Chapitre 2

Indexation et matérialisation de vues

Le travail d'optimisation de performance de l'administrateur de bases de données consiste principalement à sélectionner des structures physiques telles que les index et les vues matérialisées. Ces structures jouent un rôle particulièrement important dans les systèmes d'aide à la décision comme les entrepôts de données, dans la mesure où elles réduisent le temps de réponse à des requêtes très souvent complexes.

Dans ce chapitre, nous présentons les principales techniques d'indexation communément utilisées dans les bases et entrepôts de données relationnels, ainsi que dans les bases de données natives XML. Ensuite, nous exposons le principe de la matérialisation des vues et son exploitation afin d'améliorer le temps d'exécution des requêtes décisionnelles dans le contexte relationnel et XML.

2.1 Techniques d'indexation dans les bases et entrepôts de données relationnels

Dans les systèmes de gestion de bases de données (SGBD), l'accès aux données est d'autant plus lent que la base de données est volumineuse. Un parcours séquentiel des données est une opération lente et pénalisante pour l'exécution des requêtes, notamment dans le cas des opérations de jointure où ce parcours doit souvent être effectué de façon répétitive. La création d'un index permet d'améliorer considérablement le temps d'accès

aux données en créant des chemins d'accès directs.

Il existe deux types d'index : les index primaires (*clustered*), aussi appelés index groupants, et secondaires (*non-clustered*), aussi appelés index non-groupants. Les adresses contenues dans un index primaire sont triées suivant le placement physique sur disque des n-uplets composant la table indexée. En revanche, les adresses d'un index secondaire ne suivent pas cette organisation. Lorsqu'un index primaire est utilisé, peu de blocs disques sont parcourus et les requêtes de recherche sont ainsi résolues de manière efficace. Toutefois, ce type d'index souffre d'un coût de maintenance très élevé car il faut maintenir l'ordre du tri. Les index secondaires sont moins efficaces que les index primaires, mais moins coûteux au niveau de la maintenance. Il peut y avoir au plus un index primaire, mais plusieurs index secondaires sur une table donnée sont possibles.

Nous présentons dans les sections suivantes les principales techniques d'indexation utilisées dans les SGBD relationnels et les entrepôts de données.

2.1.1 Index en B-arbre

Un B-arbre est une liste chaînée de nœuds dont la valeur est celle de l'index. Les feuilles de l'arbre font référence à une seule valeur, si cet index est construit sur un attribut clé, ou plusieurs valeurs, si cet index est construit sur un attribut non-clé, des n-uplets de la table indexée. Cette référence spécifie l'emplacement physique du n-uplet sur le disque [BM72].

Un B-arbre offre un excellent compromis pour les opérations de recherche par clé et par intervalle, ainsi que pour les mises à jour. Ces qualités expliquent le fait que les B-arbres et leurs variantes soient systématiquement intégrés dans la plupart des SGBD.

La Figure 2.1 montre un exemple de B-arbre construit sur la table `Films` définie par le schéma `Films (Film_ID, Film_Titre, Film_Année,...)`.

2.1.2 Index de hachage

Les tables de hachage sont des structures de données très couramment utilisées en mémoire centrale pour organiser des ensembles et fournir un accès performant à leurs éléments. Nous commençons par rappeler le principe du hachage avant de présenter les spécificités apportées par le stockage en mémoire secondaire.

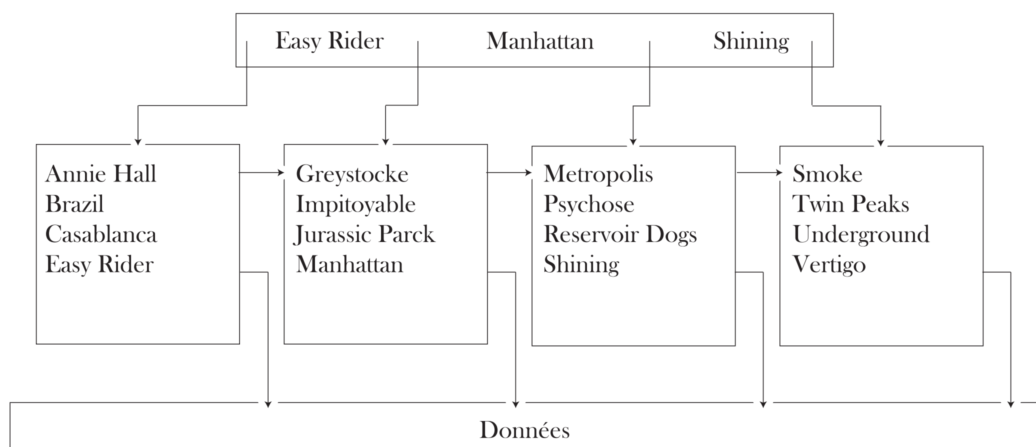


FIG. 2.1 – Index en B-arbre construit sur l'attribut `Film_Titre`

L'idée de base du hachage est d'organiser un ensemble d'éléments d'après une clé et d'utiliser une fonction, dite de hachage, qui, pour chaque valeur de clé c, donne l'adresse $f(c)$ d'un espace de stockage où l'élément doit être placé. En mémoire principale, cet espace de stockage est en général une liste chaînée et, en mémoire secondaire, un ou plusieurs blocs sur le disque.

La Figure 2.2 montre un exemple d'index de hachage construit sur la table `Films` définie dans la Section 2.1.1. La fonction de hachage est $H(titre) = rang(titre[0]) mod 5$, où $titre[0]$ désigne la première lettre du titre d'un film.

Une fonction de hachage mal conçue affecte tous les n-uplets à la même adresse et la structure dégénère vers un simple fichier séquentiel. Cela peut être le cas, avec notre fonction basée sur la première lettre du titre, pour tous les films dont le titre commence par la lettre l.

2.1.3 Index *bitmap*

Un index *bitmap* repose sur un principe très différent de celui des index en B-arbre. Alors que dans ces derniers, on trouve, pour chaque attribut indexé, les mêmes valeurs dans l'index et dans la table, un index *bitmap* considère toutes les valeurs possibles de l'attribut indexé, que la valeur soit présente ou non dans la table [OQ97]. Pour chacune de ces valeurs possibles, un tableau de bits, dit *bitmap*, est stocké. Ce *bitmap* est composé d'autant de bits qu'il y a de n-uplets dans la table indexée. Notons par A l'attribut indexé et v la valeur

