UNIVERSITÉ DU QUÉBEC À MONTRÉAL

LES INDEX BITMAP COMPRESSÉS

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

SAMY CHAMBI

JANVIER 2017

REMERCIEMENTS

Je remercie mes chers parents qui m'ont toujours soutenu dans mes études, tant moralement que financièrement, et à qui je dédie ce doctorat.

Je tiens aussi à remercier Daniel Lemire, professeur au département Science et Technologie de la Téluq, qui m'a donné la chance de m'initier à la recherche et de réaliser ce travail. Je le remercie également pour son soutien financier durant tout le long de mon doctorat, pour son dévouement dans l'encadrement de mes travaux de recherche, et pour m'avoir toujours correctement orienté dans les différents choix que j'ai eu à prendre.

Je remercie aussi Robert Godin, professeur au département d'informatique de l'UQAM, qui a toujours été présent durant mon doctorat pour me donner de précieux conseils, des suggestions et des orientations lors de la réalisation de mes travaux de recherche et lors de la rédaction de cette thèse.

Je remercie Kamel Boukhalfa, professeur au département d'informatique à l'USTHB, pour ses conseils et encouragements.

Je remercie l'équipe Druid, qui ont gentillement accepté de collaborer avec moi pour intégrer *Roaring bitmap*, l'une des solutions de cette thèse, à leur SGBD Druid.

Je remercie tous les membres de ma famille et mes amis qui m'ont soutenu et encouragé durant ce travail. Notamment, mon petit frère Sofiane, ma sœur Dina, ma chère grand-mère tittis, mes oncles Kamel, khelifa, Youcef, Hacen, Mourad et Djamel, mes cousins Anis, Amine, Yassir, Riad, Mokrane, Younes, Kiki, Anis C. et Amine C., mes amis Younes S., Mahfoud, Imad et Nassim.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

SGBD : Systèmes de Gestion de Bases de Données

WAH : *Word Aligned Hybrid Code*

BBC : *Byte aligned Bitmap Code*

Concise : *Compressed 'n' Composable Integer Set*

PLWAH : *Position List Word Aligned Hybrid*

EWAH : *Enhanced Word Aligned Hybrid*

VAL : *Variable Aligned Length*

OLAP : *On-Line Analytical Processing*

SQL : *Structured Query Langage*

JMH : Java Microbenchmark Harness

% : dénote l'opération modulo

$\in$: dénote le symbole mathématique d'appartenance

[X, Y] : dénote un intervalle de valeurs dont la borne inférieure est X et la borne supérieure est Y

RÉSUMÉ

Les index bitmap sont très utilisés dans les moteurs de recherche et les bases de données pour accélérer les opérations de recherche. Leurs principaux avantages sont leur forme compacte et leur capacité à tirer profit du traitement parallèle de bits dans les CPU (*bit-level parallelism*). Lorsque employés sur des attributs de faibles cardinalités, l'efficacité des index bitmaps en matière d'occupation d'espace mémoire et de temps de traitements comparé aux autres types d'index classiques, tels que l'arbre-B, est largement reconnue dans la littérature. Cependant, plus la cardinalité d'un attribut s'élève plus la taille et les temps de traitements de son index bitmap augmentent jusqu'à consommer plus d'espaces mémoires que les données indexées et d'importants temps de calculs. Afin de maintenir l'efficacité de ces solutions d'indexation dans ces conditions-là, plusieurs chercheurs ont proposé des travaux dans le but de réduire la taille et améliorer les temps de traitements de bitmaps indexant des attributs de larges cardinalités. Les solutions proposées dans la littérature adressant cette problématique se divisent en trois catégories : le paquetage des bitmaps, l'encodage des bitmaps et la compression des bitmaps.

Les contributions proposées dans cette thèse se classent parmi la troisième catégorie. Après avoir constaté que la plupart des techniques de compression de bitmaps introduites ces 15 dernières années se basent sur le modèle de la solution *WAH*, qui combine une compression par plages de valeurs avec une représentation bitmap sous forme de chaînes de bits alignées par mots CPU, cette thèse propose la technique *Roaring bitmap*, qui adopte un nouveau modèle pour compresser les bitmaps. Cette méthode discrétise l'espace des entiers représentés par un bitmap en des partitions de taille fixe, puis applique sur chacune une forme de compression appropriée selon la densité du groupe d'entiers. Des expériences ont été conduites pour comparer les performances temps-espace du nouveau modèle avec ceux de deux autres solutions de compression bitmap parmi les plus connues dans la littérature : *WAH* et *Concise*. Les résultats ont montré que, sur des faibles densités, la nouvelle méthode ne consomme que $\approx 50\%$ de l'espace mémoire occupé par *Concise* et $\approx 25\%$ de celui de *WAH*. Aussi, *Roaring bitmap* a pu accélérer les temps de calculs d'opérations logiques par rapport aux deux autres techniques sur tous les tests effectués, en étant de 4 à 5 fois plus performant sur des données synthétiques, et jusqu'à ≈ 1100 fois plus rapide sur des données réelles.

La librairie de *Roaring bitmap* et celles des autres solutions adoptant le modèle *WAH* qui sont disponibles au grand public ne supportent que des bitmaps d'au plus 2^{32}

($\approx$ 4 milliards) entrées. Avec l'avènement du *Big Data*, le besoin d'indexer de très larges collections de données sur lesquelles de telles librairies se révèlent impraticables est souvent rencontré. Les ingénieurs du moteur de recherche Apache Lucene ont rencontré ce problème, et ont introduit la solution *OpenBitSet*, qui peut allouer des bitmaps avec jusqu'à $64 \times 2^{32} - 1$ entrées. Cependant, cette solution reste simple et n'applique aucune forme de compression sur les bitmaps. La présente thèse propose trois nouveaux modèles de compression bitmap basés sur le format de *Roaring bitmap* et qui peuvent indexer jusqu'à 2^{64} entrées. Des expériences sur des données synthétiques comparant les performances des trois nouveaux modèles avec la solution d'Apache Lucene, *OpenBitSet*, et d'autres collections Java du paquetage Java.Util : *ArrayList*, *LinkedList*, *HashSet* et *TreeSet*, ont montré qu'*OpenBitSet* et les collections Java consomment, respectivement, jusqu'à $\approx$ 300 millions de fois et $\approx$ 1800 fois plus d'espaces mémoire comparés aux trois nouveaux modèles. Ces derniers ont également calculé des intersections entre deux ensembles d'entiers, $\approx$ 6 millions de fois, $\approx$ 63 milles fois et $\approx$ 6 fois plus rapidement par rapport à *OpenBitSet*, aux deux collections *ArrayList* et *LinkedList*, et aux deux structures *HashSet* et *TreeSet*, respectivement. En évaluant les temps pour calculer l'union de deux ensembles d'entiers, les nouvelles méthodes ont été jusqu'à $\approx$ 3 millions de fois plus performantes qu'*OpenBitSet*. Aussi, cette dernière structure de données a été jusqu'à $\approx$ 14 millions de fois plus lente pour insérer un entier généré aléatoirement que les trois solutions proposées.

Afin de valider le format de la solution *Roaring bitmap* dans un SGBD réel, cette technique d'indexation a été intégrée au moteur OLAP *Druid*. Ce système se base essentiellement sur des index bitmap compressés avec la technique *Concise* pour accélérer les temps de réponse de requêtes OLAP effectuant des analyses détaillées sur les données *(drill-down)*. Des expériences sur des données réelles ont été réalisées pour évaluer les performances de *Roaring bitmap* et de *Concise* au sein du SGBD *Druid*. Les résultats ont montré que *Roaring bitmap* a amélioré de $\approx$ 2 fois les temps de réponse de requêtes d'agrégations et près de 5 fois les temps de traitements de requêtes de recherche comparé à la solution *Concise*.

MOTS-CLÉS : index bitmap, compression, performances, opérations logiques, structures de données.

CHAPITRE I

INTRODUCTION

Les entrepôts de données relationnels sont généralement conçus avec un schéma en étoile, composé d'une table de fait liée à des tables de dimensions par des clés étrangères. Une table de faits est une collection de données qui sont généralement de valeurs numériques et qui sont gardées pour des besoins d'analyse. Ces données collectées dans une table de faits sont analysées selon des axes d'analyse représentés par les tables de dimensions qui lui sont liées.

Ces ensembles de données sont, la plupart du temps, interrogés avec des requêtes OLAP (*On-line Analytical Processing*) multidimensionnelles et complexes, de type ad hoc nécessitant d'effectuer des jointures coûteuses entre la table de fait et plusieurs dimensions. La majorité des techniques d'indexation comme les index multidimensionnels (les arbres-KD, les arbres-R, etc.) et les variantes des arbres-B (les arbres-B^+, les arbres-B^*, etc.), s'avèrent peu efficaces pour ce genre d'opérations et souffrent de ce qui est communément appelé : le fléau de la dimensionnalité (*the curse of dimmensionality*) (Bertchold *et al.*, 1998). En effet, l'une des solutions envisageables pour ce problème est de créer un index multidimensionnel sur chaque combinaison possible d'attributs. Cependant, cette approche reste impraticable car elle requiert la création d'un nombre exponentiel d'index qui nécessiterait, dans le cas de gros volumes de données, un espace de stockage ingérable en pratique. En plus, les index multidimension-

nels sont connus de n'être efficaces que sur des environnements ayant un nombre limité de dimensions (ex. : < 15) et avec des requêtes qui utilisent presque tous les attributs indexés, cas rarement rencontrés dans un contexte OLAP.

Par contre, des travaux (Stockinger et Wu, 2008; Wu *et al.*, 2006) ont montré que les index bitmap brisent cette malédiction liée à la dimensionnalité et peuvent même fournir des temps de recherche optimaux dans certaines situations. Effectivement, Wu *et al.* (2006) rapportent que des index bitmap compressés permettent de répondre optimalement à des requêtes d'intervalles à une dimension, en offrant des temps de réponse de complexité linéaire par rapport au nombre des éléments en sortie. Étant donné que le résultat obtenu sur une dimension se présente sous la forme d'un bitmap, et que plusieurs bitmaps peuvent être facilement et efficacement combinés ensembles, ces index se révèlent aussi efficaces pour des requêtes d'intervalles multidimensionnelles.

Les moteurs de recherche, tels que Lucene (Apache, 2012), utilisent souvent des index inversés pour représenter les termes apparaissant dans les documents indexés. Chaque terme d'un index inversé est associé à une liste d'entiers, dont les éléments correspondent aux identifiants des documents dans lesquels figure ledit terme. L'exécution d'une requête de recherche ayant pour but d'extraire les documents renfermant des termes spécifiques nécessiterait le calcul du résultat de l'intersection des listes d'entiers des termes recherchés. Sans une représentation efficace de ces listes d'entiers, le temps d'exécution d'une telle opération serait, généralement, d'une lenteur intolérable sur de larges listes. En représentant ces listes d'entiers par des bitmaps, des solutions (Culpepper et Moffat, 2010) ont pu répondre efficacement à de telles requêtes en exécutant des opérations ET logiques entre les bitmaps des termes recherchés. Les temps de réponse obtenus étaient beaucoup plus rapides comparés à d'autres types de représentations de listes d'entiers.

Tel que constaté, les index bitmap sont très utilisés dans les entrepôts de données et

les moteurs de recherche. Leur principal avantage réside en leur forme compacte et en leur capacité à tirer profit du traitement parallèle de bits (*bit-level parallelism*) dans les CPU (*Central Processing Unit*), permettant à ces derniers d'exécuter rapidement des traitements sur les bitmaps, ce qui finit, en général, par réduire significativement les temps de réponse des requêtes.

Pour indexer un ensemble de valeurs, un index bitmap est créé sur chaque valeur distincte de l'ensemble. Un bitmap ainsi créé sera représenté par une suite de bits (0 et 1) d'une longueur égale au nombre des éléments de l'ensemble, et indiquera les positions dans l'ensemble auxquelles apparaît la valeur distincte indexée, ceci en plaçant des bits à 1 aux positions correspondantes dans le bitmap.

L'efficacité des index bitmap par rapport aux autres types d'index est largement reconnue dans la littérature lorsqu'il s'agit d'indexer des ensembles de données de faibles cardinalités (Sharma, 2005), caractérisés par un petit nombre de valeurs distinctes, tel l'attribut genre qui est une énumération de deux valeurs distinctes : masculin et féminin. Cependant, sur des attributs de fortes cardinalités, comme un attribut représentant les différents types de produits étalés dans un supermarché pouvant atteindre des milliers de valeurs distinctes, les performances des index bitmap se dégradent considérablement. L'intérêt que portent les chercheurs à ces index a fait qu'un grand nombre de contributions ont été apportées dans la littérature pour considérer ce problème. Pour commencer, cette thèse fait état des principaux types de travaux réalisés en ce sens. Ces solutions se divisent en trois grandes catégories : l'encodage des bitmaps, le paquetage des bitmaps et la compression des bitmaps. Les deux premières solutions visent à réduire le nombre de bitmaps créés pour indexer un ensemble de données, mais engendrent des calculs un peu plus complexes et coûteux pour effectuer des traitements sur les bitmaps. La première consiste à appliquer des techniques d'encodage sur les bitmaps : l'encodage par rang (Chan et Ioannidis, 1998a), l'encodage par intervalle (Chan et Ioannidis, 1999), etc. Quant à la deuxième, elle repose sur l'idée d'indexer plusieurs

valeurs distinctes avec un seul bitmap. Une présentation détaillée de chacune de ces deux solutions avec une illustration des principales contributions apportées en ce sens dans la littérature, qui met le point sur les avantages et les inconvénients de chaque contribution est présentée dans cette thèse.

Le troisième type de contributions applique une méthode de compression sur chaque bitmap individuel. Ces approches peuvent aussi être combinées avec les deux précédents types de solutions. Différentes méthodes de compression ont été appliquées à ce jour sur les bitmaps. L'avantage commun entre les solutions de ce type, est qu'elles réduisent efficacement l'espace mémoire consommé par un bitmap non compressé. Toutefois, pour effectuer des opérations sur un bitmap compressé avec les premières techniques de compression bitmap introduites dans la littérature, une entière décompression de celui-ci devait être effectuée avant de pouvoir l'utiliser. Cette opération de décompression engendre un temps de traitement énorme, ce qui a longtemps laissé les experts en administration d'entrepôts de données et moteurs de recherche réticents à leur adoption. Ce n'est qu'en 1995, qu'Antoshenkov, un chercheur d'IBM (*International Business Machines*), a introduit la solution de compression bitmap *BBC* (*Byte aligned Bitmap Code*) (Antoshenkov, 1995), qui compresse un bitmap en appliquant une compression par plage de valeurs (*Run Length Encoding* ou RLE) sur une suite de bits alignés par blocs d'octets. Cette solution a non seulement permis de réduire significativement l'espace mémoire occupé par les bitmaps, mais avec sa capacité à effectuer des traitements sur des bitmaps dans leurs états compressés (sans une décompression au préalable), elle a réussi aussi à accélérer drastiquement les temps d'exécution d'opérations effectuées sur des bitmaps.

Suite à l'introduction de *BBC*, des chercheurs (Wu *et al.*, 2006) ont étendu l'idée d'Antoshenkov afin d'arriver à une solution beaucoup plus compatible à l'architecture des CPU, en appliquant une compression par plage de valeurs sur des suites de bits alignés par mots CPU, plutôt que par blocs d'octets. Le résultat a donné la méthode de com-

pression bitmap *WAH* (*Word Aligned Hybrid bitmap compression*) qui a montré une amélioration impressionnante des temps de calcul d'opérations logiques entre bitmaps, mais avec une consommation mémoire un peu plus grande comparativement à *BBC*. Plusieurs travaux ayant pour but d'améliorer les performances du modèle *WAH* ont suivi : *Concise* (Colantonio et Di Pietro, 2010), *PLWAH* (Deliège et Pedersen, 2010), *EWAH* (Lemire *et al.*, 2010), etc. Cette thèse fera part d'une description détaillée des importantes contributions parmi ces techniques de compression bitmap.

1.1 Les contributions

Les contributions de cette thèse s'intéresse au troisième type de solutions qui est la compression des index bitmap. Ayant observé que la plupart des techniques de compression bitmap proposées ces 15 dernières années se basent sur le modèle de *WAH* pour compresser les bitmaps, nous avons introduit un nouveau modèle de compression bitmap, appelé *Roaring bitmap*, qui représente un bitmap comme une liste d'entiers triés par ordre croissant, où chaque élément correspond à la position d'un bit à 1 dans le bitmap. Cette technique discrétise l'espace des entiers représentant un bitmap en des partitions de taille fixe. Les entiers tombant dans une partition sont organisés dans un bitmap ou dans un tableau selon la densité du groupe d'entiers. Des expériences comparant les performances de *Roaring bitmap* avec celles d'autres solutions de compression bitmap : *WAH* et *Concise* (Colantonio et Di Pietro, 2010), ont été menées sur des données synthétiques et réelles. Les résultats ont révélé de remarquables performances en matière de taux de compression et temps de traitements au profit de *Roaring bitmap*.

La librairie *Roaring bitmap* et le reste des librairies de compression bitmap adoptant le modèle *WAH* ont été développées de sorte à ne supporter que des bitmaps ne dépassant pas les 2^{32} ($\approx$ 4 milliards) d'entrées. Dans l'ère actuel des méga données (*Big Data*), les experts en recherche d'information se trouvent souvent face à l'obligation d'indexer des collections de données ayant un nombre d'entrées qui dépasse de beaucoup le seuil

supporté par les librairies existantes. Les ingénieurs du moteur de recherche Apache Lucene (Apache, 2012), par exemple, ont fait face à ce problème et ont introduit une librairie d'index bitmap *OpenBitSet* (Apache, 2010) écrite en Java et qui supporte jusqu'à $64 \times 2^{32} - 1$ entrées. En s'inspirant du format *Roaring bitmap*, trois nouveaux modèles de compression bitmap supportant jusqu'à 2^{64} entrées ont été introduits : *RoaringTreeMap*, *RoaringTwoLevels* et *LazyRoaring*. Des expériences sur des données synthétiques ont été mises en œuvre pour comparer les performances de ces trois nouveaux modèles avec celles d'*OpenBitSet* et d'autres collections Java figurant dans le paquetage Java.Util : *ArrayList*, *LinkedList*, *HashSet* et *TreeSet*. Les résultats ont montré que les trois nouveaux modèles proposés consomment beaucoup moins d'espace que les autres structures évaluées et permettent de réaliser des traitements sur des bitmaps compressés en des temps très rapides comparé au reste des solutions.

Après avoir introduit la solution *Roaring bitmap*, qui a montré de remarquables résultats en matière de compression et temps de traitements, nous nous sommes intéressé à l'évaluation des performances de *Roaring bitmap* au sein d'un système de gestion de bases de données (SGBD) réel. Le moteur OLAP *Druid* (Yang *et al.*, 2014) a été choisi pour ce projet. Ce SGBD fait un grand usage d'index bitmap compressés pour accélérer les requêtes OLAP effectuant des analyses détaillées (*drill-down*) sur les données. Auparavant, le système utilisait la solution de compression bitmap *Concise* (Colantonio et Di Pietro, 2010). Suite aux intéressantes performances observées avec *Roaring bitmap* lorsque comparé à *Concise*, une collaboration avec les développeurs de *Druid* a suivi pour intégrer *Roaring bitmap* comme une solution de compression bitmap à part entière au sein de *Druid*. Des expériences sur des données réelles du banc d'essai TPC-H (Council, 2014) ont été produites, où *Roaring bitmap* a affiché des améliorations significatives aux temps de réponse des requêtes d'analyses faisant recours aux bitmaps comparativement à la solution existante.

1.2 Organisation de la thèse

La présente thèse est organisée comme suit. Le Chapitre II présente un état de l'art sur les importantes solutions utilisées à ce jour pour réduire la taille et améliorer les temps de traitement des index bitmap. Les trois types de solutions utilisées en ce sens et les principales contributions proposées pour chacun de ces types y sont présentés : le paquetage, l'encodage et la compression des bitmaps. Le Chapitre III introduit le modèle de compression bitmap, *Roaring bitmap*, et présente les bancs d'essais réalisés pour évaluer les performances de cette technique avec celles d'autres méthodes parmi les plus connues dans la littérature. Vient ensuite le Chapitre IV, où sont proposés les trois nouveaux modèles de compression bitmap supportant des entiers de 64 bits. Des bancs d'essai comparant les performances de ces trois nouveaux formats avec celles d'autres solutions utilisées dans un tel contexte y sont présentés également. Le projet d'intégration de *Roaring bitmap* à *Druid* s'en suit au Chapitre V, où une présentation du moteur OLAP *Druid* et des expériences réalisées pour évaluer les bénéfices de *Roaring bitmap* au sein du moteur OLAP *Druid* y sont présentés. Cette thèse se termine au Chapitre VI avec une conclusion générale et les possibles travaux à entreprendre dans un futur proche.

CHAPITRE II

ÉTAT DE L'ART

2.1 Les index bitmap

Dans un ensemble de données, un index bitmap peut être créé sur chaque attribut candidat à l'indexation. L'attribut X de la Figure 2.1 possède huit valeurs distinctes. Un bitmap, qui est un tableau de bits d'une longueur égale au nombre d'entrées de l'attribut X, sera créé pour chaque valeur distincte de X. Chacune de ces valeurs distinctes représentera la clé de son bitmap correspondant. L'index bitmap construit sur X sera alors composé de 8 bitmaps : $E_0, E_1, ..., E_7$. Le $i^{\text{ème}}$ bit d'un bitmap est mis à 1 si la valeur de la $i^{\text{ème}}$ entrée de X est égale à la clé du bitmap, sinon le bit est mis à 0. Ainsi, un index bitmap peut être représenté sous la forme d'une matrice binaire, dans laquelle chaque colonne correspond à un bitmap. L'index bitmap d'un attribut à n entrées et L valeurs distinctes consommera alors un espace de stockage de $\approx n \times L$ bits.

Sur des attributs de faibles cardinalités, l'efficacité des index bitmap est largement reconnue dans la littérature (Wu et Yu, 1998; Sharma, 2005; Wu *et al.*, 2006). Cependant, le volume des index bitmap tend à s'élargir sur des attributs de fortes cardinalités jusqu'à nécessiter plus d'espace de stockage que celui réservé aux données indexées. Dans de tels cas, l'utilisation d'un index bitmap se traduirait en un nombre important d'opérations d'E/S nécessitant un temps d'exécution beaucoup plus lent comparé à ce que pourraient offrir d'autres types d'index, tels que les arbres-B (Sharma, 2005).

Index bitmap

X	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	0
4	0	0	0	0	1	0	0	0
5	0	0	0	0	0	1	0	0

Figure 2.1: L'index bitmap de l'attribut X.

Afin de préserver l'efficacité des index bitmap dans de tels contextes, plusieurs contributions scientifiques ont été introduites dans le but de réduire la taille des index bitmap et d'améliorer leurs temps de traitement. Ces propositions peuvent être regroupées en trois grandes catégories : (1) l'empaquetage, (2) l'encodage et (3) la compression des index bitmap. Les sections qui suivent expliquent plus en détail ces trois types de solutions.

2.2 Techniques de paquetage des bitmaps (*binning*)

La stratégie de paquetage des bitmaps (*binning*) permet de réduire l'espace mémoire occupé par un index bitmap et améliorer ainsi ses temps de traitements. Toutefois, l'adoption de cette stratégie peut engendrer un problème majeur qui nuit aux performances des index bitmap. Une description de ce problème connu sous le nom du problème de la vérification candidate est donnée par la suite. En dernier, quelques contributions efficaces introduites dans la littérature pour réduire les coûts liés au problème de la vérification candidate sont discutées.

2.2.1 Le paquetage des bitmaps (*binning*)

La stratégie de *binning* est adoptée dans les bases de données pour diminuer l'espace de stockage occupé par un index bitmap en réduisant le nombre de ses bitmaps à une certaine constante bien établie. Pour cela, cette stratégie crée un bitmap, non pas sur chaque valeur distincte de l'attribut indexé, mais sur des intervalles de valeurs, appelées *bins*. Les deux méthodes de *binning* les plus utilisées dans les SGBD sont : le *binning* à largeur égale (*equi-width binning*) et le *binning* à profondeur égale (*equi-depth binning*) (Rotem *et al.*, 2005). La première méthode adopte des *bins* ayant approximativement des intervalles de même largeur, et la deuxième technique essaie de répartir les entrées de l'attribut indexé le plus équitablement possible entre les différents *bins*. La Figure 2.2 illustre un exemple d'un index bitmap avec un *equi-width binning*. L'espace des valeurs de l'attribut indexé Z est $[0 - 1\,000)$. Cinq *bins* sont créés au total chacun couvrant 200 valeurs possibles de l'attribut Z. Un bitmap est alors créé sur chaque *bin*. Le $i^{\text{ème}}$ bit d'un bitmap est à 1 si la valeur de la $i^{\text{ème}}$ entrée de l'attribut Z tombe dans l'intervalle du bin correspondant au bitmap, sinon le bit est mis à 0.

Une des plus anciennes stratégies de *binning* introduites dans la littérature est le *Range-based binning* proposée par Wu et Yu (1998). Cette technique partitionne l'espace des

Z	[0 – 200)	[200 – 400)	[400 – 600)	[600 – 800)	[800 – 1000)
0	1	0	0	0	0
80	1	0	0	0	0
405	0	0	1	0	0
625	0	0	0	1	0
50	1	0	0	0	0
200	0	1	0	0	0
318	0	1	0	0	0
999	0	0	0	0	1
738	0	0	0	1	0
862	0	0	0	0	1
920	0	0	0	0	1

Figure 2.2: Index bitmap avec *binning* créé sur l'attribut Z

valeurs de l'attribut en plusieurs intervalles (*bins*) et crée un bitmap pour chaque *bin*. Vu que les données de l'attribut peuvent être distribuées entre les différents *bins* de façon non équitable, entraînant ainsi des temps de recherche non uniformes pour des requêtes différentes, les auteurs ont introduit une approche *DBEC* (*Dynamic Bucket Expansion and Contraction*) qui divise les partitions assez peuplées et fusionne les *bins* moins peuplés, afin de maintenir des temps de recherche presque uniformes entre des requêtes différentes.

2.2.2 Problème de la vérification candidate

Bien que la stratégie de *binning* réduise le nombre des bitmaps et diminue l'espace consommé par l'index, elle engendre souvent des accès coûteux aux données (généralement stockées sur le disque) lors du traitement d'une requête de recherche. En effet, pour répondre à une requête ayant comme prédicat 300 <= Z <= 700 avec l'index bitmap de la Figure 2.2, les bitmaps correspondant aux intervalles $[200-400)$, $[400-600)$ et $[600-800)$ seront scannés, car ils indexent des données inclues dans l'intervalle de valeurs spécifié par la requête. Les enregistrements indexés par le bitmap de la partition

$[400 - 600)$ respectent la condition de la requête, tandis que les deux autres bitmaps (en gris) indexent des données non spécifiées par la requête (200 et 738, appelées les fausses valeurs). Une étape de vérification supplémentaire doit alors être faite en accédant aux valeurs indexées par ces deux bitmaps, afin de vérifier et de ne retourner que les entrées respectant la contrainte de la requête. Cette étape de vérification est appelée, phase de vérification candidate (*candidate check*), et le temps qu'elle consomme domine généralement le temps d'exécution d'une telle requête de recherche (Rotem *et al.*, 2005).

2.2.3 Solutions réduisant le coût des phases de vérifications candidates

Plusieurs contributions ont été introduites dans la littérature dans le but de minimiser le temps consommé par les phases de vérification candidate dans une charge de requêtes. Le temps d'exécution d'une vérification candidate sur un bitmap dépend du nombre de pages disques chargées en mémoire lors de cette phase. Plus un bitmap renferme des bits à 1 (bits positifs [1]) indexant des valeurs pouvant être réparties sur plusieurs pages disque, plus une vérification candidate sur untel bitmap augmente les coûts des opérations d'E/S.

Koudas (2000) a introduit un algorithme de programmation dynamique qui prend en entrée une charge de requêtes à prédicat d'égalité, un attribut e, une constante K représentant le nombre maximal de bitmaps à créer, et retourne un ensemble d'au plus K *bins* optimisant le coût des vérifications candidates de la charge de requêtes sur l'attribut e. Cette solution parcourt exhaustivement tout l'espace de recherche en estimant le nombre des fausses valeurs engendrées pour chaque configuration de *bins* possible. L'algorithme proposé s'exécute en un temps de $O(n^2K)$, où n représente le nombre des valeurs distinctes de l'attribut.

---

1. Pour simplifier la lecture du reste du texte de cette thèse, l'expression "bit positif" fera référence à un bit à 1, et l'expression "bit négatif" indiquera un bit à 0.

Rotem *et al.* (2006b) ont aussi introduit *multiOptBin*, un algorithme à base de programmation dynamique ayant une meilleure complexité temporelle comparée à l'algorithme de Koudas (2000) et qui permet d'établir une configuration de *bins* optimale pour un ensemble d'attributs et de requêtes multidimensionnelles. Les auteurs ont montré qu'il était possible d'utiliser des statistiques sur les intervalles des requêtes à la place des valeurs des attributs pour construire des *bins* optimaux. Leur méthode prend en entrée un ensemble de requêtes multidimensionnelles à prédicats d'intervalles, un ensemble de données avec des attributs et une contrainte k indiquant le nombre de bitmaps à créer, puis construit une configuration de *bins* sur les attributs de l'ensemble de données qui optimise le coût des vérifications candidates de la charge de requêtes. L'algorithme en question s'exécute en un temps de $O(tkr^2)$ où t est égal au nombre d'attributs dans l'ensemble de données, et r le nombre d'intervalles dans la charge de requêtes. Les auteurs ont aussi constaté que l'ordre des dimensions lors de l'exécution des requêtes a un impact significatif sur le nombre d'E/S effectuées pendant les phases de vérifications candidates. En se basant sur la sélectivité des attributs, ils proposent une heuristique qui permet d'ordonner optimalement les dimensions pour l'exécution d'une requête.

Toutefois, *multiOptBin* alloue un même nombre de *bins* pour chaque attribut indexé, dans Rotem *et al.* (2006a) les auteurs ont amélioré *multiOptBin* en le rendant capable de déterminer le nombre optimal de *bins* pour chaque attribut. Des expériences sur des ensembles de données réelles (Rotem *et al.*, 2006a) ont confirmé le gain apporté par cette sophistication en réduisant de près de 30 % les coûts des E/S comparée à l'ancienne version de *multiOptBin*.

Les méthodes à base de programmation dynamique présentées plus haut ne sont adaptées que pour un type précis de requêtes (requêtes à prédicats d'égalité ou d'intervalle), et leurs temps d'exécution croient quadratiquement avec la taille du problème. Goyal et Sharma (2009) ont proposé une approche efficace pour tous types de requêtes (à prédicats d'égalité et d'intervalle). Les auteurs tirent avantage du fait que les requêtes ayant

des intervalles qui correspondent exactement aux intervalles des *bins* ne requièrent aucune vérification candidate. En se basant sur des statistiques obtenues de la charge de requêtes, cette solution détermine un ensemble de *bins* optimal réduisant le coût des phases de vérification candidate et ainsi le temps d'exécution de la charge de requêtes. Le principe est de créer des *bins* avec des intervalles qui correspondent exactement à ceux des requêtes fréquemment exécutées. Un seuil défini par l'administrateur permet de différencier entre les requêtes fréquentes et moins fréquentes. Cette démarche tolère que des *bins* soient entrelacés en représentant des valeurs communes, et si des trous apparaissent entre plusieurs *bins*, des *bins* additionnels sont créés. Ainsi, plusieurs combinaisons de *bins* pourraient être choisies pour répondre à une requête peu fréquente. Pour cela, les auteurs proposent un algorithme qui minimise le coût des vérifications candidates pour de telles requêtes, en faisant en sorte de sélectionner l'ensemble de *bins* ayant le nombre minimal de 1.

Le désavantage de la solution de Goyal et Sharma (2009) est qu'elle consomme plus d'espace mémoire comparée aux autres techniques de *binning*, causé par le fait qu'un nombre important de *bins* doit être créé. Mais en termes de temps de réponse aux requêtes, cette stratégie s'est révélée beaucoup plus performante qu'un *Range binning* et un *equi-width binning*. Aussi, les résultats des expériences ont montré que les temps de réponse augmentent et l'espace de stockage diminue avec la montée du seuil permettant de déterminer les requêtes fréquentes des moins fréquentes. Ainsi, un bon choix de ce seuil offrirait un compromis temps-espace intéressant pour l'utilisateur.

D'autres travaux se sont intéressés aux méthodes de regroupement (*clusterisation*) des valeurs des attributs indexés, en faisant en sorte que les valeurs qui tombent dans un *bin* soient stockées séquentiellement sur le disque. Une telle approche réduit considérablement le nombre des accès disque comparée aux cas où les valeurs d'un *bin* ne sont pas séquentiellement organisées sur le disque. Wu *et al.* (2008) ont introduit une telle méthode de regroupement dénommée *OrBic*. Les auteurs ont mené des expériences pour

comparer les performances de cette approche avec des index bitmap sans regroupement, ainsi qu'avec des index de projection sur des données réelles et synthétiques. Les résultats ont montré qu'un index bitmap avec *OrBic* consomme une quantité d'espace de stockage équivalente à celle d'un index bitmap sans *OrBic*, offre des temps de réponse $\approx$ 3 fois plus rapides qu'un index bitmap classique et pouvait aller jusqu'à $\approx$ 40 fois plus vite qu'un index de projection sur des données biaisées.

2.3 Techniques d'encodage des index bitmap

La technique d'encodage des index bitmap a été la plus étudiée parmi les trois grandes stratégies, elle consiste à encoder les index bitmap afin de réduire les temps de réponse en minimisant le nombre de bitmaps à lire pendant l'exécution de requêtes, ainsi que l'espace de stockage en diminuant le nombre de bitmaps à créer pour un attribut donné (Stockinger et Wu, 2008). Les solutions proposées dans la littérature pour l'encodage des index bitmap se divisent en deux catégories : (1) les techniques d'encodage basiques, et (2) les techniques d'encodage composées.

2.3.1 Solutions basiques d'encodage des index bitmap

Trois sortes d'encodages basiques existent dans la littérature. L'encodage par égalité représente le plus simple des encodages basiques et a été utilisé depuis l'introduction des index bitmap. Quant aux deux autres encodages basiques, ils ont été proposés par Chan et Ioannidis (1998a, 1999) et sont : l'encodage par rang et l'encodage par intervalle.

Un exemple d'un encodage par égalité est illustré avec l'index bitmap créé sur l'attribut X de la Figure 2.1. La clé de chaque bitmap correspond à une des valeurs distinctes de l'attribut. Si C est la cardinalité de l'attribut, alors l'index bitmap comptera au total C bitmaps. Ce type d'encodage reste le meilleur pour les requêtes avec prédicats d'égalité, comme « âge = 5 » ou « taille = 70 ». En effet, répondre à la requête « X = 5 » ne requiert que la lecture d'un seul bitmap E_5.

La Figure 2.3 montre un exemple d'un encodage par rang appliqué sur l'index bitmap de la Figure 2.1. Huit bitmaps seront créés au total, un pour chaque valeur distincte de l'attribut X. Les bits positifs d'un bitmap marquent les entrées de X dont la valeur est inférieure ou égale à la clé du bitmap. Dans le cas du bitmap R_2, un bit est égal à 1 s'il correspond à une entrée de l'attribut X ayant une valeur inférieure ou égale à 2, sinon le bit sera mis à 0. Ce deuxième type d'encodage s'avère efficace pour des requêtes

d'intervalles à sens unique, du genre « âge <= 4 ». En effet, pour répondre à une telle requête, seulement le bitmap R_4 dont la clé est égale à 4 sera scanné. Tandis qu'avec un encodage par égalité, un OU logique sera exécuté entre 5 bitmaps (R_0 à R_4). Même si l'on prend en compte l'optimisation qui consiste à calculer le complément du bitmap résultant lorsque plus de la moitié des bitmaps seront scannés, le nombre des bitmaps accédés dans le cas d'un encodage par égalité pour cette requête sera alors de 3 bitmaps (R_5 à R_7), ce qui est toujours plus coûteux qu'avec un encodage par rang.

D'une façon générale, une requête d'intervalle à sens unique nécessitera de scanner un seul bitmap dans le cas d'un encodage par rang, et jusqu'à $\lfloor C/2 \rfloor$ bitmaps avec un encodage par égalité, où C représente la cardinalité de l'attribut indexé.

