# Sélection Statique et Incrémentale des Index de Jointure Binaires Multiples

Rima Bouchakri *, Ladjel Bellatreche**

*Ecole nationale Supérieure d'Informatique, Alger, Algérie
r_bouchakri@esi.dz
**LISI/ENSMA Université de Poitiers, France
bellatreche@ensma.fr

**Résumé.** Les index de jointure binaires ont montré leur intérêt dans la réduction des coûts d'exécution des requêtes décisionnelles définies sur un schéma relationnel en étoile. Leur sélection reste cependant difficile vu le vaste et complexe espace de recherche à explorer. Peu d'algorithmes de sélection des index de jointure existent, contrairement à la sélection des index définis sur une seule table qui a connu un intérêt particulier auprès de la communauté des bases de données traditionnelles. La principale particularité de ces algorithmes est qu'ils sont *statiques et supposent la connaissance préalable des requêtes*. Dans cet article, nous présentons une démarche de sélection des index de jointures binaires définis sur *plusieurs attributs appartenant à des tables de dimension* en utilisant des algorithmes génétiques. Ces derniers sont utilisés dans le cadre *statique* et *incrémental* qui prévoit l'adaptation des index sélectionnés à l'arrivée de nouvelles requêtes. Nous concluons nos travaux par une étude expérimentale démontrant l'intérêt de la sélection des index de jointure binaires multiple, de l'élagage de l'espace de recherche et de l'efficacité des algorithmes génétiques dans les cas statique ou incrémentale.

## 1 Introduction

Dans les applications décisionnelles, les données importantes sont intégrées, historisées et stockées dans des entrepôts de données ($\mathcal{ED}$) souvent schématisés en modèles relationnels en étoile ou ses variantes (Kimball et Strehlo, 1995) pour des fins d'analyse en ligne. Cette dernière est effectuée à l'aide des requêtes décisionnelles complexes, dites *requêtes de jointures en étoile*. Afin de rendre efficace l'analyse décisionnelle et satisfaire les exigences des décideurs en termes de temps de réponse, il est primordiale d'optimiser ces requêtes. Cette optimisation est assurée par l'utilisation des structures d'optimisation sélectionnées lors de la phase de *conception physique*. Les *index* sont un exemple de ces structures qui ont montré leur intérêt dans les bases de données traditionnelles et les entrepôts de données. La particularité des requêtes décisionnelles et le schéma en étoile rendent les index classiques, comme les *arbres B* (Comer, 1979), les *index de jointure* (Valduriez, 1987), etc. inefficaces. Pour répondre aux limites des index traditionnels, de nouveaux index ont été introduits. On peut ainsi citer,

les *index binaires* (Chan et Ioannidis, 1998) qui optimisent les opérations de sélection définies sur des attributs appartenant à des tables de dimension, les *index de jointures en étoile* permettant de stocker le résultat d'exécution d'une jointure en étoile entre plusieurs tables (Systems, 1997) et les *Index de Jointures Binaires* ($\mathcal{IJB}$) (O'Neil et Graefe, 1995). Ils sont plus adéquats au contexte d'entreposage des données, car ils permettent à la fois d'optimiser les jointures en étoile et les opérations de sélections définies sur les tables de dimensions. Deux types d'$\mathcal{IJB}$ existent : *les index simples* (index mono-attribut) définis sur un seul attribut d'une table de dimension et *les index multiples* (index multi-attributs) définis sur plusieurs attributs issus d'une ou plusieurs tables de dimension.

Le problème de sélection des $\mathcal{IJB}$ dans sa *formalisation classique* consiste à sélectionner une configuration d'index optimisant une charge de requêtes, *connue à l'avance*, sans violer la contrainte d'espace de stockage. Ce problème est difficile à résoudre lorsqu'un nombre important d'attributs de tables de dimension est concerné par le processus d'indexation. Un nombre raisonnable de travaux traitant ce problème existe (Aouiche et al., 2005; Bellatreche et Boukhalfa, 2010; Bouchakri et al., 2010; Bellatreche et al., 2008; Stöhr et al., 2000). Suite à l'analyse de ces derniers, nous avons identifié les points traités suivants : (1) *la sélection des attributs indexables :* la plupart de ces travaux proposent d'abord d'analyser *syntaxiquement* des requêtes afin d'extraire les attributs de sélection candidats à l'indexation. La complexité du problème de sélection des $\mathcal{IJB}$ est proportionnelle au nombre d'attributs candidats (Bellatreche et Boukhalfa, 2010). (2) *L'élagage de l'espace de recherche* du problème de sélection des $\mathcal{IJB}$ qui vise à écarter les attributs ou index non pertinents afin de réduire la complexité du problème de sélection des $\mathcal{IJB}$. Cet élagage peut être réalisé *manuellement* en exploitant l'expérience de l'administrateur. Cette solution n'est pas appropriée lorsqu'un nombre important d'attributs est concerné. Pour remédier à cet élagage manuel, plusieurs travaux adoptent *l'élagage automatique* réalisé selon deux méthodes principales que nous nommons : (a) *un élagage par algorithmes* et (b) un *élagage dirigé par une technique d'optimisation*. Les travaux dans (Aouiche et al., 2005; Bellatreche et Boukhalfa, 2010) sont des exemples d'élagage basé sur des algorithmes. (Aouiche et al., 2005) proposent l'utilisation des techniques de fouille de données, où l'algorithme *Close* est utilisé (Pasquier et al., 1999). Dans (Bellatreche et Boukhalfa, 2010), les auteurs proposent l'utilisation de plusieurs stratégies automatiques d'élimination d'attributs comme : *l'attribut de forte cardinalité, l'attribut le moins utilisé*, etc. Cet élagage ne prend pas en compte la *définition globale de la requête*. Dans les travaux d'élagage automatique basé sur une technique d'optimisation, certaines études exploitent la similarité entre $\mathcal{IJB}$ et la fragmentation horizontale pour établir cet élagage (Boukhalfa et al., 2010; L.Bellatreche et al., 2007; Stöhr et al., 2000). Ils proposent d'abord de partitionner le schéma d'un $\mathcal{ED}$ en considérant toutes les requêtes, ensuite de l'indexer en ne prenant que les requêtes non bénéficiaires de la fragmentation. La façon d'identifier ces dernières est la limite principale de ces travaux. (3) *Les algorithmes de sélection des index finaux* : après la phase d'élagage, plusieurs travaux adoptent les algorithmes gloutons (Aouiche et al., 2005; L.Bellatreche et al., 2007; Bellatreche et Boukhalfa, 2010) guidés par un modèle de coût pour la sélection d'une configuration quasi-optimale d'$\mathcal{IJB}$. Récemment, un algorithme génétique a été proposé pour la sélection d'$\mathcal{IJB}$ simples (Bouchakri et al., 2010). Dans le contexte des entrepôts de données, les requêtes décisionnelles utilisent plusieurs prédicats de sélection dont les attributs sont candidats à l'indexation, d'où le besoin de développer des algorithmes pour définir des index multiples.