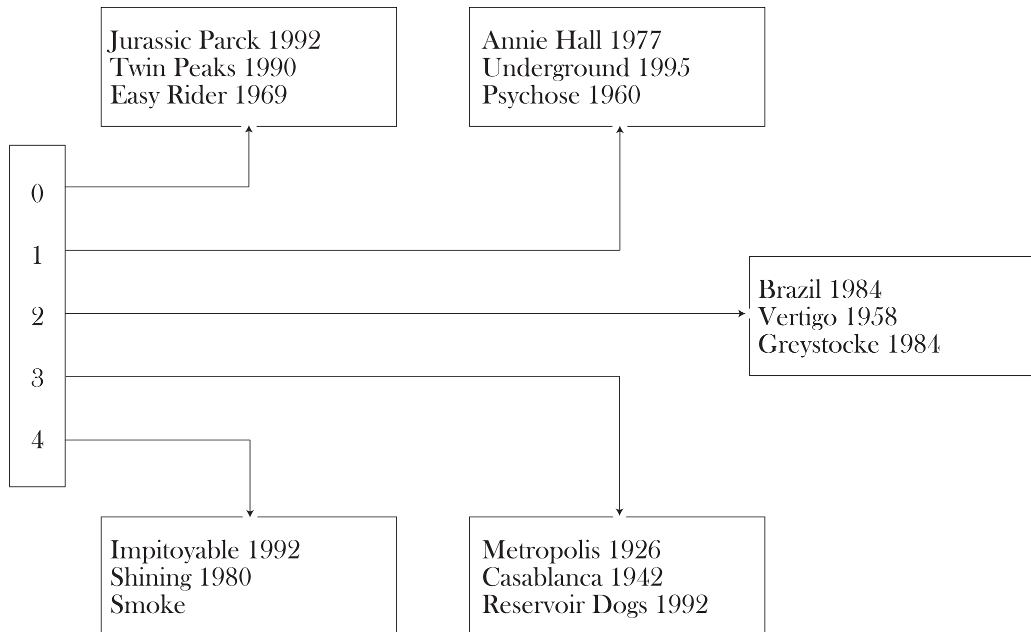


FIG. 2.2 – Index de hachage construit sur l'attribut `Film_Titre`

définissant le *bitmap*. Chaque bit associé à un n-uplet a alors la signification suivante :

- si le bit est mis à 1, l'attribut A a pour valeur v pour ce n-uplet ;
- sinon, le bit est mis à 0.

Lorsque les n-uplets dont la valeur est v sont recherchés, il suffit donc de prendre le *bitmap* associé à v, de chercher tous les bits à 1 et d'accéder ensuite aux n-uplets correspondants. Un index *bitmap* est très efficace si le nombre de valeurs possibles de l'attribut indexé est relativement faible.

Reprenons l'exemple de la table `Films` et créons un index *bitmap* sur le genre des films. L'utilisation d'un B-arbre ne donnerait pas de bons résultats car l'attribut est trop peu sélectif ; autrement dit, une partie importante de la table peut être sélectionnée quand on effectue une recherche par valeur. En revanche, un index *bitmap* est tout à fait approprié puisque le genre fait partie d'un ensemble énuméré comprenant relativement peu de valeurs. La Figure 2.1 montre l'index *bitmap* pour les valeurs `Drame`, `Science-Fiction` et `Comédie`. Chaque colonne correspond à un film[1].

[1]Il s'agit dans ce cas d'un index *bitmap* simple (*simple bitmap*). Il existe d'autres types d'index *bitmap* comme l'index *bitmap* encodé (*encoded bitmap index*).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Drame	0	0	0	1	1	1	0	0	0	0	0	0	0	0	1	0
Science-Fiction	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0
Comédie	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	1

TAB. 2.1 – Index *bitmap* construit sur les catégories des films

Pour la valeur `Drame`, les bits sont à 1 pour les films de rang 4, 5, 6 et 15. Tous les autres bits sont à zéro. Pour `Science-Fiction`, les bits à 1 sont aux rangs 2, 10 et 11. Bien entendu, il ne peut y avoir qu'un seul 1 dans une colonne, puisqu'un attribut ne peut prendre qu'une seule valeur.

Un index *bitmap* est de très petite taille comparé à un B-arbre construit sur le même attribut. Il est donc très utile dans des applications de type entrepôt de données gérant de gros volumes de données et classant les informations par des attributs catégoriels définis sur de petits domaines de valeurs. Certaines requêtes peuvent alors être exécutées très efficacement, parfois sans même recourir à la table contenant les données. Prenons l'exemple suivant : "Combien y a-t-il de films dont le genre est Drame ou Comédie ?", exprimée en SQL comme suit, `select count(*) from Films where genre in ('Drame', 'Comédie')`. Pour répondre à cette requête, il suffit de compter le nombre de 1 dans les *bitmaps* associés à `Drame` et `Comédie`.

2.1.4 Index de jointure

Valduriez a proposé un index appelé index de jointure (*join indices*) pour pré-joindre deux tables [Val87]. L'index matérialise les liens existants entre deux tables par le biais d'une table à deux attributs contenant les indentifiants [2] des n-uplets joints. Plus formellement, un index de jointure construit sur les tables R et S jointes par les attributs A de R et B de S est l'ensemble des n-uplets :

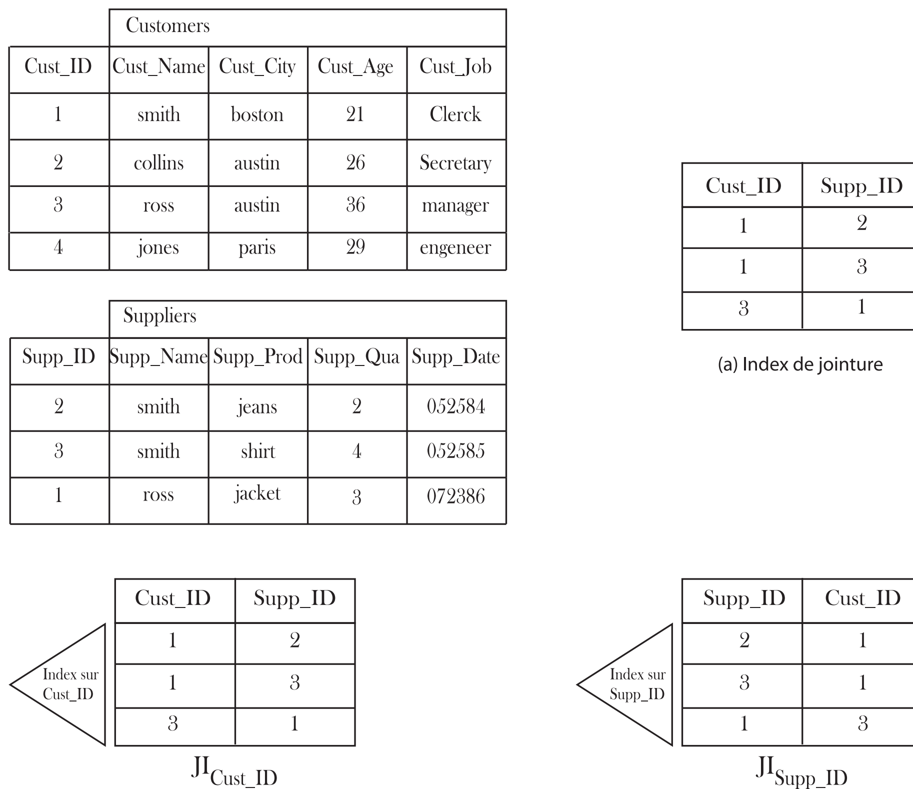
$$JI = \{(R_ID_i, S_ID_j), f(R_ID_i.A, S_ID_j.B) \text{ est vraie}\}$$

où :

[2]Chaque n-uplet d'une table est identifié de manière unique à l'aide d'un identifiant unique invariant généré par le système *surrogate*.