Pour répondre à une requête portant sur un prédicat d'égalité en utilisant un index bitmap encodé par rang, le bitmap correspondant à la valeur recherchée sera fusionné avec le bitmap qui le précède (si existant) à l'aide d'un OU exclusif. Lorsque le $2^{\text{ème}}$ bitmap n'existe pas, cas où le bitmap associé à la valeur recherchée figure à la première position de l'index, alors le traitement du premier bitmap suffira pour trouver le résultat final. Par conséquent, deux bitmaps pourront être traités au pire cas avec un encodage par rang pour satisfaire une telle requête.

L'encodage par intervalle (Chan et Ioannidis, 1999) est le plus adapté parmi les trois encodages aux requêtes d'intervalles à deux sens, ex. : « 7 < âge < 30 ». Dans cet encodage, un bitmap représente, non pas une valeur distincte de l'attribut indexé, mais plutôt un *bin* de $\lfloor \frac{C}{2} \rfloor$ valeurs, où C est la cardinalité de l'attribut indexé. Le nombre de bitmaps obtenus avec cet encodage est de $\lceil \frac{C}{2} \rceil$. Ce qui veut dire que cette solution requiert deux fois moins de bitmaps comparée aux deux encodages précédents. En plus, cet encodage assure qu'au pire cas, une seule paire de bitmaps aura besoin d'être scannée pour répondre à une quelconque requête.

X	R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7
0	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
3	0	0	0	1	1	1	1	1
6	0	0	0	0	0	0	1	1
1	0	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1
3	0	0	0	1	1	1	1	1
7	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1	1
4	0	0	0	0	1	1	1	1
5	0	0	0	0	0	1	1	1

Figure 2.3: L'index bitmap de l'attribut X avec un encodage par rang

2.3.2 Techniques d'encodage composées

Diverses solutions introduites dans la littérature combinent un ou plusieurs des trois encodages décrits précédemment. Ces contributions peuvent être classées en deux grandes classes : les techniques d'encodage multicomposantes (*multi-component encoding*) et les techniques d'encodage multiniveaux (*multi-level encoding*).

Méthodes d'encodage multicomposantes

L'encodage binaire (Wong *et al.*, 1985) fait partie de la classe des méthodes d'encodage multicomposantes qui à ce jour offrent le plus petit nombre de bitmaps parmi toutes les solutions existantes. Cette approche est d'ailleurs adoptée par le bit-sliced index (O'Neil et Quass, 1997). Avec cet encodage, le nombre des bitmaps est réduit à $\lceil \log_2 C \rceil$ bitmaps, où C est la cardinalité de l'attribut indexé. Cependant, le point négatif de cette technique est qu'elle nécessite un accès à tous les bitmaps pour répondre à une requête d'intervalle ou d'égalité. Ce qui prend généralement beaucoup plus de temps comparé aux trois encodages basiques.

Plusieurs autres travaux (Bhan *et al.*, 2012; Stockinger et Wu, 2008) ont également étudié des modèles d'encodage multicomposantes qui construisent plusieurs index bitmap pour représenter les valeurs d'un attribut indexé. Chaque index bitmap fait l'objet d'une composante individuelle qui peut être encodée avec l'un des trois encodages basiques.

(Chan et Ioannidis, 1999) ont introduit un tel encodage multicomposantes. La Figure 2.4 illustre un exemple de cet encodage lorsque deux composantes sont adoptées sur un attribut dont les valeurs $\in [0, 999]$. Pour former deux composantes pour un attribut ayant des valeurs $\in [0, C-1]$, le nombre de bitmaps dans les deux composantes, respectivement b_1 et b_2, sont choisies de sorte que $b_1 \times b_2 \geq C$. Ainsi, chaque valeur v de l'attribut indexé pourrait être écrite de la sorte : $v = c_1 \times b_2 + c_2$, avec $c_1 = v \ / \ b_2$ et $c_2 = v \,\%\, b_2$. Dans l'exemple de la Figure 2.4, la première composante (C_1) a un nombre de bitmaps $b_1 = 25$ et la deuxième composante (C_2) en possède $b_2 = 40$ bitmaps ; ce qui représente plus de bitmaps comparé à un encodage binaire qui n'aura besoin que de $\lceil \log_2 \ 1\,000 \rceil = 10$ bitmaps dans ce cas, mais moins que ce que nécessiterait un simple encodage par rang ou par égalité. Pour encoder une valeur v quelconque de l'attribut indexé, la technique procède en deux étapes : la première étape sert à sélectionner le bitmap approprié de la première composante (C_1) en calculant la division entière $v \ / \ 40$. La deuxième étape calcule le modulo $v \ \% \ 40$ afin de trouver le bon bitmap dans la deuxième composante (C_2). Enfin, les deux bits correspondant à la position de v dans sa colonne seront inversés en des bits positifs dans les deux bitmaps sélectionnés.

Pour répondre à une requête d'intervalle comme « âge <= 220 », la requête sera transcrite sous la forme : $C_1 < 5 \ OU \ (C_1 = 5 \ ET \ C_2 <= 20)$. Ainsi, si un encodage par rang est appliqué sur les deux composantes, 3 bitmaps seront accédés, tandis qu'avec un encodage binaire 10 bitmaps devront être scannés. En général, un encodage multicomposantes engendre plus de bitmaps qu'un encodage binaire, mais avec l'avantage que peu de bitmaps seront scannés lors de l'exécution de requêtes. Aussi, il crée, dans la plupart des cas, moins de bitmaps comparé à un encodage par égalité ou par rang.

	C_1						C_2					
Z	R_0	R_1	R_2		R_{23}	R_{24}	R_0	R_1	R_2		R_{38}	R_{39}
0	1	0	0		0	0	1	0	0		0	0
81	0	0	1		0	0	0	1	0		0	0
40	0	1	0		0	0	1	0	0		0	0
921	0	0	0		1	0	0	1	0		0	0
80	0	0	1		0	0	1	0	0		0	0
2	1	0	0	...	0	0	0	0	1	...	0	0
118	0	0	1		0	0	0	0	0		1	0
999	0	0	0		0	1	0	0	0		0	1
38	1	0	0		0	0	0	0	0		1	0
962	0	0	0		0	1	0	0	1		0	0
920	0	0	0		1	0	1	0	0		0	0

Figure 2.4: L'index bitmap de l'attribut Z avec un encodage multicomposantes.

Plus le nombre de composantes augmente, plus l'espace de stockage croît et moins de bitmaps auront besoin d'être scannés pour répondre à une requête. Chan et Ioannidis (1999) mentionnent que le nombre optimal de composantes qui offre le meilleur compromis temps-espace est de 2.

Méthodes d'encodage multiniveaux

D'autres solutions faisant partie des méthodes d'encodage multiniveaux et qui combinent un ou plusieurs des trois encodages basiques ont également été proposées dans le but d'améliorer le compromis entre le nombre de bitmaps stockés et scannés lors de l'exécution de requêtes (Wu *et al.*, 2001b; Sinha et Winslett, 2007; Bin et Yu-Xing, 2011). L'encodage à deux niveaux de Bhan *et al.* (2012) adopte un encodage par égalité avec un *binning* au premier niveau et un encodage binaire au second niveau. Des expériences ont montré que cette approche utilise peu de bitmaps comparée à un encodage par égalité et en scanne beaucoup moins lors de l'exécution de requêtes d'intervalles. Les auteurs ont aussi constaté que ces performances s'amélioraient en augmentant le

nombre des valeurs dans les *bins*.

Wu *et al.* (2010) ont également étudié un encodage à plusieurs niveaux formant des hiérarchies de *bins*. Les valeurs distinctes de l'attribut indexé sont divisées en un certain nombre de *bins* au premier niveau, puis chacun de ces *bins* est à son tour divisé en d'autres *bins* plus fins. Ce processus est réitéré sur les *bins* générés jusqu'à obtention de *bins* assez fins. Un index bitmap est construit sur chaque *bin* et peut être encodé avec l'un des trois encodages basiques. L'index global peut alors être fondé sur un encodage hybride bénéficiant des avantages de chacun des trois encodages basiques. Des expériences (Wu *et al.*, 2010) ont montré qu'un encodage intervalle-égalité, qui consiste à encoder les index bitmap de haut niveau avec un encodage par intervalle et les index bitmap de bas niveau avec un encodage par égalité, offre de meilleurs compromis temps-espace comparé à un encodage binaire, à un encodage par égalité, ainsi qu'à d'autres sortes de combinaisons (rang-égalité, égalité-égalité, etc.). En effet, les temps de réponse obtenus avec un encodage intervalle-égalité sont, respectivement, ≈ 4 fois et ≈ 10 fois plus rapides par rapport à un encodage binaire et à un encodage par égalité.

2.4 Techniques de compression des bitmaps

Afin de réduire la taille d'un index bitmap et d'améliorer ses temps de traitements, des chercheurs ont proposé de compresser chaque bitmap de l'index individuellement en appliquant une technique de compression de bitmaps. Cette section présente les principaux types de méthodes ayant été utilisées pour compresser des bitmaps et présente, pour chaque type rapporté, des solutions efficaces rencontrées lors du survol de la littérature ancienne et récente.

2.4.1 Compression de bitmaps avec des méthodes de compression textuelle

Le champ de recherche relatif à la compression des bitmaps est étudié depuis bien longtemps. Parmi les plus anciennes contributions, nous citons celles qui consistent à appliquer des techniques génériques de compression textuelle, comme : le codage de Huffman (Huffman, 1952) ou *LZ77* (Ziv et Lempel, 1977; Gailly, 1998) sur des vecteurs de bits. Jakobsson (1978) propose un modèle de compression bitmap basé sur un codage de Huffman. Cette solution divise un bitmap en des blocs de k bits, puis construit un arbre de Huffman sur le bitmap en se basant sur la probabilité d'apparition de chaque bloc de bits. Le bitmap est par la suite encodé sous forme d'une séquence de mots de code obtenus de l'arbre de Huffman. Cette technique offre d'impressionnants taux de compression sur des bitmaps de faibles densités. En effet, dans ces cas-là, les blocs ne contenant que des bits à 0 (des bits négatifs) ou seulement un seul bit à 1 (bit positif) apparaissent fréquemment, et le codage de Hauffman attribuera à ce type de blocs un mot de code de taille plus petite que k bits. Au final, la taille moyenne des mots de code représentant les blocs de bits du bitmap initial sera généralement inférieur à k bits.

D'autres méthodes qui combinent plusieurs techniques de compression textuelle ont aussi été introduites. Fraenkel et Klein (1985) proposent une solution de compression

bitmap, nommée *LLRUN*, qui combine un codage de Huffman avec une compression par plage de valeurs (*run-length encoding*). Cette solution représente un bitmap avec un codage différentiel, dans lequel un bit à 1 est encodé par un entier Δ, représentant la différence entre la position de ce bit est celle du précédent bit à 1 dans le bitmap (cette dernière valeur étant égale à 0 si c'est le premier bit à 1 du bitmap). Un ensemble d'entiers Δ de même taille sont groupés dans un même paquet, et chacun des paquets collectés se voit attribué un code de Huffman unique, ch. Ainsi, un entier Δ est encodé par le code ch de son paquet suivi de sa représentation binaire minimale en omettant le 1 de poids fort.

Wedekind et Harder (1976) ont proposé une technique hiérarchique pour la compression de bitmaps. Tout d'abord, le bitmap initial v_0 est divisé en r_0 blocs de k_0 bits chacun. Puis, les blocs ne contenant que des bits à 0 sont omis de la représentation. Pour garder trace de ces blocs de bits à 0 afin de pouvoir reconstruire l'index initial, un nouveau bitmap v_1 de r_0 bits est créé pour jouer le rôle d'un index de plus haut niveau. Le $i^{\text{ème}}$ bit de v_1 est à 1 si le $i^{\text{ème}}$ bloc de v_0 contient au moins un bit à 1, sinon il est mis à 0. Le bitmap v_1 est ensuite compressé de la même manière, après l'avoir divisé en r_1 blocs de k_1 bits chaque, un nouveau bitmap v_2 sera instancié pour indiquer la position des blocs de bits à 0 omis de v_1. Le même principe de compression bitmap se poursuit sur les nouveaux bitmaps générés jusqu'à obtention d'un bitmap de taille assez petite. En dernier, les blocs de bits contenant au moins un bit à 1 des différents bitmaps v_i sont concaténés dans l'ordre décroissant de i, pour aboutir à la forme compressée du bitmap initial v_0.

2.4.2 Compression de bitmaps avec des techniques de compression de listes d'entiers

Dans les bases de données et les moteurs de recherche, un bitmap est souvent remplacé par une liste d'entiers triée dans un ordre croissant où chaque élément correspond à

la position d'un de ses bits à 1. Culpepper et Moffat (2010) proposent une telle représentation pour améliorer les performances des index inversés. Cette représentation peut économiser de l'espace de stockage lorsqu'employée sur un bitmap de faible densité renfermant peu de bits positifs, mais en consomme, cependant, beaucoup plus que la représentation initiale sur des bitmaps denses ayant un fort taux de bits à 1. Afin d'améliorer l'efficacité d'une représentation bitmap par liste d'entiers, une solution serait d'appliquer des techniques de compression d'entiers sur chaque liste. Les performances des techniques de ce genre connues à ce jour varient d'une méthode à l'autre, et chaque approche vise à fournir des résultats intéressants face à l'imposant compromis entre les taux de compression et les temps d'encodage/décodage. Ce champ de recherche est exploré depuis bien longtemps, nous citons quelques contributions connues comme : Golomb et Rice coding (Rice et Plaunt, 1971), Interpolative coding (Moffat et Stuiver, 2000), Elias gamma et delta coding (Elias, 1975), Variable byte (Cutting et Pedersen, 1990), Byte-oriented encodings (Stepanov *et al.*, 2011), Simple Family (Anh et Moffat, 2010, 2005; Yan *et al.*, 2009), Binary packing (Anh et Moffat, 2010; Deveaux *et al.*, 2007), Patched coding (Zukowski *et al.*, 2006).

Pour donner un aperçu de méthodes de compression d'entiers, voyons voir plus en détails quelques-unes de ces techniques. Pour encoder un entier positif a, la méthode de Golomb (1966) procède en deux étapes : la première étape consiste à calculer $q = \lfloor \frac{a}{b} \rfloor$, où b est un paramètre fixé à l'avance. Une fois obtenu, q est encodé en unaire (Wikipedia, 2013). La deuxième phase calcule le reste de la division, $r = a \ mod \ b$, puis encode le résultat r en binaire avec au plus $\lceil \log_2 \ b \rceil$ bits.

Rice et Plaunt (1971) améliorent la technique de Golomb en prenant le paramètre b comme une puissance de 2. Cela permet de calculer q et r à l'aide de simples décalages de bits, offrant ainsi une implémentation plus rapide de la méthode de Golomb. Ces deux encodages offrent de bons taux de compression, mais ils sont connus d'être lents lorsqu'il s'agit de compresser/décompresser des entiers.

Les encodages gamma (γ) et delta (δ) d'Elias (1975) sont parmi les plus cités dans la littérature scientifique. Afin de compresser une liste d'entiers, ces deux techniques essaient d'attribuer de courts codes aux entiers de la liste. L'encodage γ d'Elias encode un entier a avec $\lfloor \log_2 a \rfloor$ bits à 0, qui représenteront le préfixe de la forme compressée, suivis de la représentation binaire minimale de a qui nécessite $1 + \lfloor \log_2 a \rfloor$ bits et dont le bit de poids fort est toujours à 1, ces derniers bits représenteront le suffixe de la forme compressée. Le nombre de bits occupés par l'entier a une fois compressé avec l'encodage γ d'Elias sera de $1 + 2 \lfloor \log_2 a \rfloor$ bits. Le Tableau 2.1 montre des exemples d'entiers encodés avec un γ code. Pour donner un meilleur aperçu de la technique d'encodage, une virgule (« , ») a été placée entre le préfixe et le suffixe de chaque code.

Selon Bertchold *et al.* (1992), le désavantage de la technique γ est que plus les entiers deviennent grands, plus le taux de compression rétrécit. L'encodage δ d'Elias est plus efficace dans les cas de larges entiers. Cette méthode encode un entier a avec l'encodage γ de l'entier $(1 + \lfloor \log_2 a \rfloor)$, suivi de la représentation binaire de a sans le bit le plus significatif. Des exemples de cet encodage sont donnés dans le Tableau 2.1. L'encodage γ consomme moins d'espace sur la majorité des entiers positifs inférieurs à 15 comparé à l'encodage δ, tandis que ce dernier n'est jamais pire que l'encodage γ sur des entiers plus grands que 15 (Bertchold *et al.*, 1992).

Les encodages δ et γ sont simples à implémenter et un seul passage sur une liste d'entiers leur suffit pour la compresser. Ces deux techniques offrent également de remarquables taux de compression tels qu'observés dans Bertchold *et al.* (1992).

Lemire et Boytsov (2013) couvrent une large sélection de techniques de compression d'entiers (anciennes et récentes) et proposent de nouvelles méthodes : *SIMD-BP128** et *SIMDFastPFOR*, qui ont montré de remarquables performances. Le lecteur désireux de connaître davantage de solutions de compression d'entiers peut se référer à leur papier.

Entier	γ	δ
5	00,101	011,01
6	00,110	011,10
7	00,111	011,11
8	000,1000	00100,000
9	000,1001	00100,001
10	000,1010	00100,010

Tableau 2.1: Exemples de codes γ et δ

2.4.3 Systèmes génériques de compression bitmap

Fraenkel et Klein (1985) ont montré qu'une large gamme de techniques de compression bitmap pouvaient être représentées par un système générique décrit comme suit :

Soit V un vecteur d'entiers positifs v_i. Pour coder un entier $a \geq 1$, il suffit de trouver le $k^{\text{ème}}$ entier de V tel que :

$$\sum_{i=1}^{k-1} v_i < a \leq \sum_{i=1}^{k} v_i.$$

Ensuite, la différence suivante sera calculée :

$$d = a - \sum_{i=1}^{k-1} v_i - 1.$$

Le $k^{\text{ème}}$ entier trouvé sera par la suite codé avec un encodage approprié et sera suivi de la représentation binaire de d qui utilise $\lfloor \log\ v_k \rfloor$ bits si $d < 2^{\lceil \log\ v_k \rceil} - v_k$, ou $\lceil \log\ v_k \rceil$ bits si $d \geq 2^{\lceil \log\ v_k \rceil} - v_k$.

Un encodage γ d'Elias par exemple, pourrait être reproduit en adoptant le vecteur d'entiers $V_\gamma = (1, 2, 4, 8, ..., 2^{i-1}, ...)$, et en codant les entiers trouvés de la liste en unaire. La méthode *LLRUN* de Fraenkel et Klein (1985) décrite plus haut pourrait être représentée avec le même vecteur d'entiers de l'encodage γ et en codant les entiers trouvés

de la liste avec un codage de Huffman portant sur la fréquence d'apparition de chaque entier sélectionné dans la liste d'entiers compressés.

Teuhola (1978) a proposé un encodage, nommé *ExpGol*, adapté aux distributions de données biaisées, souvent rencontrées dans les ensembles de données réelles. *ExpGol* représente une généralisation de l'encodage γ d'Elias et utilise de ce fait un vecteur d'entiers $V_T = (b, 2b, 4b, ..., 2^{i-1}b, ...)$, où b est un paramètre fixe dont les auteurs suggèrent de le prendre comme une petite puissance de 2.

Bertchold *et al.* (1992) évaluent les performances de plusieurs modèles de compression bitmaps génériques, dont *LLRUN*, *ExpGol*, le γ coding d'Elias ainsi qu'un encodage de Huffman (Huffman, 1952). La version originale d'*ExpGol* (Teuhola, 1978) a été conçue pour être appliquée de façon globale sur un index bitmap, en encodant tous les bitmaps de l'index avec un même vecteur d'entiers. Les auteurs de Bertchold *et al.* (1992) ont changé cette méthode en une version locale, qui permet d'encoder chaque bitmap avec son propre vecteur d'entiers. Les deux méthodes globales *LLRUN* et celle de Huffman ont également étaient modifiées en des versions locales. Cependant, consommant de larges espaces, l'attribution d'un code préfixe propre à chaque bitmap s'est révélée inefficace et, par conséquent, les auteurs ont choisi de ne générer que $\lceil \log\ N \rceil$ codes préfixe différents, où N est égal au nombre de bits dans les bitmaps non compressés. Pour encoder le bitmap t, le $\lfloor \log\ p_t \rfloor^e$ code préfixe sera employé, où p_t correspond au nombre de bits positifs dans le bitmap t. Les résultats ont montré que les versions locales des méthodes offrent de meilleurs taux de compression par rapport aux anciennes versions globales.

Bookstein et Klein (1991) proposent également de nouvelles méthodes de compression bitmap et comparent leurs performances avec celles de plusieurs autres techniques précédemment introduites. Les expériences menées ont montré que leurs nouvelles méthodes offrent les meilleurs résultats et atteignent des taux de compression touchant les

16 %. Bertchold *et al.* (1992) ont reproduit un banc d'essai similaire à celui de Bookstein et Klein (1991) et ont constaté que leurs méthodes locales ont affiché des taux de compression environnant les 30%, ce qui confirme l'efficacité de ces approches locales.

2.4.4 Compression de bitmaps par alignement d'octets (*Byte aligned Bitmap Compression*) (*BBC*)

Les techniques de compression textuelle et d'entiers, lorsque appliquées sur des bitmaps, offrent en général des taux de compression assez proches de la limite théorique, mais affichent cependant plusieurs désavantages, comme : (1) Un calcul logique ou un accès aléatoire ne peut se faire directement sur des bitmaps compressés (Johnson, 1999), car ces derniers doivent être décompressés avant de procéder à une telle opération, introduisant ainsi une surcharge qui rend les performances de ces approches moins intéressantes dans des contextes où ce genre d'opérations doit pouvoir être rapidement réalisé. En effet, pour exécuter efficacement une requête comme « âge < 200 », un index bitmap encodé par égalité entraînera une fusion logique entre 200 bitmaps. Un tel scénario nécessite une méthode qui, non seulement offre de bons taux de compression, mais qui permet aussi d'exécuter rapidement des opérations logiques entre plusieurs bitmaps. (2) Le stockage des bitmaps dans leur forme compressée augmente les coûts des modifications.

Antoshenkov (1995) a observé que la plupart des ordinateurs traitent les données binaires en des blocs de bits de taille fixe. Cela poussait les techniques de compression textuelle ou d'entiers, qui encodent les entités avec des chaînes de bits de longueurs variées, à gaspiller de l'espace lors de l'alignement de la forme compressée d'une entité sur des blocs de bits manipulables par la machine, et à perdre du temps lors des opérations de lecture où plusieurs blocs de bits doivent être analysés pour récupérer une certaine représentation compressée. L'auteur a alors constaté le besoin d'une technique de compression bitmap qui est bien adaptée aux caractéristiques matérielles de la

machine et qui peut opérer sur des bitmaps compressés sans recourir à une éventuelle décompression. Ainsi, *BBC* a été introduite comme une technique de compression bitmap qui respecte ces deux principaux critères.

BBC lit un bitmap non compressé bit par bit et l'encode en une séquence de blocs d'octets. Trois types de blocs d'octets peuvent être rencontrés lors de l'encodage d'un bitmap : un bloc propre (*gap byte*), qu'on notera *GBYTE*, dont tous les bits ont la même valeur logique. Le deuxième type de bloc d'octets suit une séquence de *GBYTE* et ne possède qu'un seul bit de valeur logique différente de celle des bits des *GBYTE*. On notera ce type de bloc *OBYTE* pour *Offset Byte*. Le troisième type de bloc est un bloc sale (*map byte*), qu'on notera *MBYTE*, et qui contient une combinaison de plusieurs 0 et 1.

BBC encode une séquence de blocs d'octets d'un même type dans un atome. Le premier bloc d'un atome est un octet de contrôle (*Control Byte*), qu'on appellera *CBYTE*. Ce dernier est composé de trois champs : un champ « type » noté (*TFIELD*), un champ « sens des bits » noté *FFIELD* et un champ « données » noté *DFIELD*.

Une séquence d'un ou de plusieurs *GBYTE* est encodée dans un atome à l'aide d'une compression par plage de valeurs. Si le nombre de *GBYTE* est inférieur à un certain seuil, alors la longueur de la séquence sera stockée dans le champ *TFIELD* du *CBYTE* de l'atome. Sinon, de 1 à 8 blocs d'octets peuvent être créés pour conserver la longueur de la séquence de *GBYTE* exprimée en octets et en bits. Le champ *FFIELD* conservera le sens des bits des *GBYTE*. De façon similaire, un atome peut encoder jusqu'à 15 *MBYTE* successifs du bitmap. La séquence des *MBYTE* est préservée dans la représentation compressée du bitmap et suit le *CBYTE* de l'atome. Le champ *DFIELD* du *CBYTE* stocke le nombre des *MBYTE* de la séquence. Quant aux *OBYTE*, un atome est généralement créé pour chaque octet de ce type et la position du bit différent dans un *OBYTE* est conservée dans le champ *DFIELD* du *CBYTE* de l'atome. Dans les cas où

un *OBYTE* suit une séquence de *GBYTE*, il sera alors stocké dans l'atome compressant les *GBYTE*.

Cela donne un aperçu des principes généraux d'une compression bitmap avec *BBC*. Concrètement, la méthode *BBC* utilise des atomes de différentes structures pour compresser un bitmap, chacune adaptée à une suite d'octets spécifiques. La technique peut utiliser en tout 8 atomes différents. Le lecteur désireux de connaître plus de détails sur la structure de chaque atome ainsi que sur les mécanismes d'encodage/décodage des bitmaps avec *BBC*, peut se référer à (Antoshenkov, 1995).

Lors d'expériences comparant les performances de *BBC* avec la méthode d'encodage différentiel *Delta* (Korn et Vo, 1995) (à ne pas confondre avec l'encodage δ d'Elias), *BBC* a offert une meilleure compression sur des fortes densités, tandis que sur des faibles densités, la compression *Delta* s'est montrée un peu plus performante. En ce qui concerne les temps d'encodage/décodage des bitmaps, l'auteur affirme que *BBC* encode ≈ 2 fois plus rapidement et décode ≈ 4 fois plus vite comparé à la compression *Delta*. Ces bonnes performances sont principalement dues à l'alignement des bitmaps en blocs d'octets.

L'auteur décrit aussi les règles suivies par *BBC* lors du calcul d'une opération logique entre deux bitmaps. Il a été rapporté, qu'en suivant ces règles, *BBC* peut effectuer une fusion logique entre deux bitmaps de 10 à 30 fois plus rapidement que des bitmaps sans compression ou compressés avec un encodage *Delta*.

Johnson (1999) compare les performances de trois techniques de compression bitmap : *ExpGol* (Bertchold *et al.*, 1992), *LZ* (Gailly, 1998) et *BBC*. L'auteur a constaté que la sélection de l'algorithme qui offre la meilleure compression parmi les trois méthodes dépend de la densité et du type de distribution des bits positifs dans le bitmap à compresser. Les résultats ont montré que sur des bitmaps de faibles densités, *ExpGol* affiche les meilleurs ratios de compression. Tandis que sur des bitmaps de fortes densités (fort

taux de bits à 1), *LZ* offre en général la meilleure compression, suivi de *ExpGol*, bien que les performances de *BBC* s'améliorent sur des distributions par grappes (*clustered distributions*) jusqu'à dépasser celles de *ExpGol*.

Les auteurs ont également évalué les temps que prennent chacune des trois techniques de compression bitmap pour faire des calculs logiques sur des bitmaps de densités variées, en utilisant 4 différents algorithmes de calculs logiques. Il a été constaté que le choix du meilleur algorithme dépend de l'opérateur logique utilisé et de la densité des données. Les auteurs rapportent qu'effectuer une opération logique directement sur des bitmaps compressés pouvait être jusqu'à 50 fois plus rapide que sur des bitmaps non compressés. Ces résultats ont été obtenus avec *BBC*, étant l'unique méthode parmi les trois qui supporte le mieux ce genre d'opérations, contrairement à *ExpGol* pour qui ce type de traitements s'avèrent complexes et *LZ* qui n'a pu être adapté pour des traitements directs sur des bitmaps compressés.

2.4.5 Compression de bitmaps par alignement de mots CPU (*Word-Aligned Hybrid code (*WAH*)*)

L'alignement des bitmaps en blocs d'octets permet à *BBC* d'effectuer des opérations logiques plus rapidement comparées aux autres techniques de compression bitmap par longueurs de bits variées, comme : *LZ77*, l'encodage γ et δ d'Elias, etc. Wu *et al.* (2001a) ont observé que la majorité des CPU modernes traitent plus efficacement les informations par mots CPU que par blocs d'octets, les auteurs ont alors eu l'idée de développer une technique de compression bitmap qui respecte un alignement par mots CPU. Ils ont alors étendu *BBC* de façon à ce qu'il compresse les bitmaps avec un alignement par mots CPU. Cette solution de compression bitmap fut nommée *Word-Aligned Bitmap Code* (*WBC*). Bien qu'une compression basée sur un alignement par blocs d'octets aboutit en général à des bitmaps plus compacts, mais en matière de calculs logiques, les performances ont montré que *WBC* a permis d'exécuter des opérations logiques ≈ 2

à $\approx$ 4 fois plus vite que *BBC* (Wu *et al.*, 2001a).

Une analyse approfondie (Wu *et al.*, 2001a, 2006) du comportement de la méthode *BBC* lors de l'exécution d'opérations logiques a permis aux auteurs de découvrir plusieurs désavantages liés aux différentes structures atomiques gérées par cette technique, rendant son traitement complexe. En effet, l'utilisation d'un *CBYTE* comme l'entête d'un atome crée une certaine dépendance entre cet octet et le reste des octets de l'atome (Wu *et al.*, 2006). Par conséquent, lors de la lecture d'un bitmap compressé avec *BBC*, le *CBYTE* de l'atome courant doit être entièrement interprété avant de pouvoir passer aux autres octets de l'atome et au *CBYTE* suivant. Aussi, la gestion d'au moins 4 types d'atomes différents et les plusieurs interprétations possibles d'un octet conduisent à faire des traitements CPU complexes. Ces inconvénients provoquent des bulles (*pipeline stall*) (Wikipedia, 2015a) sur les pipelines du CPU et finissent par ralentir les temps d'exécution.

La méthode introduite par Wu *et al.* (2006), nommée *WAH*, est une technique de compression bitmap avec alignement de bits par mots CPU. Les auteurs rapportent que contrairement à *BBC* et *WBC*, cette technique se caractérise par sa simplicité, du fait qu'elle n'utilise pas de structures atomiques complexes et n'adopte que deux types de mots CPU, des *fill* et des *literal*. Les auteurs affirment aussi que les mots de *WAH* n'affichent pas de dépendances entre eux, contrairement aux blocs d'octets du modèle *BBC*. Des expériences (Wu *et al.*, 2006, 2001a) ont montré que *WBC* et *WAH* nécessitent presque la même quantité d'espace de stockage, mais que *WAH* permet d'exécuter des opérations logiques $\approx$ 2 fois plus rapidement que *WBC* et $\approx$ 12 fois plus vite que *BBC*.

La Figure 2.5 illustre un exemple de compression bitmap avec *WAH* sur un mot CPU de 32 bits ($w = 32$ bits). La partie (a) de la Figure 2.5 montre les positions des bits positifs du bitmap à compresser. Pour des contraintes d'espace, des «...» ont été utilisés pour représenter une suite de bits à 0.

Avec un mot CPU à w bits, *WAH* divise, tout d'abord, le bitmap en des blocs de $(w-1)$ bits. Les blocs obtenus sont donnés dans la partie (b) de la Figure 2.5. Tout comme *BBC*, les suites de blocs homogènes contenant une même valeur logique seront encodées par *WAH* à l'aide d'une compression par plage de valeurs, mais dans des mots CPU de type *fill*. Quant aux blocs hétérogènes renfermant une combinaison de 0 et de 1, ils seront encodés dans des mots de type *literal*. Pour différencier entre les deux types de mots, les concepteurs ont choisi d'utiliser le bit le plus significatif de chaque mot. Si celui-ci est à 1, le mot est alors de type *fill*, sinon il est de type *literal*. Cette stratégie permet de distinguer entre les deux types de mots sans extraire le moindre bit. Le deuxième bit le plus significatif d'un mot *fill* peut être à 1 ou à 0, selon le sens des bits dans la séquence des blocs homogènes encodés. Les $(w-2)$ bits restants sauvegardent le nombre de blocs homogènes compressés dans le mot. Pour ce qui est des $(w-1)$ bits de poids faible d'un mot *literal*, ceux-ci conservent les $(w-1)$ bits du bloc de bits hétérogène représenté par le mot. La partie (c) de la Figure 2.5 donne le résultat de la compression *WAH* sur le bitmap original.

Appliqué sur un bitmap, l'encodage *WAH* instanciera, au pire cas, 2 mots CPU pour chaque bit à 1. Le premier mot sera un mot *literal* qui stockera la chaîne de bits littérale contenant le 1, et le deuxième mot, de type *fill*, encodera la séquence des bits négatifs, s'il y en a, séparant deux bits positifs du bitmap. On sait qu'un attribut à N entrées nécessitera un index bitmap contenant N bits positifs. Ainsi, la taille maximale d'un index bitmap une fois compressé avec *WAH* sera $\approx 2N$ mots CPU. Ce qui est moins volumineux comparé à l'arbre-B$^+$, un des index les plus communément utilisés dans les bases de données, qui consomme environ $3N$ à $4N$ mots CPU (Wu *et al.*, 2006).

Par définition, le temps d'une opération de recherche est optimal s'il est une fonction linéaire par rapport au nombre des éléments de l'ensemble résultant. En d'autres mots, si K est le nombre des éléments résultants, alors le temps optimal d'une opération de recherche est en $\Theta(K)$. Dans Wu *et al.* (2006), les auteurs ont montré que la méthode

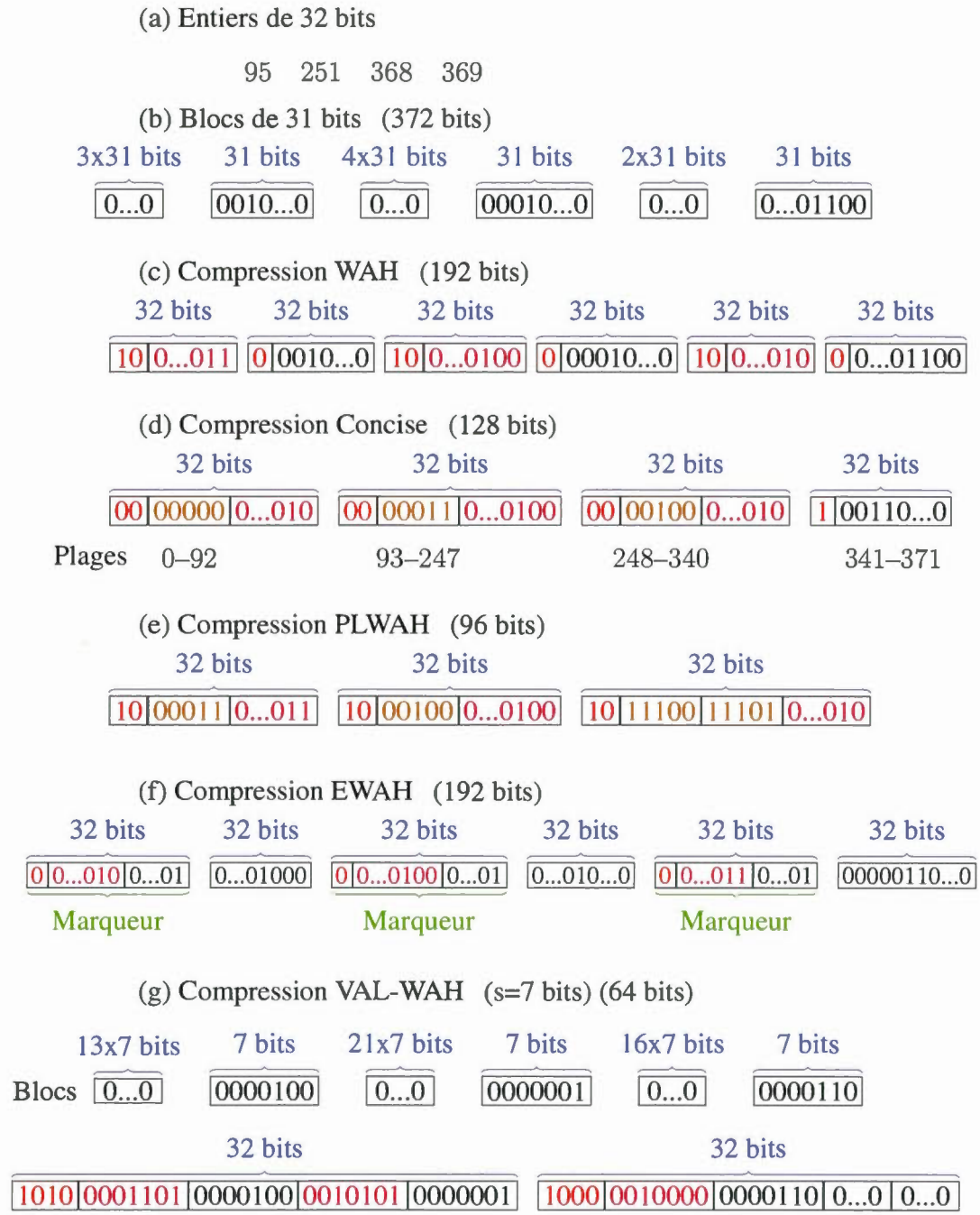


Figure 2.5: Compression du bitmap représenté par la liste d'entiers {95, 251, 368, 369} avec WAH, Concise, PLWAH, EWAH et VAL-WAH sur un mot CPU de taille w = 32 bits.

