Vers de meilleures performances avec des *Roaring bitmaps*

Samy Chambi [1], **Daniel Lemire** [2], **Robert Godin** [1]

1. *Département d'informatique, UQAM*
201, av. Président-Kennedy, Montréal, QC, Canada. H2X 3Y7
chambi.samy@gmail.com,godin.robert@uqam.ca

2. *LICEF, Université du Québec*
5800 Saint-Denis, Montréal, QC, Canada. H2S 3L5
lemire@gmail.com

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. 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 et Concise 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* 1 100 *fois pour les intersections).*

ABSTRACT. *Bitmap indexes are frequently used in data warehouses and search engines. Their effectiveness at exploiting bit-level parallelism and fast bitwise operations significantly speed up search queries. However, on high cardinality attributes, they consume a large amount of space. Hence, many bitmap compression techniques have been introduced that provide efficient space representations with better search times compared 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 and Concise. Synthetic and real data experiments have shown that our index requires* $\approx$ 25 % *of the space needed by WAH and* $\approx$ 50 % *of Concise's space, while performing bitwise operations many times faster.*

MOTS-CLÉS : *index bitmap, compression, performances.*

KEYWORDS: *index bitmap, compression, performances.*

DOI:10.3166/TSI.35.335-355

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 (P. O'Neil, 1987 ; Su *et al.*, 2013). Plusieurs travaux proposent des solutions recourant à ces index pour améliorer les performances des entrepôts de données (Bouchakri, Bellatreche, 2011 ; Bellatreche *et al.*, 2007 ; Boukhalfa *et al.*, 2010 ; Aouiche *et al.*, 2005 ; Stockinger, Wu, 2008). En effet, leur capacité à exécuter efficacement des opérations logiques entre bitmaps accélère considérablement les temps de réponse des requêtes OLAP (*On-Line Analytical Processing*) multidimensionnelles.

Dans une base de données, un index bitmap est créé sur chaque attribut candidat à l'indexation. Pour un attribut X à n entrées, un bitmap, qui est un tableau de n bits, est créé pour chaque valeur distincte de X. Une clé égale à l'une de ces valeurs distinctes est associée à chaque bitmap. Le i^e bit d'un bitmap est mis à 1, si la i^e entrée de l'attribut X équivaut à la clé du bitmap, sinon le bit est laissé à 0. De façon similaire, un bitmap peut servir à représenter efficacement un ensemble d'entiers. Pour représenter un entier $i \in [0, n[$, un 1 est placé au i^e bit du bitmap. Dans un cas échéant, un bit à 0 indique l'absence de l'entier correspondant à la position du bit dans le bitmap.

Les index bitmap sont connus pour être très performants sur des attributs de faibles cardinalités. Cependant, lorsque la cardinalité d'un attribut augmente, la taille de son index bitmap croît linéairement par rapport au nombre de ses valeurs distinctes jusqu'à occuper plus d'espace de stockage que les données indexées et jusqu'à offrir des temps de réponse plus lents que ceux d'autres types d'index, tels que l'arbre B. Afin de préserver les bonnes performances des index bitmap dans de telles situations, 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), PLWAH (Deliège, Pedersen, 2010), COMPAX (Fusco *et al.*, 2010), SPLWAH (Chang *et al.*, 2015), SECOMPAX et PLWAH+ (Chen *et al.*, 2015), etc. Ces solutions visent essentiellement à réduire la taille des index bitmap et à fournir des algorithmes de calculs logiques qui procèdent directement sur des bitmaps compressés, améliorant au final les temps de réponse des requêtes de recherche.

La plupart des techniques de compression bitmap introduites ces 15 dernières années se basent sur un même codage hybride qui combine une compression par plages de valeurs, avec une représentation bitmap sous forme de chaînes de bits alignées par mots CPU. Concise (Colantonio, Di Pietro, 2010) et WAH (K. Wu *et al.*, 2006) 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 se-

lon 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 et Concise. 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 à 0. 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. Puisque la liste possède au moins un 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.

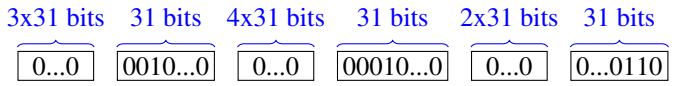
Concise adopte un mécanisme similaire pour compresser un bitmap. Toutefois, ses concepteurs introduisent un nouveau type de mot, appelé mot mixte. Lorsqu'une séquence de blocs de bits homogènes est interrompue par un seul bit (que l'on appellera *le bit pollué*) d'un bloc hétérogène, alors un mot mixte est utilisé pour coder la séquence des blocs homogènes et le bloc hétérogène dans un seul mot CPU. Le premier et le deuxième bit 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 encodés. La partie (d) de la figure 1 montre le résultat de la compression de la liste d'entiers de la partie (a) de la même figure avec Concise. Le premier mot encode la suite des 3 blocs homogènes débutant la séquence. 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. 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 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 de poids faible du mot littéral sera mis à 1, sinon il sera laissé à 0.

