Chapitre 4

Recherche de motifs fréquents fermés pour la sélection d'index

4.1 Introduction

Les entrepôts de données sont généralement modélisés selon un schéma en étoile contenant une table de faits centrale volumineuse et un certain nombre de tables dimensions représentant les descripteurs des faits [Inm02, KR02]. La table de faits contient des clés étrangères vers les clés des tables dimensions, ainsi que des mesures numériques. Avec ce type de modèle, une requête décisionnelle nécessite une ou plusieurs jointures entre la table de faits et les tables dimensions. De plus, le schéma de l'entrepôt peut comporter des hiérarchies au niveau des dimensions (schéma en flocon de neige), ce qui entraîne des jointures additionnelles. Ces jointures sont très coûteuses en terme de temps de calcul. Ce coût devient prohibitif lorsque les jointures opèrent sur de très grands volumes de données. Il est alors crucial de le réduire.

Plusieurs techniques ont été proposées pour améliorer le temps de calcul des jointures dans les bases de données, comme les jointures par hachage, par tri-fusion et la jointure imbriquée [ME92]. Cependant, ces techniques ne sont efficaces que quand la jointure s'applique à deux tables et que le volume de données est relativement faible. Lorsque le nombre de jointures est supérieur à deux, il faut alors les ordonner en fonction des tables à joindre (problème d'ordonnancement des jointures). D'autres techniques, utilisées dans les entrepôts

de données, exploitent des index de jointure pour précalculer ces dernières afin d'assurer un accès rapide aux données. L'administrateur de l'entrepôt de données a donc pour tâche cruciale de choisir les meilleurs index à construire.

Avec le développement des bases de données en général et des entrepôts de données en particulier, il est devenu très important de réduire la fonction d'administration des SGBD [WMHZ02]. Les systèmes auto-administratifs ont pour objectif de s'administrer et de s'adapter eux-mêmes, automatiquement, sans perte (ou même avec un gain) de performance. Dans le cadre du stage de DEA Extraction des Connaissance à partir des Données [Aou02], nous avons proposé une démarche de sélection automatique d'index dans les bases de données fondée sur l'extraction de motifs fréquents [ADG03a, ADG03b]. Nous avons également montré quelques pistes pour l'adaptation de cette démarche dans le contexte des entrepôts de données [ADB04]. Dans ce chapitre, nous présentons la poursuite de nos travaux dans cette voie. Partant du constat que tous les index candidats fournis par la phase d'extraction des motifs fréquents ne peuvent pas être construits en pratique (contraintes systèmes ou d'espace de stockage), nous proposons une stratégie permettant de sélectionner les plus avantageux grâce à des modèles de coût. Ces modèles nous permettent d'estimer le coût d'accès aux données à travers ces index, ainsi que les coûts de maintenance et de stockage. Un algorithme glouton exploite ces modèles afin de recommander une configuration d'index pertinente.

Notre stratégie de sélection d'index est modulaire et, de ce fait, facilement adaptable à une autre technique d'indexation. En effet, il suffit d'adapter le module de génération des index à partir des motifs fréquents fermés et d'appliquer les modèles de coût correspondant à ces index. Dans cette étude, nous avons choisi de nous focaliser sur les index *bitmap* de jointure car ils sont bien adaptés à l'environnement des entrepôts de données. En effet, les *bitmaps* de ces index rendent efficace l'exécution d'opérations courantes comme `And`, `Or`, `Not` ou `Count` qui opèrent directement sur les *bitmaps* (donc en mémoire) et non plus sur les données sources. De plus, les jointures sont préalablement calculées au moment de la création de ces index. Elles ne sont donc pas calculées lors de l'exécution des requêtes. D'autre part, l'espace disque occupé par les *bitmaps* est faible, notamment quand la cardinalité des attributs indexés n'est pas élevée [Sar97, Wu99]. Ces attributs sont souvent utilisés dans les clauses `Where` et `Group by` des requêtes décisionnelles [HLL03].

Ce chapitre est organisé comme suit. Avant de présenter le principe de notre démarche de sélection automatique d'index à base d'extraction de motifs fréquents fermés à la Section 4.3, nous discutons et motivons à la Section 4.2 notre choix d'utiliser les motifs fréquents fermés pour la sélection d'index. Nous présentons ensuite nos modèles de coût à la Section 4.4 et détaillons notre stratégie de sélection d'index à la Section 4.5. Afin de valider celle-ci, nous l'avons expérimentée sur un entrepôt de données test. Les résultats sont rapportés à la Section 4.6. Nous terminons enfin par une conclusion et une discussion à la Section 4.7.

4.2 Recherche de motifs fréquents

4.2.1 Contexte

Notre propos dans cette section n'est pas de faire une étude des algorithmes de recherche de motifs fréquents, mais de présenter comment et pourquoi nous recourons à ces algorithmes, plus particulièrement à l'algorithme Close, dans notre stratégie de sélection automatique d'index.

Rappelons que nous voulons concevoir un système qui exploite les données stockées dans une base de données afin d'en extraire des connaissances utiles au choix des index. Nous pensons que la pertinence d'un index est fortement corrélée avec la fréquence de son utilisation dans l'ensemble des requêtes d'une charge. *A priori*, la recherche des fréquents est un moyen qui nous semble approprié pour rendre compte de cette corrélation et ainsi faciliter le choix des index à construire.

Définition 4.2.1 (Motif fréquent) *Soient $I = i_1, \ldots, i_m$ un ensemble de m items et $B = t_1, \ldots, t_n$ une base de données de n transactions. Chaque transaction est composée d'un sous-ensemble d'items $I' \subseteq I$. Un sous-ensemble I' de taille k est appelé un k-itemset. Une transaction t_i contient un motif I' si et seulement si $I' \subseteq t_i$. Le support d'un motif I' est la proportion de transactions de B qui contiennent I'. Le support est donné par la formule suivante.*

$$support(I') = \frac{|\{t \in B, I' \subseteq t\}|}{|\{t \in B\}|}$$

Un motif dont le support est supérieur ou égal au seuil minimal du support *minsup*, défini par l'utilisateur, est appelé un motif fréquent. Par exemple, pour le contexte d'extraction présenté au Tableau 4.1 et une valeur de *minsup* égale à $\frac{2}{6}$, les motifs A, B, C et E sont fréquents car leur supports sont égaux à $\frac{3}{6}$, $\frac{5}{6}$, $\frac{5}{6}$ et $\frac{5}{6}$, respectivement. Par contre, le motif D n'est pas fréquent car son support est égal à $\frac{1}{6}$.

OID (Identifiant d'objet)	Motifs
1	$\{A\,C\,D\}$
2	$\{B\,C\,E\}$
3	$\{A\,B\,C\,E\}$
4	$\{B\,E\}$
5	$\{A\,B\,C\,E\}$
6	$\{B\,C\,E\}$

TAB. 4.1 – Exemple de contexte d'extraction de motifs fréquents

Définition 4.2.2 (Motif fréquent fermé) *Un motif fermé est un ensemble maximal de motifs communs à un ensemble d'objets. Un motif* $i \subseteq I$ *tel que* $support(i) \geq minsup$ *est appelé motif fréquent fermé.*

Par exemple, dans le contexte d'extraction du Tableau 4.1, le motif $\{B\ C\ E\}$ est un motif fermé car il est l'ensemble maximal d'items communs aux objets $\{2, 3, 5, 6\}$. Le motif $\{B\ C\}$ n'est pas un motif fermé car tous les objets contenant B et C (objets 2, 3, 5 et 6) contiennent également l'item E.

4.2.2 Motivations

Plusieurs algorithmes traitent le problème de la recherche des motifs fréquents. Nous citons, à titre d'exemple, Apriori [AS94, Sri96], AprioriTID [AS94], Partition [SON95] et Close [PBTL99a, PBTL99b]. Ce dernier nous intéresse plus particulièrement pour les raisons suivantes.

Dans notre cas d'étude, les objets sont des requêtes et les items sont les attributs extraits de ces requêtes. La charge de requêtes peut être volumineuse. L'algorithme Close s'avère adapté à cette volumétrie. En effet, Close est basé sur les opérateurs de fermeture de

Galois, qui permettent la détermination efficace des ensembles fermés, selon cette fermeture, dans les grandes bases de données. En comparaison des autres algorithmes existants de détermination des ensembles fermés, Close permet de réduire le nombre d'accès aux données du contexte d'extraction, qui constituent les opérations les plus coûteuses en temps lorsque ces algorithmes sont appliqués à de grands volumes de données [Pas00].

Par ailleurs, l'utilisateur d'un entrepôt de données suit en général un raisonnement logique dans un processus d'interrogation. Les données interrogées dans une session de travail sont souvent corrélées. De ce fait, les requêtes de la charge, prises dans un intervalle continu de temps, le sont aussi. Cette corrélation entre les requêtes donne lieu à un contexte d'extraction dense. Dans ce cas, l'algorithme Close est performant en temps de calcul et en espace mémoire nécessaires à la recherche des motifs fréquents.