WAH calcule optimalement des requêtes d'intervalles à une dimension. Sachant que le temps d'exécution d'une requête d'intervalle à une dimension en utilisant un index bitmap est dominé par le temps nécessaire pour effectuer les calculs logiques entre les bitmaps sélectionnés. Dans Wu *et al.* (2001a, 2006); Stockinger *et al.* (2002), il a été rapporté que le temps consommé par une opération logique entre plusieurs bitmaps compressés avec *WAH* est une fonction linéaire par rapport à la taille totale des bitmaps impliqués. Puisque la taille d'un bitmap compressé par *WAH* est de $O(N)$ mots CPU, où N représente le nombre de bits positifs dans le bitmap, le temps d'exécution d'une opération logique entre plusieurs bitmaps compressés par *WAH* sera une fonction linéaire par rapport au nombre de bits positifs dans les bitmaps; mais pour une requête sur une dimension (sur un seul attribut), ce nombre de bits positifs est égal au nombre des éléments résultants, car un OU logique sera calculé entre les bitmaps sélectionnés. Par conséquent, un index bitmap compressé avec *WAH* permet de répondre optimalement à des requêtes à une dimension.

Les tests réels conduits par Wu *et al.* (2006) reflètent d'avantage l'efficacité des index bitmap compressés avec *WAH*. Comparé à plusieurs index : *BBC*, arbre-B, index de projection (O'Neil, 1987) et un index bitmap non compressé, *WAH* a montré les meilleurs temps de réponse dans tous les tests portant sur des requêtes d'intervalle à une dimension : $\approx$ 283 fois plus rapide que l'arbre-B, $\approx$ 6 fois plus rapide que *BBC* et $\approx$ 190 fois plus rapide que l'index de projection.

Il est à noter que la remarquable avancée par rapport à *BBC* est due au fait que les tests ont été menés sur des bitmaps stockés en mémoire principale. Si les bitmaps devaient être chargés du disque avant une opération, la différence entre les performances de *WAH* et de *BBC* pourrait diminuer à cause du fait que la taille des bitmaps compressés avec *BBC* est environ 60 % plus petite que celle des bitmaps compressés avec *WAH* (Wu *et al.*, 2001a). Du coup, moins d'opérations d'E/S disque seront requises avec *BBC*.

Wu *et al.* (2006) ont également étudié les performances de *WAH* avec des requêtes multidimensionnelles. Étant donné que *WAH* offre des temps de réponse optimaux sur une dimension et que les résultats obtenus sur une dimension peuvent être efficacement combinés avec ceux d'autres dimensions, cela permet de répondre aussi efficacement à des requêtes multidimensionnelles à l'aide d'index bitmap compressés avec *WAH*. Des tests sur des données réelles comparant les performances de *WAH* avec celles d'autres index : *BBC*, arbre-B, index de projection, etc., ont montré que *WAH* est le plus rapide pour des requêtes multidimensionnelles par rapport aux autres index évalués. L'index de projection et l'arbre-B sont, respectivement, ≈ 3 fois et ≈ 6 fois plus lents que *WAH*, et ce dernier est ≈ 6 fois plus rapide que *BBC*.

2.4.6 Variantes de *WAH*

Depuis l'introduction de *WAH*, plusieurs chercheurs ont essayé d'améliorer les performances de cette technique en proposant des solutions de compression bitmap qui combinent une compression par plage de valeurs avec une représentation bitmap sous forme de chaînes de bits alignées par mots CPU : Colantonio et Di Pietro (2010); Deliège et Pedersen (2010); Lemire *et al.* (2010); Fusco *et al.* (2010); Schmidt *et al.* (2011); Guzun *et al.* (2014); Ma *et al.* (2014); Chang *et al.* (2015); Chen *et al.* (2015); Wu *et al.* (2016), etc.

Compressed 'n' Composable Integer Set (*Concise*)

Colantonio et Di Pietro (2010) ont introduit une nouvelle technique de compression bitmap appelée Concise. Cette méthode occupe, sur des faibles densités, 2 fois moins d'espace que *WAH* sans, en contrepartie, augmenter les temps de traitement sur les bitmaps. Un bitmap de faible densité est majoritairement composé de plusieurs 0 séparant deux bits positifs. Dans un tel bitmap, *WAH* créé en général deux mots CPU à la rencontre d'un bit positif (qu'on appellera *bit pollué*), un premier mot *literal* pour encoder

le bloc dans lequel apparaît le *bit pollué*, et un deuxième mot de type *fill* pour encoder la suite de blocs homogènes de valeur logique 0 séparant le bloc du *bit pollué* de celui du prochain bit positif dans le bitmap. L'idée principale de Concise est d'utiliser un nouveau type de mot dans ce genre de scénario qui encode le bloc du *bit pollué* et la séquence des blocs homogènes dans un seul mot CPU, que l'on nommera mot *mixte*.

Rappelons qu'un mot *WAH* (de taille w bits) de type *fill* utilise les $(w-2)$ bits de poids faible pour sauvegarder la longueur de la séquence de blocs homogènes encodée. En observant de plus près un bitmap compressé avec *WAH*, les auteurs ont constaté que dans la majorité des mots CPU de type *fill*, beaucoup moins de $(w-2)$ bits suffisent pour stocker la longueur de la file de blocs homogènes. Les auteurs ont alors eu l'idée de prendre les $\lceil \log_2 \ w \rceil$ bits les plus significatifs parmi les $(w-2)$ bits de poids faible d'un mot *fill*, afin d'encoder la position du *bit pollué* dans son bloc hétérogène. Notons que les positions des bits d'un bloc hétérogène varient de 1 à $w-1$ et que chacune peut être écrite sur $\lceil \log_2 \ w \rceil$ bits. La forme d'un mot *mixte* se présente comme suit : son bit de poids fort indique le type du mot : 0 pour un *mixte* ou 1 pour un *literal*, le bit suivant spécifie le sens des bits des blocs homogènes encodés dans le mot, les $\lceil \log_2 \ w \rceil$ bits qui suivent indiquent la position du *bit pollué* dans le bloc hétérogène, et le reste des bits encodent la longueur des blocs de bits homogènes. Notons qu'un mot de type *fill* est également représenté par un mot *mixte* dont les bits de position sont nuls. Une autre petite distinction par rapport à *WAH* réside au niveau du processus de division du bitmap en des blocs de $(w-1)$ bits. Concise stocke un bit positif par l'entier correspondant à sa position dans le bitmap. Dans le cas d'un bit positif situé à la $i^{\text{ème}}$ position du bitmap, Concise le représentera avec un bit positif dans le $(i \ / \ (w-1))^{\text{ème}}$ bloc (en comptant les blocs à partir de 0) au $((i \mod (w-1))+1)^{\text{ème}}$ bit de poids faible. Aussi, la position stockée dans les bits de position d'un bloc mixte est donnée par rapport aux bits de poids faible. La partie (d) de la Figure 2.5 montre le résultat de la compression Concise du même bitmap compressé précédemment avec *WAH*. Notons

également que Concise peut encoder aussi efficacement des distributions contenant des séquences de bits positifs séparant des bits négatifs. Le principe reste le même, sauf que le deuxième bit de poids fort d'un mot *mixte* représentant le sens des bits homogènes aura la valeur logique 1. Le *bit pollué* dans ces cas aura la valeur logique inverse, soit 0.

Cette sophistication permet de réduire la taille des bitmaps de faibles densités à $\approx N$ mots CPU plutôt que de $\approx 2N$ mots CPU dans le cas de *WAH*. Des expériences (Colantonio et Di Pietro, 2010) approuvent cette affirmation et montrent que Concise ne prend jamais plus de temps que *WAH* lors des calculs logiques.

Position List Word Aligned Hybrid (*PLWAH*)

PLWAH (Deliège et Pedersen, 2010) est une technique qui adopte le même principe que Concise mais qui permet de former des mots *mixtes* encodant une séquence de blocs homogènes et un bloc hétérogène pouvant contenir un ou « plusieurs » *bits pollués*. Le nombre de *bits pollués* tolérés dans un mot mixte est défini par un seuil fixe préétabli. Une petite distinction par rapport à Concise réside sur le fait que le bloc hétérogène encodé dans un mot *mixte* vient après la file dans la représentation décompressée du bitmap (non pas avant la file comme c'est le cas avec Concise). Aussi, le bit de poids fort d'un mot *PLWAH mixte* prend un 1, tandis que celui d'un mot *literal* est mis à 0. Cette solution devrait compresser plus efficacement que Concise sur des distributions formant plusieurs séquences de blocs homogènes interrompus par un bloc hétérogène contenant plusieurs *bits pollués*. La comparaison des performances de ces deux techniques reste à déterminer. Toutefois, par rapport à *WAH*, des expériences (Deliège et Pedersen, 2010) ont montré que *PLWAH* compresse $\approx$ 2 fois mieux sur des faibles densités. En terme des temps de réponse aux requêtes, *PLWAH* a montré qu'il n'a était jamais pire que *WAH* et qu'il pouvait être jusqu'à 20% plus rapide pour certaines requêtes. La partie (e) de la Figure 2.5 montre le résultat d'une compression *PLWAH* sur le précédent bitmap

compressé avec *WAH*.

Chang *et al.* (2015) ont amélioré le modèle de *PLWAH* en introduisant une nouvelle technique de compression bitmap : *SPLWAH*, qui étend le format *PLWAH* avec de nouveaux mots de code pour compresser de nouveaux patrons dans une suite de mots de 32 bits : SF, FSF et SFS. SF compresse une succession de deux mots de 32 bits, dont le premier est de type *literal* et le deuxième de type *fill*, en un seul mot CPU de 32 bits. FSF encode une suite de trois mots CPU de 32 bits, où le premier est de type *fill*, le deuxième de type *literal* et le troisième de type *fill*, en un seul mot CPU de 32 bits. Quant à SFS, ce mot de code compresse une occurrence de trois mots CPU en un seul mot CPU, lorsque le premier mot de la suite est de type *literal*, le deuxième de type *fill* et le troisième de type *literal*. Des bancs d'essais sur des données réelles ont révélé que *SPLWAH* est plus adapté aux cas de bitmaps denses, très souvent rencontrés sur des données triées, où il a affiché des taux de compression de $\approx 26\%$ meilleurs que *PLWAH*.

Enhanced Word Aligned Hybrid (*EWAH*)

Puisque *WAH* représente des blocs de 31 bits hétérogènes par des mots CPU de 32 bits, il arrive parfois, sur certaines données, que la taille du bitmap compressé avec *WAH* s'élargisse de 3% par rapport à sa taille sans compression. Lemire *et al.* (2010) ont étudié une solution de compression bitmap appelée *EWAH* qui réduit la probabilité de générer un bitmap compressé plus large que sa taille sans compression à $0,1\%$. Comme WBC, cette technique divise le bitmap en des blocs de w bits, puis compresse la séquence de blocs obtenue à l'aide d'atomes. Un atome peut être composé de deux types de mots CPU : un mot *literal* et un mot *marqueur*. Un mot *literal* conserve un bloc hétérogène de w bits. Un mot *marqueur* joue le même rôle qu'un *CBYTE* dans un atome *BBC*, et peut encoder une file de blocs hétérogènes ou une succession de blocs homogènes possiblement suivie d'une suite de blocs hétérogènes. Sur un mot CPU de

w bits, la forme d'un mot *marqueur* se présente comme suit : le bit le plus significatif représente le sens des bits de la file homogène, les $\lceil (w-1)/2 \rceil$ bits suivants enregistrent la longueur de la file des blocs homogènes et les $\lfloor (w-1)/2 \rfloor$ derniers bits stockent le nombre des blocs de bits hétérogènes, s'il y en a, qui suivent la séquence homogène. La partie (f) de la Figure 2.5 montre un exemple de compression *EWAH* sur le bitmap compressé précédemment avec *WAH*.

Sur des bitmaps de faibles densités, *EWAH* peut être moins compacte que *WAH* lors de la compression de longues suites de 0 Lemire *et al.* (2010). En effet, sur un mot CPU de 32 bits, le mot *marqueur* d'*EWAH* ne peut représenter que des files de blocs homogènes d'une longueur inférieure à 2^{16}. Lors d'expériences (Lemire *et al.*, 2010) sur des données réelles et un mot CPU de 32 bits, les auteurs ont constaté que *EWAH* était $\approx 14\%$ moins efficace que *WAH* en terme de compression des séquences de blocs homogènes, mais qu'il était de $\approx 3\%$ mieux en matière de compression des blocs hétérogènes, qui, dans leurs tests, constituaient près de la moitié des mots des bitmaps compressés. Cette dernière observation laisse prétendre l'efficacité d'*EWAH* comparé à *WAH* sur de fortes densités. Toutefois, les performances d'*EWAH* sur des files de blocs homogènes peuvent être améliorées en ajustant le nombre des bits représentant les longueurs des files. D'ailleurs, sur un mot CPU de 64 bits, les auteurs affirment qu'*EWAH* a été aussi efficace que *WAH* en compressant ces blocs de bits.

Bien que l'utilisation d'atomes crée une dépendance entre les mots CPU d'un bitmap encodé et ralentie de peu les temps d'exécution, *EWAH* peut cependant traiter un bitmap compressé contenant un fort taux de mots *literal* beaucoup plus rapidement que *WAH*. Effectivement, pour traiter un mot *literal*, *WAH* doit accéder au mot puis vérifier son bit le plus significatif avant de procéder à son traitement, tandis qu'un simple accès au mot suffit pour *EWAH*. Aussi, lors de calculs logiques, l'avantage de connaître à l'avance le nombre des mots *literal* dans un atome permet souvent à *EWAH* de sauter le traitement de toute une suite de mots CPU, tandis, que chaque mot *literal* doit impéra-

tivement être accédé avec *WAH*. Dans une opération comme un ET logique entre deux bitmaps de tailles t_1 et t_2, cet avantage fait passer la complexité d'une telle opération à $O(min(t1, t2))$ lorsque l'un des deux bitmaps est de faible densité et l'autre d'une densité assez élevée.

Cependant, lorsque le nombre des mots *marqueur* est assez important dans un bitmap, les traitements peuvent devenir plus rapides avec *WAH*. En effet, la lecture d'un mot *marqueur* requiert 3 accès : un pour connaître le sens des bits dans les blocs homogènes, un deuxième puis un troisième pour connaître les longueurs des suites de blocs homogènes et hétérogènes, respectivement. Par contre, chacun des mots de *WAH* ne nécessite, au pire, qu'un seul accès.

Variable Aligned Length (*VAL*)

On a vu qu'utiliser une petite unité de compression comme des blocs d'octets dans le cas de *BBC* permettait d'aboutir à de meilleurs taux de compression contrairement à l'adoption de plus grandes unités de compression telles que des blocs de $(w - 1)$ bits sur un mot CPU de w bits dans le cas de *WAH*. Cette dernière unité de compression est cependant plus adaptée à l'architecture des CPU modernes, ce qui se traduit en des traitements beaucoup plus efficaces comparé à des méthodes utilisant une plus petite unité de compression. Guzun *et al.* (2014) ont dernièrement introduit un système, nommé *VAL* (*Variable Aligned Length*) qui prend en compte plusieurs paramètres tirés des données et de l'utilisateur pour sélectionner la technique d'encodage (*WAH*, *EWAH*, *PLWAH*, etc.) et l'unité de compression les plus appropriées pour un bitmap donné. Ce système permet, entre autres, à l'utilisateur d'exploiter le compromis temps-espace afin d'aboutir à des configurations d'index bitmap qui satisfont ses exigences. L'avantage principal de cette solution est qu'elle tire profit d'une compression par petite unité pour réduire remarquablement les tailles des bitmaps, en plus d'un alignement des chaînes de bits par mots CPU pour assurer des traitements efficaces.

Pour encoder un bitmap, *VAL* identifie son profil (dense, peu dense, etc.) à l'aide d'une analyse, puis sélectionne la méthode d'encodage (*WAH*, *EWAH*, *PLWAH*, etc.) appropriée pour le bitmap. Voyons voir comment *VAL* effectue une compression *WAH* sur un bitmap. Avec un mot CPU de w bits, *VAL-WAH* divise le bitmap en des segments de s bits, où $s < w$. Ces segments sont par la suite encodés avec *WAH* en prenant une unité de compression égale à s bits. Une fois encodés, les blocs de $(s+1)$ bits obtenus avec *WAH* (1 bit est ajouté par *WAH* à chaque bloc pour indiquer le type du segment : *literal* ou *fill*) seront ensuite alignés dans des mots CPU de w bits. Pour accélérer les opérations de décodage, un bloc entête est placé au début d'un mot CPU et contient le bit de poids fort de chacun des blocs empaquetés dans le mot. Un octet est également stocké au début de chaque bitmap compressé pour identifier la longueur des segments s et la méthode d'encodage utilisée lors de la compression. La partie (g) de la Figure 2.5 illustre un exemple de compression *VAL-WAH* sur le bitmap des exemples précédents en prenant un $s = 7$ bits. L'adoption de segments de courtes longueurs permet à *VAL-WAH* de fournir de remarquables taux de compression, tout en assurant en général des temps de traitement comparables à ceux de *WAH*.

Un des avantages de *VAL-WAH* est qu'il offre la possibilité de compresser plusieurs bitmaps avec différentes unités de compression s, où le choix de ce dernier pour un bitmap dépend des données et de l'application. Bien qu'utiliser une unité de compression appropriée pour chaque bitmap améliore les taux de compression, ceci ne reste pas sans répercussions sur le coût des calculs logiques. En effet, soient deux mots CPU de deux bitmaps encodés avec deux unités de compression différentes. Avant de pouvoir procéder à un calcul logique entre deux de leurs blocs, il faut que ces derniers puissent être alignés en deux blocs de même taille. Ce type de transformation peut être très coûteux si s est arbitrairement choisi. Afin de réduire les coûts liés à cette surcharge lors des traitements logiques, les auteurs ont défini un ensemble représentant les seules valeurs légales pour s. Cet ensemble, noté LS, est défini comme suit :

$LS = \{2^i\ (b-1) \mid 0 \leq i \leq (log_2\ w - log_2\ b)\}$, où b est un facteur d'alignement introduit par l'utilisateur et w correspond à la taille du mot machine. LS est conçu de façon à ce que les grandes valeurs soient des multiples des plus petites valeurs, simplifiant ainsi l'alignement de deux blocs de tailles s différentes lors des calculs logiques. Par exemple : un $w = 64$ bits et un $b = 16$ donneront un ensemble $LS = \{15, 30, 60\}$.

Les petites valeurs de s compressent bien les bitmaps, mais engendrent une surcharge liée à l'alignement des blocs lors des calculs logiques. Tandis que des grandes valeurs de s compressent moins bien, mais réduisent les surcoûts dus aux alignements de blocs lors des traitements logiques. Afin de permettre à l'utilisateur d'adapter ce compromis temps-espace selon ses besoins, *VAL* lui propose d'introduire une valeur $0 \leq \gamma \leq 1$ qui sera prise en compte, en plus du profil du bitmap une fois analysé (dense, peu dense, etc.), lors du choix d'une taille de segment $s \in LS$ appropriée pour le bitmap. Plus γ est proche de 0, plus la compression sera forte et les valeurs possibles de s tendront à être plus petites, et vice-versa.

Lorsque comparé à *PLWAH32*, *EWAH32*, *EWAH64*, *WAH32* et *WAH64* lors d'expériences (Guzun *et al.*, 2014), où NomMéthodeX signifie l'adoption de la technique NomMéthode sur un mot CPU de X bits, *VAL-WAH64* avec un $\gamma = 0,2$ a donné les meilleurs taux de compression sur toutes les distributions de données testées : $60\% - 80\%$ de la taille de *PLWAH32*, $55\% - 70\%$ de l'espace occupé par *WAH32*. Quant au reste des méthodes (*WAH64*, *EWAH32/64*), elles ont montré une compression beaucoup moins efficace que les 3 autres. En comparant les temps de réponse obtenus avec les techniques précédentes, *VAL-WAH*64 s'est montré $\approx 25\%$ et $\approx 15\%$ plus rapide que *WAH32* et *PLWAH32*, respectivement. Bien que les deux versions d'*EWAH* ne compressent pas si bien, elles ont cependant affiché les meilleurs temps de réponse sur toutes les données testées. *WAH64* s'est montré plus rapide que *VAL-WAH64* sur ces tests, mais en choisissant une grande valeur γ comme 0,9, *VAL-WAH64* a été 5% plus rapide que *WAH64*, dû à une plus petite taille des index compressés.

Pour connaître laquelle des techniques offre le meilleur compromis temps-espace, les auteurs ont combiné les taux de compression et les temps de réponse en une seule métrique référencée par le *gain*. En présumant que les ratios de compression (*compress_ratio*) et les ratios d'accélération (*speedup_ratio*) sont des facteurs de performances de poids égaux, le *gain* est calculé avec la moyenne harmonique (H_m) (Wikipedia, 2016) des deux ratios comme suit :

$$gain = \frac{1}{H_m} = \frac{compress\_ratio + speedup\_ratio}{2 \times speedup\_ratio \times compress\_ratio}$$ (voir Guzun *et al.* (2014)).

Sur des données réelles, les versions d'*EWAH* avec leurs impressionnants temps de réponse ont montré les meilleurs *gains*, suivis de *VAL-WAH64*. L'inefficacité de *WAH* et *PLWAH* s'explique par le fait que ces ensembles de données sont caractérisés par de courtes files de blocs homogènes que ces deux techniques compressent moins bien. Par contre, sur une distribution de données biaisée, *VAL-WAH64* a affiché les meilleurs gains suivi, dans l'ordre, de *PLWAH32* (0,3% moins bon que *VAL-WAH64*), puis de *WAH32*, d'*EWAH32*, de *WAH64* et en dernier d'*EWAH64*. La raison en est que *VAL-WAH* et *PLWAH* compressent ces types de données beaucoup mieux que *WAH* sans pour autant affecter les temps de réponse. *EWAH* a affiché les plus faibles gains parce que, sur des bitmaps peu denses, cette méthode ne compresse pas efficacement et offre des temps de réponse ne dépassant pas significativement ceux des autres techniques.

2.4.7 Modèles hybrides de compression bitmap

On a observé d'autres types de travaux dans la littérature qui adoptent des structures de données hybrides pour compresser des bitmaps et accélérer les opérations de calculs logiques (Sidirourgos et Kersten, 2013; Uno *et al.*, 2005; Culpepper et Moffat, 2010). *RIDBit* (O'Neil *et al.*, 2007) est un système développé à des fins pédagogiques pour montrer comment l'indexation bitmap peut être intégrée dans un système de bases de données. Les auteurs adoptent un arbre-B comme structure globale indexant les clés

d'un index bitmap donné. Chacune de ces clés est associée à un bitmap qui garde trace des entrées correspondant à la valeur de sa clé dans la colonne indexée. Lorsqu'un bitmap est dense, il fait l'objet d'une structure compacte et efficace. Par contre, un bitmap peu dense gaspille de l'espace et l'application d'une méthode de compression améliorerait de beaucoup son efficacité. *RIDBit* utilise une méthode simple pour compresser un bitmap de faible densité, qui consiste à le transformer en une RID-liste (O'Neil *et al.*, 2007), qui est une liste d'entiers dont chaque élément est représenté sous 2 octets. Pour différencier un bitmap dense d'un autre moins dense, les auteurs utilisent un seuil de 1/50; lorsque la densité du bitmap descend jusqu'à ce seuil, il sera transformé en une RID-liste, et vice-versa. Bien qu'un bitmap consomme plus d'espace qu'une RID-liste sur des densités variant entre 1/16 et 1/50, il reste cependant plus rapide en terme d'exécution d'opérations logiques. Par contre, sur des densités de 1/50 et moins, une RID-liste est plus efficace en matière d'espace de stockage et de temps de calculs logiques.

Pour améliorer encore plus les performances de la structure hybride, les auteurs stockent physiquement un bitmap sous forme de plusieurs segments qui seront par la suite conservés dans des pages disques. Une page disque peut aussi contenir plusieurs RID-listes. Une deuxième optimisation adoptée par *RIDBit* consiste à représenter les segments peu denses d'un bitmap par des RID-listes. Cela aide à économiser de l'espace et à rendre la structure de données plus compacte.

Des expériences ont comparé les performances de *RIDBit* avec celles d'un autre système *FastBit* (Wu *et al.*, 2009) qui utilise la technique *WAH* pour compresser les bitmaps. Les résultats ont montré que, sur des fortes densités, les deux systèmes consomment un espace presque similaire pour stocker les index bitmap. Alors que sur des faibles densités, *RIDBit* affiche une avancée significative par rapport à la compression *WAH* de *FastBit*. Ce qui confirme l'efficacité de la compression apportée par les RID-listes sur ces taux de densités. Cependant, en évaluant les temps pris par chaque système

pour répondre à des requêtes, les résultats ont révélé que *WAH* était plus performant que *RIDBit*.

CHAPITRE III

ROARING BITMAP

Ce chapitre est tiré des deux articles ci-dessous avec quelques contenus qui ont été ajoutés, modifiés et/ou supprimés :

Chambi, S., Lemire, D. et Godin, R. (2016b). Vers de meilleurs performances avec des Roaring bitmaps. Techniques et Science Informatique (TSI), 35(3), 335 – 355. http ://dx.doi.org/10.3166/TSI.35.335-355

Chambi, S., Lemire, D., Godin, R. et Kaser, O. (2014). Roaring bitmap : un nouveau modèle de compression bitmap. Dans 10e journées francophones sur les Entrepôts de Données et l'Analyse en Ligne (EDA'14), volume 27, 37 – 50., Vichy, France. RNTI.

3.1 Introduction

La plupart des techniques de compression des index 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 d'une chaîne de bits alignée par blocs de taille fixe (octets ou mots CPU). La Figure 3.1 illustre un exemple de compression bitmap avec deux solutions adoptant un tel encodage hybride : WAH et Concise.

Bien que les techniques adoptant ce type d'encodage offrent de bons taux de compres-

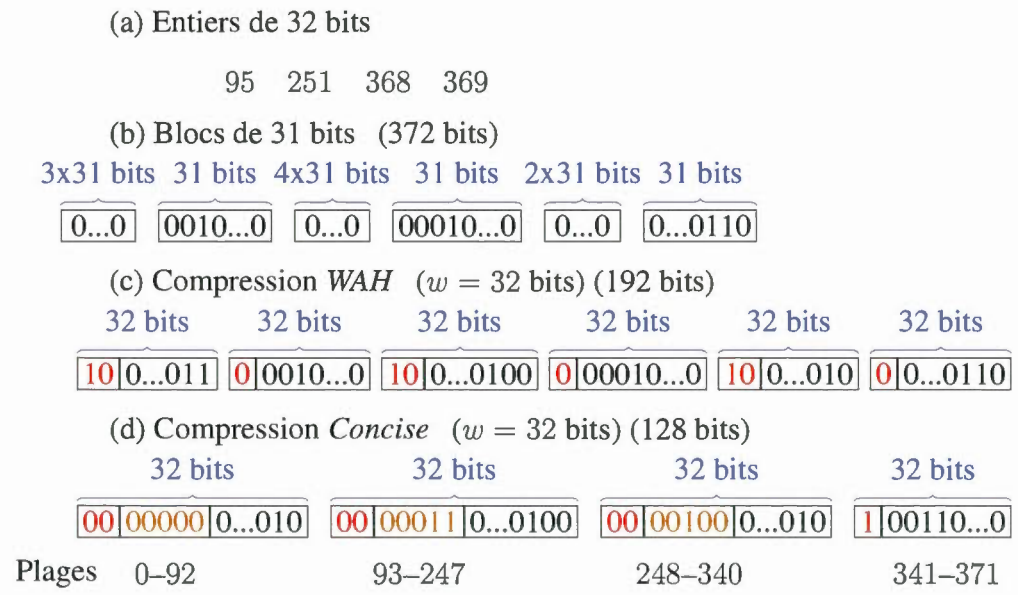


Figure 3.1: Compression de la liste d'entiers $\{95, 251, 368, 369\}$ avec *WAH* et *Concise* sur un mot CPU de taille $w = 32$ bits

sion, elles répondent moins efficacement aux opérations d'accès aléatoires. En effet, accéder à un bit aléatoire, i, 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.

Ce chapitre propose une nouvelle technique de compression bitmap, nommée *Roaring bitmap* (Chambi *et al.*, 2014, 2016d,b), adoptant un nouveau modèle hybride qui combine une compression préfixe avec plusieurs structures de données pour compresser un bitmap. En considérant un bitmap comme une liste 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 et Lemire, 2006). Des expériences ont montré que *Roaring bitmap* utilise, en moyenne, 16 bits/entier pour compresser une liste d'entiers de 32 bits sur des faibles densités, tandis que *Concise* et *WAH* requièrent, respectivement, 32 bits/entier et 64 bits/entier en moyenne sur les mêmes densités. Aussi, *Roaring bitmap* a affiché des temps de cal-

cul 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 chapitre est organisé comme suit : La Section 3.2 introduit la structure de *Roaring bitmap*. La Section 3.3 explique comment des accès aléatoires et des opérations logiques, ET ou OU, sont opérés sur des *Roaring bitmaps*. La Section 3.4 introduit la notion de *Memory-mapping* utilisée lors des expériences. La Section 3.5 présente les expériences qui ont permis d'évaluer les performances de *Roaring bitmap* sur des données réelles et synthétiques. On termine à la Section 3.6, avec une conclusion et des travaux futurs.

3.2 Roaring bitmap

Le modèle *Roaring bitmap* (Chambi *et al.*, 2014, 2016d,b) se présente sous la forme d'une structure à deux niveaux qui permet de compresser efficacement une liste d'entiers de 32 bits. Un tableau dynamique regroupe les entiers partageant les mêmes 16 bits de poids fort dans une même entrée, composée d'une clé et d'un conteneur. La clé préserve les 16 bits de poids fort du groupe 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. Ces dernières sont utilisées tel un index de premier niveau pour accélérer les accès aléatoires et les opérations logiques.

La Figure 3.2 illustre un exemple de compression d'une liste d'entiers avec *Roaring bitmap*. Lors de l'insertion d'un entier de 32 bits, une recherche binaire est lancée sur le tableau pour trouver une entrée dont la clé est équivalente aux 16 bits de poids fort de l'entier à insérer. Si une telle entrée est repérée, les 16 bits de poids faible de l'entier sont ajoutés au conteneur correspondant (voir l'insertion de 10 500 sur la Figure 3.2). Dans un cas échéant, une nouvelle entrée, composée d'un champ pour la clé et d'un

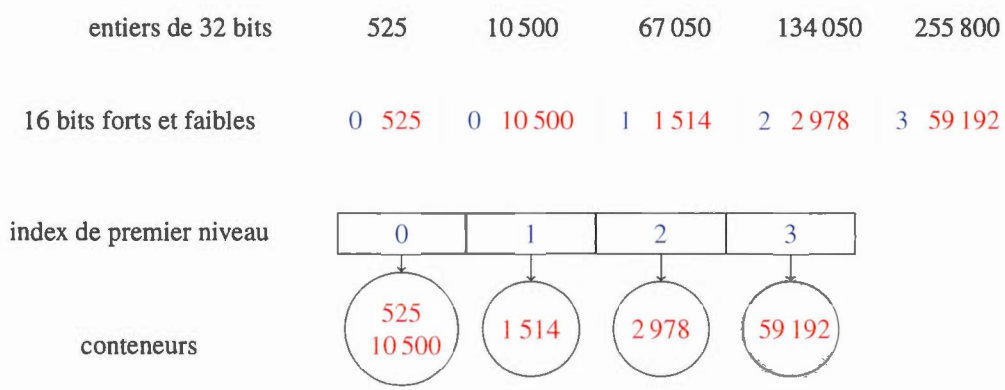


Figure 3.2: Représentation de la liste d'entiers {525, 10 500, 67 050, 134 050, 255 800 } compressée avec *Roaring bitmap*

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.

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 à représenter.

Un *conteneur-bitmap* est un bitmap de 2^{16} bits, pouvant représenter 2^{16} entiers compris dans l'intervalle [0, 65 535]. Initialement, tous les bits du bitmap sont à zéro. Pour indiquer la présence d'un éventuel entier a, le $(a \bmod 2^{16})^e$ bit correspondant à sa position dans le bitmap est mis à 1. Cette structure de données n'utilise que 1 bit pour indiquer la présence ou l'absence d'un entier de 16 bits. Cela permet aux *conteneur-bitmaps* d'être très efficaces sur des ensembles d'entiers denses. Cependant, lorsque la densité s'affaiblit, les performances se dégradent considérablement. Revenons à l'exemple de la liste des cinq entiers compressés sur la Figure 3.2. Avec des *conteneur-bitmaps*, le *Roaring bitmap* résultant consommera $(65\,536+16)\times 4$ bits, ce qui est très volumineux comparé à une représentation via un simple tableau d'entiers, qui, dans ce cas, ne nécessiterait que de 32×5 bits pour stocker un tel ensemble d'entiers. Après investigations, on a

constaté que ces cas survenaient lorsque le nombre d'entiers conservés dans un conteneur est inférieur à 2^{12} (4 096). Effectivement, nous avons besoin de moins de 2^{16} bits (taille statique d'un *conteneur-bitmap*) pour stocker $i \times 16$ bits, lorsque $i \in [1, 4\,095]$. Afin de contourner ce problème, nous utilisons des tableaux dynamiques (*conteneur-tableaux*) triés par ordre croissant, pour stocker les entiers de 16 bits d'un conteneur peu dense, ne contenant pas plus de 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 *Structured Query Language* (SQL) de type COUNT.

Si l'on compressait la liste d'entiers de la Figure 3.1 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 : $(16 + 16 \times 4)$ bits = 80 bits. Ce qui est beaucoup plus économique comparé aux 128 bits de *Concise* et 192 bits de *WAH*.

Plusieurs approches basées sur une structure de données hybride ont précédemment été proposées. Afin d'améliorer les performances de *LCM*, un algorithme de recherche de motifs fréquents, (Uno *et al.*, 2005) proposent une solution qui combine trois types de structures de données : un arbre préfixe, des bitmaps et des tableaux ; chacune ayant ses avantages et inconvénients par respect à la densité des données. Le système *RID-BIT* (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* (Wu *et al.*, 2009) qui utilise la technique *WAH* pour compresser les

bitmaps, *RIDBIT* a montré de faibles performances.

3.3 Opérations sur des *Roaring bitmaps*

3.3.1 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 par 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 binaire 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. 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. Ce calcul est réalisé efficacement à l'aide de l'instruction Java *Long.bitCount* qui fait recours à l'instruction *POPCNT* supportée par la majorité des CPU récents, notamment ceux d'Intel et d'AMD. 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, l'ensemble obtenu sera écrit dans un nouveau *conteneur-bitmap*, sinon, un nouveau *conteneur-tableau* représentera l'ensemble des entiers résultant. Calculer à l'avance la cardinalité de l'ensemble résultant permet d'éviter l'allocation inutile d'un nouveau *conteneur-bitmap* dans certains cas.