L'analyse de ces travaux nous conduit à faire les constats suivants : (1) l'élagage de l'espace de recherche effectué à partir de la définition des requêtes n'a pas reçu un grand intérêt par les travaux de sélection d'index, particulièrement pour les index multiples. En effet, si un index multiple est défini sur tous les attributs d'une requête, l'exploitation de cet index lors de l'exécution de la requête ne nécessite aucune jointure, ce qui réduit considérablement le coût d'exécution, (2) peu de classes d'algorithmes ont été explorées (*exacts, approximatifs, etc.*) pour sélectionner des $\mathcal{IJB}$, (3) peu de travaux s'intéressent à la sélection des $\mathcal{IJB}$ multiples, pourtant ces derniers présentent un avantage majeur par rapport au $\mathcal{IJB}$ simples. Certes les $\mathcal{IJB}$ simples présentent un espace de recherche moins complexe que les $\mathcal{IJB}$ multiples (Bellatreche et Boukhalfa, 2010), mais ces derniers permettent de mieux respecter la contrainte d'espace de stockage. En effet, si deux $\mathcal{IJB}$ simples sont définis sur les attributs $A_1$ et $A_2$ respectivement, ils sont plus volumineux qu'un seul $\mathcal{IJB}$ multiple défini sur les deux attributs, puisque chaque index comporte une colonne supplémentaire constituant l'identifiant (*Row Identifier* nécessitant 16 octets dans le SGBD comme Oracle). (4) Nous avons remarqué que ces travaux de sélection d'index se situent dans le cadre de sélection dite *statique*, basée sur la connaissance préalable des requêtes. Si les requêtes changent, leur optimisation devient obsolète. En conséquence, une optimisation continuelle de la charge de requêtes est primordiale afin d'adapter les index existants pour satisfaire les nouvelles requêtes. (Azefack et al., 2007) propose une sélection dynamique des $\mathcal{IJB}$ en étendant leurs travaux concernant la sélection statique basée sur l'utilisation des techniques de fouille de données (Aouiche et al., 2005). Cette sélection hérite des mêmes problèmes qui sont liés à l'utilisation de ces techniques pour l'identification des index pertinents (Bellatreche et Boukhalfa, 2010).

L'objectif de cet article est double, d'une part, il propose une *sélection statique* d'$\mathcal{IJB}$ multiples par un algorithme génétique guidé par modèle de coût avec un élagage basé sur des requêtes et d'autre part il présente une adaptation de cet algorithme pour la *sélection incrémentale*.

Cet article est structuré comme suit : la section 2 est dédiée au problème de sélection des $\mathcal{IJB}$ multiples et les travaux antérieurs de sélection. La section 3 présente la démarche de sélection des $\mathcal{IJB}$ simples et multiples par des algorithmes génétiques avec élagage basé sur des requêtes. Dans la section 4 nous abordons la sélection incrémentale des $\mathcal{IJB}$ et nous présentons l'architecture et les algorithmes mis au point. Enfin, la section 5 est consacrée à l'étude expérimentale réalisée en deux phases : des tests sur les sélections statique et incrémentale des $\mathcal{IJB}$ multiples. La section 6 conclut le papier.

## 2 Problème de sélection et travaux antérieurs

Un $\mathcal{IJB}$ permet de pré-calculer les jointures en étoile entre la table des faits et une ou plusieurs tables de dimension en utilisant un ou plusieurs attributs de ces dernières (O'Neil et Graefe, 1995; O'Neil et Quass, 1997). Il matérialise ces opérations sous formes de vecteurs binaires moins gourmands en termes d'espace de stockage, où chaque bitmap fait référence à une valeur d'un attribut indexé. Une autre caractéristique de ces index est qu'ils peuvent être compressés (Wu et al., 2006).

Le problème de sélection des $\mathcal{IJB}$ (PSI) est formalisé sous *forme statique* comme suit (Aouiche et al., 2005; Boukhalfa et al., 2010) :
Étant donné : (1) un $\mathcal{ED}$ modélisé par un schéma en étoile ayant $d$ tables de dimension $\mathcal{D} =$

$\{D_1, D_2, ..., D_d\}$ et une table des faits $\mathcal{F}$, (2) une charge de requêtes $\mathcal{Q} = \{Q_1, Q_2, ..., Q_m\}$ à partir de laquelle un ensemble d'attributs indexables $AS = \{A_1, \cdots, A_n\}$ est sélectionné et (3) un espace de stockage d'index $\mathcal{S}$. Le problème de sélection des $\mathcal{IJB}$ consiste à trouver une configuration d'index $Config_{Ci}$ qui réduit le coût d'exécution de $\mathcal{Q}$ sans violer la contrainte de stockage ($\mathcal{S}$).

Si l'administrateur souhaite sélectionner un seul index simple à partir de l'ensemble des $n$ attributs indexables, il doit évaluer $n$ possibilités d'index (chaque attribut peut donner lieu à la création d'un index simple). Le nombre de toutes les configurations d'index simples possibles est donné par l'équation suivante (Boukhalfa et al., 2010) :

$$NbSimple = 2^n - 1 \tag{1}$$

Pour $n = 20$, $NbSimple = 2^{20} - 1 = 1048575$.
D'une manière similaire, si l'administrateur souhaite sélectionner un seul $\mathcal{IJB}$ multiple parmi les $n$ attributs indexables, il doit évaluer $(2^{n-1} - 1)$ possibilités. Le nombre de toutes configurations d'index multiple possibles est donné par l'équation suivante (Boukhalfa et al., 2010) :

$$NbMultiple = 2^{2^{n-1}-1} - 1 \tag{2}$$

Vue la complexité de la sélection des $\mathcal{IJB}$ multiple, une énumération exhaustive pour générer la configuration optimale est quasi impossible pour un nombre important d'attributs indexables. Dans (Bellatreche et Boukhalfa, 2010), une méthode de sélection $\mathcal{IJB}$ multiple a été proposée. Elle consiste à définir un $\mathcal{IJB}$ pour chaque requête impliquant des attributs indexables. Cette définition réduit considérablement son coût d'exécution. Cette manière d'indexer génère des index volumineux surtout si les attributs utilisés ont une forte cardinalité. Pour satisfaire la contrainte de stockage, les auteurs proposent d'utiliser des heuristiques d'élimination basées sur des critères comme : des attributs de forte cardinalité, des attributs appartenant aux tables moins volumineuses, des attributs les moins utilisés, etc. Ces heuristiques sont inefficaces, car elles ne prennent pas en compte tous les paramètres liés au processus d'optimisation. En conséquence, le développement des algorithmes plus avancés comme les algorithmes génétiques (AG) est recommandé. Nous avons adopté les AG pour trois raisons principales : (1) leurs contributions à l'optimisation de l'opération de jointure dans le contexte des bases de données traditionnelles (Ioannidis et Kang, 1990) et la conception physique des entrepôts de données (la sélection des vues matérialisées (Zhang et Yang, 1999) et l'automatisation de la conception physique des bases de données parallèles (Rao et al., 2002), etc.). (2) leur utilisation par les optimiseurs de SGBD comme PostgreSQL et (3) la disponibilité des framework implémentant ses algorithmes (e.g., Java Genetic Algorithms Package : http ://jgap.sourceforge.net).

