Auto-administration des entrepôts de données complexes

Kamel Aouiche*
Jérôme Darmont**, Omar Boussaid**, Fadila Bentayeb**

Laboratoire ERIC, Université Lumière Lyon 2
5 avenue Pierre Mendès-France
69676 Bron Cedex
* kaouiche@eric.univ-lyon2.fr
**{jdarmont,boussaid,bentayeb}@univ-lyon2.fr
http://eric.univ-lyon2.fr

Résumé. Les requêtes définies sur les entrepôts de données sont souvent compliquées et utilisent plusieurs opérations de jointure qui sont coûteuses en terme de temps de calcul. Dans le cadre de l'entreposage de données complexes, les adaptations apportées aux schémas classiques d'entrepôts induisent des jointures supplémentaires lors des accès aux données. Ce coût devient encore plus important quand les requêtes opèrent sur de très grands volumes de données. Il est donc primordial de réduire ce temps de calcul. Pour cela, les administrateurs d'entrepôts de données utilisent en général des techniques d'indexation comme les index de jointure en étoile ou les index *bitmap* de jointure. Cela demeure néanmoins complexe et fastidieux.

La solution proposée s'inscrit dans une optique d'auto-administration des entrepôts de données. Dans ce cadre, nous proposons une stratégie de sélection automatique d'index. Pour cela, nous avons recouru à une technique de fouille de données, plus particulièrement la recherche de motifs fréquents, pour déterminer un ensemble d'index candidats à partir d'une charge donnée. Nous proposons ensuite des modèles de coût permettant de sélectionner les index engendrant le meilleur profit. Ces modèles de coût évaluent en particulier le temps d'accès aux données à travers des index *bitmap* de jointure, ainsi que le coût de maintenance et de stockage de ces index.

Mots clés : Entrepôts de données, données complexes, auto-administration, sélection d'index, motifs fréquents, modèles de coût, index *bitmap* de jointure.

1 Introduction

Les entrepôts de données classiques 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 [Inmon, 2002] [Kimball et Ross, 2002]. La table de faits contient les clés (étrangères) des tables dimensions et les mesures. La Figure 1 montre un exemple d'entrepôt de données modélisé en étoile composé de la

table de faits `Sale` et des tables dimensions `Products` et `Customers`.

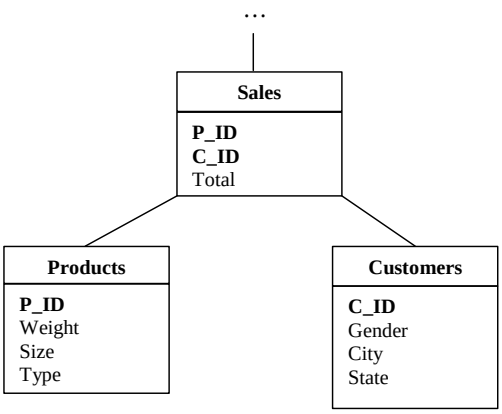

FIG. 1 – *Exemple d'entrepôt de données modélisé en étoile*

Les entrepôts de données classiques permettent d'analyser des activités représentées sous la forme de données numériques. Cependant, les données exploitées dans le cadre des processus décisionnels sont de plus en plus complexes. L'avènement du Web et la profusion de données multimédias ont en grande partie contribué à l'émergence de cette nouvelle sorte de données. Nous considérons que des données sont complexes si elles sont :

- *multiformats* (données numériques, symboliques, textes, images, sons, vidéos...) et/ou
- *multistructures* (bases de données relationnelles, collections de documents XML...) et/ou
- *multisources* (bases de données réparties, Web...) et/ou
- *multimodales* (un même phénomène décrit par plusieurs canaux ou points de vue, comme des radiographies et le diagnostic audio d'un médecin pour évaluer l'état de santé d'un patient, des données exprimées dans des échelles ou des langues différentes...) et/ou
- *multiversions* (bases de données temporelles, recensements périodiques dont les critères évoluent...).

Les données complexes sont représentées par un ensemble de descripteurs de bas niveaux et sémantiques. Leur manipulation s'opère par l'intermédiaire de ces descripteurs. Leur multiplication permet d'obtenir une information de plus en plus complète sur les données complexes elles-même, mais complique également leur exploitation. Le choix des descripteurs les plus pertinents à considérer dans le cadre d'un processus décisionnel devient alors un vrai problème. Les concepts de l'entreposage de

données demeurent cependant valides dans ce contexte. Les mesures, bien que non nécessairement numériques, restent des indicateurs pour l'analyse et cette dernière s'effectue vraisemblablement selon différentes perspectives représentées par des dimensions. La volumétrie des données et leur datation jouent également en faveur d'une approche d'entreposage.