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 superscalaires, 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 scénario.

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 rapport des 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. Une telle fusion s'exécute en un temps de $O(n_1 + n_2)$, avec deux tableaux de tailles respectives, n_1 et n_2. Sinon, une intersection *galloping* (voir Culpepper et Moffat (2010)) est appliquée, qui est connue pour être efficace dans de telles situations, en faisant passer la complexité temporelle de l'opération de fusion à $O(n_1 \log n_2)$, lorsque $n_1 << n_2$. Le résultat est finalement renvoyé dans un nouveau *conteneur-tableau*.

3.3.2 Accès aléatoires

Une opération d'accès aléatoire sur un *Roaring bitmap* commence par effectuer une recherche binaire sur les valeurs des clés de l'index de premier niveau. Si une entrée est trouvée, une deuxième recherche est lancée au niveau conteneur, soit par un accès direct dans le cas d'un *conteneur-bitmap*, ou par une recherche binaire si c'est un *conteneur-tableau*. Cette opération s'exécute en temps $O(\log_2 n)$, où n vaut au plus 2^{16} si l'on gère des entiers de 32 bits.

3.3.3 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 par rapport aux valeurs de leurs clés. À chaque itération, 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 de conteneurs triés dans un ordre croissant de cardinalités. Si la file débute par un conteneur bitmap, l'algorithme commence 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 après chaque fusion et le prochain dans la séquence, mais avec la différence que la fusion, cette fois-ci, se fera en place (*in-place*). Plus précisément, le résultat d'une union entre deux conteneurs sera stocké dans le *conteneur-bitmap* calculé à l'étape précédente, en évitant d'en générer un de nouveau à chaque fusion de deux conteneurs. Le même processus se poursuit jusqu'à la fin de la séquence de conteneurs de même clé.

Dans le cas échéant, lorsqu'un *conteneur-tableau* commence une telle séquence de conteneurs comportant une clé équivalente, 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 en place.

Le conteneur obtenu d'une opération de fusion de conteneurs de même clé sera ensuite inséré, avec sa clé, dans le *Roaring bitmap* résultant. À chaque fois qu'un conteneur est retiré de la file, son suivant (s'il y en a) dans son *Roaring bitmap* est inséré dans la file. Les traitements précédents 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, un espace d'adressage dans la mémoire virtuelle du programme en cours d'exécution est réservé afin de créer une corrélation octet par octet entre cette zone mémoire et une portion du fichier physique mappé et qui est stocké sur le disque (sans charger les données du 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 économiser 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 dé-sérialisation. Pour rendre *Roaring bitmap* opérationnel dans un contexte de *memory-mapping*, la librairie Java *Roaring bitmap* a été étendue en y incluant de nouvelles classes qui proposent des méthodes permettant d'effectuer plusieurs types de traitements (sérialisation/dé-sérialisation de données, opérations logiques, accès aléatoires, etc.) avec des bitmaps mappés en mémoire principale.

3.5 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™-8150 à 8 cœurs avec une fréquence d'horloge de 3,60 GHz et 16 GB de mémoire RAM. Pour *Concise* et *WAH*, on utilise la version 2.2 de la librairie Java ConciseSet (Colantonio, 2010). On se sert aussi de la composante Java *BitSet* pour représenter des bitmaps non compressés. Les temps de traitement sont donnés en nanosecondes. Le serveur JVM à 64 bits d'Oracle est utilisé sur un système Linux Ubuntu 12.04.1 LTS. Le code source de la librairie *Roaring bitmap* est librement accessible sur Roaring's team (2014c).

Le langage de programmation Java a été choisi principalement du fait de la disponibilité de plusieurs autres librairies de représentation de bitmaps implémentées avec ce langage, comme celles évaluées au cours de ce travail : Concise, WAH et `BitSet`, ou

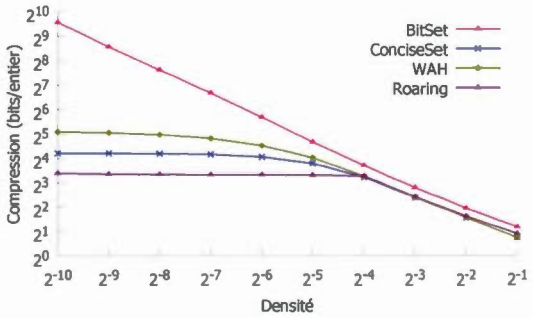
d'autres, telle que la librairie JavaEWAH. [1] En plus, sachant que de nos jours plusieurs systèmes de traitement de données massives sont implémentés en Java, comme : Hadoop (Shvachko *et al.*, 2010), Hive (Thusoo *et al.*, 2010), Druid (Yang *et al.*, 2014), etc., l'utilisation de ce langage de programmation offre pas mal d'opportunités pour intégrer la librairie *Roaring bitmap* à l'un de ces systèmes, ce qui permettrait d'évaluer ses performances dans un tel contexte.

3.5.1 Données synthétiques

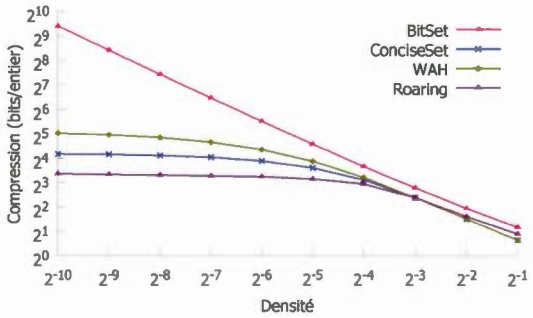
On a reproduit les expériences décrites dans Colantonio et Di Pietro (2010) en incluant des comparaisons avec *Roaring bitmap*. Deux ensembles de 10^5 entiers sont générés lors de chaque essai avec deux types de distributions : Uniforme et $\mathrm{Beta}(0{,}5, 1)$ discrétisée. Les quatre techniques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

Au départ, un test est exécuté 100 fois sans tenir compte des temps d'exécution ; nombre d'itérations assez suffisant pour permettre à la JVM d'apporter des optimisations au code. Cette première phase est connue sous le nom de l'étape d'échauffement (*warming-up phase*) qui permet d'éviter de prendre en considération les surcoûts de l'initialisation des essais dans les mesures capturées et elle est essentielle pour la pertinence des expériences. Puis, chaque essai est répété 100 fois, avant de présenter la moyenne des résultats obtenus lors de ces 100 dernières répétitions. Le code de ces bancs d'essai est

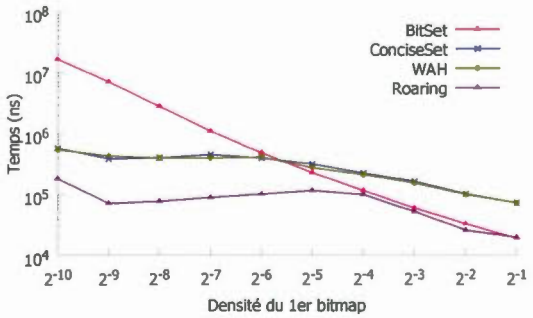
1. https ://github.com/lemire/javaewah



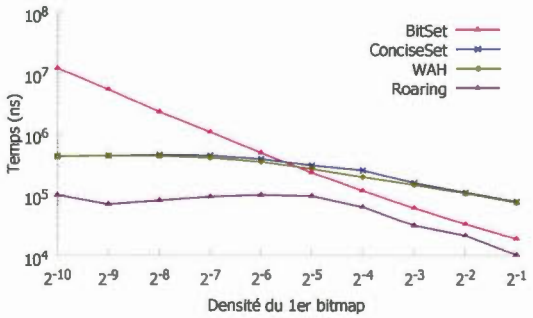
(a) Compression : distribution uniforme



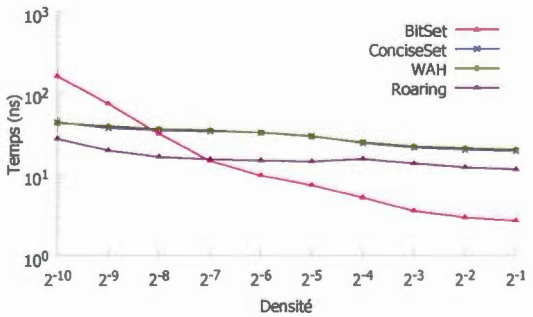
(b) Compression : distribution Beta(0.5, 1)



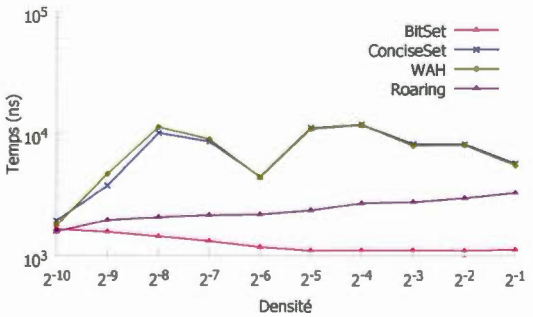
(c) Intersection : distribution uniforme



(d) Intersection : distribution Beta(0.5, 1)



(e) Insertion : distribution uniforme



(f) Suppression : distribution Beta(0.5, 1)

Figure 3.3: Compression et temps d'exécution

librement accessible sur internet : Roaring's team (2014b).

Les Figures 3.3a et 3.3b montrent le nombre moyen de bits par entier que chaque technique utilise pour stocker la première des deux listes d'entiers de 32 bits générées. Sur des bitmaps de faibles densités, *Roaring bitmap* ne consomme que $\approx 50\,\%$ d'espace

mémoire par rapport à *Concise* et $\approx 25\,\%$ par rapport à *WAH*. Avec la croissance de la valeur de max sur les densités faibles, les entiers générés tendent à devenir de plus en plus grand, poussant *BitSet* à allouer d'importants espaces de stockage afin de représenter les larges entiers.

Les tests suivants rapportent les temps moyens consommés par chaque technique pour effectuer une intersection entre deux listes d'entiers (voir Figures 3.3c et 3.3d). Les ensembles d'entiers sont représentés par deux bitmaps de densités asymétriques (l'un ayant une plus forte densité que l'autre), où la densité d_2 du deuxième bitmap est calculée à partir de la densité d du premier bitmap comme suit : $d_2 = (d-1) * x + d$; x étant un réel généré pseudo-aléatoirement de [0,1). Cette formule nous permet d'obtenir un deuxième bitmap aléatoirement plus dense. Le résultat d'une intersection est retourné dans un nouveau bitmap. Puisque *BitSet* ne supporte que des opérations en-place, on commence par copier le premier bitmap.

Tel que constaté, *Roaring bitmap* est entre 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 bitmaps encodés avec un *equality-encoding* (Stockinger et Wu, 2008) ou un *range-encoding* (Chan et Ioannidis, 1998b) 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 3.3e montre les résultats obtenus sur une distribution de données uniforme. Puisque *WAH* et *Concise* nécessitent de décoder séquentiellement les bitmaps compressés avant d'insérer chaque nouvel élément, ils mettent un temps linéaire par rapport à la taille des bitmaps compressés. Ce qui est beaucoup plus lent comparé à *Roaring bitmap*, qui effectue cette tâche en un temps logarithmique par rapport au nombre d'entrées de l'index de premier niveau et des *conteneur-tableaux* (dans les cas de densités faibles). L'allocation d'espaces ralentit *BitSet* sur les basses densités, mais il finit par accélérer sur des données denses, dépassant de beaucoup les autres techniques. Ceci s'explique par la diminution des taux d'allocations d'espaces, et du fait que des accès directs suffisent pour mettre à jour les bits. On n'a pas présenté les résultats obtenus sur une distribution $\mathrm{Beta}(0{,}5, 1)$, car des comportements similaires y ont été observés.

Dans le dernier test, on mesure le temps moyen consommé par chaque technique pour supprimer un élément sélectionné aléatoirement d'un ensemble d'entiers. Les résultats obtenus sur une distribution $\mathrm{Beta}(0{,}5, 1)$ sont présentés à la Figure 3.3f. On voit clairement que *Roaring bitmap* est beaucoup plus performant comparé aux deux autres techniques de compression bitmap. Grâce à ses accès directs, *BitSet* affiche les meilleures performances sur ces essais. Des résultats similaires ont été observés sur des données de distribution uniforme.

3.5.2 Données réelles

Les techniques d'indexation précédentes ont été comparées à nouveau sur 4 ensembles de données réelles (voir le Tableau 3.1) précédemment utilisés par Lemire *et al.* (2012) : *Census1881* (de recherche en démographie historique, 2009), *CensusIncome* (Frank et 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 [2] 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 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.

Les ensembles de données sont gardés dans leur ordre original (non trié). Tout d'abord, un index bitmap est construit sur chaque ensemble. Par la suite, des bitmaps sont sélectionnés avec 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 3.1 présente les caractéristiques des bitmaps sélectionnés.

Un 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)

---

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

	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

Tableau 3.1: Caractéristiques des bitmaps sélectionnés

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 effectuées avec des calculs en place. Une étape d'échauffement est tout d'abord lancée pour bénéficier au mieux de l'optimiseur de code de la JVM. Ensuite, chaque essai est répété un certain nombre de fois avant de présenter les temps moyens obtenus. Le code de ces essais est librement accessible sur le web (Roaring's team, 2014a).

Le Tableau 3.2a montre le facteur de croissance de l'espace mémoire lorsqu'on remplace *Roaring bitmap* par *BitSet*, *WAH* et *Concise*. Les valeurs au-dessus de 1,0 indiquent de combien *Roaring bitmap* devance la technique correspondante. Les Tableaux 3.2b–3.2c présentent les facteurs de croissance des temps de calcul des opérations logiques.

Roaring bitmap a été plus compact que *Concise* et *WAH* sur presque tous les ensembles de données testés, consommant jusqu'à deux fois moins d'espace mémoire que ces deux méthodes, excepté pour l'ensemble *Wikileaks*, qui contient de larges plages de bits à 1 qui sont incompressibles par *Roaring bitmap*. Pour ce qui est des temps de calcul des opérations logiques, *Roaring bitmap* a été plus performant que les deux solutions de compression bitmap sur tous les ensembles de données, 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.

	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

(a) Facteurs de croissance d'espaces mémoires lorsque *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

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

	CENSUS1881	CENSUSINCOME	WIKILEAKS	WEATHER
Concise	54,41	4,73	2,09	5,04
WAH	47,72	4,25	2,02	4,46
BitSet	27,06	0,24	3,57	0,38

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

Tableau 3.2: Résultats sur des données réelles

3.5.3 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 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 SGBD *Druid* (Metamx, 2015). Cette version représente une extension de la librairie ConciseSet 2.2 (Colantonio, 2010) 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 sur chaque ensemble de données. Ensuite, un *Roaring bitmap* et un bitmap *Concise* sont créés avec les entiers de chaque liste, donnant un total de 200 bitmaps compressés avec chacune des deux méthodes de compression bitmap. Les 200 bitmaps des deux techniques sont ensuite sérialisés séparément sur deux fichiers disque différents, qui seront par la suite mappés en mémoire principale. Lors d'un test, on mesure 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 compressés. 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 les temps moyens calculés. Le code de ces essais est librement accessible en ligne : Chambi (2014).

Les Tableaux 3.3a–3.3b montrent les résultats des essais évaluant la consommation

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

(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	104,0	104,0	104,0
Concise	80,0	80,0	80,0

(b) Espace moyen (en octets/bitmap) réservé en mémoire principale pour les pointeurs d'un des 200 bitmaps sérialisés sur disque et compressés avec *Roaring bitmap* ou *Concise*

Tableau 3.3: Occupation mémoire avec du *memory-mapping* sur des données réelles

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

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

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

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

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

(c) Temps moyen en millisecondes pour récupérer les positions des bits à 1 de 200 bitmaps compressés avec *Roaring bitmap* ou *Concise*

Tableau 3.4: Temps de traitements avec du *memory-mapping* sur des données réelles

d'espaces mémoire avec chaque solution. 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 compacts 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 méta-données 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 pointeurs de tailles statiques. Cela explique les résultats observés sur le Tableau 3.3b qui montre une même et infime quantité d'espace mémoire consommée par chaque modèle de bitmap compressé sur les différents ensembles de données. Toutefois, *Roaring bitmap* affiche un léger surplus de quelques octets par rapport à *Concise*, dû à sa structure un peu plus complexe.

Les Tableaux 3.4a–3.4c affichent les temps moyens que prend chaque méthode de compression bitmap pour effectuer différents types de traitements. Le Tableau 3.4a présente les résultats obtenus pour le calcul de l'union de 200 bitmaps compressés. Pour ces tests, la méthode d'union horizontale présentée précédemment à la Section 3.3.3 a été utilisée pour *Roaring bitmap* afin de calculer l'union des 200 bitmaps. Cette stratégie d'union a montré qu'elle pouvait améliorer significativement les temps de calculs comparé à l'union classique (voir le Tableau 3.5). *Roaring bitmap* affiche de remarquables performances comparé à *Concise* sur les trois ensembles de données, allant jusqu'à ≈ 129 fois plus vite sur l'ensemble WEATHER.

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

Tableau 3.5: 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 de la Sous-section 3.5.3 ont été utilisés)

Le Tableau 3.4b 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 ≈ 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'accéder aux données sélectionnées par une requête. Le Tableau 3.4c 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.

3.6 Conclusion

Ce chapitre introduit une nouvelle technique de compression bitmap, nommée *Roaring bitmap*. Des tests synthétiques et réels nous ont permis de comparer les performances de *Roaring bitmap* avec deux autres techniques de compression bitmap 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, on vise à implémenter d'autres types d'algorithmes de recherche pour améliorer les temps des intersections. Culpepper et Moffat (2010) ont montré qu'une recherche *Golomb* (Golomb, 1966) permettait d'exécuter efficacement des intersections entre *posting lists*. On souhaite aussi améliorer la vitesse des accès aléatoires en adoptant une structure de données plus compacte sur le premier niveau de l'index, tel un bitmap offrant des accès directs. On envisage également d'expérimenter la nouvelle approche sur des bancs d'essai décisionnels, comme le *Star Schema Benchmark* (O'Neil *et al.*, 2009).

Druid (Yang *et al.*, 2014) est un SGBD analytique à code libre, orienté colonne, distribué, qui permet d'effectuer des analyses multidimensionnelles complexes sur des quantités massives de données temporelles, en des temps concurrentiels par rapport à d'autres SGB connus dans la littérature, tels que Hadoop (Shvachko *et al.*, 2010). Pour assurer des traitement rapides, *Druid* fait usage, entre autres, de bitmaps compressés avec *Concise* pour indexer les données de base. Cet SGBD sérialise la grande partie de ses données sur des fichiers disque et les manipule avec du *memory-mapping*. Parmi les données sérialisées, figure des collections d'index bitmap compressés.

Les résultats positifs obtenus avec *Roaring bitmap* lorsque comparé à *Concise* sur des données réelles et dans un contexte de *memory-mapping*, nous laisse prévoir d'intégrer *Roaring bitmap* comme une technique de compression bitmap au SGBD *Druid*, et d'investiguer les performances globales du système 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 auxquelles *Druid* améliore les performances à l'aide de bitmaps compressés. Depuis, ce projet a été réalisé et le Chapitre V de cette thèse en fait la présentation.

CHAPITRE IV

ROARING 64 BITS

Ce chapitre est tiré de l'article ci-dessous avec quelques contenus qui ont été ajoutés, modifiés et/ou supprimés :

Chambi, S., Lemire, D. et Godin, R. (2016a). Nouveaux modèles d'index bitmap compressés à 64 bits. Dans 12e journées francophones sur les Entrepôts de Données et l'Analyse en Ligne (EDA'16), volume RNTI-B-12, 1 – 16., Aix-en-Provence, France. RNTI.

4.1 Introduction

La quantité d'informations générées quotidiennement de nos jours ne cesse de croître à une vitesse phénoménale. Pour indexer de telles masses de données, la majorité des librairies de représentation d'index bitmap proposées en code libre (*Open Source*) 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} entrées. En réponse à cette problématique rencontrée tant dans le milieu industriel que scientifique, nous avons introduit trois nouveaux modèles d'index bitmap compressés (Chambi *et al.*, 2016a) basés sur le modèle *Roaring bitmap* et qui peuvent indexer jusqu'à 2^{64} entrées ; un seuil plutôt raisonnable par rapport à la majorité des collections de données traitées de nos jours. Ce chapitre introduit ces trois différentes librairies et présente les bancs

d'essai mis en œuvre pour comparer les performances de ces nouvelles techniques avec d'autres collections Java, ainsi qu'avec la solution classique (sans compression) d'indexation bitmap : OpenBitSet, supportant jusqu'à $64 \times 2^{32} - 1$ entrées et qui est adoptée par le moteur de recherche Lucene (Apache, 2012).

Le chapitre est organisé comme suit : la Section 4.2 présente 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 4.3. Le chapitre se termine avec une conclusion à la Section 4.4.

4.2 Modèles d'index bitmap compressés

Les modèles d'index bitmap compressés (Chambi *et al.*, 2016a) proposés dans ce chapitre représentent chaque bit à 1 dans un bitmap par un entier de 64 bits qui indique sa position débutant possiblement à 0. Ces modèles se comptent au nombre de trois et sont définis dans les sous-sections suivantes :

4.2.1 RoaringTreeMap

Le modèle *RoaringTreeMap* combine un arbre Java TreeMap avec la structure *Roaring bitmap* pour indexer un ensemble d'entiers à 64 bits représentant les positions des bits positifs d'un bitmap. Un TreeMap est une structure de données faisant partie du paquetage des collections Java et met en œuvre l'arbre rouge-noir, qui est un arbre binaire de recherche équilibré. 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 à 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 à 32 bits, et d'un *Roaring bitmap*. *RoaringTreeMap* regroupe un ensemble d'entiers à 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 à 64 bits dans un *RoaringTreeMap* commence par effectuer un accès aléatoire dans l'arbre, qui consiste en un parcours de l'arbre en partant de la racine 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 dans le *Roaring bitmap* associé à ce nœud.

Ainsi, un accès aléatoire à un entier de 64 bits stocké dans un *RoaringTreeMap* nécessitera un temps de $O(\log n)$ pour effectuer un premier parcours dans l'arbre, où n représente le nombre de nœuds dans le *RoaringTreeMap*, plus le temps pour accéder aux 32 bits de poids faible de l'entier dans le *Roaring bitmap* associé au nœud trouvé précédemment, 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 garantit que les nœuds de l'arbre puissent être toujours par-

courus dans l'ordre croissant des valeurs de leurs clés dans un temps de complexité linéaire par rapport au nombre de nœuds dans l'arbre. Ceci combiné au tri par ordre croissant des entiers à 32 bits maintenus au sein des *Roaring bitmaps*, cela implique que le modèle *RoaringTreeMap* permet 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.

L'union de deux *RoaringTreeMaps*

L'algorithme 1 présente le pseudo-code qui calcule l'union de deux *RoaringTreeMaps*. Afin de traiter efficacement une telle opération, deux algorithmes de complexités temporelles différentes ont été mis en œuvre. Chacun des deux algorithmes est employé lors d'un scénario approprié qui dépend du type des données en entrée. Une solution similaire est utilisée au sein des optimiseurs de systèmes de gestion de bases de données pour optimiser le processus d'exécution de jointures relationnelles (Godin, 2012).

L'union de deux *RoaringTreeMaps* requiert, en partie, de procéder une fusion entre deux arbres rouge-noirs. Plusieurs solutions ont été proposées dans la littérature pour effectuer une telle fusion de manière efficace : (Brodal *et al.*, 2006), (Booth, 1992), etc. Toutefois, l'étude de cette problématique reste en dehors de la portée de cette thèse.

Les deux algorithmes prennent deux *RoaringTreeMaps* en entrée, et renvoient le résultat de l'union dans un nouveau *RoaringTreeMap*. Le premier algorithme, présenté par le pseudo-code de l'Algorithme 2, est utilisé lorsque l'un des deux arbres en entrée

Algorithm 1 Algorithme qui calcule l'union de deux *RoaringTreeMaps*.

1: **entrée** : deux *RoaringTreeMaps* A et B
2: **sortie** : un nouveau *RoaringTreeMap* C représentant l'union de A et B
    {$X.size$ représente le nombre de nœuds dans l'arbre X}
3: **if** $A.size < B.size$ **then**
4: $smallTree \leftarrow A$
5: $bigTree \leftarrow B$
6: **else**
7: smallTree$\leftarrow B$
8: bigTree$\leftarrow A$
9: **end if**
10: **if** $smallTree.size \times log_2(bigTree.size) < bigTree.size + smallTree.size$ **then**
11: $C \leftarrow$ linearithmicUnion($smallTree$, $bigTree$)
12: **else**
13: $C \leftarrow$ linearUnion($smallTree$, $bigTree$)
14: **end if**
15: **return** C

renferme beaucoup plus de nœuds que le second. Cet algorithme commence par copier le plus grand arbre dans le résultat à la ligne 3, puis parcourt le plus petit arbre en insérant chacun de ses nœuds dans l'arbre résultant aux lignes 4 et 5. Si la clé d'un nœud du petit arbre existe déjà dans l'un des nœuds de l'arbre résultant, un OU logique est exécuté entre les *Roaring bitmaps* associés aux deux nœuds de clés équivalentes, et le résultat sera stocké dans un nouveau *RoaringBitmap* qui remplacera le *Roaring bitmap* du nœud de l'arbre résultant. Dans un cas contraire, si aucune clé équivalente à celle de l'un des nœuds du petit arbre n'est trouvée dans l'arbre résultant, une copie de ce nœud incluant sa clé et son *Roaring bitmap* sera insérée dans l'arbre résultant. Ces opérations se poursuivent jusqu'à avoir parcouru tous les nœuds du plus petit arbre.

Le temps d'exécution de cet algorithme dépend du temps nécessaire pour copier le plus grand *RoaringTreeMap*, ce qui se fait efficacement en un temps de $\Theta(n_1)$, où n_1 représente le nombre de nœuds dans le plus grand *RoaringTreeMap*. Ensuite, il faut ajouter le temps pour parcourir le plus petit arbre et rechercher-insérer chacun de ses nœuds dans l'arbre résultant. Ce qui se fait en un temps linéarithmique de $O(n_2 \log(n_1+n_2))$,

Algorithm 2 LinearithmicUnion : algorithme qui fusionne deux *RoaringTreeMaps*. Les clés des deux arbres sont comparés en un temps linéarithmique.

1: **entrée** : deux *RoaringTreeMaps* A et B, avec $A.size < B.size$
2: **sortie** : un nouveau *RoaringTreeMap* C représentant l'union de A et B
3: $C \leftarrow B.clone$ {$X.clone$ crée une nouvelle copie du *RoaringTreeMap* X}
4: **for** each node e in A **do**
5: $\quad C.insert(e.key, e.RoaringBitmap)$
6: **end for**
7: **return** C

où n_2 représente le nombre de nœuds dans le plus petit arbre. Aussi, le coût relatif aux différentes opérations OU logiques calculées entre des *Roaring bitmaps* doit également être pris en considération.

Le deuxième algorithme de calcul de l'union, dont le pseudo-code est affiché par l'Algorithme 3, est utilisé lorsque les deux *RoaringTreeMaps* en entrée renferment presque le même nombre de nœuds. Cet algorithme commence par allouer un nouveau tableau dynamique vide à la ligne 3 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 de leurs clés par un algorithme de fusion à la ligne 6. 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 (aux lignes 8, 10, 14, 16, 27 et 31), 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 *OR* 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 à la ligne 20. 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.

Algorithm 3 LinearUnion : algorithme qui fusionne deux *RoaringTreeMaps*. Les clés des deux arbres sont comparés en un temps linéaire.

entrée : deux *RoaringTreeMaps* A et B
sortie : un nouveau *RoaringTreeMap* C représentant l'union de A et B
$array \leftarrow$ a new dynamic array
$n_1 \leftarrow A.NextNode$ {$X.NextNode$ retourne le prochain nœud non accédé de l'arbre X en suivant l'ordre croissant des clés}
$n_2 \leftarrow B.NextNode$
while $A.HasNext$ and $B.HasNext$ **do** {$X.HasNext$ Retourne faux si tous les nœuds de X ont déjà été parcourus. Renvoie vrai sinon}
  if $n_1.key < n_2.key$ **then**
    $array.insert(n_1.key, n_1.RoaringBitmap)$
    if $A.HasNext = faux$ **then**
      $array.insert(n_2.key, n_2.RoaringBitmap)$
    end if
    $n_1 \leftarrow A.NextNode$
  else if $n_1.key > n_2.key$ **then**
    $array.insert(n_2.key, n_2.RoaringBitmap)$
    if $B.HasNext = faux$ **then**
      $array.insert(n_1.key, n_1.RoaringBitmap)$
    end if
    $n_2 \leftarrow B.NextNode$
  else
    $array.insert(n_1.key, n_2.RoaringBitmap\ OR\ n_1.RoaringBitmap)$
    $n_1 \leftarrow A.NextNode$
    $n_2 \leftarrow B.NextNode$
  end if
end while

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

25: **while** $A.HasNext$ **do**
26: $n_1 \leftarrow A.NextNode$
27: $array.insert(n_1.key, n_1.RoaringBitmap)$
28: **end while**
29: **while** $B.HasNext$ **do**
30: $n_2 \leftarrow B.NextNode$
31: $array.insert(n_2.key, n_2.RoaringBitmap)$
32: **end while**
33: $C \leftarrow$ a new *RoaringTreemap* built from $array$
34: **return** C

nécessaire pour calculer les OU logiques entre *Roaring bitmaps*.

L'intersection de deux *RoaringTreeMaps*

Une opération d'intersection prend deux *RoaringTreeMaps* en entrée et renvoie le résultat dans un nouveau *RoaringTreeMap*. Pour calculer l'intersection de deux *RoaringTreeMaps*, deux algorithmes de complexités temporelles différentes ont été développés. Tout comme pour l'union, chacun de ces deux algorithmes est approprié pour un scénario dépendant du type des données en entrée. Le premier algorithme qui ressemble à l'Algorithme 2 est adopté lorsque l'un des deux *RoaringTreeMaps* en entrée contient beaucoup plus de nœuds que le second, et consiste essentiellement à parcourir le petit arbre en vérifiant l'existence de chacune de ses clés dans le grand arbre. Ainsi, si deux nœuds, appartenant chacun à l'un des deux arbres, possèdent une même clé, une opération ET logique sera exécutée entre leurs *Roaring bitmaps* associés, et le résultat sera stocké dans un nouveau *RoaringBitmap*. Par la suite, un nouveau nœud contenant le *Roaring bitmap* obtenu et une clé de valeur égale à celle des deux clés équivalentes trouvées précédemment sera ajouté au *RoaringTreeMap* final.

Le temps d'exécution de cet algorithme dépend du temps nécessaire pour parcourir le plus petit arbre en vérifiant l'existence de chacune de ses clés dans le plus grand arbre, ce qui se fait en un temps de $O(n_1 \log(n_2))$, où n_1 représente le nombre de nœuds

dans le plus petit des deux arbres et n_2 le nombre de nœuds dans le plus grand arbre. Il faut ajouter à cela le temps pour calculer les possibles opérations ET logiques entre des *Roaring bitmaps*.

Le deuxième algorithme qui est presque similaire à l'algorithme 3 est lancé lorsque les deux *RoaringTreeMaps* en entrée ont un nombre de nœuds presque similaire. Celui-ci 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. En plus, il faut ajouter aussi le temps consommé par les calculs des ET logiques entre *Roaring*

bitmaps.

4.2.2 RoaringTwoLevels

Le 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 celles de type COUNT.

Nommons les 48 bits de poids fort parmi les 64 bits d'une entrée par les bits communs de l'entrée, et les 16 bits de poids faible restants dans les 64 bits d'une entrée 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.

Pour accéder à un entier de 64 bits quelconque stocké dans la structure de données, une première opération de recherche binaire à temps logarithmique 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.

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 4 présente le pseudo-code effectuant ce calcul. L'algorithme commence par parcourir les entrées des deux tableaux de premier niveau des deux *RoaringTwoLevels* en entrée à la ligne 6. À chaque itération, les valeurs des bits communs de chacune des deux entrées courantes sont comparées. Si les valeurs sont différentes, l'algorithme insère (aux lignes 8, 11, 21 et 25) une copie de l'entrée dont les bits communs sont de plus petite valeur et une copie de son conteneur dans le *RoaringTwoLevels* résultant. Ensuite, l'algorithme avance d'une position sur le tableau de premier niveau de l'entrée copiée.

Dans le cas où les valeurs des bits communs des deux entrées de premier niveau comparées lors d'une itération sont équivalentes, l'algorithme ajoute, à la ligne 14, une nouvelle entrée au tableau de premier niveau du *RoaringTwoLevels* résultant. L'entrée ajoutée contient 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 à la ligne 15. 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 *RoaringTwo-*

Levels donnés en entrée.

Algorithm 4 Algorithme qui calcule l'union de deux *RoaringTwoLevels*.

entrée : deux *RoaringTwoLevels* A et B
sortie : un nouveau *RoaringTwoLevels* C représentant l'union de A et B
$C \leftarrow$ a new *RoaringTwoLevels*
$pos_1 \leftarrow 0$
$pos_2 \leftarrow 0$
while $pos_1 < A.size$ and $pos_2 < B.size$ **do** {$X.size$ représente le nombre d'éléments dans le tableau de premier niveau du *RoaringTwoLevels* X}
  if $(A[pos_1].key >> 16) < (B[pos_2].key >> 16)$ **then** {$>>$ indique l'opération de décalage non signé de bits à droite.}
    $C.insert(A[pos_1].key, A[pos_1].container)$
    $pos_1 \leftarrow pos_1 + 1$
  else if $(A[pos_1].key >> 16) > (B[pos_2].key >> 16)$ **then**
    $C.insert(B[pos_2].key, B[pos_2].container)$
    $pos_2 \leftarrow pos_2 + 1$
  else{$(A[pos_1].key >> 16) = (B[pos_2].key >> 16)$}
    $C.insert(A[pos_1].key, (A[pos_1].container\ OR\ B[pos_2].container))$
    Set the cardinality of the last entry added to C
    $pos_1 \leftarrow pos_1 + 1$
    $pos_2 \leftarrow pos_2 + 1$
  end if
end while

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 complexité linéaire par rapport au nombre des éléments dans les deux tableaux. En second lieu, les performances de l'algorithme dépendent aussi 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 se référer au chapitre introduisant la technique *Roaring bitmap*, ou bien aux références suivantes : Chambi *et al.* (2016d, 2014).

```
20: while pos_1 < A.size do
21:     C.insert(A[pos_1].key, A[pos_1].container)
22:     pos_1 ← pos_1 + 1
23: end while
24: while pos_2 < B.size do
25:     C.insert(B[pos_2].key, B[pos_2].container)
26:     pos_2 ← pos_2 + 1
27: end while
28: return C
```

L'intersection de deux *RoaringTwoLevels*

L'algorithme d'intersection ressemble à celui de l'union et 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, à la différence de l'algorithme d'union, celui-ci 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 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 en $O(log_2 n)$.

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 conte-

neur. 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 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 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.*, 2016d, 2014).

4.2.3 LazyRoaring

Nous avions vu précédemment avec le modèle *Roaring bitmap* 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 traitement effectués sur un *Roaring bitmap*. En s'inspirant de cette idée qui a été avantageuse pour *Roaring bitmap*, on a introduit ce nouveau modèle de compression bitmap 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.

L'avantage d'utiliser deux niveaux d'index revient à la possibilité d'éviter l'accès à plusieurs conteneurs pouvant renfermés en tout jusqu'à 2^{32} entiers, lorsqu'une entrée de premier niveau n'est pas considérée comme valide lors d'un quelconque traitement réalisé sur un *LazyRoaring*. Ce type d'index pourrait être très bénéfique sur des ensembles de données triées, où les bitmaps sont formés de plusieurs suites de bits à 1 séparées par de longues séquences de 0.

Accès aléatoires dans un *LazyRoaring*

Un accès aléatoire consiste en le parcours d'un *LazyRoaring* afin d'accéder à un entier de 64 bits possiblement stocké dans la structure de données. Cette opération commence par parcourir le tableau de premier niveau afin de trouver une entrée de valeur égale aux 32 bits de poids fort de l'entier à accéder. Cette première opération de recherche est réalisée avec un algorithme de recherche binaire, dans 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 avec un algorithme de recherche binaire 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 dans le 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, telle que discuté au niveau du chapitre introduisant *Roaring bitmap*, et qui nécessite, au meilleur cas, un accès en temps constant lorsque le conteneur est un bitmap, et au pire des cas, d'effectuer une recherche binaire sur le tableau trié représentant le conteneur dans un temps de complexité logarithmique par rapport au nombre des éléments du tableau.