De plus, l'ensemble des motifs fermés fréquents étant un sous-ensemble de l'ensemble des motifs fréquents[1], le nombre d'opérations nécessaires à leur extraction est inférieur au nombre d'opérations nécessaires à la recherche des motifs fréquents. D'une part, cela a pour effet de réduire considérablement le temps de calcul des motifs fréquents fermés par rapport au temps de calcul des motifs fréquents. D'autre part, le nombre d'index candidats dépend du nombre de motifs fréquents. Plus le nombre de motifs fréquents est élevé, plus on génère d'index candidats. Pour éviter la prolifération des index candidats, il est judicieux de générer ces index à partir des motifs fréquents fermés dont le nombre est moins important que les motifs fréquents. Cela permet de réduire la complexité du problème de sélection d'index.

4.2.3 Algorithme Close

L'algorithme Close parcourt l'ensemble des générateurs[2] des motifs fermés fréquents par niveaux. À l'étape $k = 1$, l'ensemble des 1-générateurs est initialisé aux 1-itemsets.

À chaque itération, l'algorithme considère un ensemble de k-itemsets générateurs. Il construit un ensemble de motifs fermés candidats qui sont les fermetures de ces k-générateurs et détermine ensuite parmi ces candidats les motifs fermés fréquents selon le seuil minimal du support *minsup*. Finalement, il crée les $(k + 1)$-générateurs qui seront utilisés lors de

[1] L'ensemble des motifs fréquents peut être généré directement à partir de l'ensemble des motifs fréquents fermés.

[2] Un motif générateur d'un motif fermé est un motif minimal (selon la relation d'inclusion) dont la fermeture par l'opérateur est ce motif fermé.

l'itération suivante afin de construire l'ensemble des motifs fermés candidats qui sont les fermetures des $(k+1)$-générateurs. Un balayage du contexte d'extraction est nécessaire durant chaque itération, afin de déterminer les fermetures des k-générateurs et calculer leur supports.

Si l'ensemble de k-générateurs fréquents est vide, l'algorithme s'arrête. Sinon, ce nouvel ensemble de $(k+1)$-générateurs est utilisé à l'itération suivante.

La Figure 4.1 montre un exemple d'extraction de motifs fermés fréquents dans le contexte du Tableau 4.1 avec Close pour un support minimal égal à $\frac{2}{6}$.

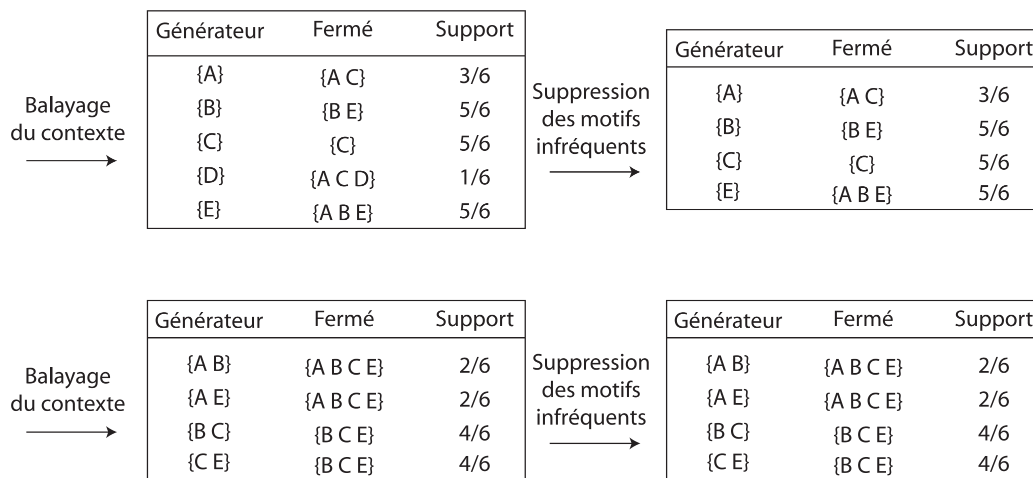

FIG. 4.1 – Extraction des motifs fermés fréquents avec Close pour un $minsup = \frac{2}{6}$

4.3 Démarche de sélection automatique d'index

L'approche que nous proposons, dont le principe général est représenté à la Figure 4.2, exploite une charge de requêtes afin d'en extraire une configuration d'index améliorant le temps d'accès aux données.

Tout d'abord, nous extrayons de la charge des attributs indexables, c'est-à-dire, les attributs utiles lors de l'exécution des requêtes s'ils sont indexés. Ces attributs sont stockés dans une matrice, dite "requêtes-attributs", qui correspond à un contexte d'extraction exploité par l'algorithme de fouille de données Close. Nous obtenons alors un ensemble de

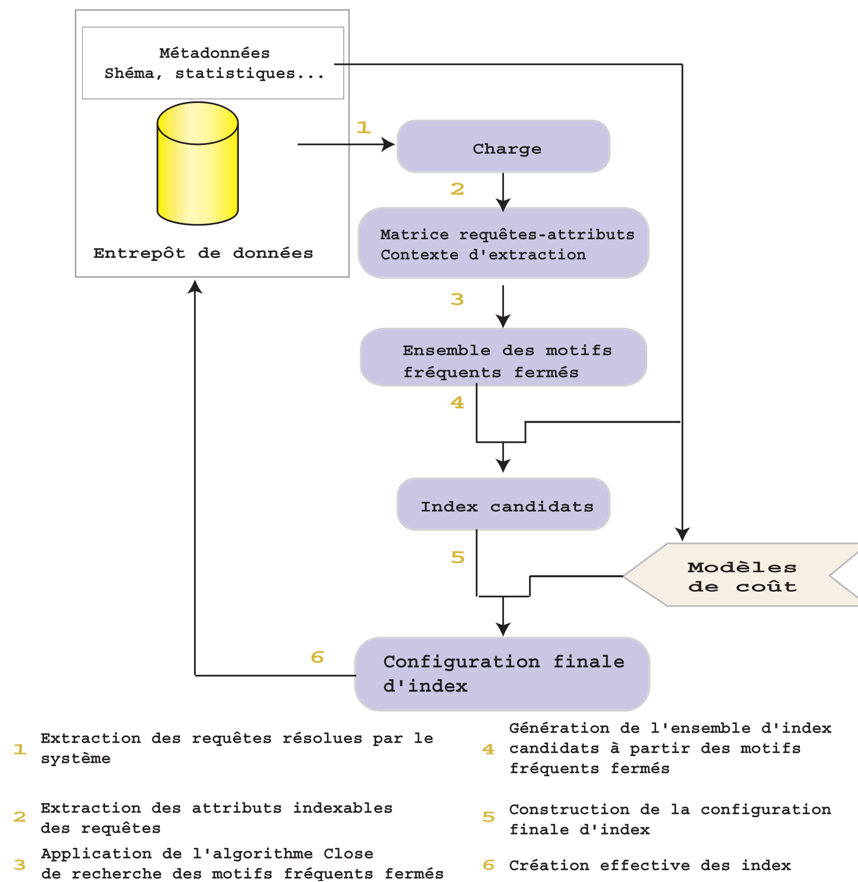

FIG. 4.2 – Architecture de notre stratégie de sélection automatique d'index

motifs fréquents fermés. Chaque motif de cet ensemble est analysé afin de générer un ensemble d'index candidats en s'appuyant sur les métadonnées (schéma : clés primaires, clés étrangères ; statistiques. . .) de l'entrepôt de données. Enfin, nous procédons à un processus d'élagage, suivant des modèles de coût, avant de construire effectivement la configuration d'index.

Nous procédons comme suit pour proposer une configuration d'index :

- extraction de la charge des requêtes,
- analyse de la charge pour extraire les attributs indexables,
- construction du contexte de recherche des motifs fréquents,
- application de l'algorithme Close sur ce contexte,

– construction de la configuration finale d'index.

Nous détaillons chacune de ces étapes dans les sections qui suivent.

4.3.1 Extraction de la charge

La première étape de notre stratégie de sélection d'index consiste à extraire du journal des transactions les requêtes adressées au SGBD. Cette tâche peut être réalisée en exploitant la capacité des SGBD actuels de sauvegarder les opérations réalisées sur le système durant une période de temps. Cette période est fixée par l'administrateur et doit être suffisante pour pouvoir anticiper les requêtes futures des utilisateurs.

La charge peut être directement obtenue à partir du journal des transactions du SGBD hôte ou bien grâce à une application externe telle que Log Explorer de Lumigent Technologies[3].

Dans l'environnement des entrepôts de données, les requêtes extraites sont décisionnelles. Dans un schéma en étoile, elles peuvent être exprimées en algèbre relationnelle comme suit :

$$q = \pi_{G,M}\sigma_S(F \bowtie D_1 \bowtie D_2 \bowtie \ldots \bowtie D_d)$$

où S est une conjonction de prédicats simples sur les attributs des tables dimensions, G est un ensemble d'attributs des tables dimensions D_i (les attributs de la clause `Group by`), et M est un ensemble de mesures agrégées, chacune étant définie en appliquant un opérateur d'agrégation sur une mesure de la table de faits F.

4.3.2 Analyse de la charge

Les requêtes SQL présentes dans la charge sont traitées par un analyseur syntaxique afin d'en extraire tous les attributs susceptibles d'être des supports d'index. Ces attributs sont ceux présents dans les clauses `Where` des requêtes. Ces attributs servent à la recherche dans les requêtes d'interrogation. Dans les systèmes décisionnels, les requêtes sont de type `Select` (*read only*) et les rafraîchissements (*batch update*) sont réalisés périodiquement [VG99]. Nous ne construisons donc la matrice "requêtes-attributs" qu'à partir des requêtes d'interrogation.