– R_ID_i, S_ID_j sont respectivement les identifiants des n-uplets des tables R et S,
– $R_ID_i.A$ (respectivement, $S_ID_j.B$) est la valeur de l'attribut A (respectivement, B) du n-uplet identifié par ID_i (respectivement, ID_j) dans la table R (respectivement, dans la table S),
– f est une fonction booléenne définissant la condition de jointure.

La Figure 2.3 (a) montre un exemple d'index de jointure construit sur les tables `Customers` et `Suppliers` vérifiant la condition de jointure `Customers.Cust_name = Suppliers.Supp_name`.



	Customers			
Cust_ID	Cust_Name	Cust_City	Cust_Age	Cust_Job
1	smith	boston	21	Clerck
2	collins	austin	26	Secretary
3	ross	austin	36	manager
4	jones	paris	29	engeneer

	Suppliers			
Supp_ID	Supp_Name	Supp_Prod	Supp_Qua	Supp_Date
2	smith	jeans	2	052584
3	smith	shirt	4	052585
1	ross	jacket	3	072386

Cust_ID	Supp_ID
1	2
1	3
3	1

(a) Index de jointure

Cust_ID	Supp_ID
1	2
1	3
3	1

Supp_ID	Cust_ID
2	1
3	1
1	3

(b) Implémentation de l'index de jointure

FIG. 2.3 – Index de jointure sur les tables `Customers` et `Suppliers`

Si l'index de jointure ne tient pas totalement en mémoire, des accès rapides à cet index via `R_ID` ou `S_ID` sont nécessaires. C'est pourquoi, Valduriez propose d'utiliser deux copies de l'index de jointure accessibles chacune par un B-arbre. La copie de l'index de jointure regroupée suivant `R_ID` (respectivement, `S_ID`) rend efficace les accès en jointure de R vers

S (respectivement, de S vers R). Cependant, la maintenance de l'index est coûteuse car il est nécessaire de maintenir les deux copies de cet index pour assurer la cohérence des données. Dans l'exemple de la Figure 2.3 (b), nous illustrons l'implémentation de l'index de jointure sur les tables `Customers` et `Suppliers` de l'exemple précédent. Les identifiants `Cust_ID` (respectivement, `Supp_ID`) de la copie JI_{Cust_ID} (respectivement, JI_{Supp_ID}) sont utilisés pour trouver les nœuds feuilles pointant vers `Cust_ID` (respectivement, `Supp_ID`).

L'algorithme proposé pour la jointure de deux tables R et S lit séquentiellement l'index de jointure page par page, compose la semi-jointure $R \ltimes JI$ (égale à $R \ltimes S$) et réalise par la suite la jointure du résultat avec $S \ltimes JI$ (égale à $S \ltimes R$). Si $R \ltimes JI$ ne tient pas en mémoire, l'opération de jointure est subdivisée en plusieurs passes.

La taille de JI dépend du facteur de sélectivité de la jointure (*join selectivity factor*). Ce facteur, noté JS est défini comme suit :

$$JS = \frac{\| R \bowtie S \|}{\| R \| \| S \|}$$

où $\| X \|$ désigne le nombre de n-uplets de la table X. Si la jointure est très sélective (c'est-à-dire, lorsque JS est petit) l'index JI est petit. Par contre, une jointure peu sélective (c'est-à-dire, lorsque JS est proche de 1), qui peut être assimilée à un produit cartésien, peut donner lieu à un index volumineux. Dans ce cas, le nombre de passes de l'algorithme de jointure est important. Cela a pour conséquence de dégrader les performances de l'index JI.

2.1.5 Index de jointure en étoile

Red Brick a proposé un index appelé index de jointure en étoile, adapté aux requêtes définies sur les entrepôts de données modélisés en étoile [Bri97] (voir exemple en Figure 2.4). Un index de jointure en étoile peut contenir toute combinaison des identifiants des n-uplets de la table de faits et des identifiants des n-uplets des tables dimensions pouvant être jointes.

Un index de jointure en étoile est dit complet s'il est construit en joignant toutes les tables dimensions de l'entrepôt avec la table de faits. S'il est construit en joignant certaines des tables dimensions avec la table de faits, il est dit partiel. L'index de jointure en étoile complet est bénéfique à n'importe quelle requête définie sur un entrepôt modélisé en étoile. Il exige cependant beaucoup d'espace de stockage et un coût de maintenance très élevé.

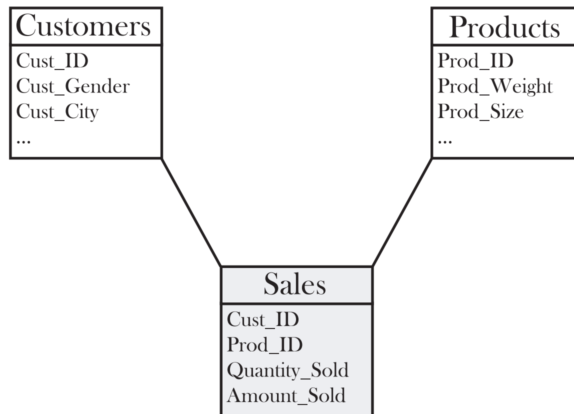


Fig. 2.4 – Entrepôt de données modélisé en étoile

	Index de jointure en étoile	
Sales_ROWID	Customers_ROWID	Products_ROWID
Sales_ROWID1	Customers_ROWID1	Sales_ROWID1
...	...	...

Fig. 2.5 – Exemple d'un index de jointure en étoile

La Figure 2.5 illustre un exemple d'un index de jointure en étoile construit sur l'entrepôt de données dont le schéma est représenté à la Figure 2.4. `Sales_ROWID`, `Customers_ROWID` et `Products_ROWID` représentent les indentifiants des n-uplets des tables `Sales`, `Customers` et `Products`. Chaque combinaison de ces indentifiants ((`Sales_ROWID1`, `Customers_ROWID1`, `Products_ROWID2`), par exemple) correspond à un n-uplet de la table de faits `Sales`.