Par conséquent, un accès aléatoire dans un *LazyRoaring* consomme, au plus, un temps de complexité logarithmique par rapport au nombre d'entrées des tableaux accédés à chaque niveau de la structure de données.

L'union de deux *LazyRoarings*

Une opération d'union prend deux *LazyRoarings* en entrée et retourne un nouveau *LazyRoaring*. L'Algorithme 5 affiche le pseudo-code de cette opération. Tout d'abord, l'algorithme commence par itérer sur les deux tableaux de premier niveau des deux bitmaps à la ligne 6 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 (aux lignes 8, 12, 22 et 27) 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 traitement et l'espace mémoire consommé durant une opération d'union. Cette optimisation est connue dans la littérature sous le nom de *copy-on-write* (Wikipedia, 2015b). 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 à comparer lors d'une itération possèdent des entiers de 32 bits équivalents, l'algorithme ajoute à la ligne 16 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 le résultat retourné par la méthode $FirstLevelEntriesUnion$ invoquée à la ligne 16 de l'algorithme, qui consiste en 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.

Algorithm 5 Algorithme qui calcule l'union de deux *LazyRoarings*.

1: **entrée** : deux *LazyRoarings* A et B
2: **sortie** : un nouveau *LazyRoaring* C représentant l'union de A et B
3: $C \leftarrow$ a new *LazyRoaring*
4: $pos_1 \leftarrow 0$
5: $pos_2 \leftarrow 0$
6: **while** $pos_1 < A.size$ and $pos_2 < B.size$ **do** {$X.size$ représente le nombre d'éléments dans le tableau de premier niveau du *LazyRoaring* X}
7: **if** $(A[pos_1].key) < (B[pos_2].key)$ **then**
8: $C.insert(A[pos_1])$ {l'objet $A[pos_1]$ n'est pas copié lors de l'insertion.}
9: $C.last.shared \leftarrow true$ {pour indiquer que le dernier élément inséré est partagé entre plusieurs *LazyRoarings*.}
10: $pos_1 \leftarrow pos_1 + 1$
11: **else if** $(A[pos_1].key) > (B[pos_2].key)$ **then**
12: $C.insert(B[pos_2])$ {l'objet $B[pos_2]$ n'est pas copié lors de l'insertion.}
13: $C.last.shared \leftarrow true$ {pour indiquer que le dernier élément inséré est partagé entre plusieurs *LazyRoarings*.}
14: $pos_2 \leftarrow pos_2 + 1$
15: **else**$\{(A[pos_1].key) = (B[pos_2].key)\}$
16: $C.insert(FirstLevelEntriesUnion(A[pos_1], B[pos_2])$
17: $pos_1 \leftarrow pos_1 + 1$
18: $pos_2 \leftarrow pos_2 + 1$
19: **end if**
20: **end while**

21: **while** $pos_1 < A.size$ **do**
22: $C.insert(A[pos_1])$ {l'objet $A[pos_1]$ n'est pas copié lors de l'insertion.}
23: $C.last.shared \leftarrow true$
24: $pos_1 \leftarrow pos_1 + 1$
25: **end while**
26: **while** $pos_2 < B.size$ **do**
27: $C.insert(B[pos_2])$ {l'objet $B[pos_2]$ n'est pas copié lors de l'insertion.}
28: $C.last.shared \leftarrow true$
29: $pos_2 \leftarrow pos_2 + 1$
30: **end while**
31: **return** C

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 la valeur des 16 bits de la plus petite des deux entrées de deuxième niveau comparées et un pointeur (*copy-on-write*) vers son conteneur, lorsque les deux entrées possèdent des entiers de 16 bits différents. Dans le cas contraire, une nouvelle entrée de deuxième niveau est ajoutée au bitmap résultant. Cette entrée contient un entier de 16 bits équivalent à ceux des deux entrées de deuxième niveau évaluées, 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 le chapitre introduisant *Roaring bitmap*.

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.

La stratégie *copy-on-write* permet à un même objet (entrée de deuxième niveau ou conteneur) de figurer dans la représentation de plusieurs bitmaps en même temps. En effet, lorsqu'un objet appartenant à un certain bitmap nécessite d'être ajouté à la représentation d'un autre bitmap, la solution classique stipule de créer une copie entière de l'objet en question, puis d'affecter cette copie au bitmap correspondant. La stratégie *copy-on-write* vise à éviter la création d'une copie entière d'un objet partagé, en fournissant plutôt un pointeur vers cet objet pour chaque bitmap l'ayant dans sa représentation. La création d'une copie entière est retardée jusqu'à ce qu'une modification d'un objet partagé ait a être effectuée dans l'un des bitmaps qui le pointent. Ainsi, l'avantage principal du *copy-on-write* réside dans le fait que si un bitmap pointant un objet partagé n'a jamais à le modifier, la copie de l'objet ne sera jamais effectuée. Cette stratégie promet de remarquables gains en matière de temps de traitements et de consommation d'espace mémoire sur de grands ensembles de bitmaps dédiés à des lectures seules, tels

que rencontrés très souvent dans les entrepôts de données.

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 $O(n_1 + n_2)$, 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, où chaque union prend un temps de $O(m_1 + m_2)$, 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 unions entre les conteneurs atteints, où le temps pour procéder chaque union dépend du type des deux conteneurs impliqués.

L'intersection de deux *LazyRoarings*

Une opération d'intersection prend deux *LazyRoarings* en entrée et retourne un nouveau *LazyRoaring*. L'algorithme est presque similaire à celui de l'union et 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, si 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'au lieu de calculer une intersection entre deux tableaux de deuxième niveau lorsque les deux entrées de deuxième niveau évaluées lors d'une itération possèdent les mêmes valeurs de 16 bits,

une intersection entre les deux conteneurs indexés par les deux entrées de deuxième niveau courantes est procédée à la place. 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 $O(n_1 + n_2)$, 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 $O(m_1 + m_2)$, 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 intersections entre les conteneurs accédés au troisième niveau, où le temps d'exécution de chaque intersection dépend du type de conteneurs utilisés (voir le chapitre introduisant *Roaring bitmap* pour plus de détails).

4.3 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 de compression bitmap avec d'autres structures de données implémentées en Java et qui sont utilisées pour représenter des ensembles d'entiers de 64 bits. Le Tableau 4.1 donne une brève description de chacune de ces structures de données avec l'espace mémoire que consomme chacun de leurs éléments.

Les expériences ont été mises en œuvre à l'aide de l'outil *Java Microbenchmark Har-*

ness (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).

4.3.1 Données synthétiques

Des expériences ont été conduites sur des bitmaps synthétiques générés en suivant deux types de distributions souvent rencontrées en pratique : uniforme et Zipf (Zipf, 1949), avec des densités d variantes de 10^{-9} à 10^{-4} par incrément d'un facteur de 10. 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 le générateur d'entiers ne lance 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. Ensuite, un nombre réel a est pseudo-aléatoirement généré de l'intervalle [0, 1[lors de chaque itération. Dans le cas d'une distribution uniforme, l'entier obtenu de $\lfloor a \times max \rfloor$ est ajouté à l'ensemble d'entiers résultant s'il n'y existe pas déjà. Tandis que dans le cas d'une distribution de Zipf, l'entier résultant de $\lfloor a^2 \times max \rfloor$ y est inséré s'il n'y est pas encore, 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 suivant une distribution uniform et Zipf. Pour les deux types de distributions, 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, deux bitmaps différents sont construits pour chaque structure de données à comparer. Par la suite, on mesure le temps moyen nécessaire pour construire un

bitmap, 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).

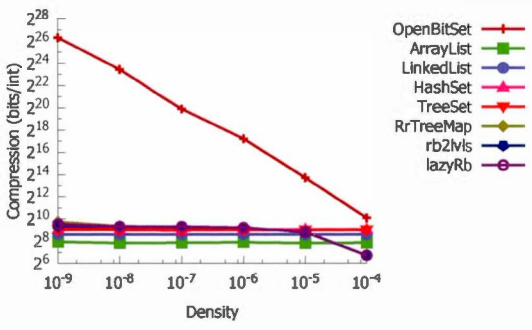
La quantité moyenne de l'espace mémoire requis pour le stockage d'un bitmap avec chaque type de structure de données a aussi été mesurée. Cependant, ces bancs d'essais n'ont pas été réalisés avec JMH, mais ont été mise en œuve par un programme Java (Chambi, 2015a).

Structure	Description
ArrayList	Un objet enveloppant un tableau dynamique, dont chaque entrée prend 28 octets : un pointeur de 4 octets vers un objet de type Long de 24 octets. Ces éléments ne sont pas forcément triés. Un même élément peut apparaître une ou plusieurs fois. Une insertion à la fin du tableau se fait dans un temps amorti constant.
LinkedList	Une liste doublement chaînée. Accepte que des éléments soient répétés plusieurs fois. Aucun ordre de tri n'est assuré. Une entrée de la liste requiert 48 octets : 12 octets utilisés par Java pour conserver les métadonnées (l'entête) d'un élément, 4 octets pour le pointeur vers l'objet Java Long de 24 octets, 4 octets pour le pointeur vers l'objet suivant et 4 octets pour le pointeur vers l'objet précédent. L'insertion d'un nouvel élément à une position donnée de la liste se fait dans un temps constant.

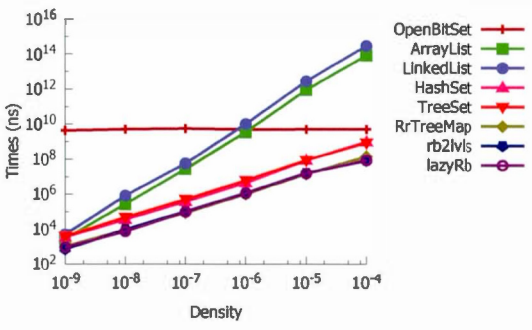
HashSet	Une table de hachage. Tous les éléments sont distincts. Aucun ordre de tri n'est assuré. Un élément nécessite 56 octets : 12 octets pour l'entête de l'entrée, 4 octets pour le stockage de la valeur de hachage de la clé, 4 octets pour le pointeur vers l'objet Java Long, 4 octets pour le pointeur vers l'objet valeur, 4 octets pour le pointeur vers la prochaine entrée de la table en cas de collision, 4 octets pour arrondir la taille d'une entrée à un multiple de 8 octets, et les 24 octets de l'objet Long pointé. Une insertion se fait dans un temps constant.
TreeSet	Un arbre rouge-noir de recherche binaire équilibré. Les clés sont de valeurs distinctes et sont triées dans leur ordre naturel ou dans l'ordre imposé par l'objet implémentant l'interface Comparator fourni lors de l'instanciation. Un nœud nécessite 64 octets : 12 octets pour l'objet entête, 1 octet pour représenter la couleur du nœud, 3 octets d'alignement, 4 octets pour le pointeur vers l'objet Java Long, 4 octets pour le pointeur vers l'objet valeur, 4 octets pour le pointeur vers le nœud fils gauche, 4 octets pour le pointeur vers le nœud fils droit, 4 octets pour le pointeur vers le nœud parent, et 4 octets d'alignement. Plus, les 24 octets de l'objet Long référencé. L'insertion d'un nouveau nœud se fait dans un temps logarithmique par rapport au nombre total des nœuds dans l'arbre.

Tableau 4.1: Description des structures de données Java utilisées dans les bancs d'essais. Ces valeurs ont été mesurées sur une JVM HotSpot 64 bits de version 1.8.0_91 employant une compression sur les pointeurs d'objets ordinaires (*CompressedOops* (Oracle, 2016)) et appliquant un alignement de 8 octets sur les objets en mémoire.

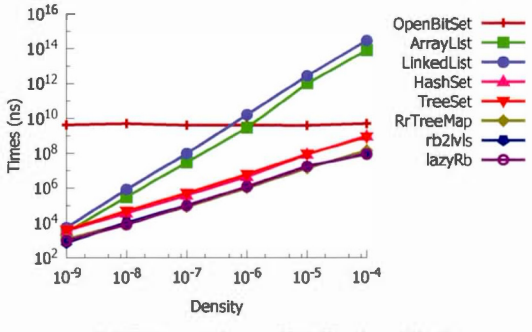
Les Figures 4.1b et 4.1c montrent les temps CPU moyens consommés par les structures de données pour calculer l'intersection de deux ensembles d'entiers de 64 bits.



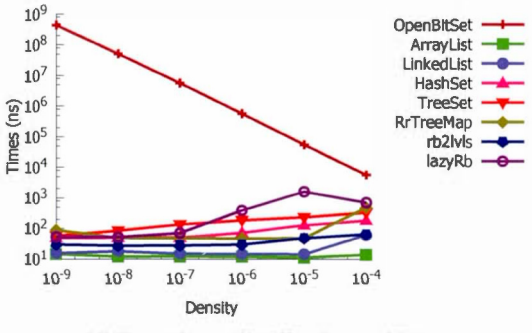
(a) Compression : distribution uniforme



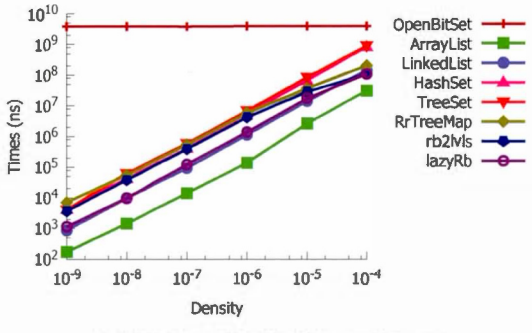
(b) Intersection : distribution uniforme



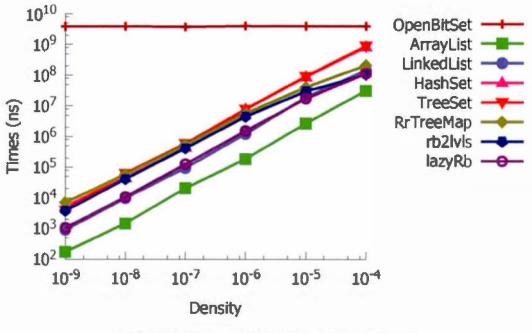
(c) Intersection : distribution Zipf



(d) Insertion : distribution uniforme



(e) Union : distribution uniforme



(f) Union : distribution Zipf

Figure 4.1: Compression et temps d'exécution

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 : uniforme et Zipf. 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. En effet, des *OpenBitSets* de tailles comparables sont obtenus à chaque fois, car chaque bitmap est construit avec la même valeur max. 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 et une grande quantité de travail du côté CPU, comparé au reste des structures de données.

Plus les densités augmentent, plus les performances des deux structures Java *ArrayList* et *LinkedList* s'approchent de celles d'*OpenBitSet*, jusqu'à les dépasser après la densité 10^{-6}, cela même si les structures Java sont près de 32 fois moins volumineuses qu'*OpenBitSet* à cette densité. Ceci s'explique par la complexité temporelle quadratique par rapport à la taille des deux ensembles en entrée d'une opération d'intersection pour ces deux structures Java, ce qui affirme la croissance fulgurante des temps de traitement avec l'augmentation du nombre des éléments dans les deux ensembles d'entiers en entrée. Toutefois, la structure *ArrayList* montre une croissance moins importante que celle de la *LinkedList* avec l'augmentation du nombre d'éléments dans les deux ensembles en entrée. Les analyses ont montré que cette différence de perfor-

mance réside principalement au niveau du temps passé à copier la structure du premier ensemble en entrée lors d'une opération d'intersection. En effet, une *Arraylist* utilise la méthode native Java *System.arraycopy* qui copie efficacement un tableau en opérant par blocs d'éléments en mémoire (avec l'instruction *memmove* du langage C), au lieu d'une boucle *for* qui traite un élément à la fois, comme c'est le cas pour la *LinkedList*. Aussi, une *ArrayList* stocke ses éléments dans un tableau de références (de pointeurs) vers des objets, qui est gardé dans un espace contigu en RAM, le rendant beaucoup plus rapide à lire qu'une *LinkedList*, dont les éléments sont des objets individuels stockés de façon dispersée en mémoire principale, et qui doivent être accédés un par un lors d'une telle opération.

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'un *HashSet* et de $O(n_1 \log n_2)$ pour le *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 près de 6 millions de fois plus rapides qu'*OpenBitSet*, $\approx$ 63 milles fois plus efficaces que les deux structures *ArrayList* et *LinkedList* et $\approx$ 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 plus fortes densités, des analyses approfondies ont révélé que les coûts d'exécution dans ces cas sont largement dominés par les temps passés à calculer des intersections entre des conteneurs. Puisque les trois nouvelles techniques adoptent le même modèle de conteneur qui stocke les 16 bits de poids faible des entiers représentés, une équivalence de 48 bits sur les hauts niveaux des deux bitmaps introduits

lors d'une opération d'intersection devra être trouvée avant de calculer une intersection entre deux conteneurs. De ce fait, le nombre d'intersections effectuées entre conteneurs est pratiquement le même pour chacune des trois nouvelles techniques lors d'une opération d'intersection entre deux bitmaps.

Sur les faibles densités, les bitmaps introduits ne possèdent en général que très peu d'entiers partageant les mêmes 48 bits de poids fort, rendant ainsi le nombre d'intersections effectuées entre conteneurs presque égal à nul. Le temps d'exécution dans ces cas est largement dominé par les comparaisons effectuées sur les hauts niveaux des trois modèles de compression bitmap. Possédant la structure haut niveau la plus simple parmi les trois modèles, qui consiste en un simple tableau renfermant un entier de 64 bits par entrée, *RoaringTwoLevels* affichent les meilleurs résultats sur ces densités-là, en étant 1, 32 fois et 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 du calcul d'une intersection entre deux bitmaps, comparé aux nœuds, stockés de façon dispersée en mémoire principale, adoptés par la méthode *RoaringTreeMap* sur le plus haut niveau.

Les Figures 4.1e et 4.1f 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 les 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 don-

nées testées. 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. 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, ce qui nécessite l'allocation de nouveaux espaces mémoires induisant d'importants temps de traitement. Étant donné que *LazyRoaring* applique une stratégie *copy-on-write* qui élimine plusieurs opérations de copie de conteneurs et de tableaux de deuxième niveau lors du calcul d'une opération d'union, cette technique de compression 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 de ≈ 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 lors du calcul d'une union du fait de son gros volume, qui engendre également une importante charge de travail au CPU. Ses performances restent néanmoins constantes sur toutes les densités pour les mêmes raisons mentionnées lors des intersections.

Bien qu'ayant 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, la structure *ArrayList* est entre 5 à 8 fois plus rapide que la *LinkedList* sur toutes les densités. Les analyses ont révélé que la *ArrayList* est beaucoup plus rapide à lire et à copier qu'une *LinkedList*. Effecti-

vement, la première structure de données adopte un tableau de références stocké dans un espace continu en mémoire, réduisant ainsi le nombre de défauts de cache (*cache misses*) lors de sa lecture et la rendant beaucoup plus rapide à traiter que la seconde, qui maintient ses éléments dans des objets individuels éparpillés en mémoire, chacun devant être accédé lors d'une telle opération. Aussi, comme indiqué précédemment, la structure *ArrayList* utilise la méthode native JNI (*Java Native Interface*) (Microsystems, 2001; Liang, 1999) *System.arraycpy* très efficace pour copier un tableau en mémoire principale, qui procède par bloc d'éléments au lieu d'un élément à la fois. Ces deux structures de données dépassent les performances des 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*. Ces deux dernières structures de données consomment beaucoup plus de temps qu'une *ArrayList* ou une *LinkedList* pour opérer une union entre deux ensembles d'entiers. Effectivement, la structure *HashSet* réalloue un nouvel espace mémoire pour la table de hachage à plusieurs reprises lors de l'insertion de nouveaux éléments au résultat, comme elle recalcule aussi la valeur de hachage de chaque élément à ajouter dans l'ensemble résultant. Quant à la structure *TreeSet*, ses performances s'expliquent par les insertions dans l'arbre résultant qui se font chacune dans un temps logarithmique par rapport au nombre de ses nœuds.

La Figure 4.1d 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 avec une densité donnée, on calcule le temps moyen que consomme chaque structure pour insérer chacun des N éléments. Ces tests donnent aussi une idée sur les performances de chaque structure de données pour construire un nouveau bitmap. Seulement les résultats sur une distribution de données uniforme ont été rapportés, car des comportements presque similaires ont été observés sur une distribution Zipf.

Parmi les trois méthodes de compression bitmap introduites, *RoaringTwoLevels* affiche les meilleures performances sur ces tests, 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 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* est un peu plus rapide que *RoaringTreeMap*, car, sur ces densités, beaucoup d'instanciations de nouveaux objets sont faites sur les plus hauts niveaux des deux structures, où *RoaringTreeMap* recense un peu plus d'objets que *LazyRoaring*. 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 l'insertion d'un nouvel entier en un temps constant, cette structure est jusqu'à 14 millions de fois plus lente que les trois nouvelles techniques de compression bitmap sur les faibles densités. Cela revient aux opérations d'allocation de nouveaux espaces mémoires qu'effectue cette structure lors de ses extensions pour peupler de nouveaux bits apparaissant sur des positions supérieurs à celles des

bits positifs courants du bitmap. Ce genre d'opérations est très observé sur des densités faibles. Plus les densités augmentent, plus les opérations d'allocation de nouveaux espaces mémoires 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 ne nécessitent pas d'importants coûts d'allocations de nouveaux espaces mémoires, ces deux structures affichent les meilleures performances sur ces bancs d'essai. Avec la *ArrayList* qui est un peu plus rapide du fait qu'elle ne se contente que d'enregistrer la référence de l'élément inséré dans l'une de ses entrées libres, tandis que la *LinkedList* alloue un nouvel objet pour chaque élément inséré. Cependant, contrairement à la *LinkedList*, la *ArrayList* nécessite d'étendre la taille de son tableau, une fois rempli, en allouant un de plus large et en y transférant ses éléments stockés, mais cette opération est effectuée de façon très efficace avec l'instruction JNI *System.array.copy* qui n'impacte presque pas les performances de la *ArrayList*.

Même si la collection Java *HashSet* permet d'effectuer des insertions en temps constants également, elle reste néanmoins plus lente que les deux structures de données précédentes. En effet, en plus de devoir créer un nouvel objet pour chaque élément à insérer, tout comme le fait la *LinkedList*, elle doit aussi étendre la taille de sa table lorsqu'elle se remplit, en allouant un nouveau tableau plus large et en y migrant les anciennes données. Mais cette dernière opération prend beaucoup plus de temps sur une *HashSet* que sur une *ArrayList*, car l'instruction JNI *System.array.copy* n'est pas utilisée et plusieurs nouvelles opérations entrent en jeu, telles que : les valeurs de hachage devant être recalculées et les listes chaînées associées aux entrées de la table ayant besoin d'être maintenues, etc. La *TreeSet* affiche des temps de réponse un 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 4.1a 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. Sur les faibles densités, la structure *OpenBitSet* est jusqu'à $\approx$ 127 mille 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 pouvant contenir un nombre de bits égal à la valeur maximale que peuvent prendre les entiers représentés, soit $max = 50$ milliards dans ces essais, environnant les 6 GB d'espace mémoire pour représenter un entier de 64 bits de valeur proche à la valeur max. Cependant, plus les densités augmentent plus les performances d'*OpenBitset* s'améliorent.

Les collections Java affichent des performances stables sur toutes les densités étudiées. Sur les faibles densités (de 10^{-9} à 10^{-5}), les deux structures *ArrayList* et *LinkedList* sont plus compactes que les trois nouveaux modèles de bitmaps, avec 3 fois et 2 fois moins d'espace consommés par ces deux structures, respectivement. Les deux autres solutions *Hashset* et *TreeSet* affichent des performances un peu plus efficaces que celles des trois nouvelles techniques ($\approx 1,5$ fois plus compactes). La contre performance des trois nouveaux modèles s'explique par le fait qu'une grande distance en moyenne sépare deux entiers successivement générés sur ces densités, poussant ainsi les conteneurs de ces trois techniques à ne stocker qu'un seul entier en moyenne sur ces très faible densités, marquant ainsi une absence presque totale de gains de compression. Avec l'absence de compression dans les trois nouveaux modèles, les coûts généraux (*overheads*) en consommation mémoire imposés par les objets employés sur les différents niveaux de chaque technique pèsent de façon significative sur son espace mémoire global, contrairement aux collections Java qui stockent les entiers sans les compresser, mais sur un seul niveau seulement tout en employant beaucoup moins d'objets.

Sur les plus fortes densités (10^{-4}), les conteneurs des trois nouvelles techniques stockent plusieurs entiers, en moyenne, affichant ainsi des gains de compression qui s'avèrent significatifs sur la quantité d'espace mémoire allouée pour chacun des trois nouveaux modèles de bitmap, faisant ainsi réduire la consommation d'espace mémoire des trois nouvelles solutions à environ 2 fois et 4 fois moins par rapport aux deux collections Java *ArrayList* et *LinkedList*, respectivement, et près de 5 fois moins comparé aux deux structures *Hashset* et *TreeSet*.

4.4 Conclusion

Dans l'ère actuelle du *Big Data*, le besoin d'indexer d'énormes ensembles de données dépassant les milliards d'entrées ne cesse d'augmenter, autant chez les chercheurs scientifiques que chez les industriels. Les librairies d'index bitmap introduites jusqu'ici dans la littérature ne sont compatibles qu'à des ensembles de données ne dépassant pas les 2^{32} entrées. Ce travail a introduit trois nouveaux modèles de compression bitmap supportant 2^{32} fois plus de possibilités que les solutions de compression bitmap actuelles, soit 2^{64} entrées.

Des expériences intensives sur deux types de distributions de données : uniforme et Zipf, ont montré des résultats allant jusqu'à $\approx$ 6 millions de fois plus vite que la solution *OpenBitset*, adoptée au sein du moteur de recherche Apache Lucene, lors du calcul d'intersections entre bitmaps, et jusqu'à $\approx$ 63 mille fois plus rapidement que des collections définies dans le langage de programmation Java. Aussi, des performances de près de $\approx$ 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 $\approx$ 14 millions de fois plus lent pour insérer un entier généré aléatoirement par rapport aux trois solutions proposées. Cela, en étant en même temps jusqu'à $\approx$ 127 mille fois et jusqu'à $\approx$ 4 fois moins gourmands en matière d'espaces mémoire consommés comparé à *OpenBitSet* et les collections Java évaluées,

respectivement.

Les trois techniques proposées ont montré une consommation d'espace mémoire assez similaire sur toutes les densités testées. Les tests évaluant les temps pour calculer l'intersection de deux bitmaps ont également révélé des performances similaires dans le cas des trois techniques. 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 $\approx$ 31 fois et jusqu'à $\approx$ 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 $\approx$ 6 fois et $\approx$ 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, on envisage d'étudier les performances des trois nouveaux modèles de compression bitmap sur des densités très fortes, peu rencontrées dans la réalité, qui n'ont pas été abordées lors des essais présentés dans ce travail faute de moyens nous donnons accès à une machine assez sophistiquée. On envisage 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).

Tel que discuté à la Sous-section 4.2.1, deux types d'algorithmes de calcul d'union et d'intersection ont été mis en œuvre pour la solution *RoaringTreeMap*. L'un étant lancé sur deux arbres de tailles asymétriques et l'autre à la rencontre de deux arbres de tailles comparables. Les essais présentés précédemment n'ont évalué que les performances du deuxième type d'algorithme. Avec ses complexités temporelles de $O(n_1 \log(n_2))$ et $O(n_2 + n_1 \log(n_1 + n_2))$ pour comparer les hauts niveaux de deux *RoaringTreeMaps* de tailles n_1 et n_2 avec $n_1 << n_2$ lors d'une intersection et d'une union, respective-

ment, le deuxième type d'algorithme pourrait révéler des performances intéressantes si comparé aux deux autres structures, *RoaringTwoLevels* et *LazyRoaring*. Cette tâche est envisagée comme travail futur.

Chaque entrée de l'index de premier niveau de la structure *RoaringTwoLevels* préserve la cardinalité d'un groupe d'entiers stockés dans le conteneur qu'elle indexe. Cela tend à faciliter le calcul de la cardinalité d'un bitmap donné, type d'opérations très sollicitées dans les SGBD. Comme travail futur, il est envisagé de comparer les performances des trois modèles proposés pour effectuer une telle opération.

Il est également prévu d'implémenter les opérations de suppression d'entiers pour chacun des trois modèles proposés, puis d'évaluer les performances de cette opération pour les trois techniques proposées, *OpenBitSet* et les collections Java utilisées lors des précédents tests.

CHAPITRE V

OPTIMISATION DE DRUID AVEC DES ROARING BITMAPS

Ce chapitre est tiré de l'article ci-dessous avec quelques contenus qui ont été ajoutés, modifiés et/ou supprimés :

Chambi, S., Lemire, D., Godin, R., Boukhalfa, K., Allen, C. R. et Yang, F. (2016c). Optimizing Druid with Roaring bitmaps. Dans 20th International Database Engineering and Applications Symposium (IDEAS'16), 77 – 86., Montreal, QC, Canada. ACM.

5.1 Introduction

De nos jours, la génération massive de données *Big Data* pouvant provenir de différents types de sources de données, comme : des organisations, des périphériques, des individus particuliers, etc., attise les entreprises du monde entier et les met face au défi de développer des solutions permettant de collecter et d'organiser efficacement de telles masses de données. Cela dans le but d'en tirer de nouvelles connaissances qui joueront, au final, un rôle majeur dans la compétitivité de la compagnie. Plusieurs solutions de ce genre ne cessent d'être introduites de la part de grandes industries telles que : Hadoop (Shvachko *et al.*, 2010), PowerDrill (Hall *et al.*, 2012) et Dremel (Melnik *et al.*, 2011) de Google, Avatara (Wu *et al.*, 2012) de LinkedIn, Rios et Lin (2012) de Twitter, etc. *Druid* (Yang *et al.*, 2014) est l'un de ces fameux systèmes. Il a été proposé très récemment par la compagnie américaine *Meta-markets* et permet le stockage et l'analyse

en temps réel d'importantes quantités de données en des temps ne dépassant pas la seconde. Des tests ont montré que *Druid* a été capable de lire, filtrer et agréger 1 milliard de lignes en seulement quelques millisecondes (Tschetter, 2011). Actuellement, ce système est utilisé chez plusieurs compagnies, notamment : NetFlix, Facebook, Google, etc. En plus de ses faramineuses capacités computationnelles, *Druid* est aussi proposé comme un projet en code libre (*Open Source*) au grand public, et s'appuie sur une vaste communauté de développeurs.

Après avoir proposé la solution *Roaring bitmap*, il était venu grand temps de chercher à valider le format de la nouvelle méthode de compression bitmap, en essayant de l'intégrer au sein d'un système de gestion de bases de données (SGBD) réel, puis d'observer et d'analyser les possibles avantages et inconvénients apportés par la technique de compression bitmap aux performances globales du SGBD. Un survol de la littérature de ce type de systèmes s'en est suivi, lors duquel on est tombé sur *Druid*, un moteur OLAP (*Online Analytical Processing*) entièrement écrit en Java et qui se base principalement sur les index bitmap compressés pour accélérer les requêtes OLAP effectuant des analyses détaillées (*drill-down*) sur les données. Auparavant, *Druid* adoptait la solution *Concise* proposée par (Colantonio et Di Pietro, 2010) comme seule méthode de compression bitmap.

Druid se base sur la fonctionnalité de *memory-mapping* pour stocker et lire les données utilisées par le système. Cette fonctionnalité donne la perception au programmeur de n'interagir qu'avec une mémoire virtuelle de la taille de l'espace disque pour lire et stocker les données du système, en dédiant les mécanismes de transfert des données entre la mémoire principale et secondaire à l'unité de gestion de la mémoire virtuelle du système d'exploitation. Pour rendre la méthode *Concise* opérationnelle dans ce contexte, une extension de la librairie originale qui ne supportait que des traitements de bitmaps alloués en mémoire centrale a été mise en œuvre par des développeurs de *Meta-markets* pour pouvoir gérer des bitmaps sérialisés sur disque. Le code source de la nouvelle li-

brairie est partagé gratuitement en ligne sur le lien suivant : Metamx (2015).

Ayant obtenu de remarquables performances avec *Roaring bitmap* lorsque comparé à *Concise* lors de précédentes expériences réalisées sur des bitmaps en mémoire primaire, la libraire *Roaring bitmap* a été étendue pour supporter la gestion de bitmaps stockés sur mémoire externe. Des expériences comparant les performances temps-espace de *Roaring bitmap* et de *Concise* dans un tel contexte ont suivi (Chambi *et al.*, 2016b), tel que présentés au premier chapitre de la thèse. Après que ces bancs d'essai aient révélé d'impressionnantes performances au profit de *Roaring bitmap*, ce travail fut proposé à la communauté derrière le système *Druid*. Suite à cela, une étroite collaboration avec l'équipe de développement logiciel de la société *Meta-markets* s'en est suivie pour faire de *Roaring bitmap* une technique d'indexation à part entière au sein du SGBD *Druid*. Par la suite, des expériences (Chambi *et al.*, 2016c) ont été conduites pour comparer les performances de *Roaring bitmap* à celles de la méthode de compression bitmap adoptée par *Druid*, *Concise*, au sein de cet SGBD. Les résultats ont été très concluants pour *Roaring bitmap*. En effet, *Roaring bitmap* a permis de traiter jusqu'à 5 fois plus rapidement que *Concise* des requêtes OLAP effectuant des analyses détaillées sur les données (*drill-down*), qui font recours aux index bitmaps lors de l'exécution. Pour faciliter la lecture du texte, le terme *Roaring* sera utilisé pour faire référence à la technique *Roaring bitmap* dans le reste du chapitre.

D'autres systèmes dédiés au traitement de données massives ont également intégré *Roaring* comme solution d'indexation : Apache Spark (Zaharia *et al.*, 2010), Kylin (Apache, 2014a), Solr (Apache, 2014b), Elastic (Grand, 2015) et Lucene (Apache, 2012), avec ce dernier qui emploie une implémentation indépendante de *Roaring*. Nous laissons l'analyse des performances de *Roaring* au sein de ces systèmes comme travaux futurs.

Ce chapitre introduit les principaux points de ce travail mené en collaboration avec l'équipe *Druid*, en présentant le système *Druid* à la Section 5.2, et ses requêtes d'ana-

lyses pouvant utiliser des bitmaps lors de l'exécution à la Section 5.3. La Section 5.4 fait état des expériences conduites au sein de *Druid* pour comparer les performances de *Roaring* à celles de *Concise*. Les résultats obtenus de ces expériences et une analyse s'ensuivent. La Section 5.5 présente la conclusion de ce chapitre.

5.2 Druid

Druid a été introduit en code libre (*Open Source*) par des développeurs de la compagnie *Meta-markets* (Metamarkets, 2015) en 2012. Après avoir constaté le besoin d'une solution qui garantit des traitements rapides pour les requêtes d'analyses OLAP (en moins d'une seconde), qui soit hautement disponible et adaptée à des contextes d'utilisation super-concurrents (1000 utilisateurs et plus), les auteurs se sont alors penchés sur les solutions *Open Source* existantes déjà dans la littérature.

L'un des plus populaires systèmes de gestion de données massives *Open Source* introduits à ce jour est la solution Hadoop (Shvachko *et al.*, 2010) de Google. Bien que Hadoop réponde parfaitement au besoin consistant à stocker de grands volumes de données et à y donner accès, il n'établit cependant aucune garantie sur la rapidité à laquelle ces données peuvent être accédées (Yang *et al.*, 2014). Hadoop est connu aussi pour être un système très disponible, néanmoins, les performances de Hadoop se dégradent considérablement lorsque plusieurs processus de chargement de données doivent être exécutés de façon concurrente (Yang *et al.*, 2014). En plus, Hadoop n'excelle pas en ce qui a trait à la vitesse à laquelle il absorbe les nouvelles données et les rend rapidement disponibles pour les requêtes des usagers (Yang *et al.*, 2014). Or, l'un des défis de la solution *Druid* est de permettre à ses utilisateurs de prendre des décisions en s'appuyant sur des données à jour, cela en procédant à l'acquisition en temps réel des nouvelles données reçues. Un des objectifs visés par les concepteurs de *Druid* est de pouvoir compléter l'acquisition d'un nouvel enregistrement en moins d'une seconde. L'ensemble des systèmes *Open Source* testés n'ont pu fournir de telles performances.