Dans le cadre d'un projet dans le domaine de la médecine du sport, nous avons conçu un entrepôt de données médicales complexes (qualitatives, numériques, textes, images...) concernant un grand ensemble de sportifs de haut niveau. Cet entrepôt, basé sur une modélisation relationnelle des données, a nécessité des adaptations des schémas en étoile classiques incluant notamment un éclatement de la table de faits en plusieurs relations. Or, avec ce type de modèle, une requête décisionnelle nécessite déjà une ou plusieurs jointures entre la table de faits et les tables dimensions. Notre adaptation pour les données complexes accentue donc ce phénomène. De plus, le schéma de l'entrepôt comporte des hiérarchies au niveau des dimensions (schéma en flocon de neige), ce qui entraîne des jointures additionnelles. La multiplication des jointures rend complexe le choix des meilleurs index à construire. Le coût de ces jointures en terme de temps de calcul devient prohibitif et s'amplifie davantage lorsque celles-ci 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, telles la jointure par hachage, par tri-fusion et la jointure imbriquée [Mishra et Eich, 1992]. Cependant, ces techniques ne sont efficaces que lorsque 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 (problème de sélection d'index).

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 Systèmes de Gestion de Bases de Données (SGBD) [Weikum *et al.*, 2002] ou du moins de fournir des outils d'aide pour l'administrateur. Ces fonctionnalités sont particulièrement critiques dans le cas des entrepôts de données complexes, du fait de l'hétérogénéité même des données stockées. 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 ce cadre, nous avons proposé une démarche de sélection automatique d'index dans les bases de données fondée sur l'extraction de motifs fréquents à partir d'une charge [Aouiche *et al.*, 2003a] [Aouiche *et al.*, 2003b], puis quelques pistes pour l'adaptation de cette démarche dans le contexte des entrepôts de données [Aouiche *et al.*, 2004]. Notons que la sélection d'index est néanmoins un processus complexe aussi bien dans le cas où les données sont simples ou complexes.

Dans cet article, nous présentons la poursuite de nos travaux dans cette voie avec pour objectif une application sur notre entrepôt de données médicales complexes. 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 et le coût de leur maintenance et de leur stockage. Un algorithme glouton exploite ces modèles afin de recommander une configuration d'index pertinente.

Par ailleurs, nous avons choisi dans ce travail 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 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 [Sarawagi, 1997] [Wu, 1999]. Ces attributs sont souvent utilisés dans les clauses `Where` et `Group by` des requêtes décisionnelles [Hu *et al.*, 2003].

Cet article est organisé comme suit. Nous présentons tout d'abord un état de l'art concernant les index de jointure et les approches proposées pour la sélection d'index dans les entrepôts de données (Section 2). Nous rappelons le principe de notre démarche de sélection automatique d'index à base d'extraction de motifs fréquents (Section 3). Nous présentons ensuite nos modèles de coût (Section 4) et détaillons notre stratégie de sélection d'index (Section 5). Afin de valider celle-ci, nous l'avons expérimentée sur un entrepôt de données test (Section 6). Nous terminons enfin par une conclusion et des perspectives de recherche (Section 7).

2 État de l'art

2.1 Techniques d'indexation dans les entrepôts de données

Au delà des index structurés en b-arbre et de leurs variantes, il existe d'autres techniques d'indexation plus adaptées à l'environnement des entrepôts de données. Ces approches prennent en compte la volumétrie des données et la complexité des requêtes décisionnelles.

Dans un premier temps, les index de jointure ont été proposés dans les bases de données [Valduriez, 1987]. Ce type d'index, utilisé pour pré-joindre deux tables, a été étendu aux entrepôts de données. En effet, les index de jointure en étoile (*star join index*) peuvent contenir toutes combinaisons de clés étrangères de la table de faits et de clés primaires des tables dimensions résultantes de la jointure de ces tables [Brick, 1997]. Un index de jointure en étoile est dit complet s'il est construit en joignant toutes les tables de dimensions avec la table de faits. Il est dit partiel s'il est construit en ne joignant que certaines dimensions avec la table de faits. En conséquence, l'index complet est bénéfique pour n'importe quelle requête mais exige beaucoup d'espace disque pour son stockage et son coût de maintenance est élevé.

Les index *bitmap* [Wu et Buchmann, 1998] sont efficaces quand la cardinalité (nombre de valeurs distinctes) de l'attribut indexé n'est pas élevée. À chaque valeur de ce dernier est associé un *bitmap* contenant autant de bits que de n-uplets de la table

indexée. Le i^{eme} bit du *bitmap* associé à une valeur est mis à 1 si cette dernière est présente dans le i^{eme} n-uplet de la table, autrement, il est mis à 0.