2.1.6 Index *bitmap* de jointure

L'index *bitmap* de jointure (*bitmap join index*) a été proposé pour pré–joindre la table de faits et les tables de dimensions dans les entrepôts de données modélisés en étoile [OG95, OQ97]. Un *bitmap* représentant les n-uplets de la table de faits est créé pour chaque valeur distincte de l'attribut de la table dimension sur lequel l'index est construit. Le $i^{\text{ème}}$ bit du *bitmap* est à un si le n-uplet correspondant à la valeur de l'attribut indexé peut être joint avec le n-uplet de rang i de la table de faits. Dans le cas contraire, le $i^{\text{ème}}$ bit est à zéro.

À partir de l'entrepôt de données représenté à la Figure 2.4 et de son extension de la Figure 2.6, nous pouvons construire un index *bitmap* de jointure sur la table de faits `Sales` en utilisant l'attribut `Cust_Gender` de la table dimension `Customers` comme illustré à la Figure 2.7. Notons que la table de faits `Sales` ne contient pas l'attribut indexé `Cust_Gender`.

	Products		
Prod_ID	Prod_Weight	Prod_Size	Prod_Type
10	10	10	A
11	50	10	B
12	50	20	A
13	50	25	C
14	30	30	A
15	50	5	B
16	50	40	D
17	5	10	H
18	50	10	I
19	50	5	E
21	40	25	I
22	50	10	F

	Customers		
Cust_ID	Cust_Gender	Cust_City	Cust_State
1	F	austin	OK
2	F	boston	OK
3	M	paris	OK
4	M	dallas	OK
5	F	austin	VA
6	F	paris	OK
7	M	boston	OK
8	F	new york	TX
9	M	paris	OK
10	F	austin	OK

	Sales	
Prod_ID	Cust_ID	Quantity_Sold
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

FIG. 2.6 – Extension de l'entrepôt de données modélisé en étoile

L'index *bitmap* de jointure est particulièrement utile pour les jointures en étoile (*star joins*). Une requête contenant des jointures en étoile a la syntaxe suivante :

```
select  *
from    F, D1, D2, ..., Dn
where   F.D1_ID = D1.D1_ID
```

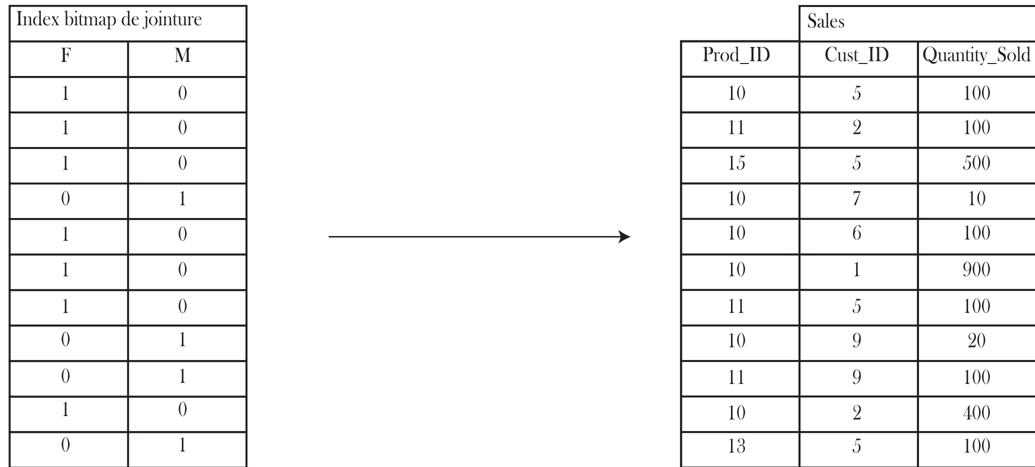
```
and F.D2_ID = D2.D2_ID
and ...
and F.Dn_ID = Dn.Dn_ID
and D1.B1 = C1
and D2.B2 = C2
and ...
and Dn.Bn = Cn
```

où `Di_ID` et `Bi` sont respectivement des attributs clés et non clés des tables dimensions `Di`, `Ci` des constantes et `F` la table de faits.

L'utilisation des index *bitmaps* de jointure par l'optimiseur de requêtes nécessite la réécriture de la requête (*star transformation*) de la manière suivante :

```
select  *
from    F, D1, D2, ..., Dn
where
        D1.D1_ID       in (welect D1.D_ID
                           from D1
                           where D1.B1 = C1)
        and D2.D2_ID   in (select D2.D2_ID
                           from D2
                           where D2.B2 = C2)
        and ...
        and Dn.Dn_ID   in (select Dn.Dn_ID
                           from Dn
                           where Dn.Bn = Cn)
```

L'index *bitmap* de jointure construit sur `Di.Bi` est exploité pour trouver les *bitmaps* vérifiant le prédicat `Di.Bi = Ci`. Cette recherche est réitérée pour chaque table utilisant les prédicats `Di.Bi = Ci`. L'intersection (opération `AND` entre les *bitmaps*) des n *bitmaps* (un *bitmap* par table dimension) donne le *bitmap* utilisé pour chercher les n-uplets de la table de faits répondant à la requête.



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
0	1

	Sales	
Prod_ID	Cust_ID	Quantity_Sold
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

FIG. 2.7 – Index *bitmap* de jointure

L'index *bitmap* de jointure de la Figure 2.7 est particulièrement intéressant pour répondre à la requête suivante.

```
select  sum(total)
from    Sales, Customers
where   Sales.Cust_ID =Customers.Cust_ID
        and Customers.Cust_Gender ='M'
```

Un index *bitmap* de jointure peut également être construit en prenant en compte simultanément plusieurs restrictions sur les valeurs d'attributs de différentes tables dimensions jointes avec la table de faits (par exemple, l'attribut `Cust_Gender` de la table `Customers` et l'attribut `Prod_type` de la table `Products`). Dans ce cas, l'index est appelé index *bitmap* de jointure multiple (*multiple bitmap join*) [VG99].

2.1.7 Index de jointure de dimensions

L'index de jointure de dimensions (*dimension join index*) est un index *bitmap* proposé pour les entrepôts de données modélisés en flocon de neige [BM01]. Le principe de cet index est de rapprocher les tables dimensions de la table de faits en réalisant le pré–calcul des jointures intermédiaires. Dans ce cas, l'index de jointure de dimensions représente un "plus court chemin" entre la table de faits et les tables dimensions indexées.

Comme dans le cas des index *bitmaps* de jointure, un *bitmap* est créé pour chaque valeur distincte de l'attribut indexé. Les valeurs des bits de ce *bitmap* sont calculées de la même manière en faisant toutes les jointures menant de la table de faits à la table dimension indexée. La réécriture des requêtes (*star transformation*) est également nécessaire pour l'utilisation de l'index de jointure de dimensions.