PLWAH (Deliège, Pedersen, 2010) est une technique de compression bitmap qui adopte un encodage assez similaire à celui de Concise. En effet, la représentation d'un bitmap compressé avec cet encodage est également formée des deux types de mots utilisés par Concise : mot mixte et mot littéral. Toutefois, PLWAH offre la possibilité d'encoder plusieurs bits pollués dans un mot mixte, pouvant ainsi améliorer les taux de compression par rapport à Concise dans certaines situations. SPLWAH (Chang *et*

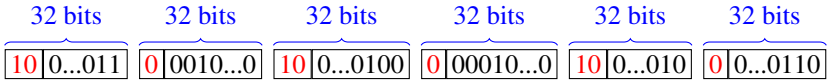
(a) Entiers de 32 bits

95 251 368 369

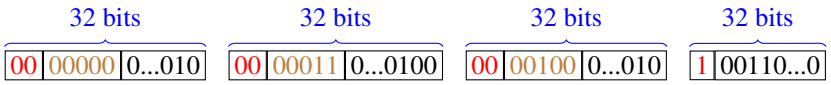
(b) Blocs de 31 bits (372 bits)



(c) Compression WAH ($w = 32$ bits) (192 bits)



(d) Compression Concise ($w = 32$ bits) (128 bits)



Plages 0–92 93–247 248–340 341–371

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

al., 2015) reprend l'idée des bits de positions adoptée au sein des deux encodages, PLWAH et Concise, et étend en plus le nombre de mots pouvant figurer dans la représentation d'un bitmap compressé à six. Effectivement, en plus des deux types de mots : littéral et propre, cette technique introduit deux autres types de mots : FS et SF, qui encodent dans un seul mot CPU une suite de blocs homogènes suivis ou précédés d'un bloc hétérogène ne contenant pas plus de quatre bits pollués. Ce modèle ajoute aussi deux autres formes de mots : FSF et SFS, qui permettent d'encoder dans un mot CPU deux successions de blocs homogènes séparées par un bloc hétérogène ayant au plus deux bits pollués, ou bien une séquence de blocs homogènes située entre deux blocs hétérogènes ne comptant pas plus de deux bits pollués. Des expériences (Chang *et al.*, 2015) ont montré que SPLWAH offre de meilleurs ratios de compression par rapport à PLWAH, WAH et d'autres techniques de compression bitmap sur des données triées.

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, nécessitera la lecture de tous les mots CPU précédant ce bit, prenant un temps $O(m)$ sur un bitmap compressé de m mots CPU.

Nous proposons une nouvelle technique de compression bitmap, nommée *Roaring bitmap* (Chambi *et al.*, 2014 ; 2015), qui adopte un modèle hybride combinant plusieurs structures de données et une compression préfixe pour représenter efficacement un bitmap. En considérant un bitmap comme un ensemble d'entiers $\in [0, n)$,

cette méthode 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 de faibles densités (Kaser, 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 1 100 fois meilleurs sur des ensembles de données réelles.

Le reste du papier est organisé comme suit : la section 2 introduit le modèle *Roaring bitmap*. La section 3, explique comment des accès aléatoires et des opérations logiques, ET et 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. L'article se termine à la section 5, avec une conclusion et des travaux futurs.

2. Roaring bitmap

Roaring bitmap est une structure d'index à deux niveaux qui permet de représenter efficacement une liste d'entiers de 32 bits. Cette structure adopte une compression préfixe au premier niveau et deux types de structures de données au second. Un tableau dynamique regroupe un ensemble d'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 d'entiers, et le conteneur stocke les 16 bits de poids faible. Le tableau est trié dans l'ordre croissant des valeurs de ses clés. Ce tableau est utilisé tel un index de premier niveau pour accélérer les accès aléatoires aux entiers représentés par un *Roaring bitmap* et les opérations logiques entre bitmaps.

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 dichotomique 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.

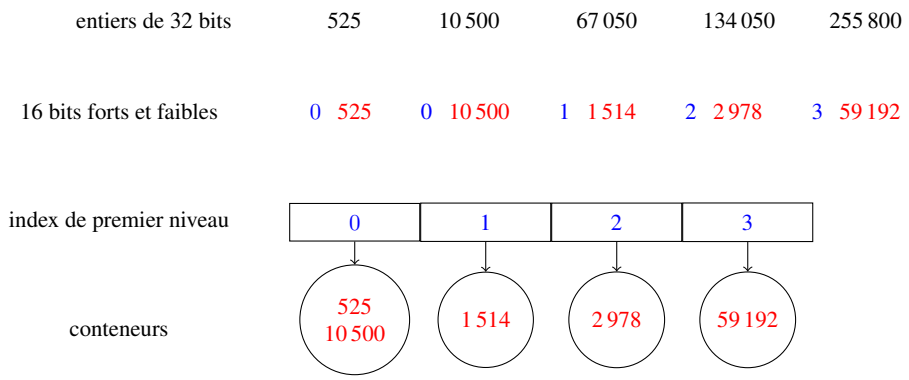


Figure 2. Représentation de la liste d'entiers {525, 10 500, 67 050, 134 050, 255 800} compressée avec Roaring bitmap

Un *conteneur-bitmap* est un bitmap de 2^{16} bits (65 536 bits), pouvant représenter 2^{16} entiers compris dans l'intervalle [0, 65 535]. Un bitmap est implémenté avec un tableau de $1\,024$ entrés chacune renfermant 64 bits du bitmap. 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, en moyenne, 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. En effet, revenons à l'exemple de la liste des cinq entiers compressés sur la figure 2. Avec des *conteneur-bitmaps*, le *Roaring bitmap* résultant consommera $(65\,536 + 16) \times 4 = 262\,208$ bits, ce qui est très volumineux comparé à une représentation via un simple tableau d'entiers, qui, dans ce cas, ne nécessiterait que de $32 \times 5 = 160$ bits pour stocker un tel ensemble d'entiers. Après investigations, il a été constaté qu'un conteneur requiert moins de 2^{16} bits (taille statique d'un *conteneur-bitmap*) pour stocker i entiers de 16 bits, lorsque $i \in [1,\ 4\,095]$. Afin de contourner ce problème, des tableaux dynamiques (*conteneur-tableaux*) triés par ordre croissant sont utilisés pour stocker les entiers de 16 bits d'un conteneur peu dense, ne contenant pas plus de 4 096 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 la liste d'entiers de la figure 1 était compressée avec *Roaring bitmap*, ce dernier créerait 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 : $(16 +$

