Nouveaux modèles d'index bitmap compressés à 64 bits

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

*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

Résumé. Les index bitmap sont très utilisés dans les entrepôts de données et moteurs de recherche pour accélérer les requêtes d'interrogation. Leurs principaux avantages sont leur forme compacte et leur capacité à tirer profit du traitement parallèle de bits dans les CPU (*bit-level parallelism*). Dans l'ère actuelle du Big Data, les collections de données deviennent de plus en plus volumineuses. Les librairies d'index bitmap compressés disponibles à ce jour dans la littérature, telles que : *Roaring bitmap*, *WAH* ou *Concise*, ne supportent qu'au plus $2^{32} \approx 4$ milliards d'entrées et sont très souvent impraticables dans de tels contextes. Après avoir constaté ce besoin tant dans le milieu industriel que scientifique, nous proposons trois nouveaux modèles d'index bitmap compressés, basés sur le format de notre précédente contribution *Roaring bitmap* et qui supportent jusqu'à 2^{64} entrées. Des expériences sur des données synthétiques ont été mises en œuvre pour comparer les performances des trois nouvelles propositions avec la solution du moteur de recherche Apache Lucene : *OpenBitSet*, et d'autres collections Java. Les résultats ont montré que les trois nouvelles techniques ont été près de ≈ 300 millions de fois et ≈ 1800 fois moins volumineuses en consommation mémoire qu'*OpenBitSet* et les collections Java, respectivement. Aussi, les trois nouveaux modèles ont calculé des opérations logiques, jusqu'à ≈ 6 millions de fois et jusqu'à ≈ 63 milles fois plus vite qu'*OpenBitSet* et les structures Java, respectivement.

1 Introduction

Les quantités d'informations générées de nos jours ne cessent de croître à une vitesse phénoménale (Richard Benjamins, 2014). Pour indexer de telles masses de données, la majorité des librairies de représentation d'index bitmap proposées à ce jour s'avèrent impraticables dans de telles situations, car elles ne peuvent être appliquées que sur des ensembles de données ne dépassant pas les $2^{32} \approx 4$ milliards d'entrées. Nous citons comme exemple, la librairie ConciseSet 2.2 (Colantonio, 2010) implémentant les modèles de compression bitmap *WAH* (Wu et al., 2006) et *Concise* (Colantonio et Di Pietro, 2010) en 32 bits, la librairie JavaEWAH (Lemire et Kaser, 2010) implémentant le modèle de la technique de compression bitmap

à 32 bits *EWAH* (Lemire et al., 2010), et la librairie RoaringBitmap (Roaring's team, 2014) mettant en œuvre le modèle de la solution de compression bitmap précédemment proposée : *Roaring bitmap* (Chambi et al., 2014, 2015). Les ingénieurs du moteur de recherche Apache Lucene (Apache, 2012) ont fait face à cette problématique, et ont introduit une librairie d'indexation bitmap : *OpenBitset* (Apache, 2010), qui supporte jusqu'à $64 \times 2^{32} - 1$ entrées. Cependant, cette solution reste simple du fait qu'elle n'adopte aucune méthode de compression, et par conséquent, ne promet pas de bonnes performances. En réponse à cette problématique rencontrée tant dans le milieu scientifique qu'industriel, nous avons introduit trois nouveaux modèles d'index bitmap compressés inspirés de la solution de compression bitmap à 32 bits, *Roaring bitmap* (Chambi et al., 2014, 2015), et qui peuvent indexer jusqu'à 2^{64} entrées ; un seuil plutôt raisonnable par rapport aux collections de données rencontrées de nos jours. Ce travail introduit ces trois librairies en présentant leurs différents formats et les mécanismes adoptés par chaque solution pour calculer des opérations logiques entre bitmaps. Les bancs d'essai mis en oœuvre pour comparer les performances de ces nouvelles techniques avec la solution d'indexation bitmap d'Apache Lucene, *OpenBitSet*, et avec d'autres collections Java définies dans l'empaquetage Java.Util sont également présentés.

Le papier est organisé comme suit : la section 2 introduit les trois nouveaux modèles d'index bitmap compressés supportant jusqu'à 2^{64} entrées. Les bancs d'essai évaluant les performances de ces index bitmap compressés, les résultats obtenus ainsi qu'une analyse de ces résultats sont présentés à la section 3. La section 4 termine avec la conclusion.

2 Modèles d'index bitmap compressés

Les modèles d'index bitmap compressés proposés représentent chaque bit positif dans un bitmap par un entier de 64 bits, qui indique la position, débutant possiblement de 0, du bit positif dans le bitmap. Ces modèles se comptent au nombre de trois et sont définis dans les sous-sections suivantes.

2.1 RoaringTreeMap

Le modèle *RoaringTreeMap* combine un arbre Java TreeMap avec la structure *Roaring bitmap* pour indexer un ensemble d'entiers de 64 bits représentant les positions des bits positifs d'un bitmap. Un TreeMap est une mise en œuvre de l'arbre rouge-noir, un arbre binaire de recherche équilibré, et fait partie des collections de données définies dans l'empaquetage Java.Util. Les différentes opérations de la structure arborescente (insertion, recherche, etc.) ont été implémentées avec les algorithmes proposés par (Cormen et al., 2001).

Pour représenter un entier de 64 bits, ce modèle le divise en deux parties. La première partie constitue les 32 bits de poids fort de l'entier, et la deuxième les 32 bits de poids faible. Un nœud d'un *RoaringTreeMap* est composé d'une clé, qui est un entier de 32 bits, et d'un *Roaring bitmap*. *RoaringTreeMap* regroupe un ensemble d'entiers de 64 bits partageant les mêmes 32 bits de poids fort dans un même nœud. La clé du nœud stocke les 32 bits de poids fort communs du groupe d'entiers, et le *Roaring bitmap* associé au nœud renferme les 32 bits de poids faible restants. Ainsi, *RoaringTreeMap* applique une forme de compression préfixe sur les 32 bits de poids fort d'un tel groupe d'entiers, pouvant sauver jusqu'à $32 \times (2^{32} - 1)$ bits pour chaque tel groupe.

Une opération d'insertion ou de recherche d'un entier de 64 bits dans un *RoaringTreeMap* commence par effectuer un accès aléatoire dans l'arbre afin de trouver un nœud comportant une clé de valeur égale à celle des 32 bits de poids fort de l'entier en question. L'adoption d'un arbre de recherche binaire équilibré pour le modèle *RoaringTreeMap* permet d'effectuer une telle opération en un temps de complexité logarithmique par rapport au nombre total des nœuds de l'arbre parcouru. Par la suite, si un tel nœud est trouvé, une deuxième opération d'insertion ou de recherche sera effectuée au niveau du *Roaring bitmap* associé à ce nœud, ce qui prend aussi un temps de complexité logarithmique par rapport au nombre d'entrées dans l'index de premier niveau du *Roaring bitmap* et au nombre d'entrées stockées dans le conteneur accédé, si ce dernier est représenté sous la forme d'un tableau trié.