Par conséquent, n'ayant trouvé aucune solution dans le monde *Open Source* qui satisfait leurs attentes, les développeurs de chez *Meta-markets* ont décidé de mettre en œuvre leur propre SGBD : *Druid*.

Druid est un système de stockage de données (*data store*) conçu pour résoudre les problèmes liés à l'acquisition rapide et à l'analyse OLAP en temps réel d'immenses ensembles de données événementielles, et pour être déployé dans des environnements hyper concurrents (1 000 utilisateurs et plus). Actuellement, *Druid* est adopté en production par plusieurs compagnies (Druid, 2015b), comme *ebay*, *Yahoo*, *NetFlix*, etc. *Druid* stocke ses données dans un format orienté colonne, adopte une architecture distribuée et non-partagée (*shared-nothing*), et fait usage d'index bitmap compressés pour accélérer les opérations OLAP de type *drill-down*. Il a également été conçu pour être un système tolérant aux pannes (*fault-tolerant*), hautement disponible, pouvant supporter des agrégations rapides, des techniques de filtrage de données flexibles et une rapide acquisition des données nouvellement reçues.

Druid stocke des données événementielles, qui une fois reçues, ne seront jamais modifiées et seront exposées en lecture seulement aux usagers. Le format de ces données est composé de trois composantes différentes :

L'estampille (*timestamp*) : Représente l'instant auquel l'événement a été généré.

Les dimensions : La deuxième composante représente l'ensemble des attributs de type chaîne de caractères utilisés par les requêtes OLAP pour filtrer les données (requêtes *drill-down*). Ainsi, ces attributs jouent le même rôle que les dimensions dans un entrepôt de données traditionnel.

Les métriques : La troisième composante représente l'ensemble des attributs jouant le rôle des mesures dans les entrepôts de données traditionnels. Ces colonnes renferment des valeurs numériques et sont utilisées par les requêtes OLAP pour effectuer des agrégations et des calculs, comme avec les fonctions SQL : count,

sum, avg, etc.

Un événement dans *Druid* est stocké dans une source de données. Cette dernière joue le même rôle qu'une table dans une base de données relationnelle. De plus, les événements d'une source de données sont divisés selon leur attribut *timestamp* en segments, chacun stockant 5-10 millions d'événements qui tombent dans un intervalle de temps bien précis : minute, heure, jour, etc. Cet intervalle est spécifié dans les paramètres initiaux de configuration d'une grappe *Druid*. La longueur de cet intervalle est la même pour tous les segments d'une source de données. La plus petite granularité possible pour l'attribut *timestamp* des événements gardés à l'intérieur des segments est le milliseconde, mais ces événements peuvent aussi être cumulés en une plus grande granularité, par exemple : minute, heure, jour. Toutefois, cette granularité doit impérativement être inférieure ou équivalente à la granularité des intervalles des segments. Également, la granularité des données à l'intérieur des segments est la même dans tous les segments d'une source de données.

Une instance de *Druid* se présente sous la forme d'une grappe (*cluster*) composée de plusieurs types de nœuds, chacun étant conçu pour faire un travail bien précis. Dans une architecture distribuée, un nœud s'exécute généralement sur une machine qui lui est dédiée (sauf si le mode d'exécution est en *standalone*) et de façon totalement indépendante par rapport au reste des nœuds de la grappe. Ainsi, un nœud en cours d'exécution ne partage aucune de ses données ou de ses ressources matérielles avec les autres nœuds de la grappe (notion d'architecture *shared-nothing* (Wikipedia, 2015c)). Les différents types de nœuds supportés par une grappe *Druid* sont :

Le nœud historique : S'occupe de charger/supprimer des segments dans son espace local, et d'exécuter des requêtes sur les segments qu'il sert.

Le nœud coordinateur : S'occupe de la répartition des segments entre les nœuds historiques du cluster.

Le nœud courtier (*broker*) : Représente l'entité à laquelle les usagers envoient leurs requêtes. Ce nœud sait comment les segments sont répartis entre les différents nœuds du cluster. Il se charge alors d'envoyer une requête vers les différents nœuds possédant des segments susceptibles de contenir des résultats valides pour une requête. Comme il se charge également de fusionner les résultats fournis par les différents nœuds sollicités lors du processus de traitement d'une requête, avant de renvoyer le résultat final à l'utilisateur.

Le nœud temps réel : Représente le point d'entrée des récents événements capturés en temps réel pour qu'ils soient insérés dans la grappe *Druid*. Ce nœud se charge de construire les segments et de les libérer aux nœuds historiques après un certain délai configurable, tout comme il répond aussi aux requêtes accédant aux données qu'il maintient localement.

Les nœuds *Overlord* et *HadoopDruidIndexer* : Les points d'entrée des données à insérer en lots. Ces nœuds ont recours à la plate-forme *Hadoop* pour paralléliser l'exécution des tâches d'acquisition de données afin d'accélérer les temps de traitements.

Pour plus d'informations sur les rôles des différents nœuds et des flux de données dans *Druid*, le lecteur peut se référer au papier Yang *et al.* (2014).

Pour exécuter une requête, l'usager l'envoie à un nœud courtier, qui la répartit ensuite vers les différents nœuds historiques et/ou temps-réels sélectionnés dans le cluster. Un nœud temps-réel exécute la requête sur les données récentes maintenues encore dans sa base locale, et un nœud historique l'exécute sur les segments qu'il stocke localement. Chaque nœud sollicité renvoie ensuite ses résultats au nœud courtier expéditeur de la requête. Afin d'améliorer les temps de réponse des requêtes OLAP, *Druid* emploie un index bitmap sur chaque attribut dimension dans un segment. Les bitmaps s'avèrent très efficaces en matière d'accélération des opérations *drill-down* filtrant les données

selon des valeurs de dimensions.

5.3 Les types de requêtes d'analyses supportées par *Druid*

Druid supporte différents types de requêtes d'analyses. Elles sont exprimées en JSON et envoyées avec des requêtes POST du protocole HTTP aux nœuds candidats. Ces requêtes se répartissent en deux grandes classes : requêtes de recherches et d'agrégations, et requêtes de métadonnées. Étant donné que seulement la première classe de requêtes peut faire usage de bitmaps lors de l'exécution, seule cette classe sera étudiée lors de ce travail. Le lecteur désireux d'en connaître davantage sur les requêtes de métadonnées peut se référer à la documentation mise en ligne : Druid (2015c). Une particularité du premier groupe de requêtes est qu'un intervalle de temps est toujours spécifié dans le corps d'une requête, pour indiquer la partition de temps qui correspond aux données ciblées par la requête.

5.3.1 Requête *GroupBy*

La requête *GroupBy* a le même objectif que celle définie dans le langage SQL, soit de retourner des valeurs numériques cumulées par groupes distincts de valeurs de dimension(s).

Un exemple d'une requête *GroupBy* est donné ci-dessous (source : Druid (2015a)) :

```
{
  "queryType": "groupBy",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "dimensions": ["country", "device"],
  "filter": {
    "type": "and",
    "fields": [
      { "type": "selector", "dimension": "carrier", "value": "AT&T" },
      { "type": "or",
        "fields": [
          { "type": "selector", "dimension": "make", "value": "Apple" },
```

```
          { "type": "selector", "dimension": "make", "value": "Samsung" }
        ]
      }
    ]
  },
  "aggregations": [
    { "type": "longSum", "name": "total_usage", "fieldName": "user_count" },
    { "type": "doubleSum", "name": "data_transfer", "fieldName": "data_transfer" }
  ],
  "postAggregations": [
    { "type": "arithmetic",
      "name": "avg_usage",
      "fn": "/",
      "fields": [
        { "type": "fieldAccess", "fieldName": "data_transfer" },
        { "type": "fieldAccess", "fieldName": "total_usage" }
      ]
    }
  ],
  "intervals": ["2012-01-01T00:00:00.000/2013-01-03T00:00:00.000"],
  "having": {
    "type": "greaterThan",
    "aggregation": "total_usage",
    "value": 100
  }
}
```

L'entête d'une requête *GroupBy* est formé de l'attribut "queryType", qui indique le type de la requête formulée, suivi du champ "dataSource", qui représente la source de données sur laquelle exécuter la requête. Le prochain champ "granularity" sert à préciser le granule de temps (granularité) auquel seront cumulés les résultats en sortie. Dans le champ "dimension", sont déclarés les attributs dimension sur lesquels seront effectués les groupements. Dans cet exemple, si l'attribut "country" possède n valeurs distinctes et l'attribut "device" en possède m, alors le résultat sera formé de $n \times m$ groupes distincts présentés par cumuls d'une journée ("granularity" : "day"). Le champ suivant permet de spécifier des filtres pour descendre plus en détail dans les données

analysées (*drill-down*), partie jouant le même rôle que la clause *WHERE* d'une requête en SQL. Les filtres sont, en général, déclarés sur des attributs dimensions, mais ils restent aussi applicables sur des attributs métriques. Pour accéder rapidement au sous-ensemble de données ciblées par le filtre, les bitmaps associés aux valeurs des attributs de dimensions spécifiées dans le filtre entrent en jeu, et plusieurs types d'opérations sont opérées sur les bitmaps : opérations OU, ET ou NON logiques, itérations sur les bitmaps, etc. En effet, dans cet exemple, un OU logique est exécuté entre les bitmaps des deux valeurs : "Apple" et "Samsung", puis un ET logique entre le bitmap résultant de l'opération logique précédente et celui de la valeur "AT&T". Ces opérations logiques seront exécutées au niveau de chaque segment accédé de la source de données.

Par la suite, vient la partie spécifiant les attributs de type métrique sur lesquels seront calculées les agrégations à renvoyer dans le résultat. Une des opérations que peu de moteurs OLAP supportent est celle que décrit le champ suivant "postAggregations". Celui-ci permet d'ajouter des attributs aux résultats, dont les valeurs sont calculées à partir des attributs métriques de la partie "aggregations", en y appliquant plusieurs sortes d'opérations arithmétiques. Dans cet exemple, un attribut "avg_usage" sera ajouté à chaque ligne du résultat et qui sera calculé à l'aide d'une division entière entre les valeurs obtenues aux attributs "data_transfer" et "data_usage".

Le prochain champ est "intervals", qui sert à définir l'intervalle (la portion) de temps sur lequel effectuer la recherche. Cette valeur aide le moteur OLAP à n'accéder qu'aux segments tombant dans cet intervalle de temps. Le dernier champ est "Having", dont la fonction reste similaire à celle de la clause "HAVING" en SQL, et qui permet de définir des restrictions sur les groupes à afficher en sortie.

Le lien ci-après donne plus de détails sur les clauses possibles d'une requête *GroupBy* : Druid (2015a).

5.3.2 Requêtes *Timeseries*

Dans la sous-section précédente, il a été montré que les requêtes *GroupBy* permettent de calculer des agrégations sur un certain intervalle de temps, présentées dans le résultat sous la forme de groupements, chaque groupe étant formé d'une combinaison distincte des valeurs de dimensions choisies. Finalement, les résultats seront présentés par cumuls d'une durée (granularité) déclarée par le biais du champ "granularity". Les requêtes *Timeseries* ne sont pas assez différentes. Elles permettent également de calculer des agrégations sur un intervalle de temps précis, mais regroupent les résultats selon la granularité spécifiée dans la requête seulement. Ci-dessous un exemple d'une requête *Timeseries* (source : Druid (2015f)) :

```
{
  "queryType": "timeseries",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "filter": {
    "type": "and",
    "fields": [
      { "type": "selector", "dimension": "sample_dimension1", "value": "
          sample_value1" },
      { "type": "or",
        "fields": [
          { "type": "selector", "dimension": "sample_dimension2", "value": "
              sample_value2" },
          { "type": "selector", "dimension": "sample_dimension3", "value": "
              sample_value3" }
        ]
      }
    ]
  },
  "aggregations": [
    { "type": "longSum", "name": "sample_name1", "fieldName": "sample_fieldName1"
        },
    { "type": "doubleSum", "name": "sample_name2", "fieldName": "sample_fieldName2"
        }
  ],
  "postAggregations": [
```

```
    { "type": "arithmetic",
      "name": "sample_divide",
      "fn": "/",
      "fields": [
        { "type": "fieldAccess", "name": "sample_name1", "fieldName": "
            sample_fieldName1" },
        { "type": "fieldAccess", "name": "sample_name2", "fieldName": "
            sample_fieldName2" }
      ]
    }
  ],
  "intervals": [ "2012-01-01T00:00:00.000/2012-01-03T00:00:00.000" ]
}
```

Comme il peut être constaté, la formulation d'une requête *Timeseries* n'est pas très différente de celle d'une requête *GroupBy*. À l'exception près qu'il n'y ait pas de dimensions à spécifier pour effectuer des groupements, ainsi que de champ "HAVING" qui est spécifique aux requêtes *GroupBy*.

5.3.3 Requêtes *TopN*

La requête *TopN* est une sorte de requête *GroupBy* qui ne permet de faire des groupements que sur une seule dimension, et qui effectue un travail en plus sur le résultat final qui consiste à ordonner les instances résultantes selon une formule spécifiée dans la requête. Comme elle permet aussi d'établir une limite sur le nombre total des instances retournées dans le résultat de la requête. Le code source ci-dessous illustre un exemple de requête TopN (source : Druid (2015g)) :

```
{
  "queryType": "topN",
  "dataSource": "sample_data",
  "dimension": "sample_dim",
  "threshold": 5,
  "metric": "count",
  "granularity": "all",
  "filter": {
    "type": "and",
```

```
  "fields": [
    {
      "type": "selector",
      "dimension": "dim1",
      "value": "some_value"
    },
    {
      "type": "selector",
      "dimension": "dim2",
      "value": "some_other_val"
    }
  ]
},
"aggregations": [
  {
    "type": "longSum",
    "name": "count",
    "fieldName": "count"
  },
  {
    "type": "doubleSum",
    "name": "some_metric",
    "fieldName": "some_metric"
  }
],
"postAggregations": [
  {
    "type": "arithmetic",
    "name": "sample_divide",
    "fn": "/",
    "fields": [
      {
        "type": "fieldAccess",
        "name": "some_metric",
        "fieldName": "some_metric"
      },
      {
        "type": "fieldAccess",
        "name": "count",
        "fieldName": "count"
      }
    ]
```

```
    }
  ],
  "intervals": [
    "2013-08-31T00:00:00.000/2013-09-03T00:00:00.000"
  ]
}
```

La syntaxe de la requête est presque similaire à celle d'une requête *GroupBy*, sauf qu'elle possède deux champs en plus, celui du "Threshold" qui indique le nombre d'éléments souhaités dans le résultat, et le champ "metric" qui spécifie comment le résultat d'une requête TopN devrait être trié. Le lien suivant donne plus de détails sur les différents composants d'une requête TopN : Druid (2015g).

5.3.4 Requêtes de recherche

Les requêtes de recherche sont des requêtes qui permettent de sélectionner les valeurs d'attributs (dimensions ou métriques) de lignes satisfaisant un critère de recherche. Une syntaxe possible d'une requête de recherche se présente comme suit (source : Druid (2015d)) :

```
{
  "queryType": "search",
  "dataSource": "sample_datasource",
  "granularity": "day",
  "searchDimensions": [
    "dim1",
    "dim2" ],
  "query": {
    "type": "insensitive_contains",
    "value": "Ke"
  },
  "sort" : {
    "type": "lexicographic"
  },
  "intervals": ["2013-01-01T00:00:00.000/2013-01-03T00:00:00.000"]
}
```

L'exemple de la requête ci-dessus contient trois nouveaux champs non rencontrés dans les exemples de requêtes précédents. Le champ "searchDimensions" indique les dimensions sur lesquelles effectuer la recherche et qui seront affichées comme attributs dans chaque entité du résultat. Le champ "query" permet de spécifier les critères de recherche à appliquer sur les dimensions du champ "searchDimensions" pour capturer les entrées qui se conforment au résultat souhaité. Quant au champ "sort", celui-ci sert à indiquer le type de tri à appliquer sur l'ensemble résultant.

Plus de détails sur la requête *search* peuvent être trouvés en suivant ce lien : Druid (2015d).

5.3.5 Requêtes de sélection

La requête "select" a le même rôle que celle du langage SQL. Elle permet de sélectionner des données depuis une source de données selon les critères de recherche définis. Ces derniers consistent en les champs filtre et intervalle. La requête permet aussi de spécifier les attributs à afficher pour les entités du résultat. Un exemple d'une requête "select" est donné ci-dessous :

```
{
   "queryType": "select",
   "dataSource": "STM",
   "dimensions":["nom", "prénom"],
   "metrics":["TotalNet", "nbPass"],
   "granularity": "all",
        "filter": {
    "type": "and",
    "fields": [
      {
        "type": "selector",
        "dimension": "arrondissement",
        "value": "Saint-Laurent"
      },
      {
        "type": "selector",
        "dimension": "Age",
```

```
        "value": "25"
      }
    ]
  },
  "intervals": ["2016-01-01/2015-01-01"]
}
```

Les deux champs "dimensions" et "metrics" servent à indiquer les colonnes à afficher pour les éléments de l'ensemble résultant. Les champs restants effectuent le même rôle qu'avec les précédentes requêtes. Le lien suivant donne plus de détails sur la requête "select" : Druid (2015e).

5.4 Expériences

Des expériences ont été conduites pour comparer les performances de *Roaring* avec celles de *Concise* sous *Druid*. Un cluster *Druid* a été lancé sur un seul nœud comportant un processeur AMD FX^{TM}-8150 de 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.

Des données d'un volume de 1 Gb provenant du banc d'essai TPC-H (Council, 2014), se présentant sous la forme d'une table relationnelle comptant en tout $\approx$ 6 millions de lignes, ont été générées pour ces expériences. Ces données ont été chargées dans *Druid* et stockées en deux sources de données. L'une dont les segments sont indexés par des bitmaps compressés avec *Concise*, et l'autre ayant des segments indexés avec des *Roarings*. Dans un premier temps, on mesure les temps moyens d'exécution de chaque type de requêtes présentées précédemment sur les deux sources de données, afin de comparer les performances des requêtes lorsque accélérées avec *Concise* et *Roaring*. L'attribut "l_shipdate" a été pris comme estampille sur laquelle les segments seront construits. Les granularités des segments et des événements ont été fixées à une journée (*day*), car l'attribut "l_shipdate" est d'une granularité journalière. Avant de commencer à calculer les

temps d'exécution moyens d'une requête, une phase de réchauffement (*warming-up*) est initiée, qui consiste à exécuter une requête plusieurs fois jusqu'à stabilisation de ses temps de réponse. Dans nos expériences, 10 répétitions ont été largement suffisantes. Après cette étape, la requête est lancée 100 fois, puis la moyenne des temps de réponse obtenus à chaque exécution est présentée. Le code de ces bancs d'essai est librement accessible en ligne (Chambi *et al.*, 2015).

Pour évaluer les performances de *Roaring* et de *Concise* sur plusieurs sortes de données, on calcule le temps d'exécution moyen de chaque type de requête avec des bitmaps de quatre densités différentes : très faible, faible, moyenne et forte.

5.4.1 Expériences avec des requêtes d'agrégations opérant des OU logiques

Une première série d'expériences a été conduite pour évaluer les temps d'exécution de requêtes qui commencent par filtrer l'ensemble de données à l'aide d'opérations booléennes de type OR en faisant recours à l'utilisation de bitmaps, puis calculent des agrégations sur l'ensemble de données réduit obtenu après la première opération. Ces requêtes sont de type : *GroupBy*, *TopN* et *Timeseries*.

La syntaxe de la requête *GroupBy* opérant des OU logiques entre des bitmaps de très faibles cardinalités sur la source de données indexée avec *Concise* est présentée par l'Algorithme VI.1 en annexe.

Cette requête effectue des opérations OR logiques entre des bitmaps de très faibles densités, dont les cardinalités varient entre $]200, 3\,000[$, pour sélectionner un sous-ensemble de données réduit de la source de données accédée. Ensuite, un *GroupeBy* sur l'attribut dimension "l_shipmode" est réalisé pour calculer les agrégations de valeurs de l'attribut métrique "L_TAX" faisant partie d'un même groupe distinct de la dimension "l_shipmode". Un attribut dimension et un autre de type métrique ont été utilisés pour ce test afin que le temps de réponse de la requête ne soit pas dominé par le temps passé

à faire les groupements et calculer les agrégations, ce qui nous aurait empêchés de distinguer entre les performances de *Roaring* et de *Concise* dans les temps de réponse capturés.

Pour ce qui est des tests sur des bitmaps de faibles cardinalités, on a changé la partie "filtre" de la requête *GroupBy* de sorte que des bitmaps de cardinalités variant entre $]1 \cdot 10^5, 9 \cdot 10^5[$ soient utilisés. La requête ainsi modifiée est présentée par l'Algorithme VI.2 en annexe.

En ce qui concerne les essais sur des bitmaps de moyennes cardinalités, 8 bitmaps de la partie "filtre" de la requête précédente ont été remplacés par de nouveaux bitmaps de cardinalités moyennes appartenant à l'intervalle $]1 \cdot 10^6, 18 \cdot 10^5[$. On s'est restreint à 8 bitmaps du fait que l'ensemble de données testé ne contient que 8 bitmaps avec de telles densités. La requête obtenue est reproduite sur l'Algorithme VI.3 en annexe.

Pour ce qui est des tests sur des bitmaps de fortes cardinalités, une démarche similaire à la précédente a été suivie, en remplaçant 3 bitmaps de moyennes densités de la requête précédente par des bitmaps de fortes densités dont les cardinalités appartiennent à l'intervalle $]22 \cdot 10^5, 3 \cdot 10^6[$. Seulement 3 bitmaps ayant de telles densités ont été trouvés dans l'ensemble de données utilisé. L'algorithme VI.4 en annexe présente la requête obtenue.

La deuxième requête utilisée est du type *TopN* et ressemble à la requête *GroupBy*. Elle prend les mêmes valeurs dans presque tous les champs de la requête, sauf à la partie "metric" où c'est l'attribut métrique "l_tax" qui sera utilisé pour déterminer l'ordre dans lequel les résultats seront présentés. Également, un seuil "Threshold" de 100 a été choisi pour préciser de ne renvoyer à l'utilisateur que les 100 premiers résultats de la requête (le Top 100). Les mêmes bitmaps de différentes densités adoptés pour la requête *GroupBy* ont été utilisés pour évaluer les performances de cette requête avec des bitmaps de très faibles, faibles, moyennes et fortes densités.

La troisième requête évaluée est de type *Timeseries* et adopte les mêmes valeurs que la requête *GroupBy* pour ses différents champs. Pour évaluer les performances de cette requête avec des bitmaps de différentes densités, le même ensemble de bitmaps utilisé sur chaque densité dans le cas de la requête *GroupBy* est repris.

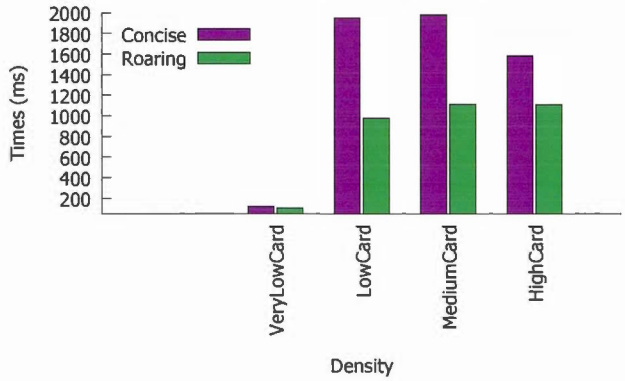
Afin de faire en sorte que le résultat soit calculé de nouveau à chaque exécution d'une requête et non pas en-cacheté après la première réponse, puis ré-accédé directement durant le reste des exécutions, le champ "context" suivant a été spécifié pour chaque requête : "context" :"useCache" :false, "populateCache" :false.

La Figure 5.1 donne les graphiques représentant les temps de réponse des trois différents types de requêtes d'agrégations évaluées. Les résultats montrent que les requêtes exécutées sur des segments indexés avec *Roaring* ont affiché des temps de réponse plus rapides que celles des requêtes lancées sur des segments indexés par *Concise*. En effet, en opérant des OU logiques entre des bitmaps de très faibles densités pour filtrer les données, les requêtes utilisant *Roaring* ont été, en moyenne, 16% [1] plus rapides dans le cas des requêtes *Timeseries* et *TopN*, et 12% plus rapides pour la requête *GroupBy* comparé aux requêtes utilisant des bitmaps du modèle *Concise*.

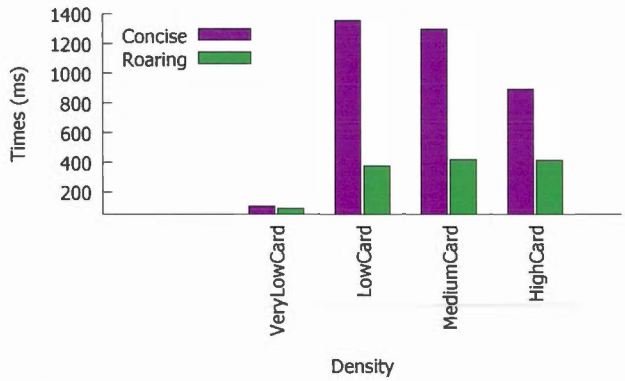
En augmentant la densité des bitmaps, *Roaring* s'est montré encore plus performant que *Concise*. Sur des bitmaps de faible densité, *Roaring* a amélioré de 72%, 67% et 50% les temps de réponse des requêtes de types : *Timeseries*, *TopN* et *GroupBy*, respectivement, comparé aux temps de réponse obtenus avec *Concise*. Alors que sur des bitmaps de cardinalités moyennes, les résultats offerts par *Roaring* dépassent ceux de *Concise* de 68%, 62% et 43%, respectivement, pour les requêtes de types : *Timeseries*, *TopN* et *GroupBy*. Cependant, sur des bitmaps de fortes densités, la différence entre les performances de *Roaring* et de *Concise* rétrécit. Cela est causé par le fait que les bitmaps introduits à

---

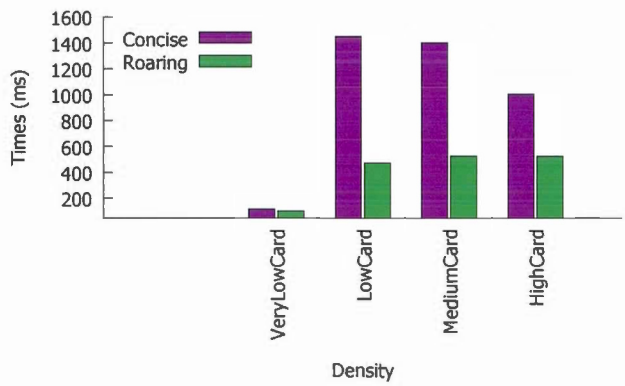
1. Une valeur de pourcentage, v, est calculée comme suit : si x est le plus petit des deux temps moyens mesurés et y le plus grand, alors $v = (1 - (x/y)) \times 100$. Au final, le résultat est arrondi au plus proche entier.



(a) Temps de réponse de requêtes *GroupBy* effectuant des OU logiques entre des bitmaps de différentes cardinalités



(b) Temps de réponse de requêtes *Timeseries* effectuant des OU logiques entre des bitmaps de différentes cardinalités



(c) Temps de réponse de requêtes *TopN* effectuant des OU logiques entre des bitmaps de différentes cardinalités

Figure 5.1: Temps d'exécution de requêtes *GroupBy* opérant des OU logiques

cette étape possèdent de longues suites de bits à 1, qui sont efficacement compressés par l'encodage RLE de *Concise*. Aussi, les développeurs du moteur OLAP *Druid* ont mis en œuvre une stratégie pour calculer efficacement l'union de plusieurs bitmaps du modèle *Concise*, qui consiste à traiter en priorité les mots CPU du type 1-fill lors de calculs de OU logiques entre plusieurs bitmaps, ce qui permet de sauter le traitement de plusieurs mots littéraux et 0-fills durant de telles opérations, résultant en des accélérations significatives dans les temps de traitement des OU logiques. Toutefois, *Roaring* reste plus efficace que *Concise* sur ces densités-là, en offrant des temps de réponse de 26%, 22% et 11% plus performants que ceux de *Concise* pour les requêtes de types : *Timeseries*, *TopN* et *GroupBy*, respectivement.

5.4.2 Expériences avec des requêtes d'agrégations opérant des ET logiques

Une deuxième série d'expériences a été réalisée pour évaluer les temps de réponse des trois types de requêtes d'agrégation lorsque exécutées sur des segments indexés avec *Roaring* ou *Concise*, mais en filtrant les données, cette fois-ci, à l'aide de ET logiques. Comme pour les précédentes expériences, on a sélectionné quatre ensembles de bitmaps, chacun ayant une densité différente : très faible, faible, moyenne et forte. Chaque requête filtre l'ensemble de données en calculant le résultat d'un ET logique entre 7 bitmaps. Les bitmaps utilisés dans une requête appartiennent chacun à une dimension différente, cela nous permet de nous assurer que le sous-ensemble de données sélectionné par le filtre d'une requête ne sera jamais vide. Cela parce que dans le cas échéant, si un bitmap nul est obtenu durant les calculs logiques, le moteur OLAP sautera le traitement des bitmaps restants non encore considérés, puis retournera un ensemble vide comme résultat. Un tel cas représente une situation à éviter à tout prix afin d'aboutir à des évaluations de performances fiables.

La requête du type *GroupBy* exécutée sur l'ensemble de données indexé avec *Concise* et qui opère des ET logiques entre des bitmaps de très faibles densités, dont les cardinalités

tombent dans l'intervalle $]0, 3 \cdot 10^3[$, est donnée dans l'Algorithme VI.5 en annexe.

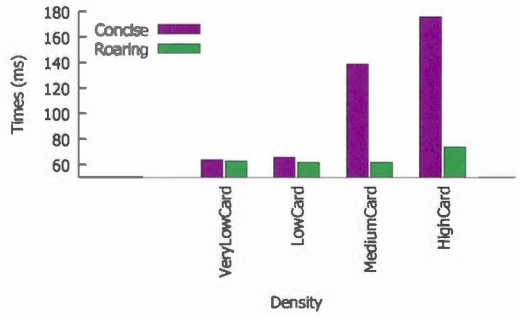
Pour tester la requête du type *GroupBy* avec des bitmaps de faibles densités, les 5 moins denses bitmaps de la requête précédente ont été remplacés avec des bitmaps de cardinalités faisant partie de $]1 \cdot 10^5, 9 \cdot 10^5[$. La requête obtenue est rapportée sur l'Algorithme VI.6 en annexe.

De la même façon, les 3 moins denses bitmaps de la requête d'avant ont été remplacés par des bitmaps de moyennes densités, qui ont des cardinalités appartenant à $]1 \cdot 10^6, 18 \cdot 10^5[$. La requête ainsi changée est présentée par l'Algorithme VI.7 en annexe.

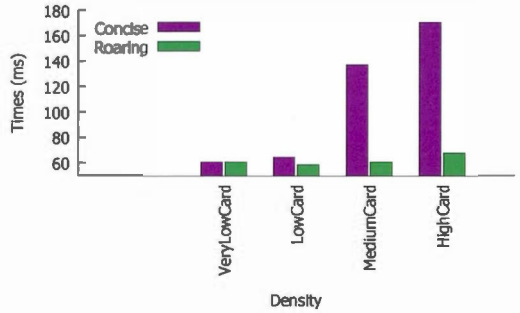
Pour les essais sur des bitmaps de fortes densités, la requête au-dessus a été reprise en remplaçant 2 de ses bitmaps les moins denses par 2 bitmaps de fortes densités comportant des cardinalités se trouvant dans $]2 \cdot 10^6, 3 \cdot 10^6[$. Les bitmaps ajoutés proviennent des deux seules dimensions de l'ensemble de données ayant des bitmaps de telles cardinalités. L'Algorithme VI.8 en annexe affiche la requête modifiée.

Les deux types de requêtes restantes : *Timeseries* et *TopN*, ont aussi été évaluées sur les mêmes densités testées pour la requête *GroupBy*. Les mêmes bitmaps choisis à chaque densité pour la requête *GroupBy* ont été utilisés pour ces deux types de requêtes à la densité correspondante. Deux exemples de ces deux types de requêtes : *Timeseries* et *TopN*, avec des bitmaps de densités moyennes sont donnés, respectivement, dans les Algorithmes VI.9 et VI.10 en annexe.

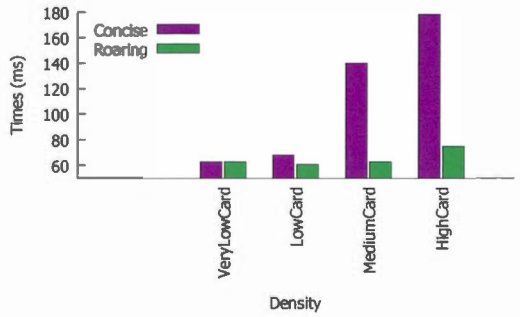
La Figure 5.2 présente les temps de réponse moyens obtenus après avoir exécuté les trois types de requêtes d'agrégations évaluées dans ces essais avec des bitmaps de différentes densités. Sur les très faibles densités, les calculs logiques se font très rapidement entre de minuscules bitmaps, rendant ainsi le temps de réponse des trois types de requêtes indépendant des calculs logiques. C'est ce qui explique les mêmes performances obtenues pour chaque type de requête lorsqu'exécutée sur des segments indexés avec



(a) Temps de réponse de requêtes *GroupBy* effectuant des ET logiques entre des bitmaps de différentes cardinalités



(b) Temps de réponse de requêtes *Timeseries* effectuant des ET logiques entre des bitmaps de différentes cardinalités



(c) Temps de réponse de requêtes *TopN* effectuant des ET logiques entre des bitmaps de différentes cardinalités

Figure 5.2: Temps d'exécution de requêtes d'agrégations opérant des ET logiques entre bitmaps

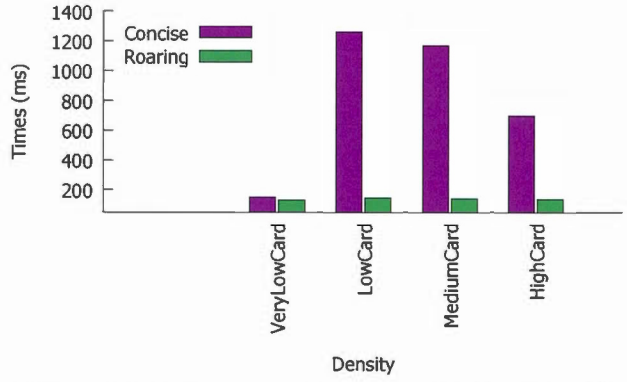
Roaring ou *Concise*.

Sur des bitmaps un peu plus denses, de faibles densités, les performances de *Concise* et de *Roaring* commencent à se démarquer, et révèlent que *Roaring* a permis de répondre 10%, 9% et 6% plus rapidement que *Concise* pour les requêtes respectives : *TopN*, *Timeseries* et *GroupBy*. Plus les densités augmentent, plus les temps de réponse augmentent substantiellement pour *Concise* et de façon assez minime pour *Roaring*, qui se comporte efficacement face à l'extensibilité des données (*scalability*). En effet, sur des bitmaps de moyennes densités, les temps de réponse atteints avec *Roaring* sont de 55% (soit plus de 2 fois) plus rapides que ceux de *Concise* pour chacune des trois types de requêtes évaluées. Sur les fortes densités, la différence entre les performances des deux modèles d'index bitmap compressés augmente encore plus, et *Roaring* a atteint une accélération de 60% pour la requête *Timeseries* et 58% pour les deux requêtes *TopN* et *GroupBy* par rapport à *Concise*.