# 3 Démarche de sélection d'$\mathcal{IJB}$ par AG

Dans cette section, nous présentons une démarche de sélection des $\mathcal{IJB}$ simples et multiples en utilisant les algorithmes génétiques (AG). Un AG est un algorithme itératif de recherche d'optimum, il manipule une population de taille constante. Cette dernière est formée de candidats appelés individus (chromosomes). La taille constante de la population entraîne un phénomène de compétition entre les individus. Chaque individu représente le codage d'une solution potentielle au problème à résoudre. Il est constitué d'un ensemble d'éléments appelés

gènes, pouvant prendre plusieurs valeurs appartenant à un alphabet non forcément numérique (Bäck, 1995). A chaque itération, appelée génération, est créée une nouvelle population avec le même nombre d'individus. Cette génération est constituée d'individus mieux "adaptés" à leur environnement tel qu'il est représenté par la fonction sélective. Au fur et à mesure des générations, les individus vont tendre vers l'optimum de la fonction sélective. La création d'une nouvelle population, à partir de la précédente, se fait par application des opérateurs génétiques que sont : la sélection, le croisement et la mutation. Ces opérateurs sont stochastiques. Notons que la principale difficulté d'employer un AG est : (1) la définition de la structure du chromosome qui est déterminante pour leur efficacité et (2) la détermination de la fonction objectif. Nous détaillons trois codages de chromosomes utilisés par nos algorithmes.

## 3.1 Codage de chromosome pour les index simples

Pour les index simples définis sur un seul attribut ($SI$), définis sur un ensemble $n$ attributs, nous proposons de coder le chromosome sous forme d'un tableau binaire de taille $n$. Une cellule vaut 1 si un index simple est défini sur l'attribut correspondant. Sinon, elle vaut 0. Le tableau 1 illustre un exemple de chromosome dit simple défini sur 5 attributs, donnant lieu à trois index simples définis sur les trois attributs : $City$, $Country$ et $Day$

TAB. 1 – *Exemple d'un chromosome pour la sélection d'$\mathcal{IJB}$ simples ($SI$)*

| City | Month | Year | Country | Day |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 |

## 3.2 Codage de chromosome pour les index multiple

Pour la sélection des index multiples ($MI$), nous avons d'abord pensé à généraliser le codage utilisé dans la sélection simple. Le nombre d'IJB multiples définis sur un ensemble de $n$ attributs indexables est $2^{n-1}-1$, ainsi la taille du chromosome est $2^{n-1}-1$ cases. Une case est à 1 si l'index multiple correspondant est sélectionné par le processus de sélection d'index, 0 sinon. Le problème de ce codage est la taille du chromosome. En effet, pour 20 attributs de sélection la taille du chromosome est $2^{20-1}-1 = 524287$ cases. Puisque chaque étape de l'algorithme génétique exploite une population de chromosomes (50, 100 chromosomes ou plus), ce codage rend le problème de sélection d'indexes par AG plus complexe. En conséquence, sa simplification est recommandée. Pour ce faire, nous proposons d'utiliser les requêtes pour limiter la taille de ce tableau. Ce codage est baptisé ($MIQ$). Dans l'ensemble des indexes candidats, nous retenons uniquement les indexes dont les attributs correspondent exactement aux attributs d'une requête donnée. En conséquence, chaque requête est associé à un $\mathcal{IJB}$. Ainsi, le nombre d'indexes candidats à la sélection (taille du chromosome) est réduit au maximum au nombre de requêtes.

**Exemple 1** *Soit un $\mathcal{ED}$ avec une table de faits $SALES$ (20 millions de tuples) et trois dimensions $CUSTOMER$, $TIME$ et $PRODUCT$. Soit une charge de trois requêtes permettant de définir les attributs avec les cardinalités suivantes : City(C :150), Country(T :30), Year(Y :20), PName(P :400), Month(M :12), Day(D :31).*

**TAB. 2** – *Exemple d'un chromosome basé requêtes pour la sélection d'$\mathcal{IJB}$ multiples ($MIQ$)*

| CYP | CMD | PM |
|---|---|---|
| 0 | 1 | 1 |

```
Q1  SELECT AVG(PriUnit)
FROM CUSTOMER C, TIME T, PRODUCT P, SALES S
WHERE C.City='Alger' AND T.Year='2008' AND P.PName='PC'
AND C.CID=S.CID AND T.TID=S.TID AND P.PID=S.PID

Q2 SELECT Count(*)
FROM CUSTOMER C, TIME T, SALES S
WHERE C.City='Oran' AND T.Year='2008' AND T.Day='20'
AND P.PName='Scanner'
AND C.CID=S.CID AND T.TID=S.TID

Q3 SELECT Max(Sold)
FROM PRODUCT P, TIME T, SALES S
WHERE P.PName='PC' AND T.Month='4'
AND C.CID=S.CID AND T.TID=S.TID AND P.PID=S.PID
```

*Les attributs candidats à l'indexation sont : $AS = \{C, M, P, Y, D\}$. La taille du chromosome pour représenter cet ensemble d'attributs est 15 ($2^{5-1} - 1$). Si nous considérons les requêtes pour instancier le chromosome initial, nous aurons un tableau de trois cellules représentant les trois $\mathcal{IJB}$ (tableau 2) : CYP, CYD et PM. En conséquence, au lieu de manipuler un chromosome de 15 cases ($2^{15} - 1 = 32767$ configurations d'IJB), nous ne manipulons que 3 $IJB$. Ces dernièrs peuvent générer $2^3 - 1 = 7$ configurations d'index. Un exemple d'une configuration d'index peut être CYP, CYD, CYD, PM ou encore CYP, CYD, PM*

L'inconvénient du codage $MIQ$ est le risque de générer des index volumineux qui pourraient violer l'espace de stockage. Pour illustrer ce problème, supposons que les trois index CYP, CYD et PM nécessitent les coûts de stockage suivants : 1,6Go, 0,9Go et 1,2Go respectivement (sans compression). Si la contrainte d'espace de stockage aux index est inférieure à 0,9Go, aucun index n'est sélectionné, en conséquence, aucune requête n'est optimisée. Il est à noter que l'espace de stockage pour un index $IJB_j$ est calculé comme suit : $Storage(IJB_j) = (\frac{\sum_{k=1}^{n_j} |A_k|}{8} + 16) \times |F|$, où $|A_k|$, $n_j$ et $|F|$ représentent respectivement la cardinalité du k-ème attribut de l'index $IJB_j$, le nombre d'attributs de l'index et la taille de la table de fait $F$ (Aouiche et al., 2005).