Une variante des index *bitmaps* sont les index *bitmap* de jointure (*bitmap join indexes*) [O'Neil et Graefe, 1995]. Un *bitmap* contenant autant de bits que de n-uplets de la table de faits est créé pour chaque valeur distincte de l'attribut indexé d'une dimension. Le i^{eme} bit du *bitmap* est mis à 1 si le n-uplet de la table de dimensions contenant la valeur associée à ce *bitmap* peut être joint avec le i^{ieme} n-uplet de la table de faits. Dans le cas contraire, ce bit est mis à 0. Dans l'entrepôt de données de la Figure 1, nous montrons un index *bitmap* de jointure sur la table de faits `Sales` en utilisant l'attribut "Gender" de la table de dimension `Customers` (Figure 2) [Vanachayobon et Gruenwald, 1999]. Notons que l'attribut indexé "Gender" ne se trouve pas dans la table indexée `Sales`.

Products

P_ID	Weight	Size	Type
10	10	10	A
11	50	10	B
12	50	10	A
13	50	10	C
14	30	10	A
15	50	10	B
16	50	10	D
17	5	10	H
18	50	10	I
19	50	10	E
21	40	10	I
22	50	10	F

Customers

C_ID	Gender	City	State
1	F	Norman	OK
2	F	Norman	OK
3	M	OKC	OK
4	M	Norman	OK
5	F	Ronoake	VA
6	F	OKC	OK
7	M	Norman	OK
8	F	Dallas	TX
9	M	Norman	OK
10	F	Moore	OK

Sales

P_ID	C_ID	Total
10	5	100
11	2	100
15	5	500
10	7	10
10	6	100
10	1	900
11	5	100
10	9	20
11	9	100
10	2	400
13	5	100

Index *bitmap* de jointure

F	M
1	0
1	0
1	0
0	1
1	0
1	0
1	0
0	1
0	1
1	0
1	0

FIG. 2 – *Exemple d'index bitmap de jointure*

Un index *bitmap* de jointure peut également être construit sur plusieurs attributs appartenant à des tables dimensions différentes, jointes avec la table de faits (par exemple, l'attribut "Gender" de la table `Customers` et l'attribut "Type" de la table `Products`). Dans ce cas, l'index est appelé index *bitmap* de jointure multiple (*multiple bitmap join*) [Vanachayobon et Gruenwald, 1999].

Par ailleurs, les index de jointure de dimensions (*dimension join*) ont été proposés pour les entrepôts de données modélisés en flocon de neige [Bizarro et Madeira, 2001]. Le principe de ces index *bitmap* est de rapprocher la table de faits des tables dimensions (hiérarchies) en réalisant le pré-calcul des jointures dans une même hiérarchie. L'index de jointure de dimensions représente un "plus court chemin" entre la table de faits et les tables dimensions.

2.2 Sélection d'index dans les entrepôts de données

Le problème de sélection d'index dans les bases de données est étudié depuis plusieurs années [Finkelstein *et al.*, 1988] [Frank *et al.*, 1992]

[Chaudhuri et Narasayya, 1998] [Agrawal *et al.*, 2000] [Valentin *et al.*, 2000] [Feldman et Reouven, 2003] [Kratica *et al.*, 2003]. Ce problème reste posé dans les entrepôts de données. Les travaux qui s'y rapportent peuvent être classés en deux familles : les algorithmes cherchant à optimiser le temps de maintenance [Labio *et al.*, 1997] et ceux cherchant à optimiser le temps d'exécution des requêtes [Gupta *et al.*, 1997] [Agrawal *et al.*, 2001] [Golfarelli *et al.*, 2002]. Dans les deux cas, l'optimisation est réalisée sous la contrainte d'espace de stockage alloué aux index par l'administrateur. Nous nous intéressons dans cet article aux trois algorithmes de la deuxième famille qui se rapprochent de notre travail.

Le premier algorithme parcourt un multi-graphe bipartie en exploitant un modèle de coût pour recommander un ensemble de vues matérialisées et d'index minimisant le coût d'exécution des requêtes [Gupta *et al.*, 1997]. Ce multi-graphe relie les index et les vues aux requêtes où ils sont présents. Le deuxième algorithme, implanté dans SQL serveur, énumère tous les index et vues matérialisées pouvant contribuer à la réduction du temps d'exécution d'un ensemble de requêtes [Agrawal *et al.*, 2001]. Des appels à l'optimiseur de requêtes permettent ensuite de réduire le nombre de candidats (index et vues). Un ensemble final d'index et de vues matérialisées est alors proposé. Le troisième algorithme se base sur une approche heuristique permettant progressivement de choisir à partir d'un ensemble de candidats (index et vues) les index les plus avantageux [Golfarelli *et al.*, 2002]. Le choix est réalisé en comparant le coût des plans d'exécution possibles, générés par l'optimiseur de requêtes, de chaque requête de la charge.

3 Démarche de sélection automatique d'index

Dans cette section, nous présentons une extension de nos travaux sur la sélection automatique d'index [Aouiche *et al.*, 2003a] [Aouiche *et al.*, 2003b] [Aouiche *et al.*, 2004]. La démarche que nous proposons (Figure 3) exploite le journal des transactions (ensemble de requêtes résolues par le SGBD) pour recommander une configuration (ensemble) 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 [Pasquier *et al.*, 1999]. Nous obtenons alors un ensemble de 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 les modèles de coût présentés dans la Section 4, avant de construire effectivement la configuration d'index. Nous détaillons chacune des ces étapes dans les sections qui suivent.