Un autre avantage de *RoaringTreeMap* vient de la propriété des arbres de recherche binaires équilibrés, qui garantie que les nœuds de l'arbre puissent être toujours parcourus dans l'ordre croissant des valeurs de leurs clés dans un temps de complexité linéaire par rapport au nombre de nœuds dans l'arbre. Combiné au tri par ordre croissant des entiers à 32 bits maintenus au sein des *Roaring bitmaps*, cela permet au modèle *RoaringTreeMap* d'itérer dans un ordre croissant sur l'ensemble des entiers à 64 bits indexés en un temps linéaire par rapport au nombre de ces entiers. Ce plus s'avère très efficace pour le calcul d'opérations ensemblistes basiques entre deux *RoaringTreeMaps*, comme : l'intersection, l'union, ou l'union exhaustive, qui peuvent s'effectuer en un temps d'ordre linéaire par rapport au nombre des entiers contenus dans les deux arbres. Sans cette propriété, une telle opération serait exécutée en un temps de complexité quadratique par rapport au nombre des éléments dans les deux ensembles.

2.1.1 L'union de deux *RoaringTreeMaps*

Une union prend deux *RoaringTreeMaps* en entrée et renvoie le résultat dans un nouveau *RoaringTreeMap*. L'algorithme commence par allouer un nouveau tableau dynamique vide qui sera rempli avec les nœuds formant le résultat de l'union. Ensuite, les deux arbres en entrée sont parcourus itérativement dans l'ordre croissant des valeurs de leurs clés. Lors d'une itération, si les deux nœuds courants ont des clés de valeurs différentes, l'algorithme copie le nœud contenant la plus petite des deux clés, l'insère dans le tableau dynamique, puis avance d'une position dans l'arbre contenant ce nœud. Sinon, si les deux nœuds comparés lors d'une itération renferment des clés de même valeur, un OU logique est calculé entre les *Roaring bitmaps* des deux nœuds, puis le résultat est retourné dans un nouveau *Roaring bitmap*. Un nouveau nœud contenant le *Roaring bitmap* obtenu et une clé de valeur égale à celle des deux nœuds comparés sera inséré dans le tableau dynamique. Ces opérations continuent jusqu'à avoir parcouru les deux arbres au complet. À la fin, les nœuds insérés jusqu'ici dans le tableau dynamique seront triés dans l'ordre croissant des valeurs de leurs clés, puis un algorithme récursif construit l'arbre *RoaringTreeMap* résultant à partir du tableau dynamique.

L'algorithme nécessite un temps de $\Theta(n_1+n_2)$ pour parcourir les deux arbres en entrée, où n_1 et n_2 représentent le nombre de nœuds dans les deux arbres, respectivement. La construction du tableau dynamique se fait en un temps de $O(n_1+n_2)$, car il pourrait y avoir au plus $O(n_1+n_2)$ insertions, et chacune d'elles se fait dans un temps amorti constant. La construction de l'arbre *RoaringTreeMap* final est réalisée avec un algorithme récursif efficace qui consomme un temps de $O(n_1+n_2)$. Ainsi, le temps total d'exécution de ce deuxième algorithme d'union est de $O(n_1+n_2)$ plus le temps nécessaire pour calculer les possibles OU logiques entre *Roaring bitmaps* (Chambi et al., 2014, 2015).

2.1.2 L'intersection de deux *RoaringTreeMaps*

Une opération d'intersection prend deux *RoaringTreeMaps* en entrée et renvoie le résultat dans un nouveau *RoaringTreeMap*. L'algorithme commence par allouer un tableau dynamique vide qui sera rempli avec les nœuds formant l'arbre résultant de l'intersection. Après, l'algorithme itère sur les nœuds des deux arbres dans l'ordre croissant des valeurs de leurs clés. À la rencontre de deux nœuds de valeurs de clés différentes, l'algorithme avance d'une position sur l'arbre contenant le nœud de plus petite valeur de clé. Sinon, dans le cas où les deux nœuds renferment deux clés de même valeur, un ET logique est exécuté entre les *Roaring bitmaps* associés aux deux clés, et le résultat est renvoyé dans un nouveau *Roaring bitmap*. Ensuite, un nouveau nœud contenant une clé de valeur égale à celle des deux clés équivalentes, et le *Roaring bitmap* retourné seront ajoutés au tableau dynamique. Ces opérations se poursuivent jusqu'à avoir complètement parcouru l'un des deux arbres au complet.

À la fin des itérations, le tableau dynamique contiendra les nœuds formant l'arbre final dans l'ordre croissant des valeurs de leurs clés. Ensuite, un algorithme récursif construira l'arbre *RoaringTreeMap* résultant.

Le temps d'exécution de ce deuxième algorithme d'intersection dépend du temps nécessaire pour parcourir l'un des deux arbres, puis pour remplir le tableau dynamique, et en dernier pour construire l'arbre *RoaringTreeMap* final. Le temps total pour accomplir les précédentes opérations est de l'ordre de $\Theta(n_1 + n_2)$ au pire cas, où n_1 et n_2 représentent, respectivement, le nombre de nœuds dans les deux arbres. Au meilleur cas, un temps de $\Theta(\min(n_1, n_2))$ est consommé par l'algorithme d'intersection, lorsqu'un parcours direct de l'un des deux arbres suffit pour arriver au résultat final. Cela sans oublier le temps consommé par les possibles calculs de ET logiques entre *Roaring bitmaps* (Chambi et al., 2014, 2015).

2.2 RoaringTwoLevels

Ce modèle utilise une structure à deux niveaux presque similaire à celle de *Roaring bitmap* pour stocker un ensemble d'entiers de 64 bits représentant les positions des bits positifs d'un bitmap. Le premier niveau est un tableau dynamique dans lequel chaque entrée renferme un entier de 64 bits et un pointeur vers un conteneur. L'ensemble des conteneurs pointés par les entrées du tableau du premier niveau forme le deuxième niveau de la structure de données.

Un groupe d'entiers de 64 bits partageant les mêmes 48 bits de poids fort sont regroupés dans une même entrée de l'index de premier niveau. Les 48 bits de poids fort communs sont stockés dans les bits de poids fort des 64 bits de l'entrée, et les 16 bits de poids faible restants du groupe d'entiers sont conservés dans le conteneur pointé par l'entrée. Pour ce qui est des 16 bits de poids faible non utilisés au niveau des 64 bits d'une entrée de premier niveau, ils serviront à garder la cardinalité du groupe d'entiers indexés par cette entrée, qui peut atteindre jusqu'à $2^{16} - 1$ entiers. Ceci permettra de calculer efficacement la cardinalité d'un bitmap en n'effectuant qu'un seul parcours du tableau de premier niveau, pouvant améliorer ainsi plusieurs sortes de requêtes utilisées dans les bases de données, comme les requêtes COUNT en SQL.