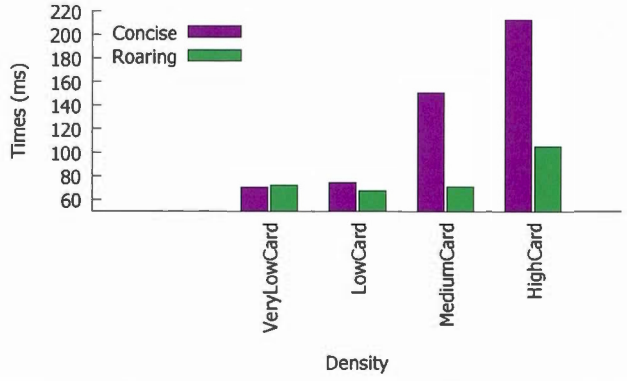
5.4.3 Expériences avec des requêtes de recherche

Dans cette série d'expériences, les performances de *Roaring* et de *Concise* ont été évaluées avec les requêtes de recherche supportées par *Druid* : *Select* et *Search*, en utilisant des bitmaps de différentes densités. Les temps de réponse de la requête du type *Select* ont été évalués en deux versions. La première opérant des ET logiques entre bitmaps pour filtrer l'ensemble de données, et la deuxième qui effectue des OU logiques entre bitmaps. La requête *Select* opérant des OU logiques entre bitmaps de très faibles densités s'exécutant sur des segments indexés avec *Roaring* est donnée sur l'Algorithme VI.11 en annexe.

Cette requête affiche les valeurs des attributs spécifiés dans les parties "dimensions" et "metrics" pour chaque élément du sous-ensemble de données sélectionné par le filtre de la requête, et permet de paginer sur le sous-ensemble de données sélectionné avec



(a) Temps de réponse de requêtes *Select* effectuant des OU logiques entre des bitmaps de différentes densités



(b) Temps de réponse de requêtes *Select* effectuant des ET logiques entre des bitmaps de différentes densités

Figure 5.3: Temps d'exécution de requêtes *Select* opérant des ET et des OU logiques entre bitmaps de différentes densités

5 enregistrements par jour à la fois. Les filtres utilisés avec les précédentes requêtes effectuant des OU et des ET logiques entre bitmaps sur les densités faibles, moyennes et fortes, ont été repris pour tester les performances de la requête *Select* sur ces densités-là avec des filtres calculant des OU et des ET logiques entre bitmaps de type *Roaring* et *Concise*. La Figure 5.3 présente les résultats obtenus.

Les différences entre les performances de *Roaring* et de *Concise* sont similaires à celles observées avec les requêtes d'agrégations, avec *Roaring* qui a accéléré les traitements de OU logiques de 12% sur des bitmaps de très faibles densités, de 88% sur des bit-

maps de faibles et moyennes densités, et de 80% sur des bitmaps de fortes densités, en moyenne, par rapport à *Concise*. Quant aux requêtes *Select* calculant des ET logiques entre bitmaps, *Roaring* a affiché des performances similaires avec celles de *Concise* sur les bitmaps de très faibles densités, tandis qu'il a amélioré les temps de réponse de 9%, 53% et 50%, en moyenne, par rapport à *Concise* sur les faibles, moyennes et fortes densités.

Des expériences ont également été conduites pour évaluer les performances de la requête *Search* sur des bitmaps de différentes densités représentés avec *Roaring* et *Concise*. Un exemple de la requête testée sur des bitmaps de très faibles densités est donné ci-dessous :

```
{
  "queryType": "search",
  "dataSource": "TPCH_benchmark_roaring",
  "granularity": "all",
  "searchDimensions": [
    "l_receiptdate",
    "l_suppkey"
  ],
  "query": {
    "type" : "insensitive_contains",
  "value" : "9"
  },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "context":{"useCache":false,"populateCache":false,"finalize":false}
}
```

Pour évaluer les performances de la requête *Search* sur différentes densités, on a sélectionné deux attributs dimensions à tester pour chaque densité. Sur les très faibles densités, les deux dimensions : *l_receiptdate* et *l_suppkey*, dont les bitmaps possèdent des cardinalités moyennes de 2 645 et 282, respectivement, ont été choisies. Le patron "9" est utilisé dans les critères de recherche pour sélectionner les bitmaps associés à des valeurs appartenant aux deux dimensions choisies et qui possèdent le chiffre 9 dans

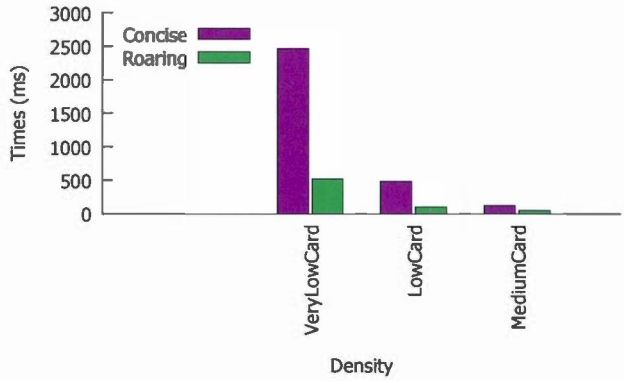


Figure 5.4: Temps d'exécution de requêtes *Search* opérant des OU logiques entre bitmaps de différentes densités

leurs représentations. Sur les faibles densités, les dimensions : *l_discount* et *l_quantity* ont été prises, ayant des bitmaps de cardinalités moyennes de 500 000 et 120 000, respectivement. Le patron "0" a été adopté comme critère de recherche. Tandis que des bitmaps de cardinalités moyennes de 1 500 000 ont été sélectionnés depuis les deux dimensions : *l_returnflag* et *l_shipinstruct*, pour effectuer des tests sur de moyennes densités. La Figure 5.4 présente les temps d'exécution moyens obtenus.

Les bitmaps de très faibles densités ont effectué des OU logiques entre 992 bitmaps indexant des valeurs ayant une occurrence dans leurs représentations du critère recherché par la requête. Les temps de réponse atteints avec *Roaring* sont de 79% ($\approx$ 5 fois) plus rapides que ceux calculés avec *Concise*. Sur les faibles densités, l'agrégation de 16 bitmaps a été faite, et les résultats de *Roaring* ont été 79% ($\approx$ 5 fois) meilleurs que celles de *Concise*. Alors que sur des bitmaps de moyennes densités, *Roaring* a calculé le OU logique de 3 bitmaps 60% ($\approx$ 3 fois) plus vite par rapport à *Concise*.

5.4.4 Expériences évaluant les temps de lecture des bits à 1 des bitmaps

Un autre aspect de performance indiquant l'efficacité d'une libraire de compression d'index bitmap est la rapidité avec laquelle elle permet d'itérer sur un bitmap pour en

extraire les positions des bits à 1, qui représenteront pour le moteur OLAP les positions des enregistrements satisfaisant les critères de recherche d'une requête donnée. Une série d'expériences a été mise en pratique pour comparer la vitesse d'exécution de cette opération-là par chacun de *Roaring* et de *Concise* au sein du moteur OLAP *Druid*. Pour simuler cet aspect de performances, on a opté pour les requêtes de type *Timeseries*, de un, parce que c'est une requête qui permet de faire des agrégations et donc qui peut se limiter à l'affichage que d'une seule valeur agrégée en sortie, ce qui aide à diminuer les coûts des entrées/sorties nécessaires pour écrire les résultats sur disque, contrairement aux requêtes de recherche qui ne font pas d'agrégations et retournent exhaustivement tous les enregistrements satisfaisant les critères de recherche de la requête. Opérations pouvant dominer le temps de réponse globale. De deux, la requête *Timeseries* n'effectue pas de groupements sur les valeurs de dimensions, comme c'est le cas pour les requêtes *GroupBy* et *TopN*, type d'opérations capables de consommer d'importants temps de traitements et dominer les temps de réponse. De ce fait, la requête *Timeseries* a été considérée comme la requête idéale qui laisse les temps de réponse dépendre entièrement du temps passé à parcourir les bitmaps.

Les opérations logiques ont été écartées aussi afin de ne pas pénaliser les temps de réponse obtenus avec *Concise*, du fait que les expériences précédentes ont révélé que cette technique a été plus lente que *Roaring* lors de ces calculs. Un seul bitmap est alors utilisé pour le filtre de chaque requête. Afin de vérifier les performances de *Roaring* et de *Concise* sur différentes densités, les essais traitent des bitmaps de cardinalités : très faibles, faibles, moyennes et fortes. Le code ci-dessous présente une requête *Timeseries* filtrant les données avec un bitmap d'une très faible cardinalité contenant 2 707 bits à 1 :

```
{
  "queryType"  : "timeseries",
  "dataSource" : "TPCH_benchmark_roaring",
  "granularity": "all",
  "context":{"useCache":false,"populateCache":false},
```

```
  "filter": { "type": "selector", "dimension": "l_shipdate", "value": "1997-06-01"
      },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
        { "type": "count", "name": "count"}
  ]
}
```

Pour les essais sur le reste des densités, seulement le bitmap du filtre de la requête est changé avec un autre correspondant à la densité testée. Le bitmap de la densité faible est celui indexant la valeur "5" de la dimension "l_linenumber", et qui renferme 643 287 bits à 1. Celui de la densité moyenne est associé à la valeur "1" de la même dimension "l_linenumber", comportant $15 \cdot 10^5$ bits à 1. Quant aux fortes densités, le bitmap de la valeur "N" appartenant à la dimension "l_returnflag" a été pris, qui a 3 043 852 bits à 1. La Figure 5.5 donne un aperçu des temps moyens capturés pour l'exécution de la requête *Timeseries* avec chacun de *Roaring* et de *Concise* sur les densités décrites précédemment.

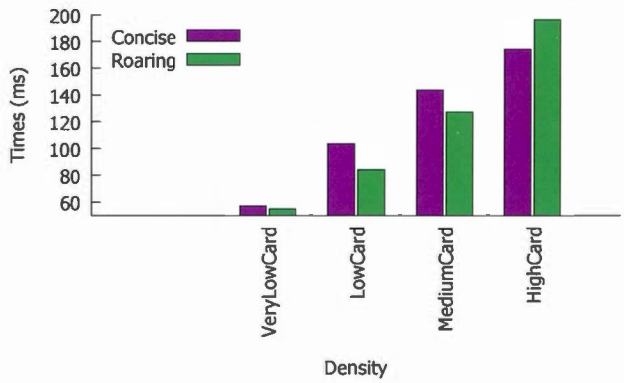


Figure 5.5: Temps d'exécution de requêtes *Timeseries* itérant sur des bitmaps de différentes densités

Sur les plus faibles densités, les résultats des deux techniques de compression bitmap sont presque nuls et équivalents. Sur de faibles densités, *Roaring* a été 18% plus rapide à itérer sur les bitmaps comparativement à *Concise*. Sur les moyennes densités, les

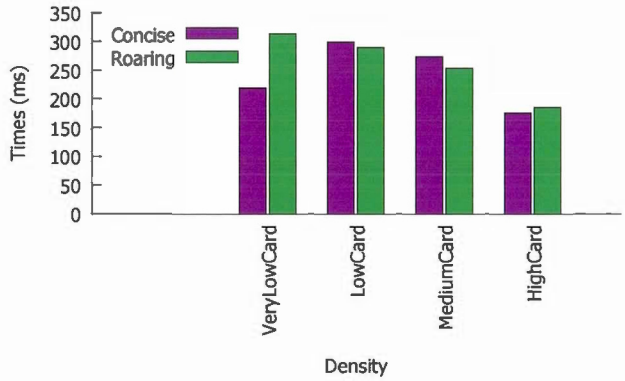


Figure 5.6: Temps d'exécution de requêtes *Timeseries* calculant le complément de bitmaps de différentes densités

temps de réponse obtenus avec *Roaring* ont surpassé ceux avec *Concise* de 11%. Sur les fortes densités, la capacité de *Concise* à compresser les longues suites de bits à 1 lui a permis d'aboutir à des temps de réponse qui dépassent ceux calculés pour *Roaring* de 11%.

5.4.5 Expériences évaluant les temps pour calculer le complément d'un bitmap

Ces expériences évaluent les temps moyens pour exécuter une requête *Timeseries* faisant une agrégation sur les valeurs de l'attribut métrique "count" accédées à l'aide du complément d'un bitmap donné comme filtre. Les bitmaps des précédents essais ont été utilisés ici pour comparer les performances obtenues avec des bitmaps de type *Roaring* et *Concise* sur différentes densités. Les résultats calculés sont rapportés sur la Figure 5.6.

Les résultats montrent que les performances de *Concise* sont nettement supérieures à celles de *Roaring* sur les plus faibles et plus fortes densités (de 30% et de 5%, respectivement). Cela revient à la présence de longues suites de bits de 0 et de 1, respectivement, sur les deux densités, et à la capacité de l'encodage RLE adopté par *Concise* à compresser une telle séquence de bits en un seul mot CPU, sur lequel il sera très fa-

cile ensuite de calculer le complément en inversant seulement le bit du mot indiquant le sens des bits compressés (0 ou 1). Tandis que *Roaring* nécessitera, en général, de créer de nouveaux conteneurs pour représenter les entiers non présents dans le bitmap original (opérations très rencontrées dans le cas des très faibles densités), d'effectuer la conversion des conteneurs du bitmap initial (tableaux vers bitmaps et bitmaps vers tableaux), et de parcourir chaque nouveau conteneur pour y ajouter les entiers manquants. Cependant, sur les faibles et moyennes densités, de telles suites de bits deviennent très rares. Dans ces cas, *Concise* est majoritairement constitué de mots littéraux, et la solution de compression bitmap procède à la lecture de chacun de ces mots afin de calculer leurs inverses, permettant à *Roaring* de montrer une légère avancée en performances par rapport à *Concise*, soit de 3% sur les faibles densités et de 7% sur les moyennes densités.

5.4.6 Expériences sur la consommation de l'espace mémoire avec chacun de *Roaring* et de *Concise*

Après le long travail réalisé pour évaluer les performances temporelles de *Roaring* et de *Concise* lors de l'exécution de requêtes d'analyses sur le SGBD *Druid*, une question ne pouvant passer inaperçue est celle concernant l'utilisation de l'espace mémoire avec chacune des deux techniques de compression bitmap évaluées. Pour donner une brève idée des performances spatiales des deux modèles de compression bitmap, on rapporte la taille totale du fichier plat stockant les index bitmap sérialisés sur disque de chacune des deux sources de données (l'une étant indexée avec *Roaring* et la seconde avec *Concise*) utilisées lors des essais précédents. La Figure 5.7 montre la taille occupée sur disque par les fichiers plats des bitmaps de type *Roaring* et *Concise*.

Bien que la méthode *Roaring* offre, en général, de remarquables temps de réponse, les résultats montrent qu'elle est d'environ 15% plus gourmande que *Concise* en matière d'occupation d'espace mémoire. Cependant, ceci n'affecte que très rarement la vitesse

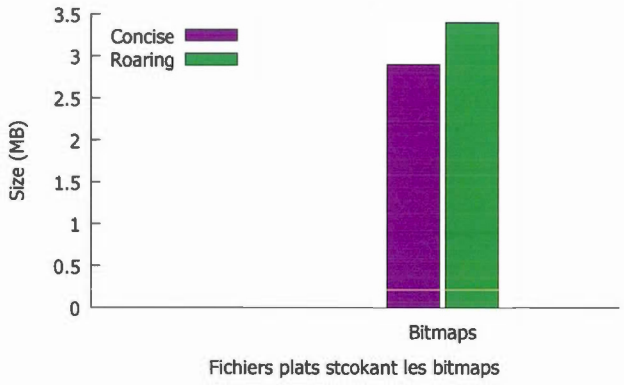


Figure 5.7: Tailles des espaces mémoire occupés par les index *Roaring* et *Concise* sur le disque

de lecture des bitmaps depuis le disque, car avec l'index de haut niveau du modèle *Roaring*, le CPU évite souvent, pour un bitmap donné, de charger des conteneurs ne participant pas aux traitements en cours d'exécution, contrairement à *Concise* pour lequel tout le bitmap doit être chargé depuis le disque afin d'être traité. Aussi, même lorsque des conteneurs d'un bitmap *Roaring* sont lus par le processeur, ce dernier pourra les traiter beaucoup plus rapidement en effectuant des traitements super-scalaires capables de traiter plusieurs suites de bits en parallèle, chose impossible à réaliser avec les mots d'un bitmap *Concise* qui nécessitent des embranchements conditionnels pour chaque mot CPU ralentissant au final les temps d'exécution.

5.5 Conclusion

Ce chapitre fait état du projet d'intégration de la solution de compression bitmap *Roaring bitmap* au moteur OLAP *Druid*, tâche réalisée en collaboration avec l'équipe de développement logiciel de la société *Meta-markets*. Tout d'abord, une introduction sur les principaux faits ayant mené à la réalisation de ce projet a été donnée. Ensuite, une présentation détaillée d'importants concepts du système de traitement de données massives *Druid* a été faite. Une section donnant un bref aperçu des différents types de requêtes d'analyses supportées par *Druid* et pouvant utiliser de bitmaps lors de l'exé-

cution s'en est suivie. La dernière section a présenté les expériences conduites pour évaluer les performances temps-espace de *Roaring* et de *Concise* au sein de *Druid*. *Roaring* a montré qu'il pouvait améliorer jusqu'à 5 fois les temps de réponse de requêtes de recherche, et jusqu'à 2 fois les temps de traitement de requêtes d'agrégations.

Après avoir constaté qu'il arrivait à *Concise* d'être plus compact que *Roaring* et de traiter des bitmaps plus rapidement lorsque ces derniers comportent de longues suites de bits à 1, un nouveau modèle de conteneurs a été développé pour *Roaring* qui est spécialement adapté à ce genre de conditions et qui est capable de compresser de longues suites de bits à 1 en appliquant une forme d'encodage RLE. Une évaluation des performances de cette nouvelle version de *Roaring* sur le moteur OLAP *Druid* est envisagée comme travaux futurs.

CHAPITRE VI

CONCLUSION

Les index bitmap sont très utilisés dans les bases de données et moteurs de recherche. Leur forme compacte et leur capacité à tirer profit du traitement parallèle de bits (*bit-level parallelism*) dans les CPU leur permettent d'accélérer les temps de réponse des requêtes. L'efficacité des index bitmap est largement reconnue lorsque utilisés sur des ensembles de faibles cardinalités. Cependant, sur des ensembles de fortes cardinalités, leurs performances spatio-temporelles se dégradent considérablement.

Plusieurs contributions scientifiques se sont penchés sur cette problématique dans le but de réduire la taille des index bitmap et d'accélérer leurs temps de traitements. Les travaux de cette thèse abordent cette problématique, en commençant par présenter un état de l'art des trois types de solutions proposées dans la littérature pour résoudre ce problème. Le premier type de solutions étudié est le paquetage des bitmaps, qui permet d'indexer plusieurs valeurs distinctes avec un seul bitmap. Bien que ces contributions réduisent l'espace mémoire occupé par les bitmaps, elles engendrent souvent un coût lié à une phase de vérification supplémentaire, appelée phase de vérification candidate, qui requiert l'accès aux données stockées sur le disque, consommant par conséquent d'importants temps de traitements qui finissent par dominer les temps de réponse des requêtes. Quelques importants travaux scientifiques introduits dans la littérature pour diminuer les temps de cette phase de vérification supplémentaire ont été présentés dans

cette thèse.

Le deuxième type de solutions traitant la problématique de la thèse a été étudié aussi. Ce type repose sur l'encodage des bitmaps afin de réduire le nombre de bitmaps à stocker et à lire lors de l'exécution de requêtes. Les propositions traitant de ce type de solutions se classent en deux grandes catégories : techniques d'encodage basiques et techniques d'encodage composées. La première catégorie renferme trois formes essentielles d'encodage des bitmaps : par égalité, par rang (Chan et Ioannidis, 1998a) et par intervalle (Chan et Ioannidis, 1999), chacun étant respectivement adapté pour les requêtes à prédicats d'égalités, d'intervalles à sens unique et à deux sens. La deuxième catégorie de techniques d'encodage regroupe des solutions combinant les encodages basiques afin de représenter les bitmaps en plusieurs composantes. Ces composantes sont construites avec moins de bitmaps comparé aux encodages basiques et en lisent encore moins lors de traitements de requêtes, améliorant ainsi l'occupation de l'espace mémoire et les temps de réponse. Ces techniques se répartissent en des encodages multicomposantes et multiniveaux.

Le troisième type de solutions, sur lequel portent les contributions de la thèse, consiste en l'application de techniques de compression sur des bitmaps individuels. Les premiers travaux introduits dans la littérature de ce type de solutions proposent des méthodes qui atteignent de forts taux de compression une fois appliquées sur des bitmaps, mais qui requièrent des temps de calculs faramineux pour exécuter des traitements sur les bitmaps une fois compressés. Cela revient au fait de devoir entièrement décompresser un bitmap avant de pouvoir y effectuer des opérations. Parmi les travaux faisant partie de cette gamme de contributions, cette thèse a commencé par un survol des méthodes de compression textuelle, comme : le codage de Huffman (Huffman, 1952), *LZ77* (Ziv et Lempel, 1977; Gailly, 1998), *LLRUN* (Fraenkel et Klein, 1985), etc. Ensuite, des techniques de compression d'entiers ont été présentées, telles que : Golomb et Rice coding (Rice et Plaunt, 1971), Interpolative coding (Moffat et Stuiver, 2000),

Elias gamma et delta coding (Elias, 1975), etc. Pour finir, des systèmes de compression génériques, par exemple : *ExpGol* (Teuhola, 1978) ont été visités.

Après avoir constaté le besoin d'une technique de compression bitmap compacte et efficace, Antoshenkov a introduit la solution *BBC* (Antoshenkov, 1995), qui combine une compression par plage de valeurs avec une représentation bitmap sous forme d'une chaîne de bits alignée par blocs d'octets. Cette méthode consomme un peu plus d'espace mémoire que les anciennes solutions, mais des expériences (Johnson, 1999) ont révélé que sa capacité à faire des traitements sur des bitmaps compressés a permis d'améliorer de 50 fois les temps d'exécution d'opérations logiques lorsque comparée à des solutions de compression traditionnelles.

Wu *et al.* (2001a) ont travaillé sur l'amélioration de la solution *BBC*, et ont fini par introduire la méthode de compression *WAH*, qui se caractérise par un format beaucoup plus simple que celui de *BBC* et qui s'adapte beaucoup mieux aux architectures des CPU grâce à un alignement de bits par mots CPU. Ainsi, *WAH* a permis d'améliorer de 12 fois les temps de calculs d'opérations logiques entre bitmaps comparé à *BBC*. Cependant, l'alignement des bits par blocs d'octets permet à *BBC* d'être plus compact que *WAH*, tel que rapporté par les expériences de Wu *et al.* (2006), qui ont révélé que les bitmaps compressés par *BBC* sont de 60% moins volumineux que ceux compressés par *WAH*.

Plusieurs travaux basés sur le modèle *WAH* ont été introduits par la suite, dans le but d'améliorer un peu plus les taux de compression et les temps de traitement de la méthode *WAH*. Cette thèse a rapporté quelques importantes contributions parmi ces solutions, comme : *Concise* et *PLWAH* qui améliorent de 2 fois les taux de compression par rapport à *WAH* sur des bitmaps de faibles densités, sans augmenter les temps de calculs. Une approche adoptant un encodage hybride pour compresser les bitmaps a été également évoquée dans la thèse. Ce travail a été réalisé au sein du SGBD *RIDBit* (O'Neil

et al., 2007) et consiste à basculer la représentation d'un bitmap vers une liste d'entiers, lorsque sa densité est jugée assez faible.

Après le survol de littérature, la première contribution scientifique de cette thèse a été introduite. Il a été constaté lors de nos recherches que la plupart des modèles de compression bitmap proposés ces 15 dernières années se basent sur le modèle de *BBC*. Cette thèse propose un nouveau modèle de compression bitmap, appelé *Roaring bitmap*, qui adopte une combinaison de plusieurs structures de données pour compresser un bitmap et accélérer ces temps de traitements. Après avoir présenté en détail les concepts de ce nouveau format de bitmaps, sa méthode de compression et les différentes stratégies adoptées pour exécuter rapidement plusieurs types d'opérations sur des bitmaps : ET ou OU logiques, accès aléatoires, etc. Des expériences ont été mises en œuvre pour comparer les performances de cette technique avec celles d'autres librairies parmi les plus connues dans la littérature : *WAH* et *Concise*. Sur des données synthétiques, les résultats ont montré que dans les cas de faibles densités, *Roaring bitmap* réduit de 50% la quantité de mémoire occupée par des bitmaps compressés avec *Concise* et de 75% celle de bitmaps compressés par *WAH*. Tandis que pour calculer des opérations ET et OU logiques entre bitmaps, *Roaring bitmap* a été de 4 à 5 fois plus rapide que *Concise* et *WAH*. Le nouveau format a été aussi plus rapide que les deux autres solutions lors d'expériences évaluant les temps moyens pour inverser un bit aléatoire (insérer ou supprimer un entier) dans un bitmap. Sur des données réelles, *Roaring bitmap* a consommé jusqu'à 2 fois moins d'espace que *WAH* et *Concise*, et a pu exécuter des opérations ET logiques jusqu'à 1100 fois plus vite que les deux autres méthodes.

Après le survol de littérature, la première contribution scientifique de cette thèse a été introduite. Il a été constaté lors de nos recherches que la plupart des modèles de compression bitmap proposés ces 15 dernières années se basent sur le modèle de *BBC*. Cette thèse propose un nouveau modèle de compression bitmap, appelé *Roaring bitmap*, qui adopte une combinaison de plusieurs structures de données pour compresser un bitmap

et accélérer ces temps de traitements. Tout d'abord, la thèse a introduit les concepts de ce nouveau format de bitmaps, sa méthode de compression et les différentes stratégies adoptées pour exécuter rapidement plusieurs types d'opérations sur des bitmaps, comme : les ET et OU logiques, les accès aléatoires, etc. Par la suite, des expériences ont été mises en œuvre pour comparer les performances de cette technique avec celles d'autres librairies parmi les plus connues dans la littérature : *WAH* et *Concise*. Sur des données synthétiques, les résultats ont montré que dans les cas de faibles densités, *Roaring bitmap* réduit de 50% la quantité de mémoire occupée par des bitmaps compressés avec *Concise* et de 75% celle de bitmaps compressés par *WAH*. Tandis que pour calculer des opérations ET et OU logiques entre bitmaps, *Roaring bitmap* a été de 4 à 5 fois plus rapide que *Concise* et *WAH*. Le nouveau format a été aussi plus rapide que les deux autres solutions lors d'expériences évaluant les temps moyens pour inverser un bit aléatoire (insérer ou supprimer un entier) dans un bitmap. Sur des données réelles, *Roaring bitmap* a consommé jusqu'à 2 fois moins d'espace que *WAH* et *Concise*, et a pu exécuter des opérations ET logiques jusqu'à 1100 fois plus vite que les deux autres méthodes.

D'autres tests ont été conduits aussi pour comparer les performances de *Roaring bitmap* avec celles de *Concise* sur des bitmaps sérialisés en mémoire externe et manipulés via du *memory-mapping*. Les résultats ont montré que *Roaring bitmap* a été jusqu'à $1,6$ fois plus compact sur disque et a permis de calculer le OU logique et le ET logique de plusieurs bitmaps, respectivement, 129 fois et 94 fois plus rapidement comparé à *Concise*. La vitesse de récupération des bits positifs d'un bitmap a également été testée, et *Roaring bitmap* a été 5 fois plus efficace que *Concise*.

La majorité des librairies de représentation de bitmaps disponibles en code libre (*Open Source*) ne supportent que des bitmaps d'au plus 2^{32} entrées. Avec le volume des collections de données qui ne cessent de croître de nos jours, ces solutions de représentation de bitmaps s'avèrent impraticables dans de nombreuses situations. Les ingénieurs

du moteur de recherche Apache Lucene (Apache, 2012) ont rencontré ce problème et ont proposé une librairie *OpenBitSet* pouvant allouer des bitmaps supportant jusqu'à $64 \times 2^{32} - 1$ entrées. Cependant, cette solution n'applique pas de forme de compression, et reste de ce fait peu efficace en matière de taux de compression et temps de traitement.

Cette thèse a introduit trois nouveaux modèles de compression bitmap basés sur le modèle *Roaring bitmap*, et qui permettent d'allouer des bitmaps ayant jusqu'à 2^{64} entrées. Le premier modèle est celui de *RoaringTreeMap*, qui adopte un arbre rouge-noir dont les nœuds sont composés d'une clé de 32 bits et d'un *Roaring bitmap*. Un groupe d'entiers de 64 bits partageant les mêmes 32 bits de poids fort sont gardés dans un nœud de l'arbre. La clé du nœud maintient les 32 bits de poids fort du groupe d'entiers et le *Roaring bitmap* associé au nœud conserve les 32 bits de poids faible.

Le deuxième est le format *RoaringTwoLevels* dont la structure combine un tableau dynamique sur le premier niveau du bitmap, et des conteneurs du modèle *Roaring bitmap* sur le niveau subséquent. Une entrée du premier niveau est composée d'une clé de 64 bits et d'un conteneur. Une telle entrée regroupe des entiers de 64 bits partageant les mêmes 48 bits de poids fort. Les 48 bits communs sont gardés dans les bits de poids fort de la clé, et les 16 bits de poids faible restants du groupe d'entiers sont conservés dans le conteneur. Les 16 bits non utilisés d'une clé servent à garder la cardinalité du groupe d'entiers indexés par une entrée de premier niveau.

La troisième structure, *LazyRoaring*, utilise des tableaux dynamiques sur les hauts niveaux de la structure avec des conteneurs du format *Roaring bitmap* sur le bas niveau. Le premier niveau contient un tableau dynamique dans lequel chaque entrée renferme une clé de 32 bits et un tableau dynamique de deuxième niveau. Une entrée de ce dernier comporte une clé de 16 bits et un conteneur du modèle *Roaring bitmap*. Un groupe d'entiers ayant les mêmes 32 bits de poids fort, sont regroupés dans une entrée du ta-

bleau de premier niveau. La clé de l'entrée gardera la valeur des 32 bits communs, et les 32 bits restants seront passés au tableau de deuxième niveau associé à l'entrée. Un tableau de deuxième niveau rassemble à son tour un groupe d'entiers de 32 bits qui ont des 16 bits de poids fort équivalents dans une même entrée. Les 16 bits partagés seront gardés par la clé de l'entrée, et les 16 bits restants seront conservés dans le conteneur pointé par l'entrée.

Des expériences sur des données synthétiques suivants deux types de distributions ont été réalisées pour évaluer les performances de ces trois nouveaux modèles avec celles de la solution d'Apache Lucen, *OpenBitSet*, et celles de collections Java proposées dans le paquetage Java.Util : *ArrayList*, *LinkedList*, *HashSet* et *TreeSet*. Les résultats ont révélé que les trois modèles proposés ont calculé des intersections jusqu'à $\approx$ 6 millions de fois plus vite qu'*OpenBitSet*, jusqu'à $\approx$ 63 milles fois plus rapidement que les deux collections Java *ArrayList* et *LinkedList*, et $\approx$ 6 fois plus efficacement que les structures *HashSet* et *TreeSet*. Lors de l'évaluation des temps moyens pour calculer une union entre deux bitmaps, les trois formats proposés ont été jusqu'à $\approx$ 3 millions de fois plus efficaces qu'*OpenBitSet*. Cette dernière structure a montré aussi qu'elle était jusqu'à $\approx$ 14 millions de fois plus lente que les trois modèles introduits pour insérer un entier de 64 bits généré aléatoirement. En matière d'occupation d'espace mémoire, la structure *OpenBitSet* et les collections Java ont été jusqu'à $\approx$ 127 mille fois et jusqu'à $\approx$ 4 fois, respectivement, plus gourmandes en consommation mémoire comparé aux trois modèles présentés.

Afin de valider le format de la solution *Roaring bitmap* au sein d'un SGBD réel, cette méthode de compression bitmap a été intégrée à un moteur OLAP disponible en code libre : *Druid*. Ce système utilise des bitmaps pour accélérer les opérations OLAP de type *drill-down*, et adoptait auparavant la solution *Concise* comme seule méthode de compression bitmap. *Druid* supporte plusieurs types de requêtes d'analyses pouvant faire recours à des bitmaps pour améliorer leurs temps de réponse. Ces requêtes d'ana-

lyses se divisent en deux classes : des requêtes d'agrégations, comme *GroupBy*, *Timeseries* et *TopN*, et des requêtes de recherche, telles que *Search* et *Select*. Des expériences ont été conduites pour comparer les temps de réponse obtenus pour ces différents types de requêtes lorsque accélérées avec *Concise* ou *Roaring bitmap*. Les résultats ont montré que *Roaring bitmap* a permis de réduire jusqu'à 2 fois les temps de réponse de requêtes d'agrégations, et jusqu'à 5 fois les temps de traitements de requêtes de recherche comparé à *Concise*. Cette thèse a présenté les principaux points de ce travail, notamment une présentation détaillée du moteur OLAP *Druid* et de ses requêtes supportées, des expériences mises en œuvre, des résultats obtenus et des analyses de ces résultats.

6.1 Travaux futurs

Lors des expériences qui ont comparé les performances de *Roaring bitmap* et de *Concise* au sein de *Druid*, il a été constaté que *Concise* a pu réaliser quelques types de traitements un peu plus efficacement que *Roaring bitmap* sur des données contenant de longues suites de bits à 1. Cela revient à l'encodage RLE adopté par *Concise* qui lui permet de représenter une telle séquence de bits en un seul mot CPU, ce qui l'aide à accélérer le parcours des bitmaps. Suite à cette observation, une nouvelle version de *Roaring bitmap* a été mise en œuvre qui supporte une forme de compression RLE au niveau des conteneurs, permettant de représenter efficacement des longues séquences de bits à 1 dans un bitmap. Une évaluation des performances de cette nouvelle version de *Roaring bitmap* au sein du moteur OLAP *Druid* est laissée comme travaux futurs.

Le modèle *LazyRoaring* représentant des entiers de 64 bits a montré de remarquables performances lors du calcul d'opérations d'unions entre bitmaps par rapport aux deux autres modèles introduits. Ceci est dû à sa stratégie *copy-on-write*, qui retarde la copie de quelques conteneurs et de quelques entrées du deuxième niveau lors d'une opération logique qu'au moment où l'un de ces objets partagés aura à être modifié pour un bitmap

quelconque. L'implémentation de cette stratégie pour le modèle *Roaring bitmap* sur des entiers à 32 bits promettrait de remarquables améliorations dans les temps de calcul d'opérations logiques.

Le format de *Roaring bitmap* divisant les données entre différents conteneurs permet de traiter ses conteneurs indépendamment les uns des autres. Sur une machine avec une architecture parallèle, les conteneurs d'un *Roaring bitmap* pourraient être répartis entre différents processeurs afin de bénéficier de calculs parallèles, ce qui pourrait améliorer les temps de traitements d'un *Roaring bitmap*. De précédents travaux (Sinha et Winslett, 2007) ont appliqué des calculs parallèles sur des solutions basées sur le modèle *WAH*, mais le format de ces méthodes ne simplifie pas une telle tâche comparé à celui de *Roaring bitmap*. Un projet mettant en œuvre des expériences comparant les performances de *Roaring bitmap* avec celles d'autres solutions basées sur le modèle *WAH*, lorsque les différentes opérations évaluées sont exécutées à l'aide de traitements parallèles, pourrait donner des résultats intéressants.

RÉFÉRENCES

Anh, V. et Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval*, *8*(1), 151–166. http://dx.doi.org/10.1023/B:INRT.0000048490.99518.5c

Anh, V. et Moffat, A. (2010). Index compression using 64-bit words. *Software : Practice and Experience*, *40*(2), 131–147. http://dx.doi.org/10.1002/spe.948

Antoshenkov, G. (1995). Byte-aligned Bitmap Code. Dans *Proceedings of the Conference on Data Compression (DCC '95)*, p. 476., Snowbird, UT. IEEE. http://dx.doi.org/10.1109/DCC.1995.515586

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

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

Apache (2014a). Kylin. Récupéré de http://kylin.apache.org/

Apache (2014b). Solr. Récupéré de http://lucene.apache.org/solr/

Bentley, J. L. et Yao, A. C. (1976). An almost optimal algorithm for unbounded searching. *Information Processing Letters*, *5*(3), 82 – 87. http://dx.doi.org/http://dx.doi.org/10.1016/0020-0190(76)90071-5. Récupéré de http://www.sciencedirect.com/science/article/pii/0020019076900715

Bertchold, S., Bohm, C. et Kriegel, H. (1992). Parameterised compression for sparse bitmaps. Dans *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '92)*, 274–285., Copenhagen, Denmark. ACM. http://dx.doi.org/10.1145/133160.133210

Bertchold, S., Bohm, C. et Kriegel, H. (1998). The pyramid-technique : Towards breaking the curse of dimensionality. Dans *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, volume 27(2), 142–153., New York, NY, USA. ACM. http://dx.doi.org/10.1145/276304.276318

Bhan, M., Rajanikanth, K. et KUMAR, S. (2012). Multi-level and Multi-component Bitmap Encoding for Efficient Search Operations. *Database Systems Journal*, *3*(4), 47–60. Récupéré de http://ideas.repec.org/a/aes/dbjour/v3y2012i4p47-60.html

Bin, H. et Yu-Xing, P. (2011). An Efficient Two-Level Bitmap Index for Cloud Data Management. Dans *IEEE 3rd International Conference on Communication Software and Networks (ICCSN 2011)*, 509–513., Xi'an, Chine. IEEE. http://dx.doi.org/10.1109/ICCSN.2011.6014776

Bookstein, A. et Klein, S. (1991). Generative models for bitmap sets with compression applications. Dans *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR '91)*, 63–71., Chicago, Illinois, USA. ACM. http://dx.doi.org/10.1145/122860.122867

Booth, H. D. (1992). An overview over red-black and finger trees.