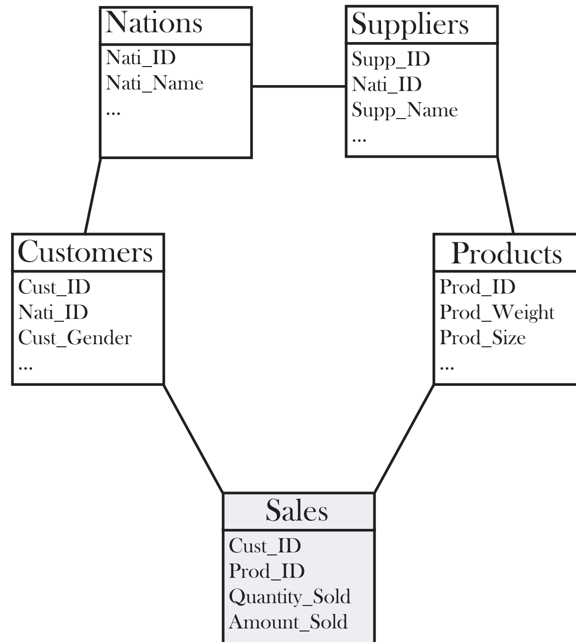


FIG. 2.8 – Entrepôt de données modélisé en flocon de neige

Dans l'entrepôt de données représenté à la Figure 2.8, deux chemins possibles mènent de la table de faits `Sales` à la table de dimension `Nations` : le premier chemin est obtenu en réalisant les jointures des tables `Sales`, `Customers` et `Nations`, et le deuxième en réalisant les jointures des tables `Sales`, `Products`, `Suppliers` et `Nations`. Deux index de jointure de dimensions `IDX_CUST` et `IDX_SUPP`, construits sur l'attribut `Nations.Nat_name`, correspondant respectivement à ces chemins peuvent être proposés pour relier la table `Sales` à `Nations`. La représentation de l'index `IDX_CUST` est donnée à la Figure 2.9.

Les index `IDX_SUPP` et `IDX_CUST` sont particulièrement intéressants pour améliorer le temps d'exécution de la requête de la Figure 2.10. Pour cette requête, sans ces index, chaque n-uplet de `Sales` doit être joint avec les n-uplets de `Customers` et `Nations` pour trouver la nationalité des clients. De la même manière, chaque n-uplet de `Sales` doit être joint avec les

	Index de jointure de dimensions	
Algeria	...	USA
0	...	1
0	...	0
0	...	1
1	...	0
0	...	1
0	...	0
1	...	1
1	...	0
0	...	1
0	...	0
1	...	1

	Sales	
Prod_ID	Cust_ID	Quantity_Sold
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

Fig. 2.9 – Index de jointure de dimensions

n-uplets de `Products`, `Suppliers` et `Nations` pour trouver la nationalité des fournisseurs. Avec les index `IDX_CUST` et `IDX_SUPP`, l'optimiseur de requêtes réécrit la requête et n'exécute pas les lignes 14 à 19 et 21 à 25 de la Figure 2.10 représentant les jointures.

2.2 Techniques d'indexation des données XML

Les bases de données natives XML sont utilisées pour stocker des documents XML. Ces données sont interrogées à l'aide de langages basés sur des expressions de chemin. Une expression de chemin est une suite d'étiquettes dans un graphe ou un arbre modélisant les données XML. Elle consiste à trouver un élément ou un nœud dans un graphe ou un arbre de données XML en parcourant le chemin décrit par l'expression.

Pour garantir des temps de traitement raisonnables pour les requêtes en expressions de chemin, des techniques d'indexation peuvent être envisagées. Les index XML peuvent être des documents XML. Ils peuvent maintenir toutes les données du document XML indexé. Plusieurs travaux se sont consacrés à la création d'un schéma ou d'un modèle pour l'indexation des données XML.

Pour illustrer les techniques d'indexation de données XML que nous présentons ici, nous utilisons le document XML décrit à la Figure 2.11. Le graphe XML correspondant est représenté à la Figure 2.12.

```
 1:  Select
 2:       supp_nation,
 3:       cust_gender,
 4:       supp_year,
 5:       sum(volume) as revenue
 6:  From
 7:       (
 8:            Select
 9:                      n1.nati_name as supp_nation
10:                      n2.nati_name as cust_nation
11:                      extract(year from supp_shipdate) as supp_year
12:                      supp_extentedprice *(1-supp_discount) as volume
13:            From
14:                      Suppliers,
15:                      Products,
16:                      Customers,
17:                      Nations n1,
18:                      Nations n2,
19:                      Sales
20:            Where
21:                      customers.cust_ID = sales.cust_ID
22:                      and products.prod_ID = sales.prod_ID
23:                      and suppliers.supp_ID = products.prod_ID
24:                      and suppliers.nati_ID = n1.nati_ID
25:                      and customers.nati_ID = n2.nati_ID
26:                      and (
27:                           (n1.nati_name = 'France' and n2.nati_name='Germany')
28:                           or (n1.nati_name = 'Germany' and n2.nati_name ='France')
29:                           )
30:                      and supp_shipdate
31:                           between date '1995-01-01' and date '1996-12-31'
32:       ) as shipping
33:  Group by
34:       supp_nation
35:       cust_nation
36:       supp_year
37:  Order by
38:       supp_nation
39:       cust_nation
40:       supp_year
```

FIG. 2.10 – Requête exploitant un index de jointure de dimensions

2.2.1 Guide de données

Le guide de données est une structure sommaire pour représenter des données semi-structurées et les données XML [YG01, KGY]. La structure de l'index consiste à décrire tous les nœuds dont l'étiquette ou nom de balise est identique. La définition du guide de données se base sur des ensembles ciblés. Un ensemble ciblé d'un chemin est l'ensemble de tous les objets (nœuds) accessibles en parcourant ce chemin. La Figure 2.13 décrit un exemple de guide de données [KGY]. Le nœud 2,4 de la Figure 2.13 représente un ensemble ciblé. En effet, les nœuds 2 et 4 de la Figure 2.12 sont accessibles par le même chemin MovieDB/actor.