3.1 Analyse de la charge

Les requêtes présentes dans la charge (journal des transactions) sont traitées par un analyseur syntaxique de requêtes SQL afin d'extraire tous les attri-

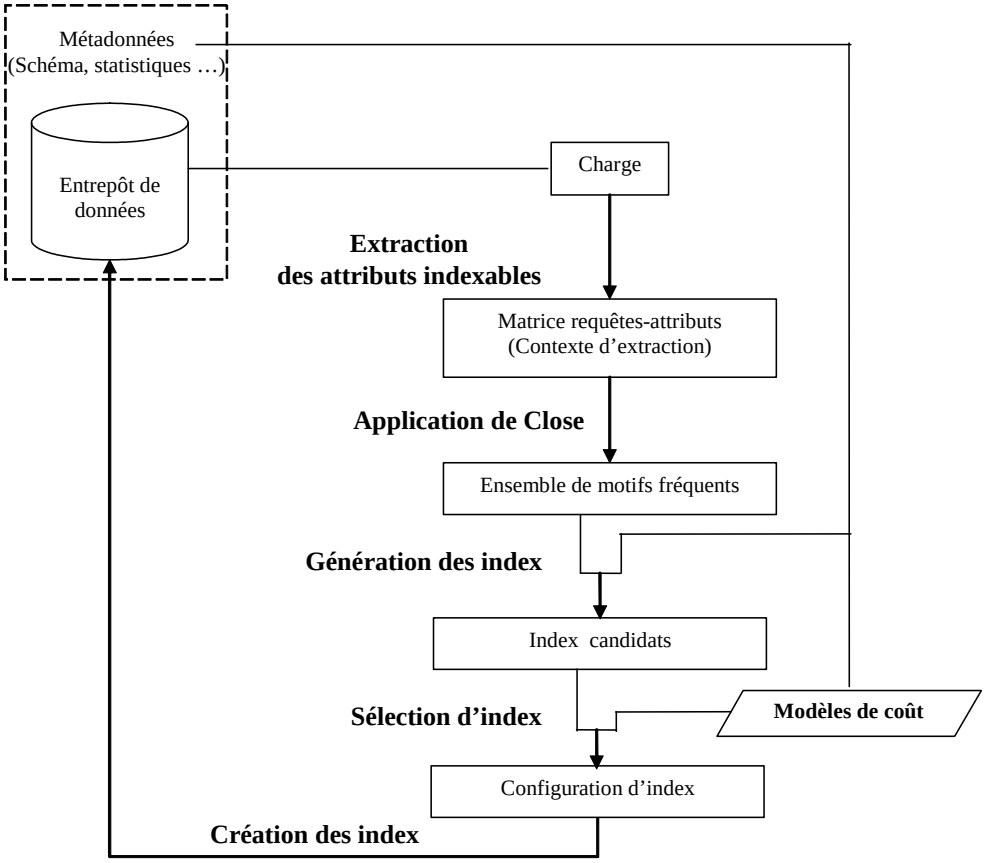

FIG. 3 – *Stratégie de sélection automatique d'index*

buts susceptibles d'être des supports d'index. Ces attributs sont ceux présents dans les clauses `Where` des requêtes (attributs servant 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 [Vanachayobon et Gruenwald, 1999]. Nous ne construisons la matrice "requêtes-attributs" qu'à partir des requêtes d'interrogation.

3.2 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 1 et son absence par 0. 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 représente un extrait d'une 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 – *Extrait d'une charge*

La matrice "requêtes-attributs" obtenue après l'analyse syntaxique de la charge est composée de dix colonnes et trois lignes (Figure 5). 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 5.1, après la présentation des modèles de coût, comment sont générés les index *bitmap* de jointure candidats à partir de ces motifs.

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 (nombre d'index par table limité dans le SGBD utilisé ou espace de stockage alloué aux index restreint). 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 d'accès aux données à travers ces index et de maintenance de ces index en terme de

	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	
	pod_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. 5 – *Matrice requêtes-attributs*

nombre d'entrées/sorties. Le Tableau 1 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. 1 – *Paramètres des modèles de coût*

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 [Wu et Buchmann, 1998] [Wu, 1999].

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 $O(|A||F|)$ [Wu, 1999].

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 de la table de dimension. La variation de la taille et les coûts de ces opérations sont présentés dans les sections suivantes.

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 $O(|A|)$ [Wu et Buchmann, 1998].
- **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 $O(|A|) + O(|F|)$ [Wu et Buchmann, 1998].

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

$$C_{maintenance} = p_T + \frac{|A||F|}{8S_p}.$$

4.2.3 Coût d'insertion dans les tables dimensions

Une insertion dans la table de 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 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.