16×4) 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. Le système *RIDBIT* (E. O'Neil *et al*., 2007) intègre une méthode de compression bitmap qui combine des bitmaps et des tableaux d'entiers. Lorsque la densité d'un bitmap est en deçà d'un seuil fixe, il est transformé en un tableau d'entiers (*RID-list*). Cependant, comparé au système FastBit (k. Wu *et al*., 2009) qui utilise la technique WAH pour compresser les bitmaps, *RIDBIT* a montré de faibles performances.

3. Opérations 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 entraînant plusieurs bitmaps candidats. Cette sous-section explique comment une opération logique d'union (OR) ou d'intersection (AND) entre deux *Roaring bitmaps* est réalisée.

Une opération logique entre deux *Roaring bitmaps* consiste à comparer les 16 bits de poids fort des entiers des deux bitmaps en parcourant leurs index de premier niveau. À la rencontre de deux entrées de valeurs équivalentes, une union ou une intersection est calculée entre les conteneurs indexés pas les deux entrées. Les 16 bits de poids fort communs et le nouveau conteneur obtenu d'un tel cas, sont ajoutés au *Roaring bitmap* résultant. Les itérateurs des deux tableaux de premier niveau sont ensuite incrémentés d'un pas vers l'avant. Dans le cas échéant, si les deux entrées de premier niveau comparées au cours d'une itération ont des valeurs différentes, l'algorithme avance d'une position sur le tableau de la plus petite des deux clés, en insérant, lors d'une union, la valeur de la clé et une copie du conteneur qu'elle indexe, dans le *Roaring bitmap* final. Pour les unions, ces itérations sont répétées jusqu'à ce que les deux index de premier niveau aient été entièrement parcourus. Tandis que pour les 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 dichotomique sur un tableau trié de n entrées, au lieu de $O(n)$ sur un tableau non ordonné.

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

Bitmap vs bitmap : Dans le cas d'une union logique, 1 024 opérations OU logique entre des blocs de 64 bits sont calculées (voir la section 2). Le résultat est stocké dans un nouveau *conteneur-bitmap*. Par contre, si une intersection logique est à réaliser, la cardinalité du résultat est tout d'abord calculée à l'aide de l'instruction Java *Long.bitCount*. Par la suite, 1 024 opérations ET logiques sont exécutées entre des blocs de 64 bits des deux bitmaps. Si la cardinalité du résultat dépasse les 4 096 (taille maximale d'un *conteneur-tableau*, voir la section 2), l'ensemble obtenu sera écrit dans un nouveau *conteneur-bitmap*, sinon, un nouveau *conteneur-tableau* représentera l'ensemble des entiers résultant.

Notons que l'instruction Java *Long.bitCount* utilise des instructions CPU très rapides, telles que l'instruction *popcnt* des processeurs Intel récents, pouvant compter le nombre de bits à 1 dans un mot CPU en une moyenne d'un seul cycle CPU. Aussi, la plupart des processeurs modernes bénéficient de calculs super-scalaires, pouvant traiter plusieurs mots CPU en parallèle (des processeurs Intel modernes peuvent traiter jusqu'à 4 mots CPU en un seul cycle CPU). Promettant ainsi un traitement efficace de ce type de calcul.

Bitmap vs tableau : Une intersection logique entre deux conteneurs différents consiste à parcourir le *conteneur-tableau* 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 de tels cas. Le résultat est retourné dans un nouveau *conteneur-tableau*.

Une union logique 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 logique, la taille de l'ensemble d'entiers résultant est tout d'abord prédite, en calculant la somme des cardinalités des deux conteneurs. Si celle-ci n'est pas supérieure à 4 096, les deux tableaux sont fusionnés et le résultat est retourné dans un nouveau *conteneur-tableau*. Sinon, les deux *conteneur-tableaux* sont parcourus pour insérer leurs éléments dans un nouveau *conteneur-bitmap*. Si la cardinalité du *conteneur-bitmap* n'est pas supérieur à 4 096, il sera transformé en un nouveau *conteneur-tableau*.

Dans le cas des intersections, si le facteur de 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, une intersection *galloping* (voir (Culpepper, Moffat, 2010)) est appliquée. 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 dichotomique 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 dichotomique si c'est un *conteneur-tableau*. Cette opération s'exécute en un temps de $O(\log_2 n)$, où n vaut au plus 2^{16}.