Nommons les 48 bits de poids fort parmi les 64 bits d'une entrée du tableau de premier niveau par les bits communs de l'entrée, et les 16 bits de poids faible restants dans les 64 bits d'une entrée de premier niveau par les bits de cardinalité de l'entrée. Le tableau dynamique du premier niveau est maintenu trié dans l'ordre croissant des valeurs des bits communs de

ses entrées. De façon similaire à *Roaring bitmap*, un conteneur au deuxième niveau peut être représenté par un bitmap d'une taille statique de 2^{16} bits, ou par un tableau dynamique d'entiers de 16 bits triés dans un ordre croissant, en fonction de la densité du conteneur. Ainsi, pour accéder à un entier de 64 bits quelconque stocké dans la structure de données, une première opération de recherche binaire à temps logarithmique par rapport au nombre des entrées est effectuée sur le tableau de premier niveau à la recherche d'une entrée avec une valeur de bits communs équivalente aux 48 bits de poids fort de l'entier à accéder. Une fois une telle entrée trouvée, une deuxième opération de recherche est effectuée au niveau du conteneur associé à l'entrée, de façon similaire à *Roaring bitmap*, en consommant au plus un temps logarithmique lorsque le conteneur est représenté par un tableau dynamique.

2.2.1 L'union de deux *RoaringTwoLevels*

L'union de deux *RoaringTwoLevels* prend deux *RoaringTwoLevels* en entrée et renvoie le résultat dans un nouveau *RoaringTwoLevels*. L'algorithme commence par parcourir les entrées des deux tableaux de premier niveau des deux *RoaringTwoLevels* en entrée. À chaque itération, les valeurs des bits communs de chacune des deux entrées courantes sont comparées. Si les valeurs sont différentes, l'algorithme insère une copie de l'entrée dont les bits communs sont de plus petite valeur et de son conteneur dans le *RoaringTwoLevels* résultant. Ensuite, l'algorithme avance d'une position sur le tableau de premier niveau de l'entrée copiée. Sinon, dans le cas où les valeurs des deux bits communs sont équivalentes, l'algorithme ajoute une nouvelle entrée au tableau de premier niveau du *RoaringTwoLevels* résultant, contenant une valeur de bits communs équivalente à celle des deux entrées comparées, et un nouveau conteneur au deuxième niveau stockant le résultat de l'union des deux conteneurs pointés par chacune des deux entrées. Par la suite, la cardinalité de la nouvelle entrée est calculée et ses bits de cardinalité sont mis à jour. Puis, l'algorithme avance d'une position sur les deux tableaux de premier niveau comparés. Ces opérations continuent jusqu'à avoir complètement parcouru les deux *RoaringTwoLevels* donnés en entrée.

Le temps d'exécution d'une union entre deux *RoaringTwoLevels* dépend, en premier lieu, du temps nécessaire pour parcourir les deux tableaux de premier niveau des deux bitmaps en entrée. Ce qui prend un temps de $\Theta(n_1 + n_2)$, avec deux tableaux de tailles n_1 et n_2. Puis, en second lieu, du temps requis pour effectuer les unions entre conteneurs, qui dépend à son tour du type de conteneur utilisé lors de chaque opération. Pour plus de détails sur les complexités temporelles des opérations d'union entre conteneurs, le lecteur peut consulter les références suivantes : Chambi et al. (2014, 2015).

2.2.2 L'intersection de deux *RoaringTwoLevels*

L'algorithme d'intersection prend deux *RoaringTwoLevels* en entrée et retourne un nouveau *RoaringTwoLevels* en sortie. Tout d'abord, l'algorithme parcourt itérativement les tableaux de premier niveau des deux *RoaringTwoLevels* en entrée. À chaque itération, les valeurs des bits communs des deux entrées courantes sont comparées. Deux cas peuvent être rencontrés. Le premier cas se présente lorsque les valeurs des bits communs des deux entrées sont différentes. Si x_1 représente les bits communs de plus petite valeur, et x_2 ceux de plus grande valeur, alors l'algorithme avance dans le tableau renfermant la valeur x_1 à la recherche d'une entrée située après l'indice de x_1 et qui détient la plus petite valeur de bits communs étant

supérieure ou égale à x_2. Cette dernière opération de recherche est réalisée à l'aide d'un algorithme de recherche exponentielle (*galloping*) dans un temps de $O(\log\ d)$, où d représente la distance traversée par l'algorithme dans le tableau (Bentley et Yao, 1976). Sachant qu'en général $d \ll n$, où n correspond à la taille du tableau, une recherche exponentielle est généralement bien plus efficace qu'une simple recherche binaire.

Par contre, dans les cas où les valeurs des bits communs des deux entrées comparées lors d'une itération sont équivalentes, une opération ET logique est exécutée entre les conteneurs indexés par les deux entrées et le résultat est renvoyé dans un nouveau conteneur. Une nouvelle entrée pointant vers ce dernier et ayant des bits communs de valeur équivalente à celles des deux entrées comparées sera insérée dans le tableau de premier niveau du *RoaringTwoLevels* résultant. L'algorithme avance ensuite d'une position sur les deux tableaux de premier niveau comparés. Ces opérations se poursuivent jusqu'à avoir parcouru l'un des deux tableaux de premier niveau.

Le temps d'exécution de l'algorithme d'intersection dépend du temps nécessaire pour comparer les tableaux de premier niveau des deux *RoaringTwoLevels* fournis en entrée, plus le temps pour calculer les possibles ET logiques entre conteneurs. La première opération s'effectue en un temps de $\Theta(n_1 + n_2)$ au pire des cas, lorsque les deux tableaux doivent être lus au complet avant que l'algorithme ne termine, sachant que n_1 et n_2 représentent le nombre d'entrées dans le premier et le deuxième tableau, respectivement. Au meilleur des cas, un seul parcours direct du plus petit des deux tableaux suffirait pour obtenir le résultat final, consommant, grâce à l'algorithme de recherche exponentielle, un temps de $O(\log(\min(n_1, n_2)))$. Pour ce qui est du deuxième type d'opérations, le temps d'exécution de celui-ci dépend du nombre total de ET logiques calculés entre conteneurs et du type de conteneur impliqué lors de chaque opération (Chambi et al., 2014, 2015).