Le guide de données peut être créé par une exploration en profondeur du graphe de données XML, tout en accumulant les ensembles ciblés des chemins visités. Chaque ensemble ciblé est stocké dans une table de hachage. Le guide de données est augmenté d'arrêtes dans une présence de références cycliques.

```
<MovieDB>
        <actor id="a1">
                <name>actor1</name>
        </actor>
        <actor id="a2" movie="m1">
                <name>actor2</name>
        </actor>
        <director id="d1">
                <name>director1</name>
                 <movie id="m1" director="d1">
                        <title>movie2</title>
                </movie>
        </director>
         <director id="d2">
                 <name>director2</name>
         </director>
         <movie id="m2" actor="a1" director="d2">
                <title>movie1</title>
         </movie>
</MovieDB>
```

Fig. 2.11 – Exemple de document XML

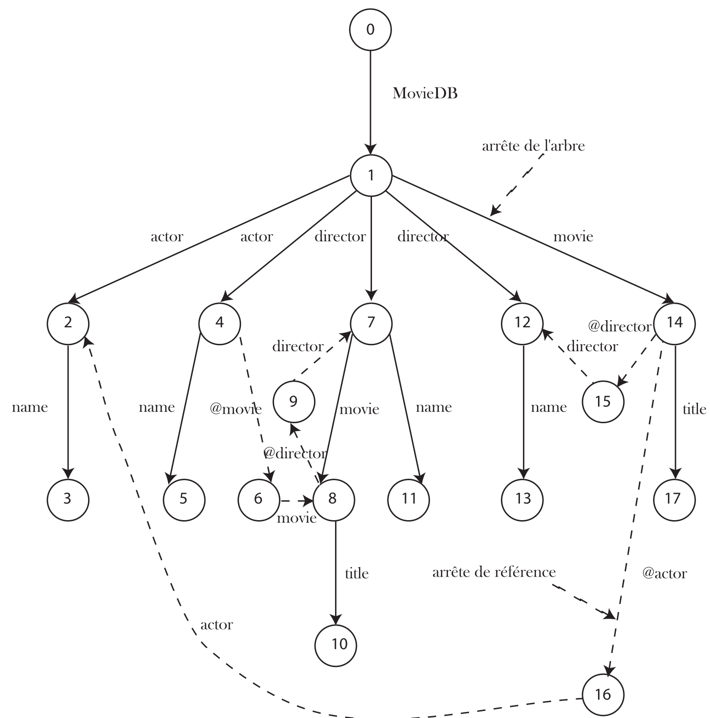


Fig. 2.12 – Graphe de données XML

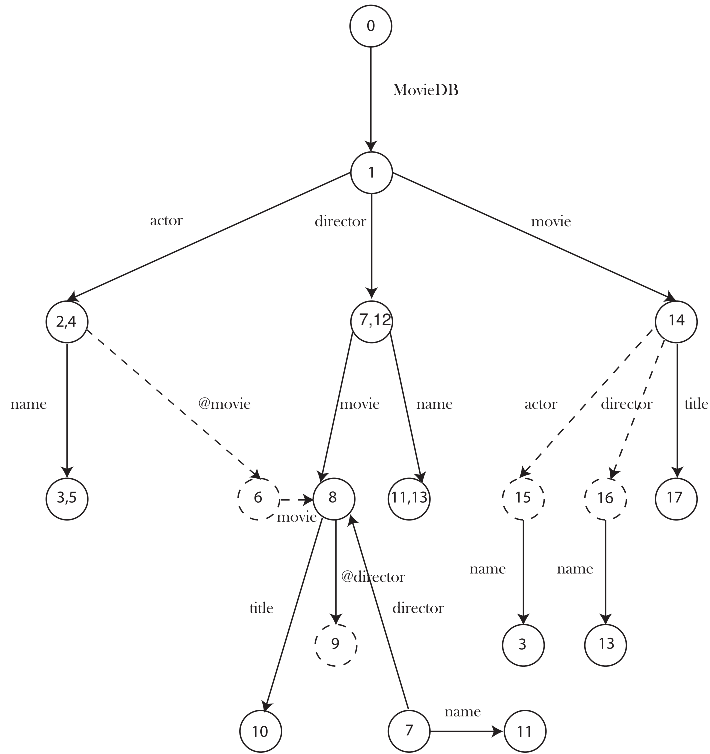


FIG. 2.13 – Guide de données

Les guides de données ont un temps de construction exponentiel et une taille pouvant dépasser la taille des données [YG01]. Le besoin d'une structure d'index est donc nécessaire pour réduire la taille de l'index et le coût d'accès aux données.

2.2.2 1-Index

Un 1-index consiste à regrouper les nœuds ayant le même ensemble de chemins entrants dans le graphe des données XML. Cela se fait par le concept de bi-simulation [KGY]. La relation symétrique binaire de bi-simulation, notée $\approx$, sur les arêtes du graphe est possible pour deux nœuds u et v de ce graphe si :

– les nœuds u et v ont la même étiquette,
– le parent de u est symétrique au parent de v,
– $u \approx v$ alors il existe u' et v' qui, respectivement, pointent vers u et v tels que $u' \approx v'$.

Le l-index pour un ensemble de données XML est défini comme un graphe étiqueté dont les nœuds sont les classes d'équivalences $[v]$ de $\approx$.

Un l-index est plus petit que l'ensemble de données, ce qui facilite l'évaluation d'une requête donnée. Comme le montre la Figure 2.14, un nœud dans le graphe d'index peut avoir plusieurs arêtes sortantes. Cependant, pour opérer rapidement, une sélection d'étiquettes ou une expression de chemins, une indexation par table de hachage ou par un B-arbre de ces étiquettes peuvent être utilisées. Le l-index, comme le guide de données, code les chemins du graphe en incluant des chemins longs et complexes [KGY].

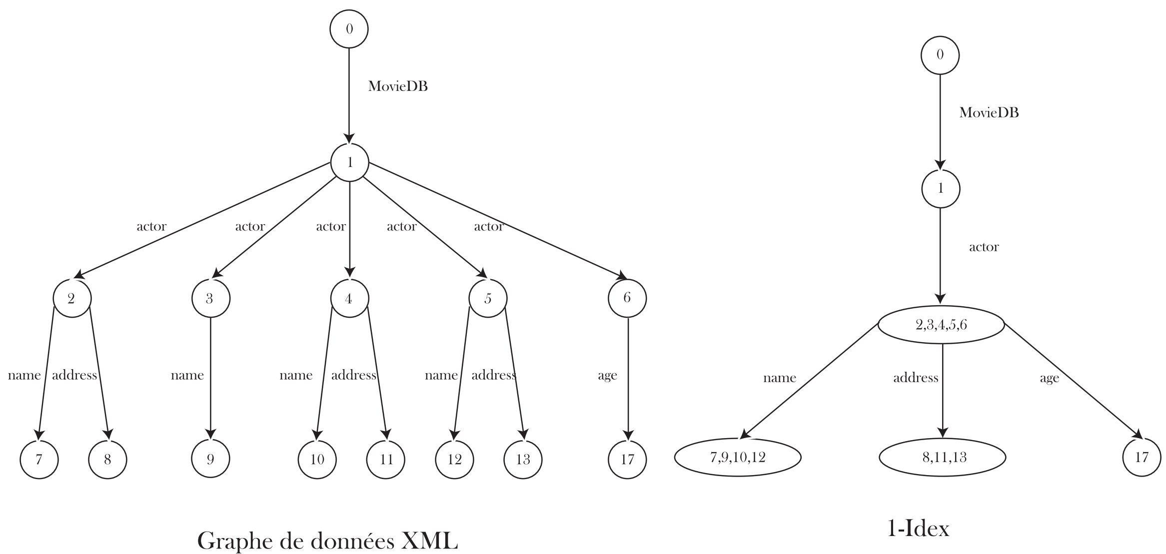


FIG. 2.14 – Exemple d'un 1-index

2.2.3 Index APEX

En général, les index stockent tous les chemins depuis l'élément racine. La taille de ces structures dépend de la profondeur d'imbrication des données XML. Cela risque de dégrader les performances si la profondeur est relativement élevée.