3.3. Union horizontale

Afin d'améliorer les temps d'exécution d'une opération d'union entraînant plusieurs *Roaring bitmaps*, une nouvelle stratégie a été mise en oeuvre, nommée *union horizontale*. Au départ, le premier conteneur de chacun des *Roaring bitmaps* à fusionner est inséré dans une file (queue) de priorités. Cette dernière garde les conteneurs triés dans un ordre croissant sur les valeurs de leurs clés. Les conteneurs dont la clé est unique dans la file sont retirés de cette dernière avant d'être ajoutés au *Roaring bitmap* résultant. Tandis que ceux ayant une même clé formeront une séquence qui sera triée de telle manière à ce que le conteneur ayant la plus grande cardinalité soit au début. Si ce dernier est un conteneur bitmap, nous commençons par effectuer une union traditionnelle (telle que discutée plus haut) entre les deux premiers conteneurs de la séquence, après les avoir retirés de la file, et un nouveau *conteneur-bitmap* renfermant le résultat de cette opération est retourné. Un scénario similaire est réitéré entre le *conteneur-bitmap* obtenu à chaque itération et le prochain élément de la séquence, mais avec la différence que les fusions se feront sur place (*in-place*). Plus précisément, le résultat d'une union entre deux conteneurs sera stocké dans le premier *conteneur-bitmap*, en évitant d'en générer un nouveau à chaque opération. Pour économiser du temps, la cardinalité d'un conteneur n'est pas calculée lorsque celui-ci est un *conteneur-bitmap* : la cardinalité ne sera calculée qu'une seule fois à la fin des opérations sur la séquence des conteneurs de même clé. Dans un cas échéant, si un *conteneur-tableau* débute une telle séquence, des unions traditionnelles seront exécutées jusqu'à ce que la cardinalité soit suffisante pour justifier un *conteneur-bitmap*, auquel cas, le reste des fusions sera complété par un calcul sur place (*in-place*). Le conteneur obtenu de la séquence d'opérations sera ensuite inséré, avec sa clé, dans le *Roaring bitmap* résultant. Lorsqu'un conteneur est retiré de la file, son suivant (s'il y en a) dans le *Roaring bitmap* correspondant y sera inséré.

Ces mêmes traitements se poursuivent jusqu'à ce qu'il n'y ait plus de conteneurs à traiter dans la file de priorités.

3.4. Memory-mapping

Le *memory-mapping* est l'une des techniques les plus adoptées à ce jour par les systèmes de gestion de données massives, elle aide à substantiellement réduire les coûts liés à l'allocation d'espace en mémoire principale et aux entrées/sorties (E/S) disque. Un des principaux avantages de cette solution est qu'elle permet à un programme en cours d'exécution de céder les tâches de lecture/écriture depuis/dans un fichier stocké sur disque à l'unité de gestion de la mémoire virtuelle du système d'exploitation (SE). En mappant un fichier externe en mémoire principale, le SE réserve

un espace d'adressage dans la mémoire virtuelle du programme en cours d'exécution afin de créer une corrélation octet par octet entre cette zone mémoire et une portion du fichier physique stocké sur le disque (sans entièrement charger le fichier en mémoire centrale). Si à un moment donné, l'application a besoin d'accéder à un certain segment de données du fichier mappé, le système d'exploitation se chargera de faire parvenir en mémoire principale les pages systèmes correspondant à la portion demandée du fichier à l'aide d'une pagination à la demande. Ce procédé permet à une application d'effectuer efficacement des accès aléatoires dans un fichier sans exiger une migration préalable de celui-ci en mémoire principale. Dans les cas d'immenses fichiers, cette méthode aide à sauver un nombre important d'accès disque comparé aux opérations d'E/S standard avec canal (*stream*).

Ainsi, le *memory-mapping* permet à un programme d'exploiter le contenu d'un fichier externe comme s'il était entièrement chargé en mémoire principale. Pour apporter des changements à un tel fichier, le processus se contente d'effectuer les modifications sur l'espace mémoire local du programme. Quant à la tâche de persistance sur disque des pages de données modifiées, elle sera prise en charge par l'unité de gestion de la mémoire virtuelle. Le mécanisme de lecture/écriture des pages de fichiers constitue l'un des éléments les plus critiques de cette unité dans un SE, et elle est considérée comme une fonction système hautement optimisée, ce qui rend cette stratégie de gestion de fichiers externes beaucoup plus efficace que les opérations standards d'E/S disque.

Les systèmes adoptant un *memory-mapping* sérialisent leurs données sur des fichiers disque. Ces fichiers sont mappés en mémoire principale lors de l'activation du système et la lecture de leurs données se fait à l'aide d'opérations de désérialisation. Pour rendre *Roaring bitmap* opérationnel dans un contexte de *memory-mapping*, sa librairie Java a été étendue en y incluant de nouvelles classes qui proposent des méthodes permettant d'effectuer plusieurs types de traitements avec des bitmaps mappés en mémoire principale, comme : la sérialisation/désérialisation de bitmaps, opérations logiques entre bitmaps, accès aléatoires dans des bitmaps, etc.

4. Expériences

Une série d'expériences a été réalisée 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™-8 150 à 8 cœurs avec une fréquence d'horloge de 3,60 GHz et 16 GB de mémoire RAM. Pour Concise et WAH, nous utilisons la version 2.2 de la librairie Java ConciseSet. [1] La composante Java `BitSet` a été prise pour représenter des bitmaps non compressés. Afin de pleinement bénéficier de l'optimiseur de code de la JVM, nous commençons par exécuter des tests sans tenir compte des temps d'exécution. Puis, nous répétons chaque essai plusieurs fois avant de présenter la moyenne des ré-

1. http://ricerca.mat.uniroma3.it/users/colanton/publications.html

sultats 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 bancs d'essais et la mise en œuvre en Java de la technique *Roaring bitmap* est librement accessible sur http://roaringbitmap.org/. Le langage de programmation Java a été choisi principalement du fait de la disponibilité de plusieurs autres librairies de représentation de bitmaps implémentées dans ce langage, comme celles évaluées au cours de ce travail : Concise, WAH et `BitSet`, ou d'autres, telle que la librairie JavaEWAH. [2] En plus, sachant que de nos jours plusieurs systèmes de traitement de données massives sont implémentés en Java, comme : Hadoop (Shvachko, Hairong *et al.*, 2010), Hive (Thusoo *et al.*, 2010), Druid (Yang *et al.*, 2014), etc., l'utilisation de ce langage de programmation offre pas mal d'opportunités pour intégrer la librairie *Roaring bitmap* à l'un de ces système, ce qui permettrait d'évaluer ses performances dans un tel contexte.