2.3 LazyRoaring

Le modèle *Roaring bitmap* (Chambi et al., 2014, 2015) a précédemment montré que le fait d'utiliser un tableau de premier niveau dont chaque entrée indexe un conteneur renfermant jusqu'à 2^{16} entiers pouvait permettre d'éviter l'accès à plusieurs conteneurs lors de différents types de traitements effectués sur un *Roaring bitmap*. En s'inspirant de cette idée qui a été avantageuse pour *Roaring bitmap*, on a introduit ce nouveau modèle d'index bitmap compressés supportant jusqu'à 2^{64} entrées. *LazyRoaring* compte trois niveaux. Le premier niveau est un tableau d'entiers de 32 bits, le deuxième niveau est constitué de plusieurs tableaux d'entiers de 16 bits, et le troisième niveau est formé de conteneurs pouvant être représentés avec des tableaux ou des bitmaps comme ceux du modèle Roaring bitmap. Chaque entrée du premier niveau pointe vers un tableau du deuxième niveau. Un tableau du deuxième niveau ne peut être pointé (indexé) que par une seule entrée du premier niveau. Une entrée d'un tableau du deuxième niveau pointe vers un conteneur, et ce dernier ne peut être indexé que par une seule entrée du deuxième niveau.

Un groupe d'entiers de 64 bits partageant les mêmes 32 bits de poids fort sont indexés par une entrée dans le tableau de premier niveau. La valeur des 32 bits de poids fort récurrents est préservée dans l'entrée du premier niveau. Les entiers rassemblés par une entrée de premier niveau sont une deuxième fois indexés par un tableau de deuxième niveau. Ce dernier regroupe dans une même entrée les entiers de 64 bits qui ont les mêmes valeurs de bits au niveau du 33^e bit de poids fort jusqu'au 48^e bit de poids fort (soit les 16 bits situés juste après les premiers

32 bits de poids fort). La valeur des 16 bits récurrents est préservée dans l'entrée de deuxième niveau. Ainsi, *LazyRoaring* applique deux fois une compression préfixe aux entiers formant un bitmap, respectivement, sur le premier et sur le deuxième niveau de la structure de données. Pour permettre des traitements efficaces sur les deux niveaux d'un *LazyRoaring*, les entiers des tableaux du premier et du deuxième niveau sont maintenus triés dans un ordre croissant. Les 16 bits restants d'un groupe d'entiers de 64 bits indexés par une entrée de deuxième niveau sont conservés dans le conteneur pointé par cette entrée.

2.3.1 Accès aléatoires dans un *LazyRoaring*

Un accès aléatoire à un entier de 64 bits stocké dans un *LazyRoaring*, consiste en un premier temps à faire une recherche binaire sur le tableau du premier niveau afin de trouver une entrée de valeur égale aux 32 bits de poids fort de l'entier à accéder. Cette opération prend un temps de $O(\log n)$, où n fait référence au nombre d'entrées dans le tableau. Après qu'une telle entrée ait été repérée, une deuxième opération de recherche binaire est effectuée sur le tableau de deuxième niveau pointé par cette entrée, dans le but de tomber sur une entrée de deuxième niveau de valeur égale aux 16 bits situés après les 32 bits de poids fort de l'entier recherché. Cette opération s'effectue dans un temps de $O(\log m)$, où m représente le nombre d'entrées dans le tableau de deuxième niveau. Une fois une telle entrée trouvée sur le deuxième niveau, une troisième opération de recherche est réalisée au niveau du conteneur indexé par l'entrée pour trouver une occurrence des 16 bits de poids faible de l'entier à accéder. Le temps d'exécution de cette dernière opération dépend du type de conteneur adopté par l'entrée en question (Chambi et al., 2014, 2015), et qui nécessite, au meilleur cas, un temps constant lorsque le conteneur est représenté par un bitmap et, au pire des cas, d'effectuer une recherche binaire sur le tableau dynamique représentant le conteneur dans un temps de complexité logarithmique par rapport au nombre d'entrées dans le tableau. Par conséquent, si n, m et h représentent les tailles des tableaux de premier, deuxième et troisième niveau, respectivement, un accès aléatoire dans un *LazyRoaring* consommera un temps de $O(\log n + \log m + \log h)$.

2.3.2 L'union de deux *LazyRoarings*

Une opération d'union prend deux *LazyRoarings* en entrée et retourne un nouveau *LazyRoaring*. L'algorithme commence par itérer sur les deux tableaux de premier niveau des deux bitmaps afin de comparer les valeurs à 32 bits de leurs entrées. Lors d'une itération, si les valeurs des entrées courantes dans les deux tableaux diffèrent, alors l'algorithme insère une nouvelle entrée dans le tableau de premier niveau du bitmap résultant, qui fait référence à l'entrée de plus petite valeur parmi les deux entrées comparées. Cette insertion est effectuée en copiant seulement le pointeur qui pointe vers la plus petite des deux entrées comparées, évitant ainsi l'allocation d'un nouvel espace mémoire pour stocker une copie du tableau de deuxième niveau et des conteneurs indexés par la plus petite des deux entrées évaluées, permettant au final de réduire les temps de traitements et l'espace mémoire consommés durant une opération d'union. Cette optimisation est inspirée de la stratégie *copy-on-write* (Wikipedia, 2015) utilisée par les systèmes d'exploitation. L'algorithme avance ensuite d'une position sur le tableau de premier niveau renfermant la plus petite valeur parmi les deux entrées comparées.

Dans le cas échéant, lorsque les deux entrées de premier niveau comparés lors d'une itération possèdent des entiers de 32 bits équivalents, l'algorithme ajoute une nouvelle entrée dans

le premier niveau du bitmap résultant, avant d'avancer d'une position dans les deux tableaux de premier niveau. Cette nouvelle entrée renferme un entier de 32 bits de valeur égale à celle des deux entrées de premier niveau évaluées, et un pointeur vers un nouveau tableau de deuxième niveau obtenu suite à une opération d'union calculée entre les deux tableaux de deuxième niveau indexés par les deux entrées de premier niveau. Cette dernière opération d'union est effectuée de façon similaire à celle réalisée sur le premier niveau d'un *LazyRoaring*, soit en ajoutant une entrée dans le deuxième niveau du bitmap résultant qui contiendra une copie du pointeur de la plus petite des deux entrées de deuxième niveau comparées, lorsque ces deux dernières possèdent des entiers de 16 bits différents pendant une itération, ou en ajoutant une nouvelle entrée de deuxième niveau au bitmap résultant qui contient un entier de 16 bits équivalent à ceux des deux entrées de deuxième niveau courantes, et un pointeur vers un nouveau conteneur résultant de l'union des deux conteneurs indexés par les deux entrées de deuxième niveau comparées. Une union entre deux conteneurs est réalisée de la même façon qu'indiqué dans les papiers introduisant le modèle *Roaring bitmap* (Chambi et al., 2014, 2015). Ces opérations sur le deuxième et le premier niveau se poursuivent jusqu'à avoir entièrement parcouru les deux tableaux de deuxième ou de premier niveau des deux bitmaps à fusionner.

