Chapitre 7

Optimisation des performances des entrepôts de données XML

7.1 Introduction

Les technologies utilisées dans les processus décisionnels, comme les entrepôts de données, l'analyse multidimensionnelle OLAP (*On-Line Analytical Processing*) et la fouille de données, sont désormais très efficaces pour traiter des données numériques ou symboliques. Cependant, diverses sources, dont le Web, présentent des données variées et hétérogènes (textes, images, sons, vidéos ou bases de données). Ces données, qualifiées de complexes [DBRA05], sont largement porteuses d'information et donc intéressantes à traiter au sein d'un processus décisionnel.

L'utilisation d'un format unique est indispensable pour faciliter la représentation de ces données. XML peut fournir une solution à ce problème. En effet, XML est un format d'échange standard pour le Web. La représentation par des documents XML de données complexes et leur modélisation multidimensionnelle peut *a priori* naturellement mener vers la conception et la construction d'entrepôts de données XML. Dans ce contexte, garantir une bonne performance des requêtes décisionnelles est un objectif crucial. Pour cela, des stratégies d'indexation et d'utilisation de vues matérialisées s'avèrent indispensables.

Nous traitons dans ce chapitre l'optimisation des performances des entrepôts de données

XML. Nous nous basons sur un entrepôt de données XML de référence, modélisé selon la spécification XCube. Nous proposons une structure d'index et une stratégie de sélection de vues pertinentes à matérialiser [Mah05]. L'index proposé permet d'éviter le calcul des jointures en maintenant efficacement les données de l'entrepôt dans sa structure de données. La stratégie de sélection de vues matérialisées XML est basée sur la classification non supervisée des requêtes XQuery d'une charge donnée. Dans ce cadre, nous adaptons nos travaux traitant de la sélection de vues matérialisées (cf. Chapitre 5) au contexte des entrepôts de données XML.

Ce chapitre est organisé comme suit. Nous présentons à la Section 7.2 l'architecture de notre entrepôt de données XML de référence. Nous détaillons à la Section 7.3 notre approche pour l'optimisation des performances des requêtes décisionnelles, à savoir notre index de jointure XML et la sélection de vues XML. Nous concluons enfin à la Section 7.4.

7.2 Architecture de l'entrepôt de données XML de référence

7.2.1 Schéma conceptuel de l'entrepôt de données de référence

Cette section présente un court état de l'art sur la modélisation des entrepôts de données XML afin de sélectionner un entrepôt de données XML de référence respectant une modélisation en étoile.

7.2.1.1 XCube

XCube utilise des documents XML pour le stockage, l'échange et l'interrogation des données d'un ou plusieurs cubes de données [HBH03]. XCube est organisé en fusion de modules ou formats : Schéma XCube, Dimensions XCube et Faits XCube.

Schéma XCube

Le schéma XCube est le format central pour décrire la structure multidimensionnelle d'un cube de données. Il modélise les dimensions et les mesures contenues dans le cube. La structure type d'un document de schéma XCube est décrite à la Figure 7.1.

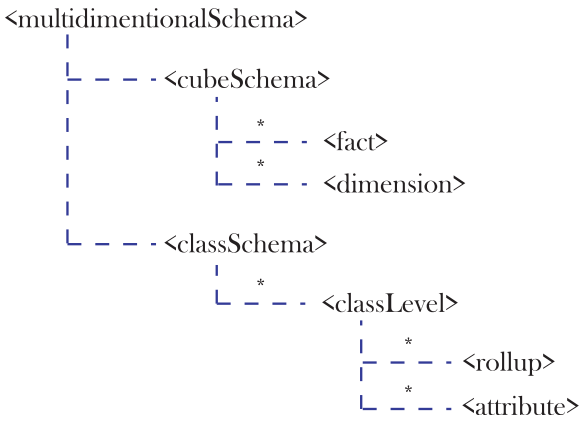

FIG. 7.1 – Schéma XCube

Les lignes discontinues de la Figure 7.1 représentent les relations père-fils et le symbole * indique que le nombre d'occurrences du fils peut être arbitraire, c'est-à-dire zéro ou plusieurs.

Il existe sous la racine *MultidimentionalSchema* deux blocs (sous-éléments) : *cubeSchema* et *classSchema*. Le bloc *cubeSchema* contient une collection de faits (l'élément *fact*) et de dimensions (l'élément *dimension*). L'élément *dimension* possède des attributs pour référencer la dimension et sa granularité la plus fine. L'élément *fact* possède un attribut qui référence le nom du fait. La section *classSchema* décrit la classification des niveaux des dimensions par l'élément *classLevel*. Ce dernier est constitué de l'élément *attribute* qui détermine l'attribut de la dimension et de l'élément *rollup* qui pointe vers le niveau supérieur dans la hiérarchie des dimensions.

La connexion entre la dimension et la classification des hiérarchies est établie par une référence depuis *cubeSchema/dimension* vers *classSchema/classLevel*. La notation a/b signifie que l'élément *b* est le fils de l'élément *a*.

Le schéma *XCube* peut aussi fournir les définitions des unités et des types des mesures, la classification des dimensions ainsi que celles des nœuds et des attributs par un schéma XML (*XML schema*). Dans ce schéma, des opérations d'agrégation peuvent être définies sur les mesures du cube.

Dimension XCube

Le format Dimension *XCube* formalise la structure des dimensions. Un document de dimension *XCube* contient des nœuds appartenant à la classification des niveaux définie sur le schéma *XCube*. La structure de base d'un document *Dimension XCube* est présentée à la Figure 7.2.