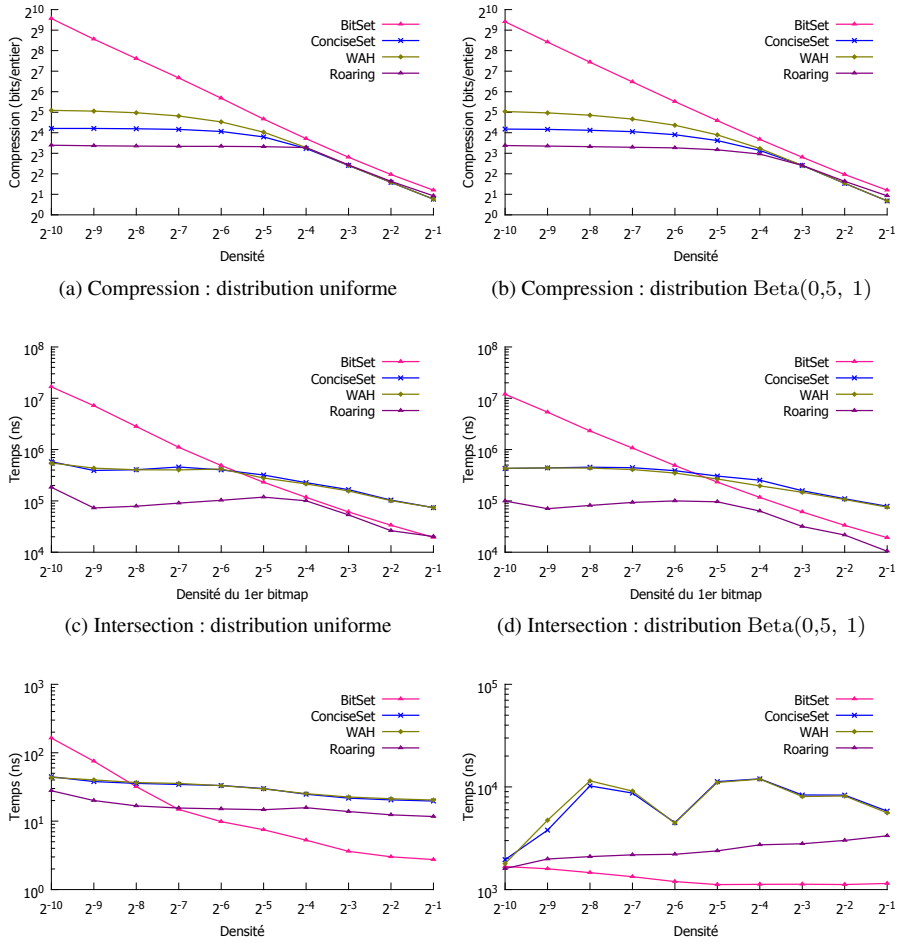
4.1. Données synthétiques

Les expériences décrites dans (Colantonio, Di Pietro, 2010) ont été reproduites pour la réalisation de ces essais. 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 (ou 2^{-1}). Tout d'abord, un nombre réel y est généré pseudo-aléatoirement de l'intervalle [0, 1). Ensuite, l'entier de $\lfloor y \times \max \rfloor$ est ajouté 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 $\mathrm{Beta}(0{,}5,\ 1)$), l'entier $\lfloor y^2 \times \max \rfloor$ y est inséré, pour pousser les entiers générés à se concentrer sur des petites valeurs.

Les figures 3(a) et 3(b) 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 grands entiers.

Les tests suivants rapportent les temps moyens consommés par chaque technique pour effectuer une intersection entre deux listes d'entiers (voir figures 3(c) et 3(d)). 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 : $d_2 = (d-1) \times x + d$; 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

2. https://github.com/lemire/javaewah



(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 $\mathrm{Beta}(0{,}5,\ 1)$

Figure 3. Compression et temps d'exécution

sur place (*in-place*), nous commençons par copier le premier bitmap. Comme il peut être 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, Wu, 2008) ou un *range-encoding* (Chan, 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.

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 3(e) 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. Nous n'avons 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, nous mesurons 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 3(f). Les résultats montrent 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 4 ensembles de données réelles (voir le tableau 1) précédemment utilisés dans (Lemire *et al.*, 2012) : *Census1881* (historique, 2009), *CensusIncome* (Frank, Asuncion, 2010), *Wikileaks* et *Weather* (Hahn *et al.*, 2004). *Census1881* représente des données provenant du recensement Canadien de l'année 1881. Cet ensemble fait un peu plus de 305 MB et renferme 4 277 807 enregistrements. *CensusIncome* a une taille de 100 MB et contient 199 523 enregistrements, c'est le moins volumineux des 4 ensembles de données. L'ensemble *Wikileaks* a été généré à partir de données publiques publiées par Google [3] et qui portent sur des textes diplomatiques confidentiels ayant été divulgués. Cet ensemble compte 1 178 559 enregistrements. L'ensemble *Weather* contient des données météorologiques prises entre 1882 et 1991. À l'origine, cet ensemble possède une taille de 9 GB et un nombre de 124 millions d'enregistrements, mais étant

3. http://www.google.com/fusiontables/DataSource?dsrcid=224453

trop large pour nos tests, seulement les données de septembre 1985 ont été utilisées, qui comptent pour un total de 1 015 367 enregistrements (Kevin et Raghu (1999) ont suivi la même approche). L'ensemble de données de très faible densité *Census2000* utilisé dans (Lemire *et al.*, 2012) 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.