Le temps d'exécution d'une opération d'union entre deux *LazyRoarings* dépend du temps nécessaire pour parcourir les tableaux de premier niveau des deux bitmaps. Ce qui se fait en un temps de $\Theta(n_1 + n_2)$, où n_1 et n_2 représentent, respectivement, le nombre d'entrées dans les deux tableaux de premier niveau. Plus, le temps pour calculer les unions entre les tableaux de deuxième niveau accédés durant l'opération, où chaque union prend un temps de $\Theta(m_1+m_2)$, avec m_1 et m_2 représentant le nombre d'éléments respectifs des deux tableaux de deuxième niveau traités. Rajouté à cela, le temps nécessaire pour effectuer les possibles unions entre les conteneurs atteints, dont chacune dépend du type des conteneurs (Chambi et al., 2014, 2015).

2.3.3 L'intersection de deux *LazyRoarings*

Une opération d'intersection prend deux *LazyRoarings* en entrée et retourne un nouveau *LazyRoaring*. L'algorithme commence par itérer sur les tableaux de premier niveau des deux bitmaps afin de comparer leurs valeurs de 32 bits. Si au cours d'une itération, les deux entrées courantes possèdent des valeurs de 32 bits différentes, l'algorithme avance d'une position sur le tableau de premier niveau contenant la plus petite des deux valeurs de 32 bits comparées. Sinon, lorsque les deux entrées courantes renferment des valeurs de 32 bits équivalentes, l'algorithme ajoute une nouvelle entrée de premier niveau au *LazyRoaring* résultant qui contient une valeur de 32 bits équivalente à celle des deux valeurs comparées, et qui pointe vers un nouveau tableau de deuxième niveau obtenu suite au calcul de l'intersection entre les deux tableaux de deuxième niveau indexés par les deux entrées de premier niveau courantes. L'intersection de deux tableaux de deuxième niveau est réalisée d'une façon similaire à celle de deux tableaux de premier niveau, sauf qu'une intersections entre deux conteneurs est calculée lorsque les deux entrées de deuxième niveau évaluées lors d'une itération possèdent les mêmes valeurs de 16 bits. Le résultat de cette dernière intersection se présente sous la forme d'un nouveau conteneur qui sera associé à la nouvelle entrée de deuxième niveau ajoutée dans le *LazyRoaring* résultant. L'intersection sur le premier ou le deuxième niveau continue jusqu'à ce que l'un des deux tableaux traités soit parcouru en entier.

Le temps d'exécution d'une opération d'intersection entre deux *LazyRoarings* dépend, en premier lieu, du temps nécessaire pour comparer les deux tableaux de premier niveau. Ce qui

se fait dans un temps de $\Theta(n_1 + n_2)$ au pire cas, et en $\Theta(min(n_1, n_2))$ au meilleur cas, où n_1 et n_2 représentent, respectivement, le nombre d'entrées dans le premier et le deuxième tableau. Suivi du temps pour comparer les tableaux de deuxième niveau lors de chaque accès au second niveau du bitmap, où chaque opération de ce type se fait dans un temps de $O(m_1 + m_2)$ au pire cas, et en $\Theta(min(m_1, m_2))$ au meilleur cas, avec deux tableaux de deuxième niveau de tailles respectives, m_1 et m_2. Reste à rajouter en dernier le temps nécessaire pour traiter les possibles intersections entre les conteneurs accédés au troisième niveau, où le temps d'exécution de chaque intersection dépend du type de conteneurs utilisés (Chambi et al., 2014, 2015).

3 Expériences

Les techniques de compression bitmap introduites dans ce chapitre ont été mises en œuvre avec le langage de programmation Java SE 8. Des expériences furent réalisées ensuite pour comparer les performances des nouvelles techniques avec celles d'*OpenBitSet* et d'autres structures de données définies dans l'empaquetage Java.Util. Les expériences ont été réalisées avec l'outil *Java Microbenchmark Harness* (JMH) (Oracle, 2015) et exécutées sur un processeur AMD FX™-8150 à Huit Cœurs avec une fréquence d'horloge de 3,60 GHz et 32 GB de mémoire RAM. Nous utilisons le serveur JVM à 64 bits d'Oracle sur un système Linux Ubuntu 12.04.1 LTS. Le code source des bancs d'essai et des trois librairies de compression bitmap introduites est librement accessible sur internet : Chambi (2015b,a,d,e,c).

3.1 Données synthétiques

Des expériences ont été conduites sur des bitmaps synthétiques générés en suivant deux types de distributions : uniforme et Zipf (Zipf, 1949), avec des densités d variantes de 10^{-9} à 10^{-4}. Un essai est conduit sur une distribution de probabilités et une densité d données. Deux bitmaps sont aléatoirement générés lors d'un essai. Pour générer un bitmap, un entier $max = 50 \times 10^9$, représentant la valeur maximale possible pour un entier, est initié avant que de lancer une suite d'itérations. Le seuil de max et des densités testées ont été fixés à ces valeurs pour des contraintes liées à l'espace mémoire disponible. Lors de chaque itération, un nombre réel a est pseudo-aléatoirement généré depuis l'intervalle [0, 1[. Dans le cas d'une distribution uniforme, l'entier obtenu de $\lfloor a \times max \rfloor$ est ajouté à l'ensemble d'entiers résultant. Tandis que dans le cas d'une distribution de Zipf, l'entier résultant de $\lfloor a^2 \times max \rfloor$ y est inséré, cela a tendance à pousser les entiers générés vers les plus petites valeurs. Colantonio et Di Pietro (2010) ont utilisé les mêmes équations pour obtenir des ensembles d'entiers sur les mêmes distributions. Pour chaque distribution, ces itérations se poursuivent jusqu'à l'obtention d'un ensemble de $N = \lfloor d \times max \rfloor$ entiers distincts. Les entiers ainsi obtenus représenteront les positions des bits positifs du bitmap à générer.

Lors d'un essai, les deux bitmaps générés sont représentés avec chaque structure de données à évaluer. Par la suite, nous mesurons le temps moyen nécessaire pour ajouter un bit à 1 à un bitmap et pour calculer l'union et l'intersection de deux bitmaps. Afin de bénéficier de l'optimiseur de code de la JVM, une phase de réchauffement (*warming-up*) qui consiste à exécuter un essai 5 fois sans prise en compte des mesures à prélever est lancée avant chaque test. Ensuite, un test est répété 5 fois avant d'afficher les moyennes des mesures capturées à chaque répétition (Chambi, 2015b). La quantité moyenne de l'espace mémoire requis pour le stockage

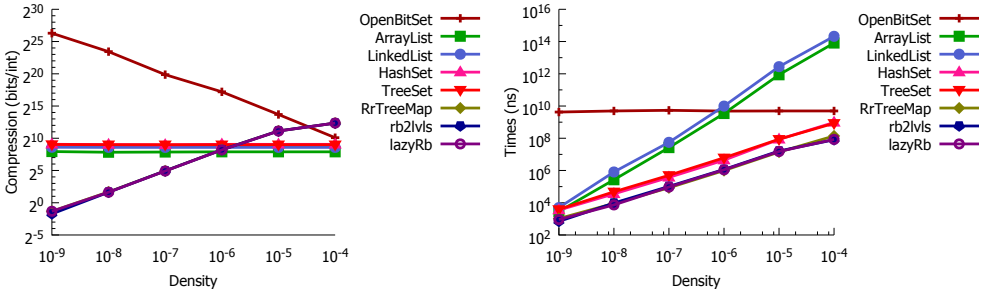
d'un bitmap avec chaque type de structure de données a été mesurée aussi. Cependant, ces bancs d'essais n'ont pas été réalisés avec JMH, mais ont été implémentés avec un programme Java (Chambi, 2015a).