APEX est un index de chemin adaptatif pour les données XML [CMS02, KGY]. Il assure un compromis entre la taille et l'efficacité. Au lieu d'indexer tous les chemins depuis la racine, APEX indexe, dans une structure, seulement les chemins fréquemment utilisés et, dans une autre structure, il maintient les sommets de la structure source. Cela réduit significativement

la taille. L'index APEX, présenté à la Figure 2.15, est composé de deux structures :

- un arbre de hachage H_{APEX}, le chemin entrant à un nœud du graphe G_{APEX}, présenté à la Figure 2.15 (a), et
- un graphe G_{APEX}, une structure pour les données XML, Figure 2.15 (b).

Chaque étiquette du graphe de données XML est conservée dans H_{APEX}, appelé `hnode` (l'attribut label). Chaque `hnode` pointe vers un nœud de G_{APEX} (l'attribut `xnode` de H_{APEX}) ou vers une table de hachage. L'extension, représentée à la Figure 2.15 (c) est l'ensemble d'arêtes dans le graphe de données qui sont assignées pour chaque `xnode` de H_{APEX}.

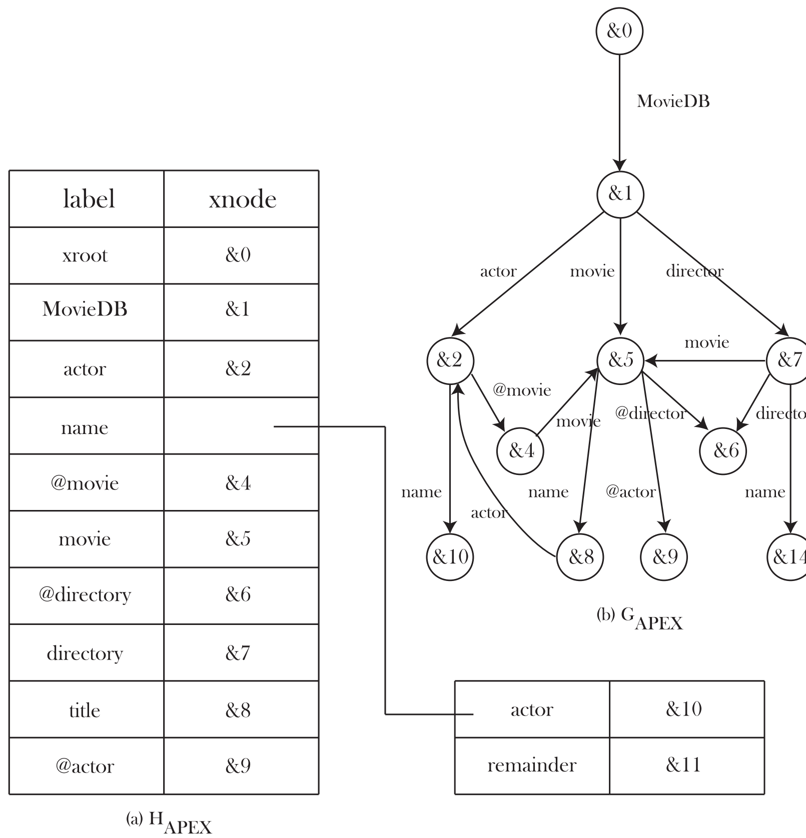


FIG. 2.15 – Exemple d'index APEX

2.2.4 Index FABRIC

L'indexation des données XML par l'index FABRIC s'effectue par un codage de chemins, partant de la racine aux nœuds feuilles [CSF+01]. Ces opérations sont assurées par des indicateurs (*designators*) qui codent les données du chemin en caractères spéciaux ou en chaînes de caractères. Un indicateur unique est donc assigné à chaque balise.

Les caractères obtenus par le codage sont insérés dans un arbre de PATRICIA qui les traite comme de simples caractères. L'index FABRIC permet une gestion efficace de grands nombres de clés complexes. C'est pourquoi, il peut être utilisé pour une recherche de chemins complexes à travers des données XML.

Un dictionnaire d'indicateurs est mis en place pour assurer la correspondance entre les indicateurs et les noms de balise. Ce dictionnaire est construit lors du parcours des données sources pour la construction de l'index. À chaque fois qu'un nouvel élément est ajouté, un nouvel indicateur est ajouté automatiquement. Les noms des balises dans une requête sont aussi transformés en indicateurs en utilisant le dictionnaire pour former une clé de recherche à travers la structure de l'index. La Figure 2.16 (d) montre un exemple d'index FABRIC pour les deux documents XML document 1 et document 2 de la Figure 2.16 (a). Le dictionnaire des indicateurs est illustré à la Figure 2.16 (b). La Figure 2.16 (c) indique le codage des chemins dans les deux documents XML.

La recherche dans l'index consiste en une recherche de chaîne de caractères clés (succession d'étiquettes). Elle commence par le nœud racine puis compare les caractères de la chaîne avec les étiquettes de la structure. Les opérations de mise à jour peuvent être traitées efficacement avec l'index FABRIC. L'ajout d'un nœud dans la structure de l'index consiste en l'ajout d'un seul nœud ou en l'ajout d'une arrête à un nœud existant.

2.3 Matérialisation de vues

Une vue matérialisée est une table contenant le résultat de l'exécution d'une requête donnée. Elle améliore le temps d'exécution des requêtes en pré–calculant les opérations les plus coûteuses comme les jointures et les agrégations, en stockant leurs résultats sur un

