'autres techniques.

TAB. 2: Résultats sur des données réelles.

5 Conclusion

On a introduit une nouvelle technique de compression bitmap, nommée *Roaring bitmap*. Des tests synthétiques et réels nous ont permis de comparer les performances de *Roaring bitmap* avec deux autres techniques de compression bitmap très connues dans la littérature : WAH et Concise. Les résultats ont montré que *Roaring bitmap* ne requiert que $\approx 25\,\%$ d'espace mémoire par rapport à WAH, et $\approx 50\,\%$ par rapport à Concise, tout en améliorant, de 4 à 5 fois, les temps de calcul des opérations logiques entre bitmaps sur des données synthétiques et jusqu'à 1100 fois sur des données réelles.

Comme travaux futurs, on vise à implémenter d'autres types d'algorithmes de recherche pour améliorer les temps des intersections. (Culpepper et Moffat, 2010) ont montré qu'une recherche *Golomb* (Golomb, 1966) permettait d'exécuter efficacement des intersections entre *posting lists*. On souhaite aussi améliorer la vitesse des accès aléatoires en adoptant une structure de données plus compacte sur le premier niveau de l'index, telle un bitmap offrant des accès directs. On envisage également d'expérimenter la nouvelle approche sur des bancs d'essais décisionnels, comme le *Star Schema Benchmark* (O'Neil et al., 2009).

Remerciements

Ces travaux ont pu être réalisés grâce à une subvention du Conseil de recherches en sciences naturelles et en génie du Canada (numéro 26143).

Références

Aouiche, K., J. Darmont, O. Boussaid, et F. Bentayeb (2005). Automatic selection of bitmap join indexes in data warehouses. In *7th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK 2005, Copenhagen, Denmark, pp. 64–73. IEEE Computer Society.

Bellatreche, L., R. Missaoui, H. Necir, et H. Drias (2007). Selection and pruning algorithms for bitmap index selection problem using data mining. In *9th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK 2007, Regensburg, Germany, pp. 221–230. IEEE Computer Society.

Bouchakri, R. et L. Bellatreche (2011). Sélection statique et incrémentale des index de jointure binaires multiples. *Revue des Nouvelles Technologies de l'Information 7e Journées francophones sur les Entrepôts de Données et l'Analyse en ligne, RNTI-B-7*, 171–187.

Boukhalfa, K., Z. Benameur, et L. Bellatreche (2010). Index de jointure binaires : Stratégies de sélection et étude de performances. *Revue des Nouvelles Technologies de l'Information EDA'10 Entrepôts de Données et Analyse en ligne, RNTI-B-6*, 355–366.

Chan, C. Y. et Y. E. Ioannidis (1998). Bitmap index design and evaluation. In *98 Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, New York, USA, pp. 355–366. ACM SIGMOD Record.

Colantonio, A. et R. Di Pietro (2010). Concise: Compressed 'n' composable integer set. *Information Processing Letters 110*(16), 644–650.

Corrales, F., D. Chiu, et J. Sawin (2011). Variable length compression for bitmap indices. In *Proceedings of the 22nd international conference on Database and expert systems applications*, DEXA'11, Berlin, Heidelberg, pp. 381–395. Springer-Verlag.

Culpepper, J. S. et A. Moffat (2010). Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems 29*(1), 1:1–1:25.

Fusco, F., M. P. Stoecklin, et M. Vlachos (2010). NET-FLi: On-the-fly compression, archiving and indexing of streaming network traffic. In *36th International Conference on Very Large Data Bases*, VLDB 2010, Singapore, pp. 1382–1393. Very Large Database Endowment.

Golomb, S. W. (1966). Run-length encodings. *IEEE Transactions on Information Theory 12*, 399–401.

Guzun, G., G. Canahuate, D. Chiu, et J. Sawin (2014). A tunable compression framework for bitmap indices. In *30th IEEE International Conference on Data Engineering*, ICDE 2014, Chicago, IL, USA. IEEE Computer Society.

Kaser, O. et D. Lemire (2006). Attribute value reordering for efficient hybrid OLAP. *Information Sciences 176*(16), 2304–2336.

Kevin, B. et R. Raghu (1999). Bottom-up computation of sparse and iceberg CUBEs. *Special Interest Group on Management Of Data Record 28*(2), 359–370.

Lemire, D., O. Kaser, et K. Aouiche (2010). Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering 69*(1), 3–28.

Lemire, D., O. Kaser, et E. Gutarra (2012). Reordering rows for better compression: Beyond the lexicographical order. *ACM Transactions on Database Systems 37*(20), 1–29.

O'Neil, E., P. O'Neil, et K. Wu (2007). Bitmap index design choices and their performance implications. In *11th International Database Engineering and Applications Symposium*, IDEAS 2007, Banff, Alta, pp. 72–84. IEEE Computer Society.

O'Neil, P. (1987). Model 204 architecture and performance. *Lecture Notes in Computer Science 359*, 40–59.

O'Neil, P., E. O'Neil, X. Chen, et S. Revilak (2009). The star schema benchmark and augmented fact table indexing. In *First TPC Technology Conference on Performance Evaluation and Benchmarking*, TPCTC 2009, Lyon, France, pp. 237–252. Springer.

Stockinger, K. et K. Wu (2008). Bitmap indices for data warehouses. In J. Wang (Ed.), *Data Warehousing and Mining: Concepts, Methodologies, Tools, and Applications*, pp. 1590–1605. Hershey, PA: IGI Global.

Su, Y., G. Agrawal, J. Woodring, K. Myers, J. Wendelberger, et J. P. Ahrens (2013). Taming massive distributed datasets: data sampling using bitmap indices. In *Proceedings of the 22nd international symposium on High-performance Parallel and Distributed Computing*, HPDC '13, New York, NY, USA, pp. 13–24. ACM.

Uno, T., M. Kiyomi, et H. Arimura (2005). LCM Ver.3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, OSDM '05, New York, NY, USA, pp. 77–86. ACM.

Wu, K., K. Stockinger, et A. Shoshani (2008). Breaking the curse of cardinality on bitmap indexes. In *20th International Conference on Scientific and Statistical Database Management*,

SSDBM 2008, pp. 348–365. Springer.

Summary

Bitmap indexes are increasingly used in data warehouses and search engines. Their leverage for bit-level parallelism and fast bitwise operations significantly speed up search queries. However, on high cardinality attributes, they consume a large amount of space, which deteriorates time-space's performances. Hence, many bitmap compression techniques have been introduced to compress bitmaps and provide efficient space representations with better search times comparing to other indexing schemes such as B-trees. In this work, we introduce a new bitmap compression scheme, called *Roaring bitmap*, and compare it to two well-known bitmap encoding techniques: WAH (Word Aligned Hybrid compression scheme) and Concise (Compressed 'n' Composable Integer Set). Synthetic and real data experiments shown that our index requires $\approx 25\,\%$ of the space needed by WAH and $\approx 50\,\%$ of Concise space. While performing bitwise operations many times faster (up to $1100\times$ with intersections).