Les figures 1b et 1c montrent les temps CPU moyens consommés par les structures de données pour calculer l'intersection de deux ensembles d'entiers de 64 bits. Sachant que la structure *OpenBitSet* et les collections Java ne proposent que des algorithmes d'intersection à caractère en place (*in-place*), on commence, pour calculer une intersection entre deux de ces structures de données, par copier la première structure en entrée, puis on effectue une intersection en place entre la copie obtenue et la deuxième structure de données en entrée. La méthode *retainAll* a été utilisée pour calculer cette dernière intersection dans le cas des collections Java, et la méthode *and* est utilisée dans le cas d'*OpenBitSet*.

Les graphiques ont montré des résultats presque similaires sur les deux distributions. Les temps de traitements croient linéairement avec le nombre des entiers contenus dans les deux ensembles fusionnés, sauf pour *OpenBitSet* dont les performances restent stables sur les différentes densités. Sur de très faible densités (entre 10^{-9} à 10^{-7}), *OpenBitSet* affiche les plus faibles performances (entre ≈ 6 millions et ≈ 57 fois plus lent que les 3 nouvelles méthodes de compression bitmap). Ceci est essentiellement dû à son grand volume qui induit un nombre important d'opérations d'allocation de nouveaux espaces mémoires comparé au reste des structures de données.

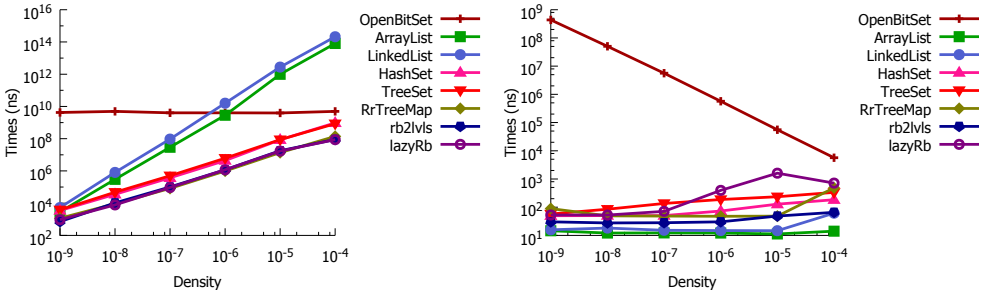
Les deux structures Java *ArrayList* et *LinkedList* sont aussi très lentes et croient très vite à cause de la complexité temporelle quadratique par rapport à la taille des deux ensembles en entrée qu'elles consomment pour réaliser une opération d'intersection. Leurs performances deviennent plus pires que celle d'*OpenBitSet* après la densité 10^{-6}. Les deux structures Java, *HashSet* et *TreeSet*, affichent des résultats presque équivalents à ceux des deux précédentes collections Java sur la plus faible densité testée (10^{-9}), mais croient moins vite sur des densités de plus en plus fortes, avec la structure *HashSet* qui reste toujours plus rapide qu'un *TreeSet*. Cette observation revient aux complexités temporelles d'une opération d'intersection avec chaque structure de données, qui est de $\Theta(n_1)$ dans le cas d'une *HashSet* et de $O(n_1 \log n_2)$ pour la *TreeSet*, où n_1 représente le nombre d'éléments dans le premier ensemble et n_2 le nombre d'éléments du second.

Les trois nouvelles techniques de compression bitmap introduites affichent les meilleures performances sur toutes les densités testées, en étant jusqu'à ≈ 6 millions de fois plus rapides qu'*OpenBitSet*, jusqu'à $\approx$ 63 milles fois plus efficaces que les deux structures *ArrayList* et *LinkedList* et jusqu'à ≈ 6 fois plus performantes que les collections Java *HashSet* et *TreeSet*. Les trois techniques affichent des performances presque similaires sur toutes les densités. Sur les faibles densités, les bitmaps introduits ne possèdent en général que très peu d'entiers partageant les mêmes 48 bits de poids fort, et le temps d'exécution dans ces cas est largement dominé par les comparaisons effectuées sur les hauts niveaux des trois modèles. Possédant la structure haut niveau la plus simple parmi les trois modèles, qui consiste en un simple tableau, *RoaringTwoLevels* affichent les meilleurs résultats sur ces densités-là, en étant jusqu'à $\approx 1,32$ fois et jusqu'à $\approx 1,62$ fois plus rapide que *LazyRoaring* et *RoaringTreeMap*, respectivement. Il est suivi de près par *LazyRoaring*, qui utilise des tableaux sur les hauts niveaux ce qui permet d'organiser les données dans un espace contigu en mémoire centrale, causant ainsi moins de défauts de cache (*caches misses*) induisant moins de transfert de données entre le CPU et la mémoire RAM lors des calculs, comparé aux nœuds, stockés de façon dispersée en



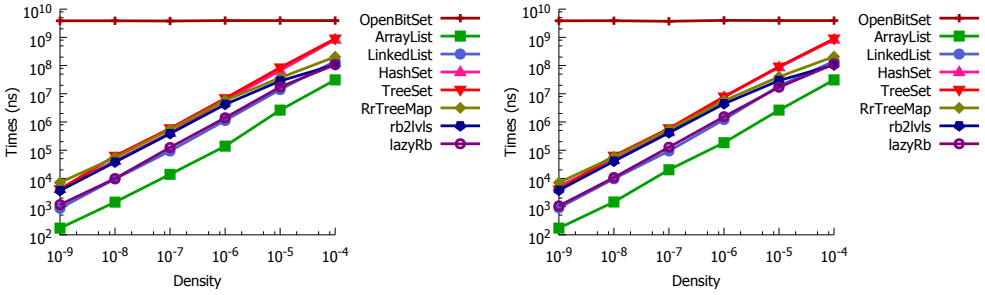
(a) Compression : distribution uniforme

(b) Intersection : distribution uniforme



(c) Intersection : distribution Zipf

(d) Insertion : distribution uniforme



(e) Union : distribution uniforme

(f) Union : distribution Zipf

FIG. 1: Compression et temps d'exécution

mémoire principale, de *RoaringTreeMap* adoptés sur le plus haut niveau.