Nous gardons l'ordre original (non trié) des ensembles de données. Tout d'abord, un index bitmap est construit sur chaque ensemble. Par la suite, des bitmaps sont sélectionnés à l'aide d'une approche similaire au *Stratified Sampling* : 150 échantillons d'attributs sont choisis par remplacement. Ensuite, 150 bitmaps sont collectés en sélectionnant aléatoirement un bitmap de chaque attribut. Puis, l'ensemble des 150 bitmaps obtenus est divisé 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`, nous commençons par copier le premier bitmap, puis le reste des opérations sont réalisées avec des calculs sur place (*in-place*). Le tableau 2a montre le facteur de croissance de l'espace mémoire lorsque *Roaring bitmap* est remplacé par `BitSet`, WAH et Concise. Les valeurs au-dessus de 1,0 indiquent de combien *Roaring bitmap* devance la technique correspondante. Les tableaux 2b–2c présentent les facteurs de croissance des temps de calcul des opérations logiques.

Tableau 1. Caractéristiques des bitmaps sélectionnés

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Lignes	4 277 807	199 523	1 178 559	1 015 367
Densité	$1{,}2 \cdot 10^{-3}$	$1{,}7 \cdot 10^{-1}$	$1{,}3 \cdot 10^{-3}$	$6{,}4 \cdot 10^{-2}$
Bits/entier	18,7	2,92	22,3	5,83

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 1 100 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.

4.3. Essais avec du Memory–mapping

Les tests présentés précédemment ont été conduits sur des bitmaps entièrement chargés en mémoire centrale. Afin de montrer l'efficacité de notre solution dans un

Tableau 2. Résultats sur des données réelles

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

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	2,23	1,40	0,82	1,35
WAH	2,45	1,63	0,83	1,46
BitSet	36,56	2,85	50,29	3,37

(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	1 160,17	6,97	8,10	7,33
WAH	1 016,28	6,22	8,04	6,42
BitSet	895,47	0,36	35,30	0,55

(c) Facteurs de croissance des temps de calcul de OU 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

contexte de *memory-mapping*, des essais qui comparent les performances de Roaring bitmap avec celles de Concise dans un tel contexte ont été produits. La version de la librairie Java ConciseSet utilisée lors de ces tests est celle proposée par les concepteurs du système de gestion de bases de données (SGBD) Druid (Yang *et al.*, 2014). Cette version représente une extension de la librairie ConciseSet 2.2 développée par Colantonio et Di Pietro (2010) et inclut de nouvelles fonctionnalités qui permettent de gérer des bitmaps compressés avec Concise et stockés sur des fichiers disque mappés en mémoire principale.

Pour ces essais, trois des ensembles de données réelles du précédent test ont été utilisés. Tout d'abord, 200 listes d'entiers de différentes tailles sont construites avec chaque ensemble de données. Ensuite, un Roaring bitmap et un ConciseSet sont créés avec les entiers de chaque liste, donnant un total de 200 bitmaps compressés pour chacune des deux méthodes de compression bitmap. Les 200 bitmaps compressés avec une technique de compression bitmap sont ensuite sérialisés dans un fichier disque, qui sera par la suite mappé en mémoire principale. Lors d'un test, nous mesurons pour chaque méthode de compression bitmap : l'espace disque consommé par la sérialisation de l'ensemble des bitmaps compressés, l'espace moyen requis en mémoire centrale pour lire un bitmap compressé, ainsi que les temps moyens pour effectuer l'union, l'intersection et la récupération des positions des bits positifs de 200 bitmaps compres-

sés. Au départ, une série de tests préalables est lancée afin de remplir le cache et de bénéficier au mieux de l'optimiseur de code de la JVM. Par la suite, chaque test est répété 100 fois avant de présenter les tailles et temps moyens calculés. Les tableaux 3a–3e montrent les résultats obtenus. Le code de ces essais est librement disponible sur internet à l'adresse suivante : https://github.com/samytto/MemoryMappedBitmaps.

Tableau 3. Résultats avec du memory-mapping sur des données réelles

(a) Espace disque moyen (en ko/bitmap) occupé par la sérialisation de 200 bitmaps compressés avec *Roaring bitmap* ou Concise

	CENSUS1881	CENSUSINCOME	WEATHER
Roaring bitmap	9,76	11,57	42,67
Concise	15,66	12,40	46,20

(b) Espace moyen (en octets/bitmap) réservé en mémoire principale pour lire un des 200 bitmaps compressés avec *Roaring bitmap* ou Concise

	CENSUS1881	CENSUSINCOME	WEATHER
Roaring bitmap	104,0	104,0	104,0
Concise	80,0	80,0	80,0

(c) Temps moyen en millisecondes pour effectuer l'union de 200 bitmaps compressés avec *Roaring bitmap* ou Concise

	CENSUS1881	CENSUSINCOME	WEATHER
Roaring bitmap	5	6	6
Concise	307	96	709

(d) Temps moyen en millisecondes pour effectuer l'intersection de 200 bitmaps compressés avec *Roaring bitmap* ou Concise

	CENSUS1881	CENSUSINCOME	WEATHER
Roaring bitmap	0,11	0,09	0,19
Concise	7	2	18

(e) Temps moyen en millisecondes pour récupérer les positions des bits positifs de 200 bitmaps compressés avec *Roaring bitmap* ou Concise

	CENSUS1881	CENSUSINCOME	WEATHER
Roaring bitmap	8	89	166
Concise	43	120	280

Les résultats montrent que l'espace occupé par la sérialisation des bitmaps compressés avec Concise dépasse celui requis pour les *Roaring bitmaps*. Ceci s'explique par la bonne compression de la méthode *Roaring bitmap* qui offre des bitmaps compressés plus compactes comparé à Concise.