Afin d'améliorer le codage précédent, nous proposons un autre, appelé $MIQ^*$. Il consiste à ajouter à chaque requête dont les attributs sont indexés, les sous index qui peuvent être générés à partir de ses attributs. Pour une requête donnée, un *sous index est défini comme une partition d'un index multiple*. L'intérêt de cette approche est de palier au problème des $\mathcal{IJB}$ très volumineux tout en respectant l'élagage par requêtes.

**Exemple 2** *Considérons l'ED et les requêtes définies dans l'exemple 1. Rappelons que les attributs indexables sont City(C :150), Country(T :30), Year(Y :20), PName(P :400), Month(M :12), Day(D :31). Les sous index extraits à partir de l'index multiple $CYP$ défini pour la requête*

**TAB. 3** – *Exemple d'un chromosome basé requêtes amélioré pour la sélection d'$\mathcal{IJB}$ multiples ($MIQ*$)*

| **CYP** | CY | CP | YP | **CYD** | CD | YD | **PM** |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

*$Q_1$ sont : $C, Y, P, CY, CP$ et $YP$. Une fois tous les sous index définis pour chaque requête, le chromosome est structuré comme le montre le tableau 3. Il est à noter que les doublants sont supprimés. Si la contrainte d'espace de stockage est inférieure à 0.9Go, les deux requêtes $Q_1$ et $Q_2$ peuvent être optimisées par l'index $CY$ commun au deux, dont la taille est 710 Mo.*

De ce fait, l'optimisation de la charge de requêtes est améliorée avec $MIQ^*$. Si un $\mathcal{IJB}$ est défini sur les attributs d'une requête donnée est volumineux, il pourrait être remplacé par un sous index moins volumineux. De plus, à partir du chromosome du tableau 3, l'espace de recherche que le problème de sélection doit parcourir représente $2^8 - 1 = 255$ configurations. Pour les 5 attributs indexables de l'exemple 1, cette nouvelle représentation de l'espace de recherche est un bon compromis entre l'espace de recherche total ($2^{2^{5-1}-1} - 1 = 2^{15} - 1 = 32767$ configurations) et l'espace de recherche généré par $MIQ$ ($2^3 - 1 = 7$ configurations) qui peut s'avéré non efficace pour l'optimisation des requêtes.

## 3.3 Implémentation de notre AG

Afin de guider la sélection d'index par l'AG, nous avons utilisé un modèle de coût qui permet de définir la fonction objectif. Ce dernier a été décrit dans (Aouiche et al., 2005) pour le cadre de la sélection des $\mathcal{IJB}$ basée sur les techniques de fouille de données. Le modèle de coût est présenté comme suit :
Soient $Config_{ci}$ et $N_{ci}$ les index sélectionnés et leur cardinalité respectivement. Afin d'évaluer la qualité de cette configuration d'index, deux coût sont utilisés : le coût de stockage des index de $Config_{ci}$ et le coût de la charge de requêtes en présence de $Config_{ci}$. Nous rappelons que le stockage de l'index $IJB_j$ de $Config_{ci}$ défini sur $n_j$ attributs est donné par :

$$Storage(IJB_j) = (\frac{\sum_{k=1}^{n_j} |A_k|}{8} + 16) \times |F| \quad (3)$$

Le coût d'exécution d'une requête $Q_i$ $(1 \leq i \leq m)$ en présence de $IJB_j$ est :

$$Cost(Q_i, IJB_j) = log_t(\sum_{k=1}^{n_j} |A_k|) - 1 + \frac{\sum_{k=1}^{n_j} |A_k|}{t-1} + d\frac{||F||}{8PS} + ||F||(1 - e^{-\frac{Nr}{||F||}}) \quad (4)$$

où $||F||$, $Nr$, $PS$, $d$ et $t$ sont resp. le nombre de pages occupées pas la table $F$, le nombres de tuples accédés par $IJB_j$ , la taille d'une page, le nombre de vecteurs bitmaps utilisés pour évaluer $Q_i$ et le rand du B-arbre défini sur l'index. Le coût total d'exécution des $m$ requêtes en présence de $Config_{ci}$ est : $Cost(Q, Config_{ci}) = \sum_{i=1}^{m} \sum_{j=1}^{N_{ci}} Cost(Q_i, IJB_j)$. Afin de pénaliser un chromosome qui génère des index violant la contrainte d'espace, une fonction

de pénalité est introduite faisant partie de la fonction objectif du AG : $Pen(ConfigI_{ci}) = \frac{storage(Config_{ci})}{S}$ où $storage(Config_{ci}) = \sum_{j=1}^{N_{ci}} storage(IJB_j)$. Enfin, la fonction objectif est définie comme suit :
$F(Config_{ci}) = \{^{Cost(Q,Config_{ci}) \times Pen(Config_{ci}), si\ Pen(Config_{ci}) > 1}_{Cost(Q,Config_{ci}), Pen(Config_{ci}) \leq 1}$

Une fois le codage du chromosome et la fonction objectif définis, la sélection d'index par AG est réalisée en trois étapes : (1) Codage de la configuration d'index en chromosome suivant l'espace de recherche d'index ($SI$, $MIQ$ ou $MIQ^*$), (2) définition de la fonction objectif et (3) la sélection d'index par algorithme génétique : pour ce faire, nous avons utilisé une API JAVA nommé *JGAP* (Java Genetic Algorithms Package) qui permet d'implémenter cet algorithme. JGAP reçoit en entrée le chromosome et la fonction objectif et sélectionne la configuration optimale (quasi optimale) d'index en se basant sur le principe d'AG suivant : l'AG génère une population initiale qui représente plusieurs chromosomes. A partir de cette population, l'AG effectue des opérations de croisement, mutation et sélection afin de générer les nouvelles populations. Chaque configuration d'index (chromosome) va être évaluée, par la fonction objectif, afin d'estimer le bénéfice apporté par celle-ci pour l'optimisation de la charge de requêtes. La configuration d'index qui réduit le plus le coût d'exécution de cette charge va être sélectionnée en fin de processus.

L'algorithme suivant décrit les détails de cette sélection pour le cas simple et multiple.