[3] Log Explorer est en téléchargement sur le site `http://www.lumigent.com/LogExplorer`.

Illustrons quelques règles permettant de proposer des attributs indexables à travers les exemples suivants. Soit un prédicat de restriction qui a l'une des formes suivantes : $t.a$ `between` v_1 `and` v_2, $t.a\,\theta\,v_2$, $t.a$ `like` v_1 `ou` $t.a$ `in` $(v_1, v_2, ...)$

où :

- t est le nom ou l'alias d'une table ;
- a est un attribut de cette table ;
- θ est l'un des opérateurs $=$, $\neq$, $<$, $>$, $\leq$ ou $\geq$;
- v_1 v_2 sont des valeurs constantes ne faisant pas référence à une table (nom ou alias d'une table).

Plusieurs cas peuvent conduire à proposer des index "inutiles", c'est-à-dire des index candidats qui ne seront pas exploités pour répondre aux requêtes de la charge.

1. Un prédicat de la forme $E(t.a) = v_1$, où $E(t.a)$ est une expression en $t.a$ et, est, par exemple, $t.a^2 + 3t.a$. Dans un tel cas, $t.a$ n'est pas l'opérande immédiat de l'opérateur de comparaison. Le SGBD n'utilise pas d'index sur l'attribut a pour chercher les n-uplets vérifiant ce prédicat.
2. Un prédicat de la forme $t.a \neq v_1$ n'utilise pas d'index construits sur a car toutes les données sont parcourues sauf v_1, si elle existe dans la table.
3. Un prédicat de comparaison de chaînes de caractères utilisant `like` n'exploite pas l'existence d'un index pour chercher les n-uplets vérifiant ce prédicat si la recherche s'effectue sur une sous-chaîne (par exemple, dans le cas d'utilisation d'un joker au début `like '%chaîne'`) plutôt que sur la totalité de la chaîne.

4.3.3 Construction du contexte d'extraction

À partir des attributs extraits dans l'étape précédente, nous construisons une matrice "requêtes-attributs" qui a pour lignes les requêtes de la charge et pour colonnes les attributs à indexer. L'existence d'un attribut indexable dans une requête est symbolisée par un et son absence par zéro. Nous illustrons la construction de cette matrice à travers l'exemple suivant.

Soit un entrepôt de données composé de la table de faits `Sales` et de cinq tables dimensions `Channels`, `Customers`, `Products`, `Times` et `Promotions`. La Figure 4.3 représente un

extrait de charge composé de trois requêtes.

```
(1) select sales.time_id, sum(quantity_sold), sum (amount_sold)
    from sales, times
    where sales.time_id = times.time_id
    and times.fiscal_year = '2000'
    group by sales.time_id ;
(2) select sales.prod_id, avg(amount_sold)
    from sales, products, promotions
    where sales.prod_id = products.prod_id
    and sales.promo_id = promotions.promo_id
    and promotions.promo_category = 'newspaper'
    group by sales.prod_id ;
(3) select sales.cust_id, avg(amount_sold)
    from sales, customers, products, times
    where sales.cust_id = customers.cust_id
    and sales.prod_id = products.prod_id
    and sales.time_id = times.time_id
    and times.fiscal_year = '2000'
    and customers.cust_marital_status ='single'
    and products.prod_category ='Women'
    group by sales.cust_id ;
...
```

FIG. 4.3 – Extrait de charge

La matrice "requêtes-attributs" obtenue après l'analyse syntaxique de la charge est composée de dix colonnes et de trois lignes (Figure 4.4). Elle est subdivisée suivant les tables utilisées dans la charge pour des raisons de clarté et de lisibilité. Cette matrice, exploitée par l'algorithme Close, donne lieu à un ensemble de motifs fréquents fermés. Nous détaillons dans la Section 4.5.1, après la présentation des modèles de coût à la Section 4.4, comment sont générés les index *bitmap* de jointure candidats à partir de ces motifs.

4.4 Modèles de coût

Généralement, le nombre d'index candidats est d'autant plus important que la charge en entrée est volumineuse. La création de tous ces index peut ne pas être réalisable en pratique à cause des contraintes systèmes telles que le nombre limité d'index par table ou la taille de l'espace de stockage alloué aux index. Pour pallier ces limitations, nous proposons des modèles de coût permettant de ne conserver que les index les plus avantageux. Ces modèles estiment l'espace en octets occupé par les index *bitmap* de jointure, les coûts

	Customers		Promotions		Products	
	cust_id	marital_status	promo_id	promo_category	prod_id	prod_category
(1)	0	0	0	0	0	0
(2)	0	0	1	1	1	0
(3)	1	1	0	0	1	1

	Sales				Times	
	prod_id	cust_id	promo_id	time_id	time_id	fiscal_year
(1)	0	0	0	1	1	1
(2)	1	0	1	0	0	0
(3)	1	1	0	1	1	1

FIG. 4.4 – Matrice requêtes-attributs

d'accès aux données à travers ces index et les coûts de maintenance de ces index en terme de nombre d'entrées/sorties. Le Tableau 4.2 résume les notations adoptées dans l'élaboration des modèles.

Symbole	**Description**
$\mid X \mid$	Nombre de n-uplets de la table X ou cardinalité de l'attribut X
S_p	Taille en octets d'une page disque
p_X	Nombre de pages nécessaires pour stocker la table X
$S_{pointeur}$	Taille en octets du pointeur d'une page
m	Ordre d'un B-arbre
d	Nombre de *bitmaps* utilisés pour évaluer une requête donnée
$w(X)$	Taille en octets d'un n-uplet de la table X ou de l'attribut X

TAB. 4.2 – Paramètres des modèles de coût

4.4.1 Taille d'un index *bitmap* de jointure

L'espace requis pour stocker un index *bitmap* simple dépend linéairement de la cardinalité de l'attribut indexé et du nombre de n-uplets de la table à laquelle il appartient. L'espace de stockage d'un index *bitmap* construit sur un attribut A de cardinalité $|A|$ appartenant à une table T composée de $|T|$ n-uplets est égal à $\frac{|A||T|}{8}$ octets [WB98, Wu99].

Les index *bitmap* de jointure sont construits sur des attributs de tables dimensions et

chaque *bitmap* contient autant de bits que de n-uplets de la table de faits F. La taille de l'espace de stockage requis est donc $S = \frac{|A||F|}{8}$ octets.

Le temps de construction d'un index *bitmap* dépend à la fois de la cardinalité de l'attribut sur lequel il est construit et le nombre de n-uplets de la table. La complexité est donc en $O(|A||F|)$ [Wu99].

4.4.2 Coût de maintenance d'un index *bitmap* de jointure

Les opérations de mise à jour ont un impact direct sur la taille des index *bitmap* de jointure, notamment lorsque leur nombre est élevé. Ces opérations peuvent être réalisées au niveau de la table de faits ou des tables dimensions. La variation de la taille et les coûts de ces opérations sont présentés dans les sections suivantes.

4.4.2.1 Variation de la taille d'un index *bitmap* de jointure

Les mises à jour des données entraînent systématiquement celles des index. Cela peut engendrer des variations dans l'espace de stockage. Une mise à jour peut être avec ou sans expansion du domaine de l'attribut indexé. On parle d'expansion quand la mise à jour provoque l'ajout d'une nouvelle valeur au domaine de cet attribut. Dans le cas contraire, la mise à jour est sans expansion. Par exemple, si le domaine de l'attribut `type` de la table `Products` est $\{A, B, C\}$ alors la commande SQL `insert into Products (type) values ('K')` provoque l'expansion du domaine de cet attribut.

- **Mise à jour sans expansion** : Dans ce cas, un nouveau bit correspondant au n-uplet inséré est ajouté à chaque *bitmap* déjà créé. La valeur du bit inséré est à 1 dans le *bitmap* correspondant à la valeur insérée et elle est à 0 dans les *bitmaps* restants. La variation de l'espace de stockage est $\Delta S = \frac{|A|(|F|+1)}{8} - \frac{|A||F|}{8} = \frac{|A|}{8}$ et la complexité est en $O(|A|)$.
- **Mise à jour avec expansion** : Le domaine de l'attribut indexé est étendu avec la nouvelle valeur insérée. Un nouveau *bitmap* correspondant à cette valeur est alors créé. La variation de l'espace de stockage est $\Delta S = \frac{(|A|+1)(|F|+1)}{8} - \frac{|A||F|}{8} = \frac{|A|}{8} + \frac{|F|+1}{8}$ et la complexité est en $O(|A|) + O(|F|)$.

4.4.2.2 Coût d'insertion dans la table de faits

Soit un index *bitmap* de jointure construit sur l'attribut A de la table dimension T. Lors d'une insertion dans la table de faits, il faut tout d'abord trouver le n-uplet de la table T pouvant être joint avec celui inséré dans la table de faits F. Au pire, toute la table T est parcourue (p_T pages sont lues). Il faut ensuite mettre à jour les *bitmaps* de l'index. Au pire, tous les *bitmaps* sont parcourus : $\frac{|A||F|}{8S_p}$ pages sont lues, où S_p dénote la taille d'une page disque. Le coût de maintenance de l'index *bitmap* de jointure est donc $C_{maintenance} = p_T + \frac{|A||F|}{8S_p}$.