Brodal, G. S., Makris, C. et Tsichlas, K. (2006). Purely functional worst case constant time catenable sorted lists. In *Proc. 14th Annual European Symposium on Algorithms*, volume 4168 de *Lecture Notes in Computer Science* 172–183. Springer Verlag, Berlin

Chambi, S. (2014). Benchmarking Roaring and Concise compressed bitmaps using memory–mapped files. Récupéré de https://github.com/samytto/MemoryMappedBitmaps

Chambi, S. (2015a). Benchmarking space consumptions for 64–bit bitmap compression schemes. Récupéré de 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. Récupéré de https://bitbucket.org/samytto/datastructures64-bitints/src/432c0bce7ca3/test/src/main/java/microbenchmarks/?at=master

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

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

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

Chambi, S., Allen, C. et Lemire, D. (2015). BenchmarkingRoaringOnDruid. Récupéré de https://github.com/samytto/BenchmarkingRoaringOnDruid

Chambi, S., Lemire, D. et Godin, R. (2016a). Nouveaux modèles d'index bitmap compressés à 64 bits. Dans *12e journées francophones sur les Entrepôts de Données et l'Analyse en Ligne (EDA'16)*, volume RNTI-B-12, 1–16., Aix-en-Provence, France. RNTI.

Chambi, S., Lemire, D. et Godin, R. (2016b). Vers de meilleurs performances avec des Roaring bitmaps. *Techniques et Science Informatique (TSI)*, *35*(3), 335–355. http://dx.doi.org/10.3166/TSI.35.335-355

Chambi, S., Lemire, D., Godin, R., Boukhalfa, K., Allen, C. R. et Yang, F. (2016c). Optimizing Druid with Roaring bitmaps. Dans *20th International Database Engineering and Applications Symposium (IDEAS'16)*, 77–86., Montreal, QC, Canada. ACM.

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

Chambi, S., Lemire, D., Owen, K. et Godin, R. (2016d). Better bitmap performance with Roaring bitmaps. *Software Practice and Experience (SPE)*, *46*(5), 709–719. http://dx.doi.org/10.1002/spe.2325

Chan, C. et Ioannidis, Y. (1998a). Bitmap index design and evaluation. Dans *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, volume 27(2), 355–366., New York, NY, USA. ACM. http://dx.doi.org/10.1145/276304.276336

Chan, C. et Ioannidis, Y. (1999). An efficient bitmap encoding scheme for selection queries. Dans *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, volume 28(2), 215–226., New York, NY, USA. ACM. http://dx.doi.org/10.1145/304182.304201

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

Chang, J., Chen, Z., Zheng, W., Cao, J., Wen, Y., Peng, G. et Huang, W. (2015). SPLWAH : A bitmap index compression scheme for searching in archival internet traffic.

Dans *2015 IEEE International Conference on Communications (ICC)*, ICC 2015, 7089–7094., London, England. IEEE. http://dx.doi.org/10.1109/ICC.2015.7249457

Chen, Z., Wen, Y., Cao, Y., Zheng, W., Chang, J., Wu, Y., Ma, G., Hakmaoui, M. et Peng, G. (2015). A survey of bitmap index compression algorithms for Big Data. *Tsinghua Science and Technology*, *20*, 100–115. http://dx.doi.org/10.1109/TST.2015.7040519

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

Colantonio, A. et Di Pietro, R. (2010). Concise : Compressed 'n' Composable Integer Set. *Information Processing Letters*, *110*(16), 644–650. http://dx.doi.org/10.1016/j.ipl.2010.05.018

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

Council, T. P. P. (2014). TPC BENCHMARK H. Récupéré de http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf

Culpepper, J. S. et Moffat, A. (2010). Efficient set intersection for inverted indexing. *ACM Transactions on Information Systems (TOIS)*, *29*(1), 1 :1–1 :25. http://dx.doi.org/10.1145/1877766.1877767

Cutting, D. et Pedersen, J. (1990). Optimization for dynamic inverted index maintenance. Dans *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, 405–411., Brussels, Belgium. ACM. http://dx.doi.org/10.1145/96749.98245

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

Deliège, F. et Pedersen, T. B. (2010). Position List Word aligned Hybrid : optimizing space and performance for compressed bitmaps. Dans *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*, 228–239., Lausanne, Switzerland. ACM. http://dx.doi.org/10.1145/1739041.1739071

Deveaux, J., Rau-Chaplin, A. et Zeh, N. (2007). Adaptive tuple differential coding. Dans *18th International Conference on Database and Expert Systems Applications (DEXA '07)*, volume 4653 de *Lecture Notes in Computer Science*,

109–119., Regensburg, Germany. Springer. http://dx.doi.org/10.1007/978-3-540-74469-6_12

Druid (2015a). Groupby query. Récupéré de http://druid.io/docs/latest/querying/groupbyquery.html

Druid (2015b). Powered by Druid. Récupéré de http://druid.io/druid-powered.html

Druid (2015c). Querying. Récupéré de http://druid.io/docs/0.8.2/querying/querying.html

Druid (2015d). Search query. Récupéré de http://druid.io/docs/latest/querying/searchquery.html

Druid (2015e). Select query. Récupéré de http://druid.io/docs/latest/development/select-query.html

Druid (2015f). Timeseries query. Récupéré de http://druid.io/docs/latest/querying/timeseriesquery.html

Druid (2015g). TopN query. Récupéré de http://druid.io/docs/latest/querying/topnquery.html

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, *21*(2), 194–203. http://dx.doi.org/10.1109/TIT.1975.1055349

Fraenkel, A. et Klein, S. (1985). Novel compression of sparse bit-strings. In A. Apostolico (dir.), *Combinatorial Algorithms On Words*, volume 12 de *NATO ASI Series* book section 3, 169–183. Berlin : Springer

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

Fusco, F., Stoecklin, M. P. et Vlachos, M. (2010). NET-FLi : On-the-fly compression, archiving and indexing of streaming network traffic. Dans *36th International Conference on Very Large Data Bases (VLDB 2010)*, volume 3, 1382–1393., Singapore. Very Large Database Endowment. Récupéré de http://www.comp.nus.edu.sg/$\sim$vldb2010/proceedings/files/papers/I01.pdf

Gailly, J. M. Adler, M. (1998). Zlib. Récupéré de http://www.zlib.net/

Godin, R. (2012). *Les systèmes de gestion de bases de données par l'exemple*, chapitre Évaluation des requêtes relationnelles. Loze-Dion, (3 éd.).

Golomb, S. (1966). Run-length encodings. *IEEE Transactions on Information Theory*, *12*(3), 399–401. http://dx.doi.org/10.1109/TIT.1966.1053907

Goyal, N. et Sharma, Y. (2009). New binning strategy for bitmap indices on high cardinality attributes. Dans *Proceedings of the 2nd Bangalore Annual Compute Conference*, p. Article No. 22., Bangalore, India. ACM. http://dx.doi.org/10.1145/1517303.1517327

Grand, A. (2015). Frame of reference and Roaring bitmaps. Récupéré de https://www.elastic.co/blog/frame-of-reference-and-roaring-bitmaps

Guzun, G., Canahuate, G., Chiu, D. et Sawin, J. (2014). A tunable compression framework for bitmap indices. Dans *30th IEEE International Conference on Data Engineering (ICDE 2014)*, 484–495., Chicago, IL, USA. IEEE. Récupéré de http://user.engineering.uiowa.edu/~gguzun/papers/Guzun_ICDE_2014.pdf

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

Hall, A., Bachmann, O., Bussow, R., Ganceanu, S. et Nunkesser, M. (2012). Processing a trillion cells per mouse click. Dans *The 38th International Conference on Very Large Data Bases*, volume 5 de *VLDB 2012*, 1436–1446., Istanbul, Turkey. VLDB.

Huffman, D. (1952). A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, *40*(9), 1098–1101. http://dx.doi.org/10.1109/JRPROC.1952.273898

Jakobsson, M. (1978). Huffman coding in bit-vector compression. *Information Processing Letters*, *7*(6), 304 – 307. http://dx.doi.org/http://dx.doi.org/10.1016/0020-0190(78)90023-6

Johnson, T. (1999). Performance measurements of compressed bitmap indices. Dans *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*, volume 27(2), 278–289., San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. Récupéré de http://dl.acm.org/citation.cfm?id=645925.671351

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

Kevin, B. et Raghu, R. (1999). Bottom-up computation of sparse and iceberg CUBEs. *sigmod-record*, *28*(2), 359–370.

Korn, D. et Vo, K. (1995). Vdelta : Differencing and compression. In B. Krishnamurthy (dir.), *Practical Reusable Unix Software*, volume 1 book section Libraries and File System Architecture, 57–58. New York, NY, USA : John Wiley & Sons

Koudas, N. (2000). Space efficient bitmap indexing. Dans *Proceedings of the ninth international conference on Information and knowledge management*, 194–201., McLean, VA (near Washington, DC), USA. ACM. http://dx.doi.org/10.1145/354756.354819

Lemire, D. et Boytsov, L. (2013). Decoding billions of integers per second through vectorization. *Software : Practice and Experience*. http://dx.doi.org/10.1002/spe.2203

Lemire, D., Kaser, O. et Aouiche, K. (2010). Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, *69*(1), 3–28. http://dx.doi.org/10.1016/j.datak.2009.08.006

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

Liang, S. (1999). *The Java Native Interface : programmer's guide and specification* (1st éd.). Addison-Wesley.

Ma, G., Guo, Z., Li, X., Chen, Z., Cao, J. et Guo, X. (2014). Breadzip : a combination of network traffic data and bitmap index encoding algorithm. Dans *IEEE International Conference on Systems, Man and Cybernetics (SMC)*, SMC, 3235–3240., San Diego, CA. IEEE. http://dx.doi.org/10.1109/SMC.2014.6974426

Melnik, S., Gubarev, A., Long, J., Romer, G., Shivakumar, S., Tolton, M. et Vassilakis, T. (2011). Dremel : Interactive analysis of web-scale datasets. Dans *The 36th International Conference on Very Large Data Bases*, volume 3 de *VLDB 2011*, 330–339., Singapore. VLDB.

Metamarkets (2015). Metamarkets. Récupéré de https://metamarkets.com/

Metamx (2015). Extendedset. Récupéré de https://github.com/metamx/extendedset

Microsystems, S. (2001). Using Native Code. Récupéré de http://www.mastercorp.free.fr/Ing1/Cours/Java/java_lesson1/doc/Tutorial/performance/JPNativeCode_fm.htm

Moffat, A. et Stuiver, L. (2000). Binary interpolative coding for effective index compression. *Information Retrieval*, *3*(1), 25–47. http://dx.doi.org/10.1023/A:1013002601898

O'Neil, E., O'Neil, P. et Wu, K. (2007). Bitmap index design choices and their performance implications. Dans *11th International Database Engineering and Applications Symposium (IDEAS '07)*, 72–84., Banff, Alberta, Canada. IEEE. http://dx.doi.org/10.1109/IDEAS.2007.4318091

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

O'Neil, P., O'Neil, E., Chen, X. et Revilak, S. (2009). The star schema benchmark and augmented fact table indexing. Dans *First TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2009)*, 237–252., Lyon, France. Springer. http://dx.doi.org/10.1007/978-3-642-10424-4_17

O'Neil, P. et Quass, D. (1997). Improved query performance with variant indexes. Dans *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, 38–49., New York, NY, USA. ACM. http://dx.doi.org/10.1145/253260.253268

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

Oracle (2016). Compressedoops. Récupéré de http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html

Rice, R. F. et Plaunt, J. (1971). Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology, 19*(6), 889–897. http://dx.doi.org/10.1109/TCOM.1971.1090789

Rios, M. et Lin, J. (2012). Distilling massive amounts of data into simple visualizations : Twitter case studies. Dans *The 6th International AAAI Conference on Weblogs and SocialL Media*, ICWSM 2012, 22–25., Dublin, Ireland. ICWSM.

Roaring's team (2014a). Benchmarking Roaring bitmap on real data. Récupéré de https://github.com/lemire/RoaringBitmapReal/blob/master/src/main/java/org/roaringbitmap/experiments/BenchmarkReal3.java

Roaring's team (2014b). Benchmarking Roaring bitmap on synthetic data. Récupéré de https://github.com/lemire/RoaringBitmapSynth/blob/master/src/main/java/org/roaringbitmap/experiments/colantonio/AsymmetricBenchmark.java

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

Rotem, D., Stockinger, K. et Wu, K. (2005). Optimizing candidate check costs for bitmap indices. Dans *Proceedings of the 14th ACM international conference on Information and knowledge management (CIKM '05)*, 648–655., Bremen, Germany. ACM. http://dx.doi.org/10.1145/1099554.1099718

Rotem, D., Stockinger, K. et Wu, K. (2006a). Minimizing I/O costs of multi-dimensional queries with bitmap indices. Dans *18th International Conference on Scientific and Statistical Database Management (SSDBM '06)*, 33–44., Vienne, Autriche. IEEE. http://dx.doi.org/10.1109/SSDBM.2006.33

Rotem, D., Stockinger, K. et Wu, K. (2006b). Optimizing I/O costs of multi-dimensional queries using bitmap indices. Dans *16th International Conference on Database and Expert Systems Applications (DEXA '05)*, volume 3588 de *Lecture Notes in Computer Science*, 220–229., Copenhagen, Denmark. Springer. http://dx.doi.org/10.1007/11546924_22

Schmidt, A., Kimming, D. et Beine, M. (2011). A Proposal of a New Compression Scheme of Medium-Sparse Bitmaps. *International Journal on Advances in Software*, *4*(3), 401–411.

Sharma, V. (2005). Bitmap Index vs. B-tree Index : Which and When? Récupéré de http://www.oracle.com/technetwork/articles/sharma-indexes-093638.html

Shvachko, K., Hairong, K., Radia, S. et Chansler, R. (2010). The Hadoop Distributed File System. Dans *26th Symposium on Mass Storage Systems and Technologies*, MSST 2010, 1–10., Incline Village, NV. IEEE. http://dx.doi.org/10.1109/MSST.2010.5496972

Sidirourgos, L. et Kersten, M. (2013). Column imprints : a secondary index structure. Dans *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 893–904., New York, NY, USA. ACM. http://dx.doi.org/10.1145/2463676.2465306

Sinha, R. et Winslett, M. (2007). Multi-Resolution bitmap indexes for scientific data. *ACM Transactions on Database Systems (TODS)*, *32*(3), Article No.16. http://dx.doi.org/10.1145/1272743.1272746

Stepanov, A., Gangolli, A., Rose, D. et Ernst, R. (2011). SIMD-based decoding of posting lists. Dans *Proceedings of the 20th ACM international conference on Information and knowledge management (CIKM '11)*, 317–326., Glasgow, UK. ACM. http://dx.doi.org/10.1145/2063576.2063627

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

Stockinger, K., Wu, K. et Shoshani, A. (2002). Strategies for processing ad hoc queries on large data sets. Dans *Proceedings of the 5th ACM international workshop on Data Warehousing and OLAP (DOLAP '02)*, 72–79., New York, NY, USA. ACM. http://dx.doi.org/10.1145/583890.583901

Teuhola, J. (1978). A compression method for clustered bit-vectors. *Information Processing Letters*, *7*(6), 308–311. http://dx.doi.org/http://dx.doi.org/10.1016/0020-0190(78)90024-8

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H. et Murthy, R. (2010). Hive - a petabyte scale data warehouse using Hadoop. Dans *26th International Conference on Data Engineering*, ICDE 2010, 996–1005., Long Beach, CA, USA. IEEE. http://dx.doi.org/10.1109/ICDE.2010.5447738

Tschetter, E. (2011). Introducing Druid : Real-Time Analytics at a Billion Rows Per Second. Récupéré de http://druid.io/blog/2011/04/30/introducing-druid.html

Uno, T., Kiyomi, M. et Arimura, H. (2005). LCM Ver.3 : Collaboration of array, bitmap and prefix tree for frequent itemset mining. Dans *Proceedings of the 1st international workshop on open source data mining : frequent pattern mining implementations (OSDM '05)*, 77–86., Chicago, Illinois, USA. ACM. http://dx.doi.org/10.1145/1133905.1133916

Wedekind, V. et Harder, T. (1976). *Datenbanksysteme II*. Eicklingen, Germany : Bibl. Institut, Mannheim.

Wikipedia (2013). Codage unaire. https://fr.wikipedia.org/wiki/Codage_unaire.

Wikipedia (2015a). Bubble (computing). Récupéré de https://en.wikipedia.org/wiki/Bubble_(computing)

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

Wikipedia (2015c). Shared-nothing architecture. Récupéré de https://en.wikipedia.org/wiki/Shared_nothing_architecture

Wikipedia (2016). Harmonic mean. Récupéré de https://en.wikipedia.org/wiki/Harmonic_mean

Wong, H., Liu, H., Olken, F., Rotem, D. et Wong, L. (1985). Bit-transposed files. Dans *Proceedings of the 11th international conference on Very Large Data Bases (VLDB '85)*, volume 11, 448–457., Stockholm, Suede. VLDB Endowment. Récupéré de http://dl.acm.org/citation.cfm?id=1286802

Wu, k., Ahern, S., Bethel, E., Chen, J., Childs, H., Cormier-Michel, E., Geddes, C., Gu, J., Hagen, H., Hamann, B., Koegler, W., Lauret, J., Meredith, J., Messmer, P., Otoo, E., Perevoztchikov, V., Poskanzer, A., Prabhat, Rubel, O., Shoshani, A., Sim, A., Stockinger, K., Weber, G. et Zhang, W.-M. (2009). FastBit : An efficient compressed bitmap index technology. Dans *Journal of Physics : Conference Series*, volume 180 de *IOPscience*, 1–10.

Wu, K., Otoo, E. et Shoshani, A. (2001a). A performance comparison of bitmap indexes. Dans *Proceedings of the 10th International Conference on Information and Knowledge Management (CIKM '01)*, 559–561., New York, NY, USA. ACM. http://dx.doi.org/10.1145/502585.502689

Wu, K., Otoo, E. et Shoshani, A. (2006). Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, *31*(1), 1–38. http://dx.doi.org/10.1145/1132863.1132864

Wu, k., Otoo, E. et Soshani, A. (2001b). Compressed bitmap indices for efficient query processing. *Tech. rep. LBNL-47807, Lawrence Berkeley National Laboratory, Berkeley, CA*, 1–16.

Wu, K., Shoshani, A. et Stockinger, K. (2010). Analyses of Multi-Level and Multi-Component Compressed Bitmap Indexes. *ACM Transactions on Database Systems (TODS)*, *35*(1), Article No. 2. http://dx.doi.org/10.1145/1670243.1670245

Wu, K., Stockinger, K. et Shoshanie, A. (2008). Breaking the curse of cardinality on bitmap indexes. Dans *20th International Conference on Scientific and Statistical Database Management (SSDBM '08)*, volume 5069 de *Lecture Notes in Computer Science*, 348–365., Hong Kong, China. Springer. http://dx.doi.org/10.1007/978-3-540-69497-7_23

Wu, K. et Yu, p. (1998). Range-based bitmap indexing for high cardinality attributes with skew. Dans *The 22nd Annual International Computer Software and Applications Conference (COMPSAC '98)*, 61–66., Vienne, Autriche. IEEE. http://dx.doi.org/10.1109/CMPSAC.1998.716637

Wu, L., Sumbaly, R., Riccomini, C., G.Koo, Kim, H., Kreps, J. et Shah, S. (2012). Avatara : OLAP for web-scale analytics products. Dans *The 38th International Conference on Very Large Data Bases*, volume 5 de *VLDB 2012*, 1874–1877., Istanbul, Turkey. VLDB.

Wu, Y., Chen, Z., Wen, Y., Zheng, W. et Cao, J. (2016). COMBAT : A new bitmap index coding algorithm for Big Data. *Tsinghua Science and Technology*, *21*(2), 136–145.

Yan, H., Ding, S. et Suel, T. (2009). Inverted index compression and query processing with optimized document ordering. Dans *Proceedings of the 18th international conference on World wide web (WWW '09)*, 401–410., Madrid, Espagne. ACM. http://dx.doi.org/10.1145/1526709.1526764

Yang, F., Tschetter, E., Lèautè, X., Ray, N., Merlino, G. et Ganguli, D. (2014). Druid : A real–time analytical data store. Dans *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, 157–168., New York, NY, USA. ACM. http://dx.doi.org/10.1145/2588555.2595631

Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S. et Stoica, I. (2010). Spark : cluster computing with working sets. Dans *2nd USENIX Workshop on Hot Topics in Cloud Computing*, HotCloud'10, 1–7., Boston, MA, USA. ACM. http://dx.doi.org/10.1145/1739041.1739071

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

Ziv, J. et Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, *23*(3), 337–343. http://dx.doi.org/10.1109/TIT.1977.1055714

Zukowski, M., Heman, S., Nes, N. et Boncz, P. (2006). Super-scalar RAM–CPU cache compression. Dans *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, p. 59., Atlanta, Georgia, USA. IEEE. http://dx.doi.org/10.1109/ICDE.2006.150

ANNEXE

Algorithme VI.1: Requête GroupBy opérant des OU logiques entre des bitmaps de très faibles cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": { "type": "or",
      "fields": [
       { "type": "selector", "dimension": "l_receiptdate", "value": "1994-12-03" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1993-03-04" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1993-05-06" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1993-05-23" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1994-10-12" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1993-01-04" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1992-10-20" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1992-07-10" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1993-02-25" },
       { "type": "selector", "dimension": "l_receiptdate", "value": "1993-08-05" },
       { "type": "selector", "dimension": "l_suppkey", "value": "5099" },
       { "type": "selector", "dimension": "l_suppkey", "value": "5055" },
       { "type": "selector", "dimension": "l_suppkey", "value": "4900" },
       { "type": "selector", "dimension": "l_suppkey", "value": "4605" },
       { "type": "selector", "dimension": "l_suppkey", "value": "5682" },
       { "type": "selector", "dimension": "l_suppkey", "value": "4981" },
       { "type": "selector", "dimension": "l_suppkey", "value": "5229" },
       { "type": "selector", "dimension": "l_suppkey", "value": "4997" },
       { "type": "selector", "dimension": "l_commitdate", "value": "1993-10-06" },
       { "type": "selector", "dimension": "l_commitdate", "value": "1992-11-21" },
       { "type": "selector", "dimension": "l_commitdate", "value": "1993-05-06" },
    { "type": "selector", "dimension": "l_commitdate", "value": "1994-07-20" } ]},
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "L_TAX", "fieldName": "L_TAX_doubleSum" }
  ] }
```

Algorithme VI.2: Requête GroupBy opérant des OU logiques entre des bitmaps de faibles cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "or",
      "fields": [
          { "type": "selector", "dimension": "l_quantity", "value": "35" },
          { "type": "selector", "dimension": "l_quantity", "value": "25" },
          { "type": "selector", "dimension": "l_quantity", "value": "32" },
          { "type": "selector", "dimension": "l_quantity", "value": "23" },
          { "type": "selector", "dimension": "l_quantity", "value": "16" },
          { "type": "selector", "dimension": "l_quantity", "value": "9" },
          { "type": "selector", "dimension": "l_quantity", "value": "41" },
          { "type": "selector", "dimension": "l_quantity", "value": "1" },
          { "type": "selector", "dimension": "l_quantity", "value": "36" },
          { "type": "selector", "dimension": "l_quantity", "value": "42" },
          { "type": "selector", "dimension": "l_discount", "value": "0.05" },
          { "type": "selector", "dimension": "l_discount", "value": "0.07" },
          { "type": "selector", "dimension": "l_discount", "value": "0.02" },
          { "type": "selector", "dimension": "l_discount", "value": "0.01" },
          { "type": "selector", "dimension": "l_discount", "value": "0.10" },
          { "type": "selector", "dimension": "l_discount", "value": "0.04" },
          { "type": "selector", "dimension": "l_discount", "value": "0.09" },
          { "type": "selector", "dimension": "l_discount", "value": "0.03" },
          { "type": "selector", "dimension": "l_linenumber", "value": "7" },
          { "type": "selector", "dimension": "l_linenumber", "value": "6" },
          { "type": "selector", "dimension": "l_linenumber", "value": "5" },
          { "type": "selector", "dimension": "l_linenumber", "value": "4" }
       ]
     },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "L_TAX", "fieldName": "L_TAX_doubleSum" }
  ]
}
```

Algorithme VI.3: Requête GroupBy opérant des OU logiques entre des bitmaps de moyennes cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "or",
      "fields": [
      { "type": "selector", "dimension": "l_shipinstruct","value": "COLLECT COD" },
      { "type": "selector", "dimension": "l_shipinstruct","value": "DELIVER IN
          PERSON" },
      { "type": "selector", "dimension": "l_shipinstruct","value": "TAKE BACK
          RETURN" },
      { "type": "selector", "dimension": "l_returnflag","value": "A" },
      { "type": "selector", "dimension": "l_returnflag","value": "R" },
      { "type": "selector", "dimension": "l_quantity","value": "9" },
      { "type": "selector", "dimension": "l_quantity","value": "41" },
      { "type": "selector", "dimension": "l_quantity","value": "1" },
      { "type": "selector", "dimension": "l_quantity","value": "36" },
      { "type": "selector", "dimension": "l_quantity","value": "42" },
      { "type": "selector", "dimension": "l_discount", "value": "0.05" },
      { "type": "selector", "dimension": "l_discount", "value": "0.07" },
      { "type": "selector", "dimension": "l_discount", "value": "0.02" },
      { "type": "selector", "dimension": "l_discount", "value": "0.01" },
      { "type": "selector", "dimension": "l_discount", "value": "0.10" },
      { "type": "selector", "dimension": "l_discount", "value": "0.04" },
      { "type": "selector", "dimension": "l_linenumber", "value": "1" },
      { "type": "selector", "dimension": "l_linenumber", "value": "2" },
      { "type": "selector", "dimension": "l_linenumber", "value": "3" },
      { "type": "selector", "dimension": "l_linenumber", "value": "4" },
      { "type": "selector", "dimension": "l_linenumber", "value": "5" },
      { "type": "selector", "dimension": "l_linenumber", "value": "6" }
      ]
  },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "L_TAX", "fieldName": "L_TAX_doubleSum" }
  ]
}
```

Algorithme VI.4: Requête GroupBy opérant des OU logiques entre des bitmaps de fortes cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "or",
      "fields": [
      { "type": "selector", "dimension": "l_shipinstruct","value": "COLLECT COD" },
      { "type": "selector", "dimension": "l_shipinstruct","value": "DELIVER IN
          PERSON" },
      { "type": "selector", "dimension": "l_shipinstruct","value": "TAKE BACK
          RETURN" },
      { "type": "selector", "dimension": "l_returnflag","value": "A" },
      { "type": "selector", "dimension": "l_returnflag","value": "R" },
      { "type": "selector", "dimension": "l_quantity","value": "9" },
      { "type": "selector", "dimension": "l_quantity","value": "41" },
      { "type": "selector", "dimension": "l_quantity","value": "1" },
      { "type": "selector", "dimension": "l_quantity","value": "36" },
      { "type": "selector", "dimension": "l_quantity","value": "42" },
      { "type": "selector", "dimension": "l_returnflag", "value": "N" },
      { "type": "selector", "dimension": "l_returnflag", "value": "F" },
      { "type": "selector", "dimension": "l_linestatus", "value": "O" },
      { "type": "selector", "dimension": "l_discount", "value": "0.01" },
      { "type": "selector", "dimension": "l_discount", "value": "0.10" },
      { "type": "selector", "dimension": "l_discount", "value": "0.04" },
      { "type": "selector", "dimension": "l_linenumber", "value": "1" },
      { "type": "selector", "dimension": "l_linenumber", "value": "2" },
      { "type": "selector", "dimension": "l_linenumber", "value": "3" },
      { "type": "selector", "dimension": "l_linenumber", "value": "4" },
      { "type": "selector", "dimension": "l_linenumber", "value": "5" },
      { "type": "selector", "dimension": "l_linenumber", "value": "6" }
      ]
  },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "L_TAX", "fieldName": "L_TAX_doubleSum" }
  ]
}
```

Algorithme VI.5: Requête GroupBy opérant des ET logiques entre des bitmaps de très faibles cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "and",
    "fields": [
       { "type": "selector", "dimension": "l_shipdate", "value": "1997-06-01" },
       { "type": "selector", "dimension": "l_commitdate", "value": "1997-05-12" },
       { "type": "selector", "dimension": "l_receiptdate","value": "1997-06-02" },
       { "type": "selector", "dimension": "l_suppkey", "value": "4623" },
       { "type": "selector", "dimension": "l_comment", "value": "c packages"},
       { "type": "selector", "dimension": "l_partkey", "value": "22120" },
       { "type": "selector", "dimension": "l_orderkey", "value": "5050562" }
        ]
      },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type":"doubleSum", "name":"l_extendedprice", "fieldName":"
        L_EXTENDEDPRICE_doubleSum" }
  ]
}
```

Algorithme VI.6: Requête GroupBy opérant des ET logiques entre des bitmaps de faibles cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "and",
    "fields": [
          { "type": "selector", "dimension": "l_shipdate", "value": "1997-06-01" },
          { "type": "selector", "dimension": "l_commitdate", "value": "1997-05-12"
              },
          { "type": "selector", "dimension": "l_shipmode", "value": "AIR" },
```

```
        { "type": "selector", "dimension": "l_quantity", "value": "34" },
        { "type": "selector", "dimension": "l_tax", "value": "0.08"},
        { "type": "selector", "dimension": "l_linenumber", "value": "6" },
        { "type": "selector", "dimension": "l_discount", "value": "0.05" }
      ]
    },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "l_extendedprice", "fieldName": "
      L_EXTENDEDPRICE_doubleSum" }
  ]
}
```

Algorithme VI.7: Requête GroupBy opérant des ET logiques entre des bitmaps de moyennes cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "and",
    "fields": [
        { "type": "selector", "dimension": "l_shipinstruct","value": "COLLECT COD
          " },
        { "type": "selector", "dimension": "l_shipmode", "value": "AIR" },
        { "type": "selector", "dimension": "l_returnflag", "value": "R" },
        { "type": "selector", "dimension": "l_quantity", "value": "35" },
        { "type": "selector", "dimension": "l_tax", "value": "0.06"},
        { "type": "selector", "dimension": "l_linenumber", "value": "1" },
        { "type": "selector", "dimension": "l_discount", "value": "0.05" }
      ]
    },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "l_extendedprice", "fieldName": "
      L_EXTENDEDPRICE_doubleSum" }
  ]
}
```

Algorithme VI.8: Requête GroupBy opérant des ET logiques entre des bitmaps de fortes cardinalités

```
{
  "queryType"  : "groupBy",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "dimensions": ["l_shipmode"],
  "filter": {
    "type": "and",
    "fields": [
          { "type": "selector", "dimension": "l_shipinstruct","value": "COLLECT COD
              " },
          { "type": "selector", "dimension": "l_shipmode", "value": "AIR" },
          { "type": "selector", "dimension": "l_returnflag", "value": "N" },
          { "type": "selector", "dimension": "l_linestatus", "value": "O" },
          { "type": "selector", "dimension": "l_tax", "value": "0.06"},
          { "type": "selector", "dimension": "l_linenumber", "value": "1" },
          { "type": "selector", "dimension": "l_discount", "value": "0.05" }
        ]
     },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum",   "name": "l_extendedprice", "fieldName": "
       L_EXTENDEDPRICE_doubleSum" }
  ]
}
```

Algorithme VI.9: Requête Timeseries opérant des ET logiques entre des bitmaps de moyennes cardinalités

```
{
  "queryType"  : "timeseries",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "filter": {
    "type": "and",
    "fields": [
    { "type": "selector", "dimension": "l_shipinstruct","value": "COLLECT COD" },
    { "type": "selector", "dimension": "l_shipmode", "value": "AIR" },
    { "type": "selector", "dimension": "l_returnflag", "value": "R" },
    { "type": "selector", "dimension": "l_quantity", "value": "35" },
```

```
    { "type": "selector", "dimension": "l_tax", "value": "0.06"},
    { "type": "selector", "dimension": "l_linenumber", "value": "1" },
    { "type": "selector", "dimension": "l_discount", "value": "0.05" }
    ]
  },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "l_extendedprice", "fieldName": "
        L_EXTENDEDPRICE_doubleSum" }
  ]
}
```

Algorithme VI.10: Requête TopN opérant des ET logiques entre des bitmaps de moyennes cardinalités

```
{
  "queryType"  : "TopN",
  "dataSource" : "TPCH_benchmark_concise",
  "granularity": "all",
  "threshold": 100,
  "dimension":"l_shipmode",
  "metric": "l_extendedprice",
  "filter": {
    "type": "and",
    "fields": [
    { "type": "selector", "dimension": "l_shipinstruct","value": "COLLECT COD" },
    { "type": "selector", "dimension": "l_shipmode", "value": "AIR" },
    { "type": "selector", "dimension": "l_returnflag", "value": "R" },
    { "type": "selector", "dimension": "l_quantity", "value": "35" },
    { "type": "selector", "dimension": "l_tax", "value": "0.06"},
    { "type": "selector", "dimension": "l_linenumber", "value": "1" },
    { "type": "selector", "dimension": "l_discount", "value": "0.05" }
    ]
  },
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ],
  "aggregations": [
    { "type": "doubleSum", "name": "l_extendedprice", "fieldName": "
        L_EXTENDEDPRICE_doubleSum" }
  ]
}
```

Algorithme VI.11: Requête Select opérant des OU logiques entre des bitmaps de moyennes cardinalités

```
{
  "queryType"  : "select",
  "dataSource" : "TPCH_benchmark_roaring",
  "granularity": "day",
  "dimensions": ["l_shipmode", "l_shipdate"],
  "metrics":["L_TAX", "L_QUANTITY"],
  "context":{"useCache":false,"populateCache":false},
  "filter": {
    "type": "or",
    "fields": [
    { "type": "selector", "dimension": "l_receiptdate","value": "1994-12-03" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1993-03-04" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1993-05-06" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1993-05-23" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1994-10-12" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1993-01-04" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1992-10-20" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1992-07-10" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1993-02-25" },
    { "type": "selector", "dimension": "l_receiptdate","value": "1993-08-05" },
    { "type": "selector", "dimension": "l_suppkey", "value": "5099" },
    { "type": "selector", "dimension": "l_suppkey", "value": "5055" },
    { "type": "selector", "dimension": "l_suppkey", "value": "4900" },
    { "type": "selector", "dimension": "l_suppkey", "value": "4605" },
    { "type": "selector", "dimension": "l_suppkey", "value": "5682" },
    { "type": "selector", "dimension": "l_suppkey", "value": "4981" },
    { "type": "selector", "dimension": "l_suppkey", "value": "5229" },
    { "type": "selector", "dimension": "l_suppkey", "value": "4997" },
    { "type": "selector", "dimension": "l_shipdate", "value": "1997-06-01" },
    { "type": "selector", "dimension": "l_shipdate", "value": "1994-09-16" },
    { "type": "selector", "dimension": "l_shipdate", "value": "1998-03-23" },
    { "type": "selector", "dimension": "l_shipdate", "value": "1995-05-30" }
    ]
  },
  "pagingSpec":{"pagingIdentifiers": {}, "threshold":5},
  "intervals": [ "1980-12-31T23:59:59.999/2005-01-30T00:00:00.000" ]
}
```