La sérialisation sur disque de la majorité des données d'un bitmap et l'exploitation de ces données via du *memory-mapping* permet au programme, lors du chargement d'un bitmap sérialisé, de n'allouer de l'espace en mémoire principale que pour les métadonnées dudit bitmap, qui se comptent au nombre d'une centaine d'octets. L'allocation d'espace en mémoire physique pour le reste des données sérialisées ne se fera qu'au moment opportun, lors duquel le système d'exploitation se chargera de faire parvenir les données nécessaires en mémoire principale à la demande de l'application. De ce fait, l'espace occupé par un bitmap lors de son chargement initial en mémoire centrale sera composé essentiellement de quelques variables de tailles statiques. Ce qui explique la même et infime quantité d'espace consommée par un bitmap compressé avec l'une ou l'autre des deux techniques sur chaque ensemble de données. Toutefois, *Roaring bitmap* affiche un léger surplus de quelques octets par rapport à Concise, dû à sa structure un peu plus complexe.

Le tableau 3c présente les résultats obtenus pour le calcul de l'union de 200 bitmaps compressés à l'aide d'unions horizontales. Cette stratégie a montré qu'elle pouvait réduire significativement les temps d'exécution d'opérations d'unions entraînant plusieurs *Roaring bitmaps* (voir tableau 4). *Roaring bitmap* affiche de remarquables performances comparé à Concise sur les trois ensembles de données, allant jusqu'à $\approx$ 129 fois plus vite sur l'ensemble WEATHER.

Tableau 4. Temps moyen en millisecondes pour calculer l'union de 200 bitmaps compressés avec Roaring bitmap *en utilisant une union traditionnelle et horizontale (les mêmes tests, matériel physique et données des essais décrits dans cette sous-section ont été utilisés)*

	CENSUS1881	CENSUSINCOME	WEATHER
Union horizontale	5	6	6
Union traditionnelle	20	9	36

Le tableau 3d présente les temps moyens consommés par *Roaring bitmap* et Concise pour effectuer l'intersection de 200 bitmaps compressés. Les résultats montrent l'avancée de *Roaring bitmap* par rapport à Concise sur les trois ensembles de données, avec un facteur de $\approx$ 94 sur les données de WEATHER.

Les derniers tests calculent les temps moyens consommés par *Roaring bitmap* et Concise afin de récupérer les positions des bits positifs de 200 bitmaps compressés. Ce type d'opération est effectué dans les SGBD afin d'accèder aux données sélectionnées par une requête. Le tableau 3e illustre les résultats obtenus. Encore une fois, *Roaring bitmap* a atteint de bien meilleures performances par rapport à Concise sur tous les ensembles de données, en étant près de 5 fois plus rapide sur les données de l'ensemble CENSUS1881.

5. Conclusion

Cet article a introduit une nouvelle technique de compression bitmap, nommée *Roaring bitmap*. Des tests sur des données synthétiques et réelles ont permis de comparer les performances de *Roaring bitmap* avec celles de deux autres techniques de compression bitmap 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 1 100 fois sur des données réelles.

Comme travaux futurs, nous envisageons d'implémenter d'autres types d'algorithmes de recherche pour améliorer les temps de calculs des intersections entre bitmaps. (Culpepper, Moffat, 2010) ont montré qu'une recherche *Golomb* (Golomb, 1966) permettait d'exécuter efficacement des intersections entre *posting lists*. Nous souhaitons aussi améliorer la vitesse des accès aléatoires en adoptant une structure de données qui offre des accès directs aux conteneurs et qui est en même temps compacte, tel un bitmap par exemple. Nous projetons également d'expérimenter *Roaring bitmap* sur des bancs d'essais décisionnels, comme le *Star Schema Benchmark* (P. O'Neil *et al*., 2009).

Druid (Yang *et al*., 2014) est un SGBD analytique à code libre, écrit en Java, orienté colonne, distribué, qui permet d'effectuer des analyses OLAP multidimensionnelles sur des quantités massives de données temporelles et en des temps concurrentiels par rapport à d'autres systèmes de traitements de données massives connus dans la littérature, comme : Hadoop (Shvachko, Kuang *et al*., 2010). Pour assurer des temps de traitement rapides, Druid fait usage, entre autres, de bitmaps compressés avec Concise pour indexer les données de base. Ce SGBD sérialise la grande partie de ses données sur des fichiers disques et les manipule avec du *memory-mapping*. Parmi les données sérialisées, figurent des collections d'index bitmap compressés. Les résultats positifs obtenus avec *Roaring bitmap* lors de sa comparaison avec Concise sur des données réelles et dans un contexte de *memory-mapping*, nous laisse prévoir d'intégrer dans un futur proche *Roaring bitmap* comme une technique de compression bitmap au SGBD Druid. Ensuite, des analyses de performances sous ce système seront conduites, afin de recenser les avantages et les inconvénients apportés par *Roaring bitmap* en matière de temps de réponse à différents types de requêtes, de temps de création et de modification d'ensembles d'index bitmap compressés, de quantités d'espaces disque et mémoire consommées par les collections d'index bitmap compressés, ainsi que d'autres fonctionnalités dont Druid améliore les performances à l'aide de bitmaps compressés.