4.4.2.3 Coût d'insertion dans les tables dimensions

Une insertion dans la table dimension T peut provoquer ou non une expansion du domaine de l'attribut indexé A. En cas de non expansion, la table de faits est parcourue pour rechercher les n-uplets pouvant être joints avec celui qui est inséré dans la table T. Ce parcours nécessite la lecture de p_F pages. Il faut ensuite mettre à jour les *bitmaps* de l'index avec un coût égal à $\frac{|A||F|}{8S_p}$. En cas d'expansion, il faut ajouter le coût de création du nouveau *bitmap* ($\frac{|F|}{8S_p}$ pages). Le coût de maintenance est donc $C_{maintenance} = p_F + (1 + \xi)\frac{|A||F|}{8S_p}$, où ξ est égal à 1 s'il y a une expansion et à 0 dans le cas contraire.

4.4.3 Coût d'accès aux données

Nous proposons deux modèles de coût pour estimer le nombre d'entrées/sorties nécessaires pour accéder aux données. Dans le premier modèle, nous ne prenons aucune hypothèse sur la façon dont sont implémentés physiquement les index *bitmap* de jointure. Dans le deuxième modèle, nous supposons que l'accès aux *bitmaps* de l'index se fait à travers un B-arbre comme c'est le cas dans le SGBD Oracle, par exemple. Nous détaillons chacun de ces modèles dans les sections suivantes.

4.4.3.1 Accès direct aux *bitmaps*

Deux phases sont nécessaires pour évaluer une requête exploitant un index *bitmap* : la phase de parcours des *bitmaps* de l'index et la phase de lecture des n-uplets de la table

indexée.

La phase de parcours des *bitmaps* de l'index correspond à toutes les opérations d'entrées/sorties nécessaires pour rechercher les *bitmaps* permettant d'évaluer une requête donnée. La phase de lecture des n-uplets de la table indexée inclut des opérations d'entrées/sorties additionnelles permettant de lire directement les données à partir du disque. Nous supposons que les données sont uniformément distribuées. Cette hypothèse est raisonnable et souvent adoptée dans l'élaboration des modèles de coût [CBC93a].

Nombre d'entrées/sorties pendant la phase de parcours des *bitmaps*

Au pire, tous les *bitmaps* doivent être parcourus pour rechercher le *bitmap* correspondant à une valeur de l'attribut indexé. Dans les SGBD, les opérations d'entrées/sorties portent sur une page de données plutôt que sur un n-uplet. Cela signifie que lorsqu'un n-uplet d'une page est accédé, toute cette page est lue. Si S_p est la taille d'une page disque, alors le nombre de pages parcourues pour lire un seul *bitmap* est $\frac{|F|}{8S_p}$.

Le coût en terme d'entrées/sorties nécessaires pour rechercher un seul *bitmap* est $\frac{|A||F|}{8S_p}$. Lorsque la lecture de d *bitmaps* est nécessaire, alors le coût de la phase de parcours des *bitmaps* de l'index est $C_{parcours} = d\frac{|A||F|}{8S_p}$. La valeur de d est égale au nombre de prédicats appliqués sur l'attribut indexé et liés par l'opérateur `or` ou la cardinalité de la liste d'une clause `in`. Par exemple, la valeur de d de l'attribut indexé A est égale à 2 dans les deux clauses suivantes : `A=5 or A=10`, `A in (5,10)`.

Nombre d'entrées/sorties pendant la phase de lecture des n-uplets

Pour un index *bitmap* construit sur l'attribut A, le nombre de n-uplets lus est égal à $\frac{|F|}{|A|}$ (nous supposons à nouveau les données uniformément distribuées).

De façon plus générale, le nombre total de n-uplets lus pour une requête utilisant d *bitmaps* peut être donné par $N_r = d\frac{|F|}{|A|}$. Étant donné le nombre de n-uplets lus, défini par la formule précédente, le nombre d'entrées/sorties de la phase de lecture est $C_{lecture} = p_F(1 - e^{-\frac{N_r}{p_F}})$ [OQ97], où p_F désigne le nombre de pages nécessaires pour stocker la table de faits.

Nombre total d'entrées/sorties

Le nombre total d'entrées/sorties est la somme du nombre d'entrées/sorties calculé dans la phase de parcours des *bitmaps* et du nombre d'entrées/sorties dans la phase de lecture des n-uplets, c'est-à-dire, $C_{index} = d\frac{|A||F|}{8S_p} + p_F(1 - e^{-\frac{N_r}{p_F}})$.

Dans cette formule, nous constatons que le coût de la phase de parcours des *bitmaps* (opérations sur les index *bitmaps*) est élevé quand la cardinalité des attributs indexés est grande. D'autre part, le coût de la phase de lecture diminue lorsque la cardinalité augmente.

4.4.3.2 Accès aux *bitmaps* par B-arbre

Dans ce modèle, nous supposons que l'accès aux index *bitmaps* est réalisé à travers un B-arbre (métaindexation) dont les nœuds feuilles pointent vers les *bitmaps* (Figure 4.5). Le coût d'utilisation des index *bitmap* de jointure en terme d'entrées/sorties pour évaluer une requête d'interrogation peut être écrit comme suit : $C = C_{descente} + C_{parcours} + C_{lecture}$, où $C_{descente}$ désigne le coût de descente du B-arbre de la racine jusqu'aux nœuds feuilles, $C_{parcours}$ dénote le coût de parcours des nœuds feuilles afin de trouver les clés de recherche correspondantes et le coût de lecture des *bitmaps* associés, et enfin $C_{lecture}$ donne le coût de lecture des n-uplets de la table indexée.

Le coût de descente du B-arbre dépend de sa hauteur. La hauteur d'un B-arbre construit sur un attribut A est $log_m|A|$, où m désigne l'ordre du B-arbre. Cet ordre est égal à $K + 1$ où K représente le nombre de clés de recherche dans chaque nœud du B-arbre. Le nombre K est égal à $\frac{S_p}{w(A)+S_{pointeur}}$, où $w(A)$ et $S_{pointeur}$ sont, respectivement, la taille en octets de l'attribut A et du pointeur d'une page. Si nous n'ajoutons pas le niveau des nœuds feuilles du B-arbre, alors le coût de descente du B-arbre est $C_{descente} = log_m|A| - 1$.

Le coût de parcours des nœuds feuilles est $\frac{|A|}{m-1}$ (au pire, tous les nœuds feuilles sont lus). L'accès aux données est réalisé via les bits mis à 1 de chaque *bitmap*. Dans ce cas, il faut lire chaque *bitmap*. Le coût de lecture de d *bitmaps* est $d\frac{|F|}{8S_p}$. Ainsi, le coût de parcours des nœuds feuilles est $C_{parcours} = \frac{|A|}{m-1} + d\frac{|F|}{8S_p}$.

Le coût de lecture des n-uplets de la table indexée est calculé de la même façon que dans la phase de lecture des n-uplets du modèle de coût de la Section 4.4.3.1.

En résumé, le coût d'évaluation d'une requête exploitant un index *bitmap* de jointure est

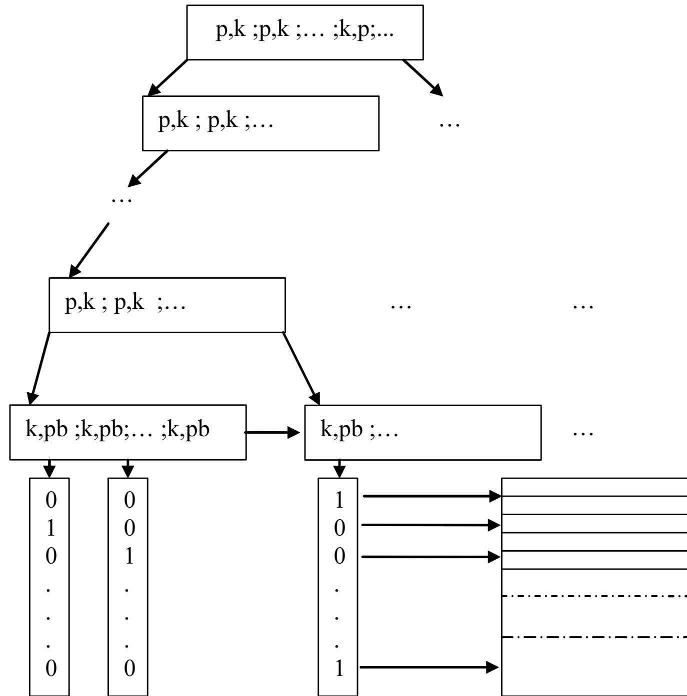

FIG. 4.5 – Index *bitmap* de jointure indexé par un B-arbre

$C_{index} = log_m|A| - 1 + \frac{|A|}{m-1} + d\frac{|F|}{8S_p} + p_F(1 - e^{-\frac{N_r}{p_F}})$.

4.4.4 Coût de jointure

Dans les cas où les index *bitmap* de jointure ne sont pas utilisés pour évaluer une requête, nous supposons que les jointures sont réalisées par hachage. Le nombre d'entrées/sorties nécessaires pour joindre les tables R et S est alors $C_{hachage} = 3\ (p_S + p_R)$ [ME92].