**Algorithme de sélection d'**$\mathcal{IJB}$
**Entrées :**
$\mathcal{Q}$ : charge de m requêtes
$\mathcal{S}$ : espace de stockage des $\mathcal{IJB}$ s
$AS$ : ensemble d'attributs de sélection (1, n)
$\mathcal{ED}$ : données relatives au modèle de coût (taille des tables, page système, etc.)
**Sortie :** Configuration finale d'index $C_f$.
**Notations :**
$ChromosomeIJB$ : chromosome de configuration d'index candidats
$Coder\_Chromosome$ : Coder le chromosome selon la démarche suivie ($SI$, $MIT$, $MIQ$, $MIQ^*$)
$FitnessIJB$ : fonction objectif pour l'AG
$JGAP$ : API JAVA qui permet d'implémenter l'algorithme génétique
**Début**
$ChromosomeIJB := Coder\_Chromosome(Q, A)$ ;
$FitnessIJB = Genetic\_FitnessFonction(A, S, ED)$ ;
$C_f = JGAP(ChromosomeIJB), FitnessIJB, Q)$ ;
**Fin**

# 4 Sélection incrémentale d'IJB multiples

Les travaux qui traitent du problème de la sélection d'index se basent sur une sélection statique permettant la réalisation d'une sélection d'une structure d'optimisation lors de la phase de conception physique d'un $\mathcal{ED}$. Par conséquent, elle ne permet pas de faire face aux changements pouvant survenir sur l'entrepôt, principalement l'exécution de nouvelles requêtes qui n'existent pas dans la charge en cours. Ainsi, une fois la sélection statique réalisée et un ensemble d'index sélectionnés, *une sélection incrémentale d'index* doit être réalisée continuellement sur l'ED à chaque exécution d'une nouvelle requête. La sélection incrémentale vise à mettre à jour la configuration courante d'index (ajout, suppression ou remplacement d'index), dans le but de prendre en compte l'optimisation de la nouvelle requête exécutée.

Considérons un ED, une charge de requêtes et un ensemble d'$IJB$ implémenté sur l'entrepôt appelé $Config_{IJB}$. Supposons un ensemble de nouvelles requêtes exécutées successivement sur l'ED. Aucune nouvelle requête n'existe dans la charge de requêtes actuelle, car si c'est le cas, aucune sélection incrémentale n'est nécessaire. L'arrivée de chaque requêtes $Q_i$ déclenche alors le processus de sélection incrémentale, dont l'architecture est illustré sur la figure 1

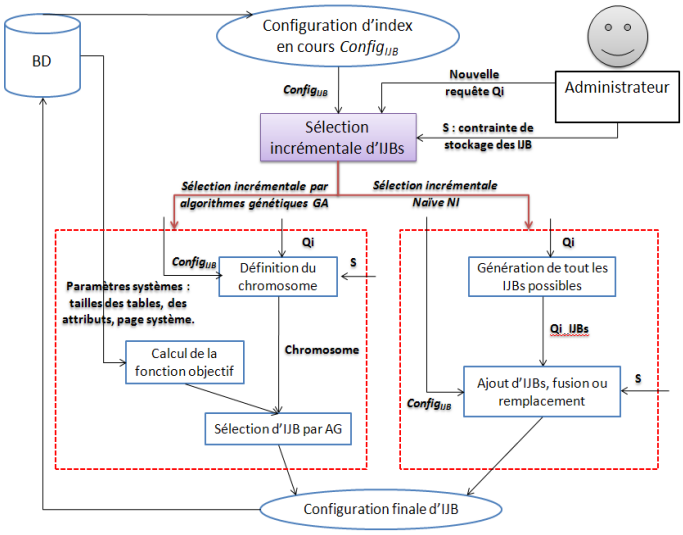


FIG. 1 – *Architecture de sélection incrémentale des IJB multiples*

Pour répondre à cette sélection incrémentale, nous proposons une approche naïve (NI). Elle part d'une configuration d'index $Config\_IJB$. A l'arrivée de chaque nouvelle requête $Q_i$, elle effectue les actions suivantes (figure 1) :

1. Extraire la configuration courante d'index $Config\_IJB$ et générer les sous index de la requête $Q_i$ appelé l'ensemble $Q_i\_IJB$.
2. Tant que l'espace de stockage d'index $\mathcal{S}$ n'est pas saturé, ajouter à $Config\_IJB$ un sous index de $Q_i$ appelé $Qi\_IJB_j$.

3. Si l'espace $\mathcal{S}$ est saturé, effectuer une fusion des index de $Config\_IJB$. La fusion est réalisée par la recherche dans $Config\_IJB$ des index qui forment une fusion d'un index de $Qi\_IJB_j$. Par exemple, la fusion des deux index CY et YD donne l'$\mathcal{IJB}$ CYD.
4. Si aucune fusion n'est possible, remplacer dans $Config\_IJB$ l'index le moins bénéfique pour la charge de requêtes, par le sous index le plus bénéfique de $Q_i$.

Si l'espace de stockage d'index est saturé, l'opération de fusion d'index est effectuée afin d'optimiser une requête. Cette fusion peut générer un index volumineux qui pourrait être ignoré par le processus de sélection, même s'il est bénéfique.

Dans le souci de garder le même environnement d'algorithme, nous avons étendu notre AG défini pour la sélection statique pour répondre à l'aspect incrémental. L'AG adapte le chromosome afin de prendre en compte les nouveaux index générés par la requête en cour de traitement. De ce fait, une nouvelle sélection d'index est réalisée afin de choisir la meilleure configuration d'index pour toute la charge y compris la nouvelle requête. Le déroulement du processus de sélection est réalisé comme suit (figure 1) :

A l'arrivée de chaque nouvelle requête $Q_i$, toute la structure du chromosome est redéfinie afin de prendre en compte les sous index de $Q_i$. Par la suite, l'AG guidé par le modèle de coût, permet de sélectionner les $\mathcal{IJB}$ quasi-optimaux. La principale particularité de cet algorithme est qu'il se base sur la manipulation du codage qui devient la clé de support de l'incrémental.

# 5 Expérimentation

Afin de comparer les différentes stratégies de sélection d'$\mathcal{IJB}$ simples et multiples, statiques et incrémentales, nous avons réalisé des tests de comparaison sur un ED réel issu du benchmark APB1 (Council, 1998) sous le SGBD Oracle 11g. Cet entrepôt est composé d'une table de faits *Actvars*(24 786 000 tuples) et quatre tables de dimension *Prodlevel* (9000 tuples), *Custlevel* (900 tuples), *Timelevel* (24 tuples) et *Chanlevel* (9 tuples). Nous avons considéré une charge de requêtes de jointures en étoile contenant **70 requêtes** avec **18 attributs** de sélection *(Line, Day, Week, Country, Depart, Type, Sort, Class, Group, Family, Division, Year, Month, Quarter, Retailer, City, Gender et All)* avec les cardinalités : 15, 31, 52, 11, 25, 25, 4, 605, 300, 75, 4, 2, 12, 4, 99, 4, 2, 3 respectivement. Afin d'effectuer la sélection des index par algorithmes génétiques, nous avons utilisé l'API JGAP. Nos tests se déroulent sur deux phases : nous présentons d'abord les expérimentations menées sur la sélection statique des $\mathcal{IJB}$, puis nous réalisons des tests sur la sélection incrémentales d'$\mathcal{IJB}$.