4.3 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.

4.3.1 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.

Nombre d'entrées/sorties pour 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 $\frac{|A||F|}{8S_p}$. Si la lecture de d *bitmaps* est requise, 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 pour 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 $\frac{|F|}{|A|}$ (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 $C_{lecture} = p_F(1 - e^{-\frac{N_r}{p_F}})$ [O'Neil et Quass, 1997], 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 celui de la phase de lecture des n-uplets :

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

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.

4.3.2 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 6). 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

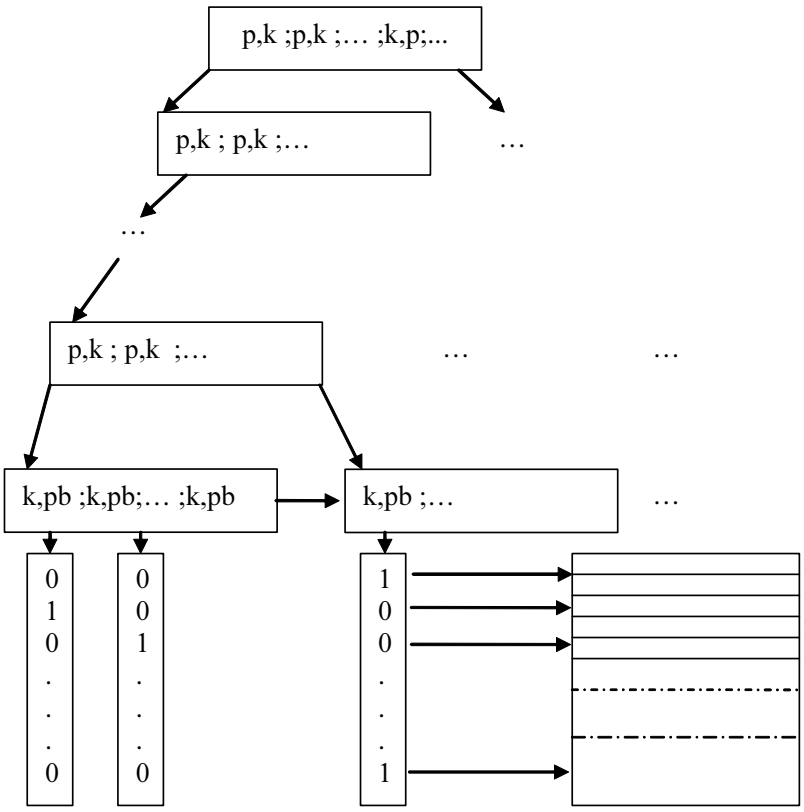

FIG. 6 – *Index bitmap de jointure indexé par un b-arbre*

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 4.3.1.

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

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

4.4 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 $C_{hachage} = 3\ (p_S + p_R)$ [Mishra et Eich, 1992].

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 3). Un algorithme glouton exploite ensuite une fonction objectif basée sur les modèles de coût présentés à la Section 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.

5.1 Construction de l'ensemble d'index candidats

À partir de l'ensemble de motifs fréquents (Section 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 7 montre un exemple d'une commande SQL permettant de construire un index *bitmap* de jointure sur l'attribut `FISCAL_YEAR` de la table de 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. 7 – *Commande SQL de construction d'un index bitmap de jointure*

Nous considérons un motif fréquent $< Table.attribut_1,...,Table.attribut_n >$ comme étant 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 7.

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 le moins 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.

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 S l'ensemble d'index final à créer.

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

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

– 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 plus grand nombre de jointures est le plus 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.

- $C_{/S}(Q)$ représente le coût d'exécution des requêtes de la charge lorsque l'ensemble d'index S 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 de requête. Lorsqu'un index i_j est ajouté à S, $C_{/S\cup\{i_j\}}(Q) = \sum_{k=0}^{|Q|} C(q_k,\{i_j\})$ représente le coût d'exécution des requêtes de la charge en considérant les index de l'ensemble $S\cup\{i_j\}$. 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 $C(q_k,\{i\})$ (coûts 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 mises à jour de l'index i_j est lors du rafraîchissement de l'entrepôt de données. La probabilité de mise à jour $p(i_j)$ de l'index i_j est égale à

$$\frac{1}{nombre\ d'index}\frac{\%rafraichissement}{\%interrogation}$$

  où le rapport $\frac{\%rafraichissement}{\%interrogation}$ 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.

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 $R_{/S}(i_j) = \frac{P_{/S}(i_j)}{taille(i_j)}$ 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 moins d'espace possible.

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 :

$$H_{/S}(i_j) = \begin{cases} P_{/S}(i_j) & \text{si } \frac{espace\_restant}{espace\_stockage} > \alpha, \\ R_{/S}(i_j) & \text{sinon.} \end{cases}$$

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, on pénalise 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 ait plusieurs index apportant une amélioration importante au temps d'exécution des requêtes et occupant beaucoup d'espace.