4.5 Stratégie de sélection d'index *bitmap* de jointure

Notre stratégie de sélection d'index procède en plusieurs étapes. Dans un premier temps, l'ensemble des index candidats est construit à partir de l'ensemble des motifs fréquents obtenus à partir de la charge (Section 4.3). Un algorithme glouton exploite ensuite une fonction objectif basée sur les modèles de coût présentés à la Section 4.4 afin d'élaguer les index les moins avantageux. Le détail de ces étapes et de la construction de la fonction

objectif est donné dans les sections suivantes.

4.5.1 Construction de l'ensemble d'index candidats

À partir de l'ensemble de motifs fréquents (Section 4.3) et du schéma de l'entrepôt de données (clés étrangères de la table de faits, clés primaires des tables dimensions, etc.), nous construisons un ensemble d'index candidats.

L'instruction de construction d'un index *bitmap* de jointure est composée de trois clauses : On, From et Where. La clause On est composée des attributs sur lesquels est construit l'index (attributs non clés des tables dimensions), la clause From contient les tables jointes et la clause Where est constituée des prédicats de jointure. La Figure 4.6 montre un exemple d'une commande SQL permettant de construire un index *bitmap* de jointure sur l'attribut Fiscal_Year de la table dimension Times (Sales est la table de faits), sous Oracle 9i.

```
create bitmap index BIJ_TIMES
on sales (times.fiscal_year)
from sales,times
where sales.time_id=times.time_id
```

Fig. 4.6 – Commande SQL de construction d'un index *bitmap* de jointure

Nous considérons un motif fréquent $< Table.attribut_1, ..., Table.attribut_n >$ comme une suite finie d'éléments de la forme $Table.attribut$. Chaque motif fréquent est analysé pour déterminer les différentes clauses d'un index *bitmap* de jointure. Tout d'abord, nous extrayons les éléments contenant les clés étrangères de la table de faits. Les motifs ne contenant pas de tels éléments ne donnent pas d'index *bitmap* de jointure car ces éléments sont nécessaires pour définir les clauses From et Where de l'index. Nous recherchons ensuite, dans le motif fréquent, les éléments contenant les clés primaires des tables dimensions afin de construire la clause Where. Les tables trouvées au cours de cette analyse constituent la clause From. Les éléments contenant des attributs non clés des tables dimensions forment la clause On. Si de tels éléments n'existent pas, l'index *bitmap* de jointure ne peut pas être construit car la clause On est formée des attributs non clés des tables dimensions. Par exemple, le motif fréquent <Times.Time_id,Sales.Time_id,Times.Fiscal_Year> donne l'index *bitmap* de

jointure BIJ_TIMES dont la commande SQL est présentée dans la Figure 4.6.

4.5.2 Fonctions objectifs

Dans cette section, nous décrivons trois fonctions objectifs évaluant la réduction ou l'augmentation du coût d'exécution, en terme d'entrées/sorties, des requêtes de la charge lorsqu'un nouvel index est créé. Le coût d'exécution d'une requête est assimilé au coût de calcul des jointures par hachage si aucun index *bitmap* de jointure n'est exploité ou au coût d'accès aux données à travers cet index dans le cas contraire. Le coût d'exécution des requêtes de la charge est la somme de tous les coûts d'exécution de ses requêtes.

La première fonction objectif privilégie les index apportant le plus de profit lors de l'exécution des requêtes, la deuxième fonction privilégie les index apportant le plus de profit tout en occupant un minimum d'espace de stockage et la troisième fonction combine les deux premières afin de sélectionner dans un premier temps les index apportant le plus de profit et de ne conserver dans un deuxième temps que ceux qui occupent le moins d'espace de stockage lorsque celui-ci devient faible. La première fonction est utilisable quand l'espace de stockage n'est pas limité, la deuxième fonction est utile quand l'espace de stockage est faible et la troisième est intéressante quand cet espace est moyennement grand. Nous détaillons dans la suite chacune de ces fonctions.

4.5.2.1 Fonction objectif profit

Soient $I = \{i_1, ..., i_n\}$ un ensemble d'index *bitmap* de jointure candidats, $Q = \{q_1, ..., q_m\}$ un ensemble de requêtes (la charge) et $Config_I$ l'ensemble d'index final à créer.

La fonction objectif profit, notée P, est définie comme suit :

$$P_{/Config_I}(i_j) = \lambda \left(C(Q, Config_I) - C(Q, Config_I \cup \{i_j\}) - \beta\, C_{maintenance}(\{i_j\}) \right), i_j \notin Config_I.$$

- Le coefficient λ estime le rapport entre le coût des jointures évitées grâce à l'index i_j lors de l'exécution des requêtes de la charge exploitant cet index et le coût total des jointures de toutes les requêtes de cette charge. Plus la valeur de λ est grande, meilleur est l'index. En effet, un index évitant le calcul d'un grand nombre de jointures est avantageux. Le coefficient λ est calculé comme suit : $\lambda = |Q| support(i_j) \frac{hachage(tables(i_j))}{\sum_{i=1}^{|Q|} hachage(tables(q_i))}$,

où $|Q|$, $support(i_j)$, $hachage(tables(i_j))$ et $\sum_{i=1}^{|Q|} hachage(tables(q_i))$ sont respectivement le nombre de requêtes de la charge, le support du motif fréquent générateur de l'index i_j, le coût de jointure par hachage des tables pré-jointes dans i_j et le coût total des jointures par hachage des tables jointes dans les requêtes de la charge.

- $C(Q, Config_I)$ représente le coût d'exécution des requêtes de la charge lorsque l'ensemble d'index $Config_I$ est utilisé. Si cet ensemble est vide, $C(\emptyset, Q)$ est égal à la somme des coûts de calcul des jointures par hachage des tables de chaque requête. Lorsqu'un index i_j est ajouté à $Config_I$, $C(Q, Config_I \cup \{i_j\}) = \sum_{k=0}^{|Q|} C(q_k, \{i_j\})$ représente le coût d'exécution des requêtes de la charge en considérant les index de l'ensemble $Config_I \cup \{i_j\}$. Si la requête q_k exploite l'index i_j alors le coût $C(q_k, \{i_j\})$ est égal à C_{i_j} (coût d'accès aux données à travers cet index). Dans le cas contraire, $C(q_k, \{i_j\})$ est égal à la valeur minimum entre $C_{hachage}$ (coût de calcul des jointures par hachage des tables jointes par la requête q_k) et les valeurs de $C(q_k, \{i\})$ (coût d'exécution des requêtes q_k exploitant $i \in S$ avec $i \neq i_j$).
- Le coefficient $\beta = |Q|\, p(i_j)$ permet d'estimer le nombre de fois où l'index i_j est mis à jour. La probabilité de mise à jour $p(i_j)$ de l'index i_j est égale à $\frac{1}{nombre\ d'index} \frac{\%\text{rafraîchissement}}{\%interrogation}$, où le rapport $\frac{\%\text{rafraîchissement}}{\%interrogation}$ représente la proportion de mises à jour de l'entrepôt de données par rapport aux interrogations.
- $C_{maintenance}(\{i_j\})$ représente le coût de maintenance de l'index i_j.

4.5.2.2 Fonction objectif ratio profit/espace

Si la sélection d'index est réalisée sous la contrainte de l'espace de stockage alloué aux index, la fonction objectif ratio profit/espace $R_{/Config_I}(i_j) = \frac{P_{/Config_I}(i_j)}{taille(i_j)}$ est utilisée. Cette fonction calcule le profit apporté par l'index i_j par rapport à l'espace de stockage $taille(i_j)$ qu'il occupe. La fonction ratio profit/espace privilégie les index accélérant le mieux l'accès aux données tout en occupant le plus petit d'espace possible.

4.5.2.3 Fonction objectif hybride

La contrainte sur l'espace de stockage peut être relaxée si l'espace alloué aux index est relativement élevé. La fonction objectif hybride H ne pénalise les index "gourmands" en

espace que si le rapport $\frac{espace_restant}{espace_stockage}$ est supérieur à un seuil α donné ($0 < \alpha \leq 1$), où *espace_restant* et *espace_stockage* sont respectivement l'espace restant après l'inclusion de l'index i_j et l'espace réservé pour stocker tous les index. Cette fonction est calculée en combinant les fonctions P et R comme suit :

$$H_{/Config_I}(i_j) = \begin{cases} P_{/Config_I}(i_j) & \text{si } \frac{espace_restant}{espace_stockage} > \alpha, \\ R_{/Config_I}(i_j) & \text{sinon.} \end{cases}$$

L'intérêt de cette fonction est de prendre dans un premier temps les index améliorant le plus le temps d'exécution des requêtes sans prendre en compte la contrainte sur l'espace de stockage. Lorsque le rapport entre l'espace restant et l'espace de stockage atteint le seuil critique indiqué par la valeur de α, sont pénalisés les index occupant beaucoup d'espace disque. Cela est intéressant lorsque l'administrateur dispose d'un espace moyennement grand pour stocker les index et qu'il existe plusieurs index qui apportent une amélioration importante au temps d'exécution des requêtes et occupent beaucoup d'espace.

4.5.3 Construction de la configuration d'index

Notre algorithme de sélection d'index (Algorithme **??**) se base sur une recherche gloutonne dans l'ensemble des index candidats I donné en entrée. La fonction objectif F doit être l'une des fonctions P, R ou H décrites dans la section précédente. Si la fonction R est utilisée, nous ajoutons en entrée de l'algorithme la taille de l'espace de stockage M réservé aux index. Si la fonction H est utilisée, nous y ajoutons le paramètre α.