## 5.1 Tests sur la sélection statique des IJB

Nous présentons ci-dessous deux types d'expériences, une évaluation théorique effectuée en utilisant le modèle de coût théorique, que nous avons défini dans la section 3, et une validation sous Oracle 11g qui utilise un modèle de coût réel pour estimer le coût d'exécution des requêtes. Nos tests visent à déterminer la stratégie de sélection statique d'index la plus performante pour optimiser la charge de requêtes. Pour cela nous avons implémenté la démarche de sélection d'index simples $SI$ et deux démarches de sélection d'index multiples à savoir $MIQ$ et $MIQ^*$.

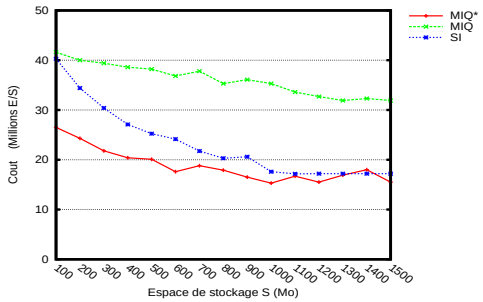


**FIG. 2** – *Coût d'exécution des requête Vs. espace de stockage S*

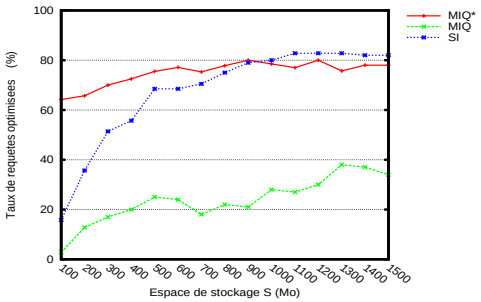


**FIG. 3** – *Taux de requêtes optimisées Vs. espace de stockage S*

Dans l'étude théorique, nous utilisons le modèle de coût défini précédemment afin de comparer les stratégies ($SI$, $MIQ$ et $MIQ^*$). Ce modèle de coût estime le nombre d'entrées/sorties (E/S) nécessaire pour exécuter une requête et permet de guider l'AG afin de trouver la configuration optimale (quasi-optimale) d'$\mathcal{IJB}$. Il est à noter que le coût total des 70 requêtes sans optimisation est de $42.5\ millions\ E/S$

Dans la première expérimentation, nous exécutons l'AG avec les trois stratégies et varions la contrainte d'espace de stockage (S) de 100Mo à 1500Mo. Pour chaque stratégie et chaque valeur de S, nous calculons le coût d'exécution de la charge (figure 2) et le taux des requêtes optimisées (figure 3) en présence des index générés. Cette expérimentation montre que la stratégie $MIQ$ n'est pas bénéfique, à cause de la structure du chromosome. En effet, chaque $\mathcal{IJB}$ est construit avec tous les attributs d'une requête donnée, ce qui génère des index volumineux qui violent la contrainte d'espace de stockage. Nous avons comparé les résultats des stratégies $SI$ et $MIQ^*$ et selon les valeurs de S, nous classifions les résultats comme suit :

1. $0$ - $900Mo$ : la stratégie $MIQ^*$ donne une meilleure optimisation avec un coût de la charge qui démarre à $27\ millions\ E/S$ avec $62\%$ des requêtes optimisées ($S = 100Mo$), contrairement à $SI$ qui démarre avec un coût de $40\ millions\ E/S$ et seulement $18\%$ des requêtes sont optimisées. En effet, selon le chromosome de $MIQ^*$, pour chaque $\mathcal{IJB}$ défini sur tous les attributs d'une requête, des sous index existent avec moins d'attributs et donc moins d'espace de stockage requis. De plus, deux index simples définis sur deux attributs $A_1$ et $A_2$ sont plus volumineux qu'un index multiple défini sur les même attributs. De ce fait, il y a davantage d'index pour optimiser chaque requête sans violation de la contrainte d'espace de stockage.
2. $900$ - $1500Mo$ : Les deux stratégies $SI$ et $MIQ^*$ sont bénéfiques. Le coût total de la charge de requêtes est réduit à $16\ millions\ E/S$ avec un taux de $80\%$ de requêtes optimisées. En effet, pour $SI$, 16 attributs sur 18 sont sélectionnés par l'AG afin de créer 16 $\mathcal{IJB}$, ce qui couvre l'optimisation d'une majorité des requêtes ($80\%$). Notons que les deux stratégies $SI$ et $MIQ^*$ donnent des résultats similaires mais nous devons choisir comme stratégie $MIQ^*$ car elle permet une meilleure optimisation de la charge de requêtes sur toutes les valeurs de S.

Afin de évaluer l'influence du nombre d'attributs indexables candidats à la sélection d'index, nous varions ce nombre de 2 à 18 sous une contrainte d'espace $S = 0,8Go$. Pour chaque

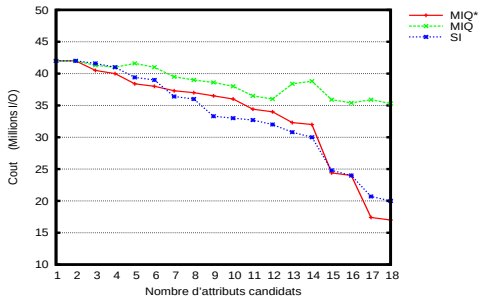


**FIG. 4** – *Coût d'exécution vs. Nombre d'attributs candidats à la sélection*

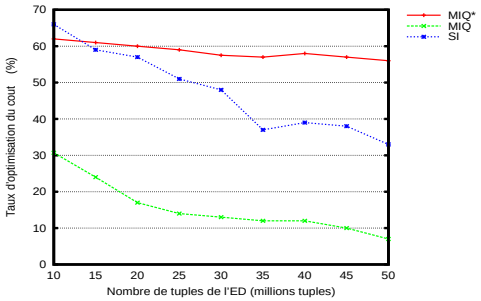


**FIG. 5** – *Coût d'exécution vs. Taille de l'entrepôt*

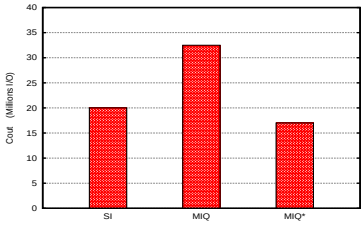


**FIG. 6** – *Le coût réel des requêtes sous Oracle11g ($SI$, $MIQ$ et $MIQ^*$)*

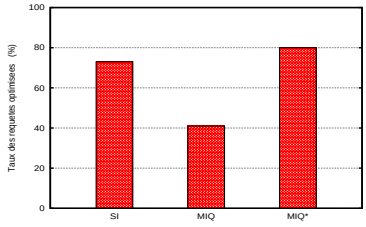


**FIG. 7** – *Le taux d'optimisation des requêtes sous Oracle11g ($SI$, $MIQ$ et $MIQ^*$)*

nombre d'attributs, nous exécutons la sélection d'$\mathcal{IJB}$ par AG selon les trois stratégies. La figure 4 illustre les résultats en coût d'exécution des requêtes. Les tests montrent que l'augmentation du nombre d'attributs indexables influe positivement sur l'optimisation. En effet, plusieurs index sont sélectionnés et le coût est réduit à $20\ millions\ E/S$ pour $SI$ et jusqu'à $17\ millions\ E/S$ pour $MIQ^*$.