5.3 Construction de la configuration d'index

L'algorithme de sélection d'index (Algorithme 1) 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 M 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 S. Cela permet de prendre en compte les interactions qui peuvent exister entre les index.

Algorithme 1 *Construction de la configuration d'index*

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

Ces itérations sont répétées jusqu'à ce qu'il n'y ait plus d'amélioration de la fonction objectif ($F_{/S}(i) \leq 0$) ou que tous les index soient sélectionnés ($I - S = \emptyset$). Si la fonction R ou H est utilisée, l'algorithme s'arrête également quand l'espace de stockage disponible est saturé.

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 2 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 l'algorithme 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. Notons que dans ces expérimentations, nous nous basons volontairement sur des données classiques, afin d'assurer une bonne compréhension de la démarche que nous présentons. Cette dernière reste néanmoins valable dans le cas des données complexes.

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	0,0001

TAB. 2 – *Caractéristiques de l'entrepôt de données test*

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 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 *bitmap* par b-arbre (Section 4.3.2) car c'est la méthode utilisée dans Oracle 9i. La charge est composée de quarante requêtes décisionnelles comportant plusieurs jointures. Nous avons mesuré le temps total d'exécution des requêtes de cette charge selon que l'on ait construit ou pas les index et selon que l'on ait appliqué ou pas les 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.

6.1 Expérimentations avec la fonction profit

La Figure 8 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 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.

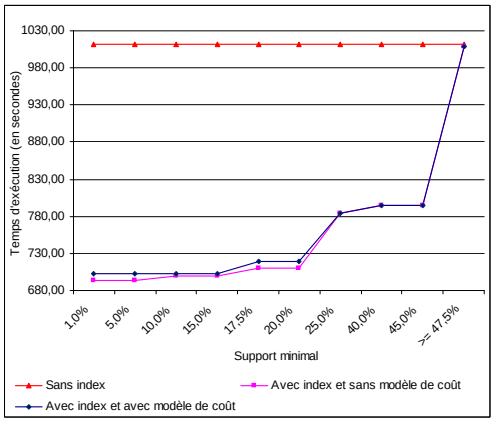

FIG. 8 – *Fonction profit*

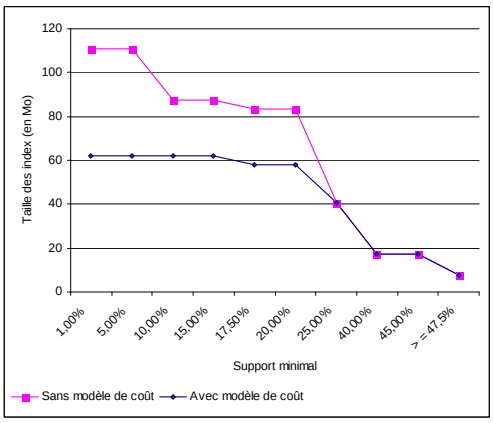

FIG. 9 – *Espace de stockage des index*

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 de 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 9. 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.

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 le plus grand nombre de motifs fréquents et, par conséquent, le plus 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 10 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.

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 $\alpha = 0{,}7$. Nous ne représentons donc à la Figure 11 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 pré-calculent 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.

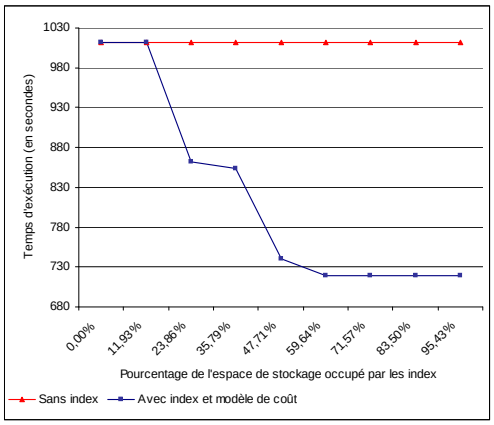

FIG. 10 – *Fonction ratio profit/espace*

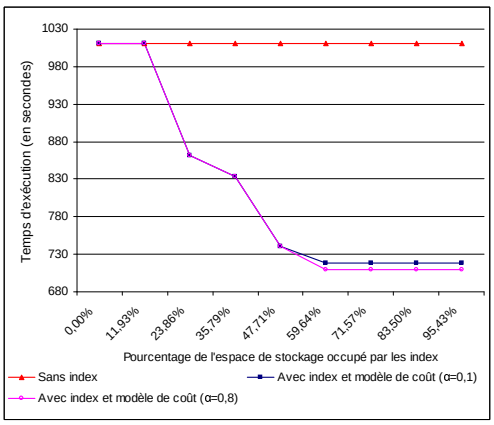

FIG. 11 – *Fonction hybride*

7 Conclusion et perspectives

Dans cet article, nous avons présenté une stratégie automatique de sélection d'index *bitmap* de jointure dans les entrepôts de données complexes. Cette stratégie exploite dans un premier temps les motifs fréquents obtenus par l'algorithme Close à partir d'une charge donnée afin de générer un ensemble d'index *bitmap* de jointures 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 exploitées 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).