À la première itération de l'algorithme, les valeurs de la fonction objectif sont calculées pour chaque index de l'ensemble I. Le coût d'exécution des requêtes de la charge Q (premier terme de la fonction F) est égal au coût total de calcul des jointures par hachage des tables jointes dans chaque requête. L'index i_{max} maximisant la fonction objectif, s'il existe ($F_{/S}(i_{max}) > 0$), est ensuite ajouté à l'ensemble S. Si l'une des fonctions R ou H est utilisée, l'espace de stockage S est diminué de l'espace occupé par i_{max}.

Les valeurs de la fonction objectif F sont ensuite recalculées pour chaque index restant dans $I - S$ puisqu'elles dépendent de l'ensemble d'index sélectionnés $Config_I$. Cela permet de prendre en compte les interactions qui peuvent exister entre les index.

Ces itérations sont répétées jusqu'à ce qu'il n'y ait plus d'amélioration de la fonction

Algorithme 9 Construction de la configuration d'index

$Config_I \leftarrow \emptyset$
répéter
$i_{max} \leftarrow \emptyset$
$F_{max} \leftarrow 0$
pour tout $i_j \subseteq I - Config_I$ **faire**
si $F_{/Config_I}(i_j) > F_{max}$ **alors**
$F_{max} \leftarrow F_{/Config_I}(i_j)$
$i_{max} \leftarrow i_j$
fin si
fin pour
si $F_{/Config_I}(i_{max}) > 0$ **alors**
$Config_I \leftarrow Config_I \cup \{i_{max}\}$
fin si
jusqu'à $(F_{/Config_I}(i_{max}) \leq 0$ ou $I - Config_I = \emptyset)$

objectif ($F_{/Config_I}(i) \leq 0$) ou que tous les index aient été sélectionnés ($I - Config_I = \emptyset$). Si l'une des fonctions R ou H est utilisée, l'algorithme s'arrête également lorsque l'espace de stockage disponible est saturé.

4.6 Expérimentations

Afin de valider notre stratégie de sélection d'index *bitmap* de jointure, nous l'avons appliquée sur un entrepôt de données test implanté au sein du SGBD Oracle 9i installé sur un PC sous Windows XP Pro doté d'un processeur Pentium 4 à 2.4 GHz, d'une mémoire centrale de 512 Mo et d'un disque dur IDE de 120 Go. Cet entrepôt de données est composé d'une table de faits `Sales` et de cinq tables dimensions `Customers`, `Products`, `Promotions`, `Times` et `Channels`. Le Tableau 4.3 détaille le nombre de n-uplets et la taille en Mo de chaque table de cet entrepôt. Nous avons mesuré pour différentes valeurs du support minimal de Close le temps d'exécution des requêtes de la charge. En pratique, ce paramètre permet de limiter le nombre d'index candidats à générer en ne sélectionnant que ceux qui sont les plus fréquemment sollicités par la charge. Le pseudo code du protocole d'expérimentation est détaillé à l'algorithme 10.

Pour calculer les différents coûts, nous avons fixé respectivement les valeurs des paramètres S_p (taille d'une page disque) et de $S_{pointeur}$ (taille du pointeur d'une page) à 8 Ko

Table	**Nombre de n-uplets**	**Taille (Mo)**
Sales	16 260 336	372,17
Customers	50 000	6,67
Products	10 000	2,28
Times	1 461	0,20
Promotions	501	0,04
Channels	5	0,000 1

TAB. 4.3 – Caractéristiques de l'entrepôt de données test

Algorithme 10 Protocole des expérimentations

{Exécution à froid (sans chronométrage)}
pour tout requête de la charge **faire**
  Exécuter la requête courante
fin pour
{Exécution à chaud}
pour $i \leftarrow 1$ à nombre_de_répétitions **faire**
  pour tout requête de la charge **faire**
    Exécuter la requête courante
    Calculer le temps d'exécution de la requête courante
  fin pour
fin pour
Calculer le temps d'exécution moyen global

et 4 Ko. Ces valeurs sont celles indiquées dans le fichier de configuration d'Oracle. Nous n'avons appliqué dans ces expérimentations que le modèle d'accès aux *bitmaps* par B-arbre (Section 4.4.3.2) car c'est la méthode utilisée dans Oracle 9i. La charge est composée de quarante requêtes décisionnelles comportant plusieurs jointures (cf. Annexe A). Nous avons mesuré le temps total d'exécution des requêtes de cette charge avec et sans index et avec ou sans application des modèles de coût exploités par les trois fonctions objectifs : profit, ratio profit/espace et hybride. Nous avons également mesuré l'espace disque occupé par les index.

4.6.1 Expérimentations avec la fonction profit

La Figure 4.7 montre que les index sélectionnés améliorent le temps d'exécution des requêtes de la charge, avec ou sans application des modèles de coût, jusqu'à ce que le support

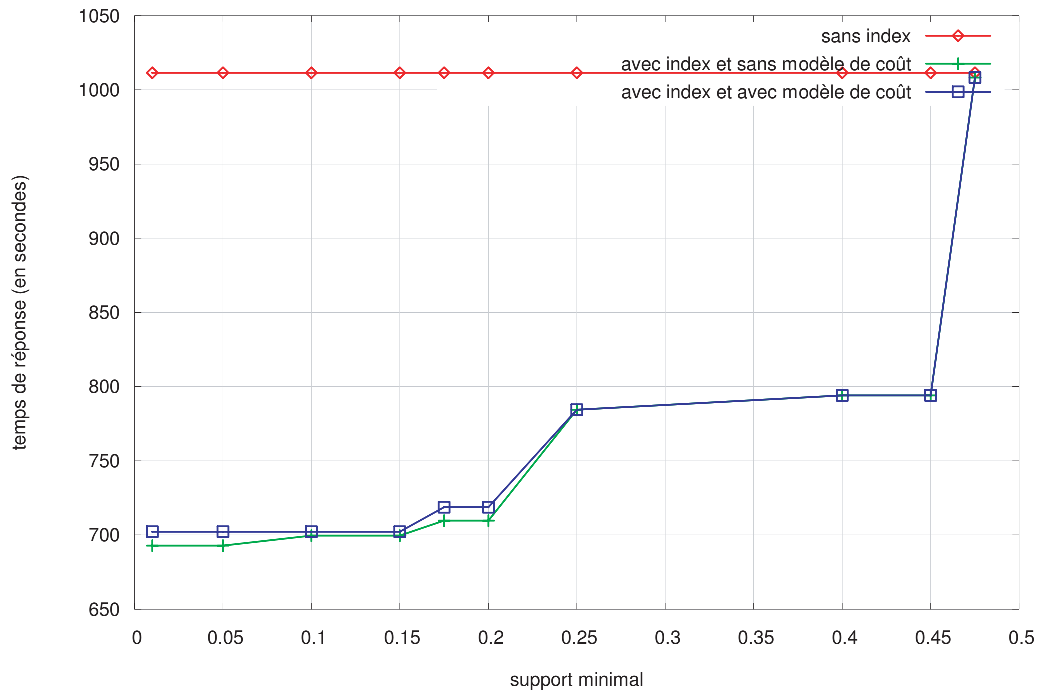

FIG. 4.7 – Fonction profit

minimal dépasse 47,5%. De plus, ce temps se dégrade de manière continue au fur et à mesure que le support minimal augmente. Cela est dû au fait que le nombre d'index diminue quand le support minimal augmente. Pour les grandes valeurs du support (plus de 47,5%), le temps d'exécution est proche de celui obtenu sans index. Ce cas est prévisible, dans le sens où, pour un support minimal très grand, aucun index ou très peu d'index candidats sont générés.

Le gain maximal en temps total d'exécution de la charge dans les deux cas est respectivement de 30,50% et 31,85%. Malgré une légère baisse de 1,32% du gain en temps d'exécution lorsque les modèles de coût sont utilisés (nombre plus réduit d'index construits), nous constatons en contre partie un gain significatif en terme d'espace de stockage (égal à 32,79% dans le cas le plus favorable) comme le montre la Figure 4.8. La réduction du nombre d'index est intéressante lorsque la fréquence de rafraîchissement de l'entrepôt de données est élevée car le coût de rafraîchissement est proportionnel au nombre d'index construits. D'autre part, le gain en espace de stockage permet à l'administrateur de limiter l'espace disque alloué aux index.