Remerciements

Ces travaux ont pu être réalisés grâce à une subvention du conseil de recherches en sciences naturelles et en génie du Canada (numéro 26 143). Nos sincères remerciements à Owen Kaser pour son aide et sa collaboration durant la réalisation de ce projet.

Bibliographie

Aouiche K., Darmont J., Boussaid O., Bentayeb F. (2005). Automatic Selection of Bitmap Join Indexes in Data Warehouses. In *7th International Conference on Data Warehousing and Knowledge Discovery*, p. 64–73. Copenhagen, Denmark, IEEE Computer Society.

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

Bouchakri R., Bellatreche L. (2011). Sélection Statique et Incrémentale des Index de Jointure Binaires Multiples. *Revue des Nouvelles Technologies de l'Information*, vol. 7e Journées francophones sur les Entrepôts de Données et l'Analyse en ligne, RNTI-B-7, p. 171–187.

Boukhalfa K., Benameur Z., Bellatreche L. (2010). Index de Jointure Binaires : Stratégies de Sélection et Étude de Performances. *Revue des Nouvelles Technologies de l'Information*, vol. EDA'10 Entrepôts de Données et Analyse en ligne, RNTI-B-6, p. 355–366.

Chambi S., Lemire D., Godin R., Kaser O. (2014, juin). Roaring bitmap : nouveau modèle de compression bitmap [Conference Proceedings]. In *10e journèes francophones sur les Entrepôts de Données et l'Analyse en ligne (EDA'14)*, vol. 27(2), p. 37–50. Vichy, France, RNTI.

Chambi S., Lemire D., Owen K., Godin R. (2015, mars). Better bitmap performance with Roaring bitmaps. *Software Practice and Experience (SPE)*, vol. 46, n° 5, p. 709–719.

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

Chang J., Chen Z., Zheng W., Cao J., Wen Y., Peng G. *et al*. (2015). Splwah: A bitmap index compression scheme for searching in archival internet traffic. In *2015 IEEE International Conference on Communications (ICC)*, p. 7089–7094. London, England, IEEE.

Chen Z., Wen Y., Cao Y., Zheng W., Chang J., Wu Y. *et al*. (2015). A survey of bitmap index compression algorithms for big data. *Tsinghua Science and Technology*, vol. 20, p. 100–115.

Colantonio A., Di Pietro R. (2010, jul). Concise: Compressed 'n' Composable Integer Set. *Information Processing Letters*, vol. 110, n° 16, p. 644–650.

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

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

Deliège F., Pedersen T. B. (2010). Position List Word Aligned Hybrid: optimizing space and performance for compressed bitmaps. In *Proceedings of the 13th International Conference on Extending Database Technology*, p. 228–239. New York, NY, USA, ACM.

Frank A., Asuncion A. (2010). *UCI machine learning repository* Web Page. http://archive.ics.uci.edu/ml.

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

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

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

Hahn C., Warren S., London J. (2004). *Edited synoptic cloud reports from ships and land stations over the globe* Web Page. http://cdiac.ornl.gov/ftp/ndp026b/.

historique P. de recherche en demographie. (2009). *The 1852 and 1881 historical censuses of Canada* Web Page. http://www.prdh.umontreal.ca/census/en/main.aspx.

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

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

Lemire D., Kaser O., Aouiche K. (2010). Sorting Improves Word-aligned Bitmap Indexes. *Data & Knowledge Engineering*, vol. 69, n° 1, p. 3–28.

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

O'Neil E., O'Neil P., Wu K. (2007, Sept). Bitmap Index Design Choices and Their Performance Implications. In *the 11th International Database Engineering and Applications Symposium*, p. 72–84. Banff, Alta, IEEE Computer Society.

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

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

Shvachko K., Hairong K., Radia S., Chansler R. (2010). The Hadoop Distributed File System. In *26th Symposium on Mass Storage Systems and Technologies*, p. 1–10. Incline Village, NV, IEEE.

Shvachko K., Kuang H., Radia S., Chansler R. (2010). The Hadoop distributed file system [Conference Proceedings]. In *26th Symposium on Mass Storage Systems and Technologies (MSST)*, p. 1–10. Incline Village, NV, USA, IEEE.

Stockinger K., Wu K. (2008). Bitmap Indices for Data Warehouses. In J. Wang (Ed.), *Data Warehousing and Mining: Concepts, Methodologies, Tools, and Applications*, p. 1590–1605. Hershey, PA, IGI Global.

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

Thusoo A., Sarma J. S., Jain N., Shao Z., Chakka P., Zhang N. *et al*. (2010). Hive - a petabyte scale data warehouse using Hadoop. In *26th International Conference on Data Engineering*, p. 996–1005. Long Beach, CA, USA, IEEE.

Uno T., Kiyomi M., Arimura H. (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*, p. 77–86. New York, NY, USA, ACM.

Wu k., Ahern S., Bethel E., Chen J., Childs H., Cormier-Michel E. *et al*. (2009). Fastbit: An efficient compressed bitmap index technology. In *Journal of Physics: Conference Series*, vol. 180, p. 1–10. United Kingdom, IOP Publishing Ltd.

Wu K., Otoo E. J., Shoshani A. (2006). Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, vol. 31, n° 1, p. 1-38.

Yang F., Tschetter E., Lèautè X., Ray N., Merlino G., Ganguli D. (2014, juin). Druid : A Real–time Analytical Data Store [Conference Proceedings]. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, p. 157–168. New York, NY, USA, ACM.

Article soumis le 1/12/2014.

Accepté le 17/04/2016.