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.

Notre travail montre que l'idée d'utiliser des techniques de fouille de données pour l'auto-administration des entrepôts de données complexes est une approche prometteuse. Face à la volumétrie et à la complexité intrinsèque des données traitées, il est en effet devenu crucial de fournir des outils d'aide à l'administrateur de l'entrepôt, notamment en ce qui concerne l'optimisation des performances.

Ce travail ouvre par ailleurs plusieurs axes de recherche future. Dans un premier temps, il paraît indispensable de continuer à tester notre stratégie pour mieux évaluer la surcharge qu'elle engendre pour le système, que ce soit en terme de génération des index ou de leur maintenance. Il sera notamment nécessaire de la confronter à notre entrepôt de données médicales complexes.

Il serait également très intéressant de comparer notre approche de manière plus systématique à d'autres démarches de sélection d'index, que ce soit par des calculs de complexité des heuristiques de génération de configurations d'index (surcharge) ou des expérimentations visant à évaluer la qualité de ces configurations (gain en temps d'exécution et surcharge due à la maintenance des index), notamment lorsque les données indexées sont complexes.

Étendre ou coupler nos travaux à d'autres techniques d'optimisation de performances (vues matérialisées, gestion de cache, regroupement physique, etc.) constitue également une voie de recherche prometteuse. En effet, dans le contexte des entrepôts de données et à plus forte raison lorsque les données sont complexes, c'est principalement en conjonction avec d'autres structures physiques (principalement les vues matérialisées) que l'indexation permet d'obtenir des gains de performance significatifs. Dans ce cadre, une classification effectuée sur le contexte d'extraction permettrait de construire des groupes de requêtes ayant de fortes similarités. Chaque groupe de requêtes pourrait alors être un point de départ pour la matérialisation des vues. De plus, il serait intéressant d'intégrer des méthodes permettant de partager efficacement l'espace de stockage disponible entre les index et les vues matérialisées [Bellatreche *et al.*, 2000].

Finalement, le travail que nous proposons ici exploite principalement des descripteurs de données complexes pour l'indexation. Il n'est spécifique aux données complexes qu'en raison des modifications d'architecture de l'entrepôt que nous avons introduites pour stocker nos données médicales. Il serait sans doute intéressant d'exploiter les

données complexes elles-même, par exemple en prenant en compte des descripteurs de bas niveaux, comme la couleur ou la texture pour des images. L'application de techniques de fouille permettant de déterminer quelles données sont les plus susceptibles d'être utilisées conjointement pourrait aussi permettre de produire un regroupement physique de ces données sur disque.

Summary

The queries defined on data warehouses are often intricate and use several join operations that induce an expensive computational cost. In the field of complex data warehousing, adaptations from the classical data warehouse schemas induce additional joins at data access time. This cost becomes even more prohibitive when queries access very large volumes of data. It is thus crucial to reduce data access costs. To improve response time, data warehouse administrators generally use indexing techniques such as star join indices or bitmap join indices. This is nevertheless still complex and fastidious.

The solution we propose lies in the field of data warehouse auto-administration. In this framework, we propose an automatic index selection strategy. To release this selection, we turn to a data mining technique; more precisely frequent itemset mining, in order to determine a set of candidate indices from a given workload. Then, we propose several cost models that help creating an index configuration composed by the indices providing the best profit. These models particularly evaluate the cost of accessing data using bitmap join indices, and the cost of updating and storing these indices.

Keywords: Data warehouses, complex data, auto-administration, index selection, frequent itemset, cost models, bitmap join indexes.

Références

[Agrawal *et al.*, 2000] S. Agrawal, S. Chaudhuri, et V.R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *26th International Conference on Very Large Data Bases (VLDB 2000), Cairo, Egypt*, pages 496–505, 2000.

[Agrawal *et al.*, 2001] S. Agrawal, S. Chaudhuri, et V.R. Narasayya. Materialized view and index selection tool for Microsoft SQL Server 2000. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2001), Santa Barbara, USA*, page 608, 2001.

[Aouiche *et al.*, 2003a] K. Aouiche, J. Darmont, et L. Gruenwald. Frequent itemsets mining for database auto-administration. In *7th International Database Engineering and Application Symposium (IDEAS 2003), Hong Kong, China*, pages 98–103, 2003.

[Aouiche *et al.*, 2003b] K. Aouiche, J. Darmont, et L. Gruenwald. Vers l'auto–administration des entrepôts de données. *Revue des Nouvelles Technologies de l'Information*, (1):1–12, 2003.

[Aouiche *et al.*, 2004] K. Aouiche, J. Darmont, et O. Boussaid. Sélection automatique d'index dans les entrepôts de données. In *1er atelier Fouille de Données Complexes*

dans un processus d'extraction des connaissances (EGC 2004), Clermont-Ferrand, France, pages 91–102, 2004.