Les figures 1e et 1f montrent les temps moyens pour calculer l'union de deux bitmaps avec chacune des structures de données évaluées dans ces essais. Tout comme pour l'intersection, dans le cas d'*OpenBitSet* et des collections Java, on commence par créer une copie du premier ensemble en entrée, puis on calcule une union en place entre la copie obtenue et le deuxième ensemble en entrée. La méthode *addAll* a été utilisée pour calculer cette dernière union dans les cas des collections Java, et la méthode *or* est utilisée pour *OpenBitSet*. Les résultats obtenus sont presque identiques sur les deux types de distributions de données testées. Le temps d'exécution d'une union avec chacune des trois nouvelles techniques de compression bitmap est essentiellement dominé par les temps passés à copier les entrées de haut niveau et des conteneurs depuis les bitmaps introduits en entrée vers le bitmap résultant. Étant donné que *LazyRoaring* applique une stratégie *copy-on-write* qui élimine plusieurs opérations de copie d'objets lors des calculs, cette technique a montré les meilleurs résultats parmi les trois formats proposés sur les deux distributions de données, en étant jusqu'à ≈ 6 fois plus rapide que *RoaringTreeMap* et ≈ 3 fois plus performante que *RoaringTwoLevels*. Ce dernier devance jusqu'à ≈ 2 fois *RoaringTreeMap* du fait qu'il n'utilise qu'un seul niveau d'indexation au-dessus des conteneurs, lui permettant ainsi de réduire le nombre des opérations d'allocation de nouveaux espaces mémoire utiles pour copier les hauts niveaux de la structure.

Comparé à *OpenBitSet*, les trois nouvelles techniques de compression bitmap affichent de remarquables performances, allant de ≈ 3 millions de fois à ≈ 34 fois plus vite. La contre-performance d'*OpenBitSet* est essentiellement causée par le taux excessif d'allocation de nouveaux espaces mémoires. Les deux structures de données *ArrayList* et *LinkedList* ont la même complexité linéaire de $\Theta(n_1 + n_2)$ pour calculer une union entre deux ensembles d'entiers de tailles n_1 et n_2. Toutefois, dans ces tests, la structure *ArrayList* est entre ≈ 5 à ≈ 8 fois plus rapide que la *LinkedList* sur toutes les densités. Ces deux structures de données prennent moins de temps que les trois nouveaux modèles de compression bitmap sur presque toutes les densités, du fait qu'elles n'effectuent pas de travail supplémentaire pour préserver l'ordre des éléments dans l'ensemble résultant. Cependant, les deux modèles de compression bitmap *RoaringTwoLevels* et *LazyRoaring* dépassent de peu la *LinkedList*, de $\approx 1,32$ fois pour la première et de $\approx 1,26$ fois pour la seconde, sur les plus fortes densités ($d = 10^{-4}$).

Les deux collections Java *HashSet* et *TreeSet* se suivent de près, mais la *HashSet* affiche de meilleures performances que la *TreeSet* grâce à ses insertions qui se font chacune en un temps constant, contrairement au temps logarithmique requis pour un *TreeSet*. De ce fait, une union entre deux ensembles de tailles n_1 et n_2, respectivement, se fait en un temps de $\Theta(n_1 + n_2)$ avec la structure *HashSet*, et en $O(n_1 + n_2 \log(n_1))$ avec la structure *TreeSet*. Cependant, ces deux dernières structures de données consomment plus de temps qu'une *ArrayList* ou une *LinkedList* pour opérer une union entre deux ensembles d'entiers.

La figure 1d montre le temps moyen que prend chaque structure de données pour insérer un nouvel entier positif e à un ensemble d'entiers S, telle que $\forall x \in S : e > x$. Après avoir aléatoirement généré un ensemble de N entiers distincts, on calcule le temps moyen que consomme chaque structure pour insérer chacun des N éléments. Les résultats sur les deux types de distributions sont semblables.

Parmi les trois nouveaux modèles, *RoaringTwoLevels* affiche les meilleures performances, en étant jusqu'à ≈ 31 fois et jusqu'à ≈ 11 fois plus rapide que *LazyRoaring* et *RoaringTreeMap*, respectivement. Cela s'explique par le fait que cette structure n'exécute une recherche

binaire que sur un seul niveau d'indexation pour trouver le conteneur dans lequel insérer les bits de poids faible de l'entier ou pour en créer un nouveau conteneur, contrairement aux deux autres structures de données qui nécessitent de faire une recherche binaire sur deux niveaux d'indexation avant d'atteindre le niveau conteneur. Sur les plus faibles densités, *LazyRoaring* et un peu plus rapide que *RoaringTreeMap*, car beaucoup d'instanciations de nouveaux objets sont faites sur le plus haut niveau des deux structures, où *RoaringTreeMap* recense un peu plus d'objets. Tandis que sur les fortes densités, les allocations sont généralement effectuées sur le deuxième niveau des deux structures, dans lequel *LazyRoaring* possède un peu plus d'objets à créer que *RoaringTreeMap*, ce qui le pousse à être plus lent sur ces densités.

Bien qu'*OpenBitSet* effectue une insertion en temps constants, cette structure est jusqu'à 14 millions de fois plus lente que les trois nouvelles techniques sur les faibles densités. Cela revient au grand taux d'allocation de nouveaux espaces mémoires qu'effectue cette structure pour peupler des bitmaps peu denses. Plus les densités augmentent, plus les opérations d'allocations diminuent et les temps d'*OpenBitSet* s'améliorent.

Avec des insertions en temps de $O(1)$ pour la *LinkedList* et en temps amorti constant pour la *ArrayList*, qui n'allouent pas d'importantes quantités d'espaces mémoires, ces deux structures affichent les meilleures performances sur ces essais. Bien que la collection Java *HashSet* effectue des insertions en temps constants également, elle reste néanmoins plus lente que les deux structures de données précédentes. La *TreeSet* affiche des temps de réponse un peu plus lents que ceux de la *HashSet* en raison du nombre de comparaisons à faire dans l'arbre avant de trouver la position à laquelle insérer le nouvel entier, qui est de l'ordre de $O(\log n)$ sur un arbre à n nœuds.