Pour la troisième expérimentation, nous évaluons les performances d'optimisations par $\mathcal{IJB}$ selon l'augmentation de la taille de l'entrepôt de données. En effet, l'ED évolue continuellement et rend l'exécution de la charge de requêtes de plus en plus complexe, il faut donc qu'une stratégie d'optimisation soit efficace devant cette complexité. Ainsi, nous exécutons la sélection des $\mathcal{IJB}$ avec $SI$, $MIQ$ et $MIQ^*$, sous une contrainte d'espace $S = 0,8Go$, et varions le nombre de tuple de la table de faits de 10 millions tuples à 50 millions tuples. Le taux d'optimisation du coût de la charge de requêtes est illustré sur la figure 5. Cette figure montre que le taux d'optimisation du coût diminue avec l'augmentation de la taille de l'ED, car les requêtes deviennent de plus en plus complexes. Nous remarquons que la dégradation des performances est plus significative pour les stratégies $SI$ et $MIQ$ que pour $MIQ^*$, ce qui nous amène à conclure que la stratégie $MIQ^*$ peut mieux faire face au problème d'évolution continuelle de l'entrepôt.

Afin de valider notre sélection statique des $\mathcal{IJB}$ (simples et multiples), nous réalisons des tests sous Oracle 11g avec le banc d'essai APB1 (Council, 1998). Nous utilisons la charge des 70 requêtes en étoile. A travers le modèle de coût théorique, nous exécutons les trois stratégies

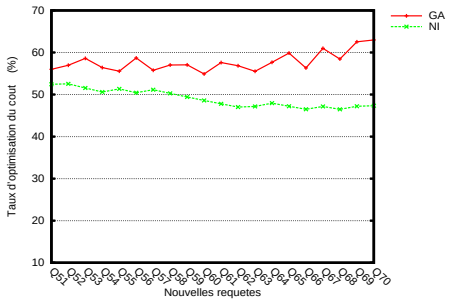


**FIG. 8** – *Taux de réduction du coût d'exécution des requêtes : NI vs. GA*

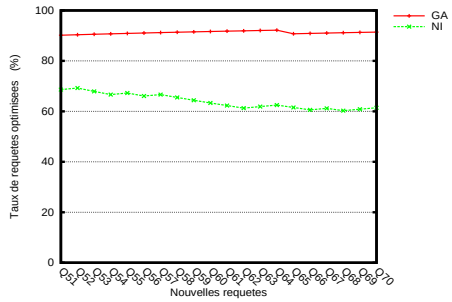


**FIG. 9** – *Taux d'optimisation de la charge de requêtes : NI vs. GA*

de sélection ($SI$, $MIQ$ et $MIQ^*$) avec une contrainte d'espace de stockage $S = 0,8Go$. Chaque stratégie génère une configuration d'$\mathcal{IJB}$ que nous implémentons effectivement sur l'entrepôt de données. Après chaque implémentation, nous calculons le *Coût Réel* des requêtes en utilisant le modèle de coût réel. Ce modèle représente le coût réel des 70 requêtes exécutées sur l'ED après implémentation des index. Afin de calculer ce coût, nous avons développé une classe JAVA appelée ORACLECOST qui fait appel à l'Optimiseur Oracle à travers l'opération EXPLAIN PLAN. Cette opération estime le coût réel d'exécution d'une charge de requêtes, en se basant sur des statistiques, sans une réelle exécution, et stocke les résultats dans une table système d'Oracle appelée PLAN_TABLE. Par la suite, notre classe JAVA ORACLECOST accède à cette table et récupère le coût des requêtes. Les figures 6 et 7 montrent respectivement le coût réel de la charge de requêtes et le taux des requêtes optimisées.

Les tests sous Oracle11g montrent que la sélection basée $MIQ^*$ donne de meilleurs résultats que les deux autres stratégies (17.1 $millions$ $E/S$ et $80\%$ des requêtes sont optimisées). Nous concluons également que le modèle de coût théorique est proche du modèle réel et estime bien le coût de la charge de requêtes

## 5.2 Tests sur la sélection incrémentale des IJB multiples

L'étude incrémentale est réalisée avec ajout successif de 20 nouvelles requêtes en considérant une contrainte d'espace de stockage $S = 1Go$. Afin d'avoir une bonne base de comparaison, nous supposons la charge des 50 requêtes optimisées avec un ensemble de multi $\mathcal{IJB}$ sélectionnés préalablement par algorithmes génétiques. Sur cette base d'index et pour chaque requête nouvellement exécutée, nous réalisons deux sélections incrémentales : une basée NI et une autre basée GA. Pour chaque sélection (NI et GA), et pour chaque nouvelle requête, nous relevons quatre informations : (1) le coût d'exécution de toute la charge de requêtes en cour, à partir duquel est calculé le taux d'optimisation du coût, (2) le taux des requêtes ayant été optimisées, (3) le nombre d'index sélectionnés et enfin (4) le nombre d'attributs utilisés par ces index.

Les figures 8 et 9 montrent respectivement le taux de réduction du coût d'exécution et le taux de requêtes optimisées, selon la sélection incrémentale par NI et par GA. Nous remarquons que les algorithmes génétiques permettent d'apporter la meilleure optimisation de

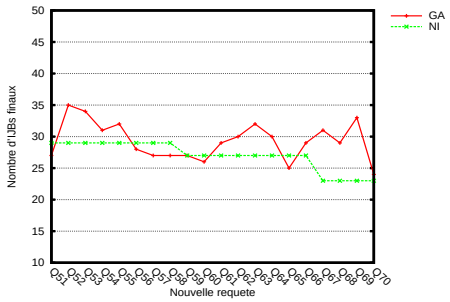


**FIG. 10** – *Nombre d'$\mathcal{IJB}$ sélectionnés : NI vs. GA*

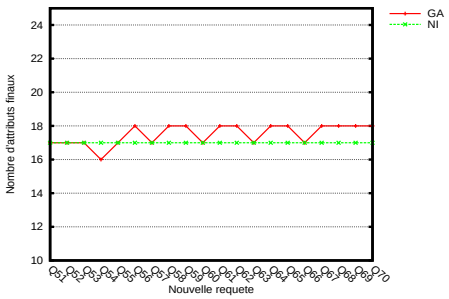


**FIG. 11** – *Nombre d'attributs figurant dans les $\mathcal{IJB}$ sélectionnés : NI vs. GA*

la charge de requêtes. En effet, GA permet en moyenne une réduction de $58\%$ du coût total des requêtes avec $90\%$ des requêtes optimisées, contre $49\%$ de réduction de coût et $62\%$ des requêtes optimisées pour NI. De plus, contrairement à la sélection NI qui dégrade les performances, la sélection GA apporte une amélioration continue du coût de la charge pour chaque nouvelle requête. Cela est dû aux modifications incrémentales apportées au chromosome, prenant en compte les nouveaux index créés à partir de chaque nouvelle requête exécutée, ce qui permet de couvrir l'optimisation d'un plus grand nombre de requêtes et améliorer les requêtes déjà optimisées.