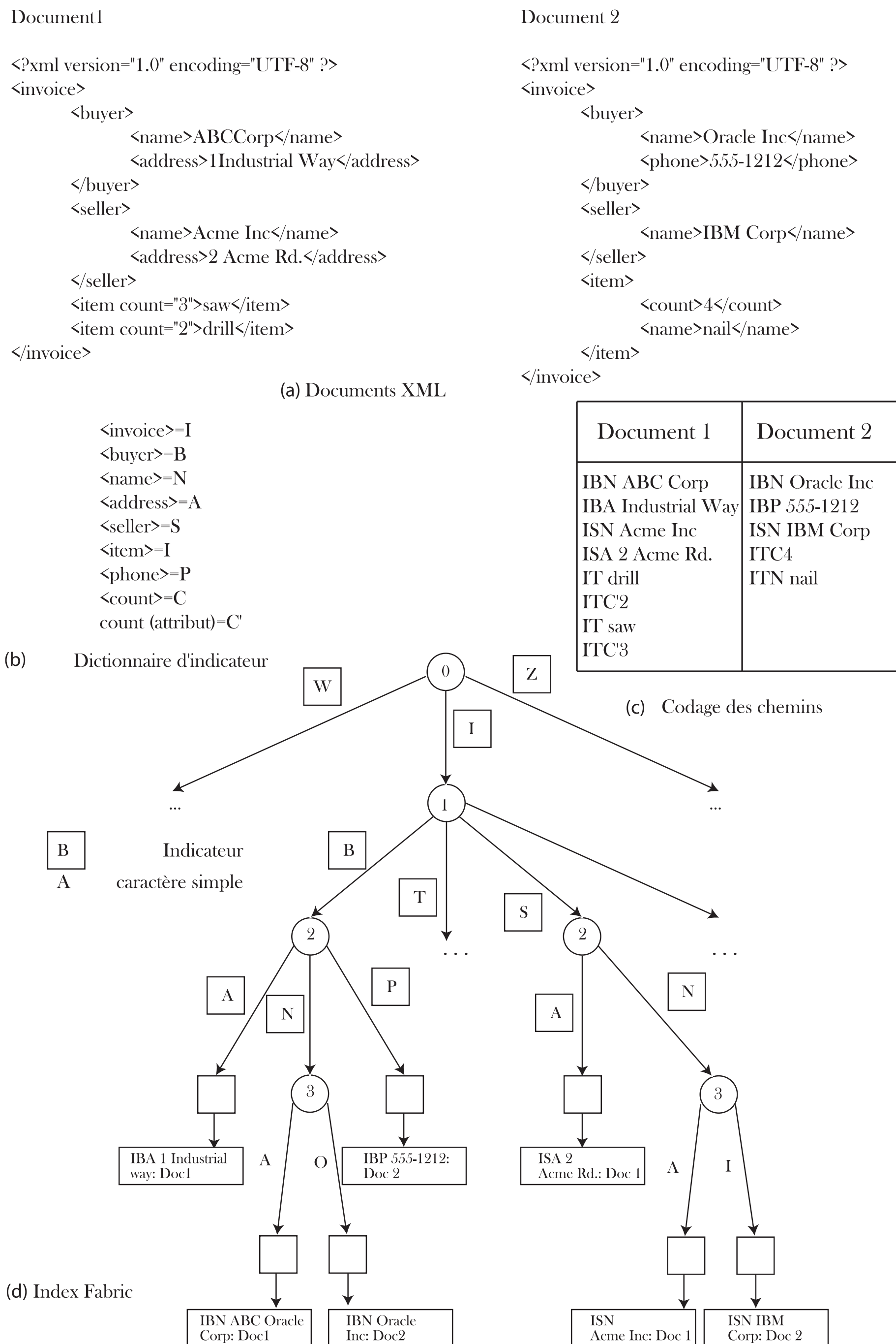


FIG. 2.16 – Exemple d'index FABRIC

support de stockage (matérialisation). De ce fait, l'exécution de certaines requêtes nécessite seulement un accès aux vues matérialisées au lieu des données sources. Cela réduit considérablement le coût d'exécution de ces requêtes car la taille des vues matérialisées est significativement moins importante que celles des données sources. Cependant, la mise à jour des données implique systématiquement celle des vues matérialisées calculées à partir de ces données afin de conserver la cohérence et l'intégrité des données. Cela induit une surcharge du système liée au coût de maintenance des vues matérialisées. De plus, la matérialisation des vues requiert un espace de stockage additionnel que l'administrateur alloue à ces vues.

Nous présentons à la Figure 2.17 un exemple de création d'une vue matérialisée dérivée de l'entrepôt de données de la Figure 2.6 à l'aide du langage SQL.

```
create materialized view mv
as
select      prod_weight, cust_gender, sum(quantity_sold) as sold
from        sales, products, customers
where       sales.prod_id = porducts.prod_id
            and sales.cust_id = customers.cust_id
group by    prod_weight, cust_gender
```

Prod_Weight	Cust_Gender	Sold
10	F	1500
50	F	800
10	M	30
50	M	100

Fig. 2.17 – Exemple de vue matérialisée

Cette vue matérialisée est particulièrement intéressante pour réduire le temps d'exécution de la requête suivante.

```
select      prod_weight, sum(quantity_sold)
```

```
from        sales, products, customers
where       sales.prod_id = porducts.prod_id
            and sales.cust_id = customers.cust_id
            and cust_gender = 'M'
group by    prod_weight
```

En effet, au lieu de réaliser les jointures des tables `Sales`, `Products` et `Customers` et d'en extraire les données vérifiant les conditions de la clause `Where` de cette requête, il suffit d'extraire les données nécessaires à partir de la vue matérialisée. Ceci est moins coûteux que les jointures. En contrepartie, la mise à jour d'au moins une des tables `Sales`, `Products` ou `Customers` implique la mise à jour de cette vue matérialisée.

```
<viewXML>
    <view id='mv'>
        <Cell>
            <fact name='sold' value ='1500' />
            <dimension name='pord_weight' value='10' />
            <dimension name='cust_gender' value='F' />
        </Cell>
        <Cell>
            <fact name='sold' value ='800' />
            <dimension name='pord_weight' value='50' />
            <dimension name='cust_gender' value='F' />
        </Cell>
        <Cell>
            <fact name='sold' value ='30' />
            <dimension name='pord_weight' value='10' />
            <dimension name='cust_gender' value='M' />
        </Cell>
        <Cell>
            <fact name='sold' value ='100' \>
            <dimension name='pord_weight' value='50' />
            <dimension name='cust_gender' value='M' />
        </Cell>
    </view>
</viewXML>
```

Fig. 2.18 – Exemple de vue XML

De nombreux travaux traitent des problématiques concernant les vues matérialisées dans le contexte des entrepôts de données. Nous pouvons distinguer deux axes principaux de recherche :

– la *maintenance incrémentale* des vues matérialisées qui se propose de répercuter les

mises à jour survenues au niveau des données sources sans recalculer complètement les vues ;
- la *sélection des vues* à matérialiser qui propose des algorithmes permettant de déterminer une configuration de vues à matérialiser dans l'entrepôt de données de telle sorte que le coût d'exécution des requêtes soit optimal.

De manière analogue, dans le contexte d'entrepôt de données XML, une vue XML peut être matérialisée sous forme d'un document XML. La Figure 2.18 représente un exemple de vue XML. Cette vue est composée de cellules (élément *Cell*). Chaque cellule est caractérisée par plusieurs dimensions (éléments *dimension*) et les mesures observées sur ces dimensions (élément *fact*).