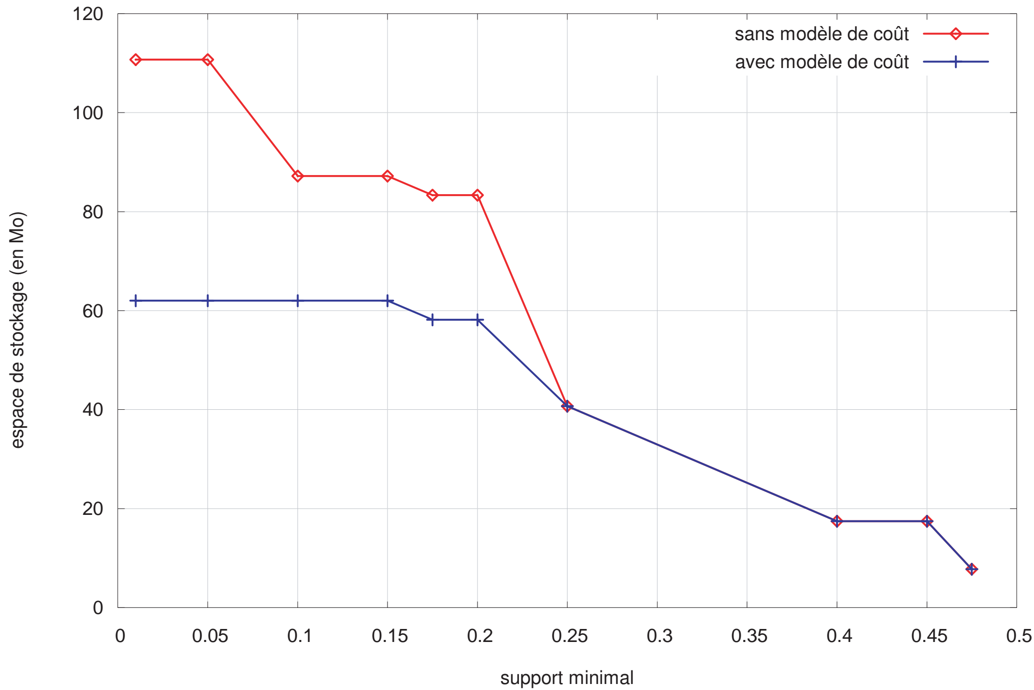

FIG. 4.8 – Espace de stockage des index

4.6.2 Expérimentations avec la fonction ratio profit/espace

Dans ces expérimentations, nous avons fixé la valeur du support minimal à 1%. Cette valeur donne un grand nombre de motifs fréquents et par conséquence un grand nombre d'index *bitmap* de jointure candidats. Cela permet de faire varier l'espace de stockage dans un plus grand intervalle. Nous avons mesuré le temps d'exécution des requêtes en fonction du pourcentage de l'espace de stockage alloué aux index. Ce pourcentage est calculé par rapport à l'espace total occupé par tous les index.

La Figure 4.9 montre que le temps d'exécution diminue quand l'occupation de l'espace de stockage augmente. Cela est prévisible, dans le sens où on peut créer un plus grand nombre d'index et donc améliorer davantage le temps d'exécution. Nous constatons aussi que le gain maximal en temps d'exécution égal à 28,95% est atteint pour une occupation de l'espace de stockage de 59,64%. Cela signifie qu'en fixant l'espace de stockage à cette valeur et en appliquant la fonction objectif ratio profit/espace, nous obtenons des gains en temps d'exécution proches de ceux obtenus en appliquant la fonction objectif profit (30,50%). Ce

cas est intéressant quand l'administrateur ne dispose pas de beaucoup d'espace pour stocker les index.

4.6.3 Expérimentations avec la fonction hybride

Nous avons reproduit les expérimentations précédentes avec la fonction objectif hybride. Nous avons fait varier les valeurs du paramètre α entre 0,1 et 1 par pas de 0,1. Les résultats obtenus dans ces expérimentations avec α allant de 0,1 à 0,7 et α allant de 0,8 à 1 sont respectivement égaux à ceux obtenus avec $\alpha = 0,1$ et $\alpha = 0,8$. Nous ne représentons donc à la Figure 4.10 que les résultats obtenus avec $\alpha = 0,1$ et $\alpha = 0,8$. Cette figure montre que pour $\alpha = 0,1$, on se rapproche des résultats obtenus en utilisant la fonction objectif ratio espace/profit, et pour $\alpha = 0,8$, on se rapproche des résultats obtenus en utilisant la fonction objectif profit. Le gain maximal en temps d'exécution est égal à 28,95% et 29,85% pour les valeurs 0,1 et 0,8 de α.

Nous expliquons ces résultats par le fait que les index *bitmap* de jointure construits sur plusieurs attributs nécessitent un espace de stockage important. En revanche, comme ils précalculent un plus grand nombre de jointures, ils améliorent davantage le temps d'exécution de ces dernières. L'espace alloué aux index se remplit donc très vite au bout de quelques itérations de l'algorithme glouton de sélection d'index. Ceci explique pourquoi le paramètre α n'influe pas significativement sur l'algorithme de sélection et donc sur les résultats de ces expérimentations.

Nous notons également que la majorité des données nécessaires pour répondre aux requêtes de la charge sont principalement retrouvées via un accès à partir de nos index au lieu d'un accès séquentiel aux données sources de l'entrepôt de données. Dans la Figure 4.11, nous représentons le taux d'exploitation des index par les requêtes de la charge en fonction du pourcentage de l'espace de stockage alloué à ces index et de la nature de la fonction objectif utilisée. Nous entendons taux d'exploitation le rapport entre le nombre de requêtes résolues en utilisant les index sélectionnés par notre stratégie et le nombre total de requêtes de la charge. Le pourcentage de l'espace de stockage est calculé par rapport à l'espace occupé par tous les index obtenus en appliquant la fonction profit. Rappelons que cette fonction donne le plus grand nombre d'index car elle privilégie les index améliorant le mieux le temps d'exécution sans tenir compte de l'espace de stockage.

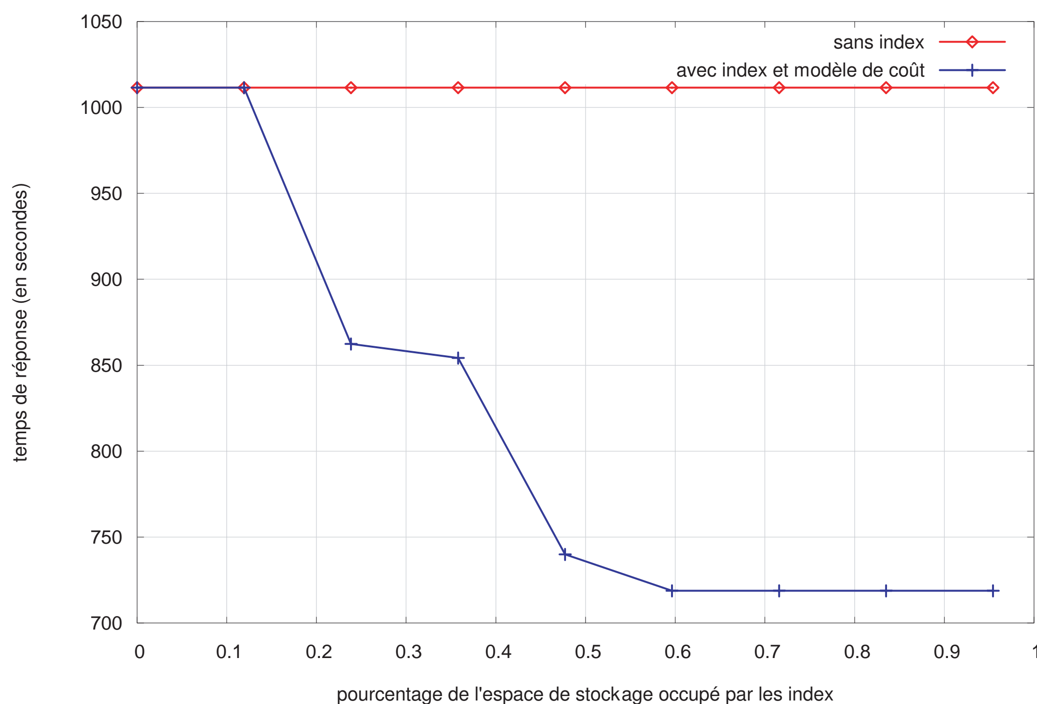

FIG. 4.9 – Résultat avec la fonction ratio profit/espace

Pour une occupation de 23, 85% de l'espace de stockage, 30% pour la (fonction ratio profit/espace) et 47, 5% (pour la fonction hybride) des requêtes sont résolues en exploitant les index sélectionnés. Cela montre que même pour un espace de stockage restreint, notre stratégie de sélection permet de construire des index qui sont pertinents pour les requêtes. Ce taux augmente en fonction de l'espace de stockage. Plus on dispose d'espace, plus on crée d'index et par conséquent plus on améliore le temps d'exécution des requêtes. Cette expérimentation montre que la sélection d'index, orientée analyse syntaxique de la charge, est efficace pour garantir l'exploitation des index par les requêtes de la charge.