La figure 10 illustre le nombre d'index finaux générés par les deux sélections incrémentales, à l'issu de l'arrivée de chaque nouvelle requête. Nous pouvons noter que le nombre d'index diminue pour la sélection NI, car celle-ci se base sur le principe de fusion d'index lorsque la contrainte d'espace de stockage est violée. Concernant GA, chaque arrivée d'une requête donne lieu à une nouvelle sélection d'index sur un nouveau chromosome, ce qui explique la fluctuation du nombre d'index. La même remarque peut être faite concernant le nombre d'attributs (figure 11). Pour NI, à partir du moment où l'opération réalisée est la fusion d'index, les mêmes attributs subsistent, contrairement à GA, ce qui montre un changement dans les attributs choisis.

# 6 Conclusion

Dans ce travail, nous avons abordé la sélection statique et incrémentale des index de jointure binaires. Concernant la sélection statique, nous avons montré la nécessité d'élaguer l'espace de recherche des index afin de réduire sa complexité. Nous avons proposé un nouveau type d'élagage basé sur les requêtes. Nous avons également identifié le besoin de développer des algorithmes avancés pour la sélection multiple des index de jointure binaire. Pour ce faire, nous avons proposé un algorithme génétique qui peut être adapté aux index simples et multiples. Plusieurs codage de chromosomes on été étudiés. Nous voulons avoir un codage flexible qui prend en considération les requêtes définies sur l'entrepôt. L'implémentation de nos algorithmes génétiques est réalisée à l'aide d'une API JGAP. La flexibilité de notre codage nous a permis de développé un algorithme génétique pour satisfaire la sélection incrémentale. Une architecture supportant nos méthodes de sélection a été définie. Elle peut être connectée

directement à un SGBD. Une étude expérimentale intensive a été conduite pour montrer l'intérêt de nos contributions. Ces tests ont montré l'intérêt de l'utilisation des index multiples, de la sélection des index par algorithmes génétiques et de l'élagage par requêtes et par requêtes améliorées.

Comme perspectives, il serait intéressant d'approfondir la sélection incrémentale et définir une architecture pour que les SGBD puissent la supporter. Une autre piste consiste à définir des structures de données dynamiques afin répondre aux différentes évolutions d'un entrepôt de données.

## Références

Aouiche, K., O. Boussaid, et F. Bentayeb (2005). Automatic selection of bitmap join indexes in data warehouses. In *7th International Conference on Data Warehousing and Knowledge Discovery (DAWAK'05)*, pp. 64–73.

Azefack, S., K. Aouiche, et J. Darmont (2007). Dynamic index selection in data warehouses. *4th International Conference on Innovations in Information Technology (Innovations 07), Dubai*.

Bäck, T. (1995). *Evolutionnary algorithms in theory and practice*. Oxford University Press, New York.

Bellatreche, L. et K. Boukhalfa (2010). Yet another algorithms for selecting bitmap join indexes. In *International Conference on Data Warehousing and Knowledge Discovery (DaWaK'2010)*, pp. 105–116.

Bellatreche, L., R. Missaoui, H. Necir, et H. Drias (2008). A data mining approach for selecting bitmap join indices. *Journal of Computing Science and Engineering 2*(1), 206–223.

Bouchakri, R., L. Bellatreche, et K. Boukhalfa (2010). Une approche par k-means de sélection multiple de structures d'optimisation dans les entrepôts de données. In *6ème Journée Francophone sur les Entrepôts de données et l'Analyse en ligne (EDA'10), Revue des Nouvelles Technologies*, pp. 207–222.

Boukhalfa, K., L. Bellatreche, et B. Ziani (2010). Index de jointure binaires : Stratégies de sélection et étude de performances. In *6éme Journée Francophone sur les Entrepôts de données et l'Analyse en ligne (EDA'10), Revue des Nouvelles Technologies*, pp. 175–190.

Chan, C. Y. et Y. E. Ioannidis (1998). Bitmap index design and evaluation. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 355–366.

Comer, D. (1979). The ubiquitous b-tree. *ACM Comput. Surv. 11*(2), 121–137.

Council, O. (1998). Apb-1 olap benchmark, release ii. *http ://www.olapcouncil.org/research/bmarkly.htm*.

Ioannidis, Y. et Y. Kang (1990). Randomized algorithms algorithms for optimizing large join queries. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 9–22.

Kimball, R. et K. Strehlo (1995). Why decision support fails and how to fix it. *SIGMOD Record 24*(3), 92–97.

L.Bellatreche, K.Boukhalfa, et M. Mohania (2007). Pruning search space of physical database design. In *18th International Conference On Database and Expert Systems Applications (DEXA'07)*, pp. 479–488.

O'Neil, P. et G. Graefe (1995). Multi-table joins through bitmapped join indices. *SIGMOD Record 24*(3), 8–11.

O'Neil, P. et D. Quass (1997). Improved query performance with variant indexes. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 38–49.

Pasquier, N., Y. Bastide, R. Taouil, et L. Lakhal (1999). Discovering frequent closed itemsets. In *International Conference on Database Theory (ICDT)*, pp. 398–416.

Rao, J., C. Zhang, G. Lohman, et N. Megiddo (2002). Automating physical database design in a parallel database. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 558–569.

Stöhr, T., H. Märtens, et E. Rahm (2000). Multi-dimensional database allocation for parallel data warehouses. In *Proceedings of the International Conference on Very Large Databases*, pp. 273–284.

Systems, R. B. (1997). Star schema processing for complex queries. *White Paper*.

Valduriez, P. (1987). Join indices. *ACM Transactions on Database Systems 12*(2), 218–246.

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

Zhang, C. et J. Yang (1999). Genetic algorithm for materialized view selection in data warehouse environments. *Proceeding of the International Conference on Data Warehousing and Knowledge Discovery (DAWAK'99 )*, 116–125.

## Summary

Bitmap join indexes have been largely used in the context of data warehouse to optimize complex queries. Their selection remains hard, since it needs to explore a large search space. Only few of classes of algorithms were proposed to deal with the problem of bitmap join index selection. These algorithms are static and do not take into account the changes of data warehouses. In this paper, we first propose a new genetic algorithm to select bitmap join indexes defined on multiple attributes belonging to various dimension tables in the static case. Secondly, an extension of this algorithm is proposed to consider the incremental case. Finally, intensive experiments are conducted to show the efficiency of our proposal in the static and incremental cases.