[Bellatreche *et al.*, 2000] L. Bellatreche, K. Karlapalem, et M. Schneider. On efficient storage space distribution among materialized views and indices in data warehousing environments. In *9th International Conference on Information and Knowledge Management (CIKM 2000), McLean, USA*, pages 397–404, 2000.

[Bizarro et Madeira, 2001] P. Bizarro et H. Madeira. The dimension-join: A new index for data warehouses. In *16th Simpósio Brasileiro de Banco de Dados (SBDD 2001), Rio de Janeiro, Brazil*, pages 259–273, 2001.

[Brick, 1997] R. Brick. Star schema processing for complex queries. White paper, 1997.

[Chaudhuri et Narasayya, 1998] S. Chaudhuri et V.R. Narasayya. Autoadmin 'what-if' index analysis utility. In *ACM SIGMOD International Conference on Management of Data, Seattle, USA*, pages 367–378, 1998.

[Feldman et Reouven, 2003] Y.A. Feldman et J. Reouven. A knowledge–based approach for index selection in relational databases. *Expert System with Applications*, 25(1):15–37, 2003.

[Finkelstein *et al.*, 1988] S.J. Finkelstein, M. Schkolnick, et P. Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988.

[Frank *et al.*, 1992] M.R. Frank, E. Omiecinski, et S.B. Navathe. Adaptive and automated index selection in RDBMS. In *3rd International Conference on Extending Database Technology (EDBT 1992), Vienna, Austria*, volume 580 of *Lecture Notes in Computer Science*, pages 277–292, 1992.

[Golfarelli *et al.*, 2002] M. Golfarelli, S. Rizzi, et E. Saltarelli. Index selection for data warehousing. In *4th International Workshop on Design and Management of Data Warehouses (DMDW 2002), Toronto, Canada*, pages 33–42, 2002.

[Gupta *et al.*, 1997] H. Gupta, V. Harinarayan, A. Rajaraman, et J. D. Ullman. Index selection for OLAP. In *13th International Conference on Data Engineering (ICDE 1997), Birmingham, U.K.*, pages 208–219, 1997.

[Hu *et al.*, 2003] X. Hu, T.Y. Lin, et E. Louie. Bitmap techniques for optimizing decision support queries and association rule algorithms. In *7th International Database Engineering and Application Symposium (IDEAS 2003), Hong Kong, China*, pages 34–43, 2003.

[Inmon, 2002] W.H. Inmon. *Building the Data Warehouse*. John Wiley & Sons, third edition, 2002.

[Kimball et Ross, 2002] R. Kimball et M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, second edition, 2002.

[Kratica *et al.*, 2003] J. Kratica, I. Ljubić, et D. Tošić. A genetic algorithm for the index selection problem. In *Applications of Evolutionary Computing, Essex, England*, volume 2611 of *LNCS*, pages 281–291, 2003.

[Labio *et al.*, 1997] W. Labio, D. Quass, et B. Adelberg. Physical database design for data warehouses. In *13th International Conference on Data Engineering (ICDE 1997), Birmingham, U.K.*, pages 277–288, 1997.

[Mishra et Eich, 1992] P. Mishra et M. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

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

[O'Neil et Quass, 1997] P.E. O'Neil et D. Quass. Improved query performance with variant indexes. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 1997), Tucson, USA*, pages 38–49, 1997.

[Pasquier *et al.*, 1999] N. Pasquier, Y. Bastide, R. Taouil, et L. Lakhal. Discovering frequent closed itemsets for association rules. In *7th International Conference on Database Theory (ICDT 1999), Jerusalem, Israel*, volume 1540 of *LNCS*, pages 398–416, 1999.

[Sarawagi, 1997] S. Sarawagi. Indexing OLAP data. *Data Engineering Bulletin*, 20(1):36–43, 1997.

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

[Valentin *et al.*, 2000] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, et A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *16th International Conference on Data Engineering (ICDE 2000), San Diego, USA*, pages 101–110, 2000.

[Vanachayobon et Gruenwald, 1999] S. Vanachayobon et L. Gruenwald. Indexing techniques for data warehouses' queries. Technical report, The University of Oklahoma, School of Computer Science, 1999.

[Weikum *et al.*, 2002] G. Weikum, A. Monkeberg, C. Hasse, et P. Zabback. Self-tuning database technology and information services: from wishful thinking to viable engineering. In *28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China*, pages 20–31, 2002.

[Wu et Buchmann, 1998] M.C. Wu et A.P. Buchmann. Encoded bitmap indexing for data warehouses. In *14th International Conference on Data Engineering (ICDE 1998), Orlando, USA*, pages 220–230, 1998.

[Wu, 1999] M.C. Wu. Query optimization for selections using bitmaps. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 1999), Philadelphia, USA*, pages 227–238, 1999.