4.7 Conclusion et discussion

La sélection automatique d'index est un aspect important dans l'automatisation des tâches d'un système de gestion de bases de données. Dans ce chapitre, nous avons décrit notre démarche et montré comment sélectionner une configuration d'index à partir d'une charge afin d'optimiser les performances. L'originalité de notre stratégie de sélection d'index

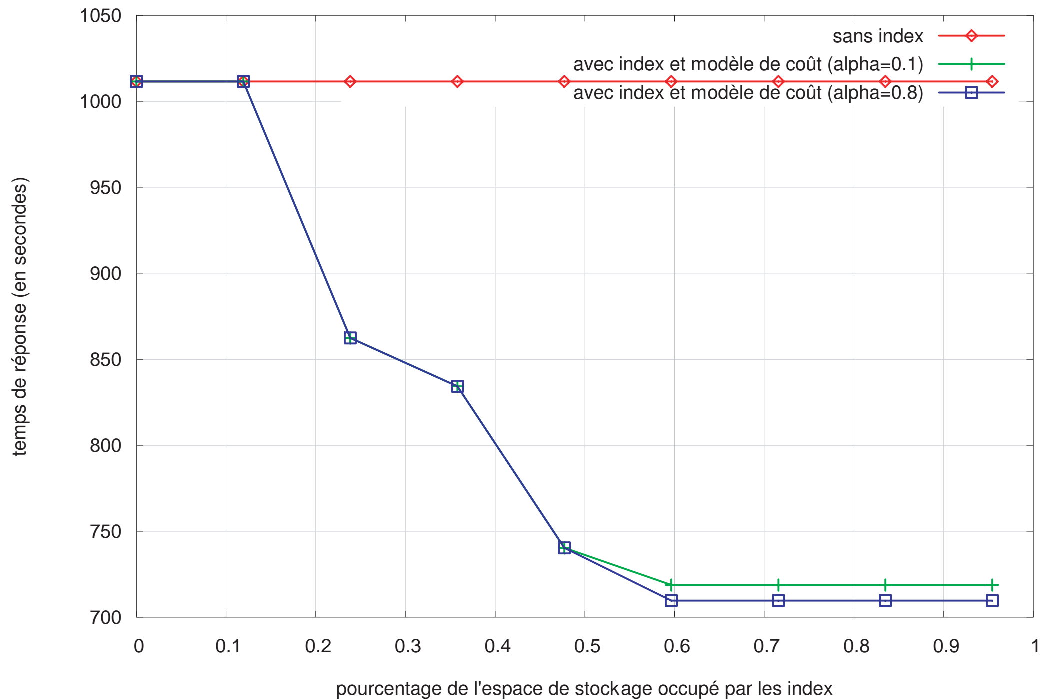

FIG. 4.10 – Résultat avec la fonction hybride

repose sur l'utilisation d'une technique d'extraction de connaissances à partir des données pour déterminer la configuration d'index à créer.

Dans ce chapitre, nous avons présenté une stratégie de sélection automatique d'index *bitmap* de jointure dans les entrepôts de données. Cette stratégie exploite dans un premier temps les motifs fréquents fermés obtenus par l'algorithme Close à partir d'une charge donnée afin de générer un ensemble d'index candidats. Elle s'appuie dans un deuxième temps sur des modèles de coût afin de ne conserver, parmi ces candidats, que ceux qui sont les plus avantageux. Ces modèles estiment le coût d'accès aux données à travers les index *bitmap* de jointure, ainsi que le coût de maintenance et de stockage de ces index. Nous avons également proposé trois fonctions objectifs (profit, ratio profit/espace et hybride) combinant les modèles de coût afin d'évaluer le coût d'exécution des requêtes exploitant ou non les index. Ces fonctions sont utilisées par un algorithme glouton de sélection d'index afin de recommander une configuration d'index pertinente. Cela permet à notre stratégie de s'adapter aux contraintes imposées par le système (nombre d'index limité par table) ou par l'administrateur de l'entrepôt de données (taille de l'espace de stockage alloué aux index).

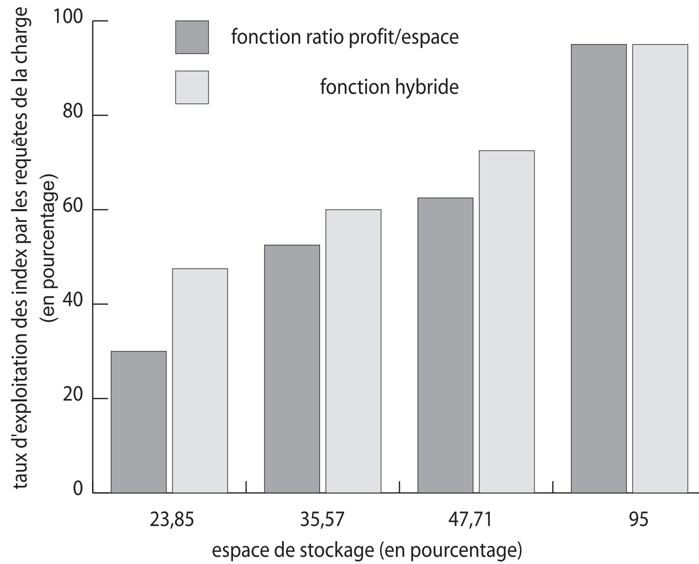

FIG. 4.11 – Taux d'exploitation des index par les requêtes de la charge

Nos résultats expérimentaux montrent que l'application des modèles de coût à notre stratégie de sélection d'index réduit le nombre d'index sélectionnés sans dégrader significativement les performances. Cette réduction garantit en revanche un gain substantiel en terme d'espace de stockage alloué aux index, ainsi qu'une diminution des coûts de maintenance de l'entrepôt de données lors des rafraîchissements. Dans la suite, nous discutons notre approche de sélection d'index et la positionnons par rapport aux travaux de recherche existants dans ce domaine. Notre propos s'articule autour de la méthode de construction de l'ensemble d'index candidats et de leur type (mono ou multi-attributs, index primaire ou secondaire, technique d'indexation), de l'évaluation du coût d'exploitation des index et de l'algorithme de sélection d'index.

La construction de l'ensemble d'index candidats est automatique et se fait en deux étapes. La première étape consiste à analyser syntaxiquement la charge de requêtes de manière similaire à celle proposée dans [CN97]. À l'inverse des approches manuelles [KY87, FON92, CBC93a, CBC93b, Gun99, KLT03], notre approche peut être déployée sur une échelle réelle. De plus, notre analyseur de requêtes exploite certaines connaissances en administration et optimisation des SGBD pour proposer des candidats comme c'est le cas dans [FR03]. Cette connaissance est très utile pour cibler les index candidats pertinents. La deuxième étape de construction des index candidats exploite la recherche des motifs fréquents fermés pour

générer les index candidats. Ces index peuvent être mono ou multi-attributs. L'utilisation de l'algorithme Close permet de générer des index mono-attribut et multi-attributs à la volée et évite de passer par un processus à plusieurs itérations dans lequel sont créés les index mono-attributs à la première itération, les index à deux attributs à la deuxième itération et ainsi de suite pour les index de taille supérieure comme c'est le cas dans [CN97].

La sélection des index primaires et secondaires a été étudiée dans les travaux de Choenni *et al.* [CBC93a, CBC93b]. Cependant, la majorité des SGBD actuels, comme Oracle, impose que les index primaires soient construits sur les clés primaires des tables. C'est pour cette raison que cet aspect n'est pas pris en compte dans notre approche.

Gündmn [Gun99] a introduit l'usage des techniques d'indexation dans le processus de sélection d'index. Cet aspect est très intéressant car la qualité d'un index, en terme d'espace de stockage et d'efficacité, dépend de la technique d'indexation adoptée. La plupart des travaux rapportés au Chapitre 3 ne proposent que les B-arbres comme techniques d'indexation. Ce type d'index est certes largement répandu dans les SGBD actuels, mais il est limité lorsqu'il s'agit d'indexer des données volumineuses et des attributs ayant une cardinalité peu élevée. Or, ce cas de figure est typique dans l'environnement des entrepôts de données qui est le cadre de notre présente étude. C'est pourquoi, nous nous focalisons sur la sélection des index *bitmap* de jointure, plus adaptés et efficaces pour réduire le coût des requêtes décisionnelles. Il est à noter que notre approche peut s'adapter facilement aussi bien à l'environnement des bases de données [ADG03a] qu'à l'environnement des entrepôts de données [ADBB05], comme nous l'avons montré dans [ADB04].

La sélection d'index utilise soit une fonction mathématique [KY87, FON92, CBC93a, CBC93b, Gun99, KLT03, FR03], soit fait appel à l'optimiseur de requêtes du SGBD [ACN00, VZZ$^+$00, GRS02]. Une fonction de coût est basée sur des hypothèses théoriques, par exemple, "la fréquence d'insertion et de suppression des n-uplets d'une table est telle que le nombre total des n-uplets dans cette table reste constant pour deux choix consécutifs d'un ensemble d'index" ou "le nombre d'index n'est pas limité par table", qui ne sont pas toujours réalisables dans la pratique. Par contre, l'optimiseur de requêtes utilise des statistiques et un modèle de coût inhérent à un SGBD donné. La sélection d'index est donc dépendante du SGBD et doit être adaptée pour un autre SGBD. De plus, le coût de communication avec l'optimiseur de requêtes peut être important si le nombre d'index candidats à évaluer est grand.

Finalement, notre approche prend en compte l'interaction qui peut exister entre les index. En effet, le coût d'exploitation d'un index peut varier d'une itération à une autre de l'algorithme de sélection d'index. Par exemple, l'attrait d'un index peut augmenter si un autre index en interaction avec ce dernier a été préalablement sélectionné. Cet aspect n'est pas pris en compte dans les travaux assimilant le problème de sélection d'index au problème du sac à dos [Gun99, FR03] ou utilisant les algorithmes génétiques [KLT03]. La raison est qu'une fois les index "mis" dans le sac ou injecté dans l'algorithme génétique, leur coût n'évolue plus. Cela peut conduire à une solution de moins bonne qualité comparée à celle obtenue en prenant en compte les interactions entre les index.