La figure 1a rapporte le nombre moyen de bits utilisés par chaque structure de données pour représenter un entier de 64 bits sur une distribution de données uniforme. Des performances équivalentes ont été observées sur une distribution Zipf. Les trois modèles de compression bitmap introduits affichent des performances similaires sur toutes les densités, variant de $\approx 0,3$ bit/entier à ≈ 5225 bits/entier. Sur les faibles densités, la structure *OpenBitSet* est jusqu'à ≈ 300 millions de fois plus volumineuse que les trois nouvelles techniques de compression bitmap pour représenter un entier de 64 bits. Cela revient à l'absence de méthode de compression de bits à 0 au sein d'*OpenBitSet*, ce qui le pousse à devoir allouer un tableau environnant les 6 GB d'espace mémoire pour représenter un entier de 64 bits de valeur proche à max. Cependant, plus les densités augmentent plus les performances d'*OpenBitset* s'améliorent, jusqu'à devenir plus compact que les trois nouveaux modèles de compression bitmap sur les plus fortes densités. Les collections Java affichent des performances stables sur toutes les densités étudiées. Mais, également pour des raisons d'absence de compression, ces structures consomment jusqu'à ≈ 1800 fois plus d'espace mémoire comparé aux trois techniques de compression bitmap sur les faibles densités.

Les meilleurs résultats pour les trois nouveaux modèles de compression bitmap ont été obtenus sur des bitmaps de faibles densités. En augmentant les densités, les surcoûts imposés par les index de haut niveau des trois techniques prennent de plus en plus d'ampleur provoquant la croissance de la quantité d'espace mémoire allouée par chaque modèle jusqu'à dépasser, après la densité 10^{-6}, celles requises par le reste des structures de données (consommant jusqu'à ≈ 22 fois plus d'espace que les collections Java à la densité 10^{-4}).

4 Conclusion

Ce travail a introduit trois nouveaux modèles d'index bitmap compressés supportant jusqu'à 2^{64} entrées. Des expériences sur des données synthétiques générées avec deux types de distributions, uniforme et Zipf, ont montré des résultats allant jusqu'à ≈ 6 millions de fois et jusqu'à ≈ 63 milles fois plus vite que la solution *OpenBitset*, adoptée au sein du moteur de recherche Apache Lucene, et de collections Java, respectivement, lors de calculs d'intersections entre bitmaps. Aussi, des performances de près de ≈ 3 millions de fois plus efficaces que celles d'*OpenBitSet* ont été observées lors d'essais évaluant les temps d'exécution d'opérations d'unions entre bitmaps. *OpenBitSet* a été également jusqu'à ≈ 14 millions de fois plus lent pour insérer un entier généré aléatoirement par rapport aux trois solutions proposées. Ces trois dernières ont été ≈ 300 millions de fois et ≈ 1800 fois plus compactes en matière d'occupation d'espace mémoire comparé à *OpenBitSet* et les collections Java, respectivement.

Les trois techniques proposées ont montré une consommation d'espace mémoire et des temps pour calculer l'intersection de deux bitmaps assez similaires sur toutes les densités testées. Avec son format simple adoptant un unique niveau d'indexation, *RoaringTwoLevels* a montré les meilleures performances en ce qui s'agit de l'insertion de nouveaux entiers de 64 bits ordonnés par ordre croissant. Plus précisément, *RoaringTwoLevels* a inséré jusqu'à ≈ 31 fois et jusqu'à ≈ 11 fois plus rapidement des entiers par rapport à *LazyRoaring* et *RoaringTreeMap*, respectivement. Avec sa stratégie *copy-on-write*, qui élimine des opérations de copies d'objets durant les calculs d'unions, *LazyRoaring* a été le plus efficace parmi les trois modèles, en étant jusqu'à ≈ 6 fois et jusqu'à ≈ 3 fois plus performant que *RoaringTreeMap* et *RoaringTwoLevels*, respectivement. En revanche, *RoaringTreeMap* reste la technique la plus simple à mettre en oœuvre.

Pour les travaux futurs, bien que peut rencontrées dans la réalité, nous envisageons d'étudier les performances des trois nouveaux modèles de compression bitmap sur des densités très fortes, non abordées lors des essais précédents. Comme il est envisagé également de réaliser des essais sur de gros ensembles de données réelles (50 Téraoctets et plus) tirées du *Star Schema Benchmark* (O'Neil et al., 2009).

Références

Apache (2010). OpenBitSet. `https://lucene.apache.org/core/3_0_3/api/core/org/apache/lucene/util/OpenBitSet.html`.

Apache (2012). Apache Lucene. `http://lucene.apache.org/core/`.

Bentley, J. L. et A. C. Yao (1976). An almost optimal algorithm for unbounded searching. *Information Processing Letters 5*(3), 82 – 87.

Chambi, S. (2015a). Benchmarking space consumptions for 64–bit bitmap compression schemes. `https://bitbucket.org/samytto/datastructures64-bitints/src/432c0bce7ca3be7ff457e8a7cd7e4cf2e48a69c0/src/Main.java?at=master&fileviewer=file-view-default`.

Chambi, S. (2015b). Benchmarking time measurements for 64–bit bitmap compression schemes with JMH. `https://bitbucket.org/samytto/datastructures64-`

bitints/src/432c0bce7ca3/test/src/main/java/microbenchmarks/?at=master.

Chambi, S. (2015c). LazyRaoring's scheme code. https://bitbucket.org/samytto/lazyroaring64-bits.

Chambi, S. (2015d). RaoringTreeMap's scheme code. https://bitbucket.org/samytto/roaringtreemap/overview.

Chambi, S. (2015e). RaoringTwoLevels' scheme code. https://bitbucket.org/samytto/roaring64bits2levels.

Chambi, S., D. Lemire, R. Godin, et O. Kaser (2014). Roaring bitmap : un nouveau modèle de compression bitmap. In *10e journèes francophones sur les Entrepôts de Donnèes et l'Analyse en Ligne (EDA'14)*, Volume 27(2), Vichy, France, pp. 37–50. RNTI.

Chambi, S., D. Lemire, K. Owen, et R. Godin (2015). Better bitmap performance with Roaring bitmaps. *Software Practice and Experience (SPE)*.

Colantonio, A. (2010). ConciseSet code. http://ricerca.mat.uniroma3.it/users/colanton/publications.html.

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

Cormen, T. H., C. Stein, R. L. Rivest, et C. E. Leiserson (2001). *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

Lemire, D. et O. Kaser (2010). JavaEWAH. https://github.com/lemire/javaewah.

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

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.

Oracle (2015). Java Microbenchmark Harness. http://openjdk.java.net/projects/code-tools/jmh/.

Richard Benjamins, V. (2014). Big data: from hype to reality? In *The 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14)*, WIMS '14, New York, NY, USA, pp. 1–2. ACM.

Roaring's team (2014). Roaring bitmap. http://roaringbitmap.org/.

Wikipedia (2015). Copy-on-write. https://en.wikipedia.org/wiki/Copy-on-write.

Wu, K., E. Otoo, et A. Shoshani (2006). Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS) 31*(1), 1–38.

Zipf, G. (1949). Human behavior and the principle of least effort. *Addison–Wesley*.