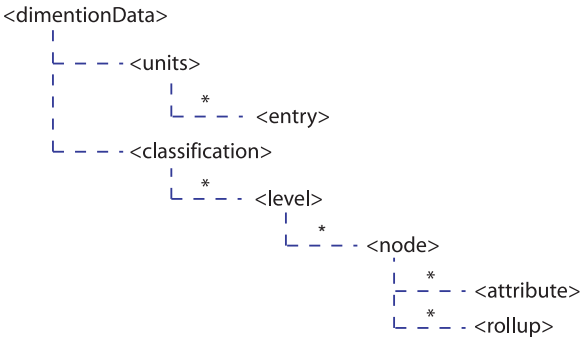

FIG. 7.2 – Schéma Dimension XCube

La racine *dimensionData* a deux fils : l'élément *units*, importé depuis les unités définies dans le schéma XCube et l'élément classification. Chaque élément *Level* est composé d'éléments *node*. Ces éléments décrivent les attributs (élément *attribute*) des dimensions et leurs hiérarchies (élément *rollup*).

Faits XCube

Un fait *XCube* définit la collection de cellules d'un cube de données. Chaque cellule est constituée, comme le montre la Figure 7.3, de deux sous-éléments : *dimension*, représentant les coordonnées multidimensionnelles et *fact* renseignant la valeur des mesures du fait.

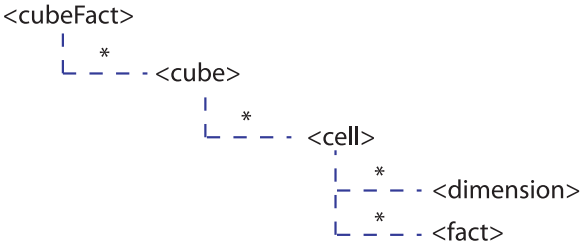

FIG. 7.3 – Schéma Fact XCube

En plus de la description des données du cube, *XCube* contient d'autres formats. *XCube-Text* fournit des descriptions textuelles et des commentaires sur les formats schéma *XCube* et dimension XCube. *XCubeQuery* est un moyen d'échange d'informations entre un serveur et un client. Il fournit au site du client les données nécessaires à un besoin défini. Il ne constitue pas un langage d'interrogation. *XCubeFunction* est en cours de développement. Il devrait permettre d'interroger un serveur XCube afin d'obtenir des cubes de données complets.

7.2.1.2 DAWAX (DAta Warehouse for XML)

DAWAX est une architecture générale pour entreposer des documents XML [BB00, BB03a]. L'entrepôt de documents XML est défini comme un ensemble de vues matérialisées XML. Le système se base sur trois modules principaux :

1. un module de spécification de l'entrepôt de données, utilisé pour sa conception ;
2. un module d'implémentation, utilisé pour le stockage des données XML dans une base de données relationnelle, la gestion de l'extraction des données et leur maintenance ;
3. un module de gestion de requêtes pour l'interrogation de l'entrepôt.

Module de spécification de l'entrepôt de données

L'entrepôt est constitué d'un ensemble de vues XML. Deux sortes de vues peuvent être distinguées : des vues de sélection et des vues composées qui créent de nouveaux éléments et attributs à partir de plusieurs sources. Des fonctions d'agrégation sont utilisées pour définir les nouvelles valeurs.

Une vue est aussi associée à une DTD (*Document Type Definition*). Elle est construite à partir de la définition de la vue ou des DTD sources. La vue est constituée :

1. d'un modèle résultat qui spécifie la structure du résultat. Ce modèle utilise des variables de fragments qui sont des collections de modèles (*patterns*) définissant la même variable dans différentes sources et fournissent l'union de ces données ;
2. l'élément Jointure pour relier les fragment ;
3. l'élément `Group by` pour le regroupement des résultats.

Module de gestion des métadonnées

La spécification d'un entrepôt de données est stockée dans un document XML. Ce document contient les métadonnées de l'entrepôt, qui définissent des informations sur le stockage des données, leur localisation (références, URL - *Uniform Resource Locator*) des sources des données et la spécification des vues.

Afin de fournir une intégration de vues sources hétérogènes, l'entrepôt de données est considéré comme un document XML contenant les résultats de toutes les vues, comme l'illustre la ligne suivante de la DTD : `< !ELEMENT datawarehouse (view1, view2, . . . , viewN) >`.

Module de stockage et gestion des données de l'entrepôt

Le stockage des données s'effectue par *mapping* dans une base de données relationnelle. Il en est de même pour les vues, grâce à trois tables : `Patterns` pour le stockage des modèles, `Fragments` pour le stockage des fragments et `Views` pour le stockage des vues.

7.2.1.3 Travaux de Pokorný

Pokorný adopte le schéma en étoile classique [KR02, Inm02] d'un entrepôt de données pour l'environnement XML [Pok02]. La Figure 7.4 montre la modélisation en étoile d'un entrepôt de données XML selon Pokorný.

Les dimensions sont des documents XML. Au niveau conceptuel, une dimension particulière est modélisée comme une séquence de DTD qui sont logiquement associées. Ce lien logique est similaire à l'intégrité référentielle définie dans les bases de données relationnelles.

L'approche proposée commence par reconstruire la hiérarchie des dimensions à partir de n collections de documents XML $C_1, \ldots, C_n$, dotées des DTD $DTD_1, \ldots, DTD_n$, respectivement. Pokorný suppose que chaque collection de documents XML est localement homogène, et donc dotée d'une seule DTD. Chaque DTD peut être la source d'un ou plusieurs niveaux de hiérarchie d'une dimension D. Les éléments PCDATA (*Description Parsed Character Data*) de la DTD, choisis par l'utilisateur, sont utilisés pour décrire la dimension D. Ces éléments extraits de la DTD constituent une sous-DTD, dénotée DTD_D, de la dimension

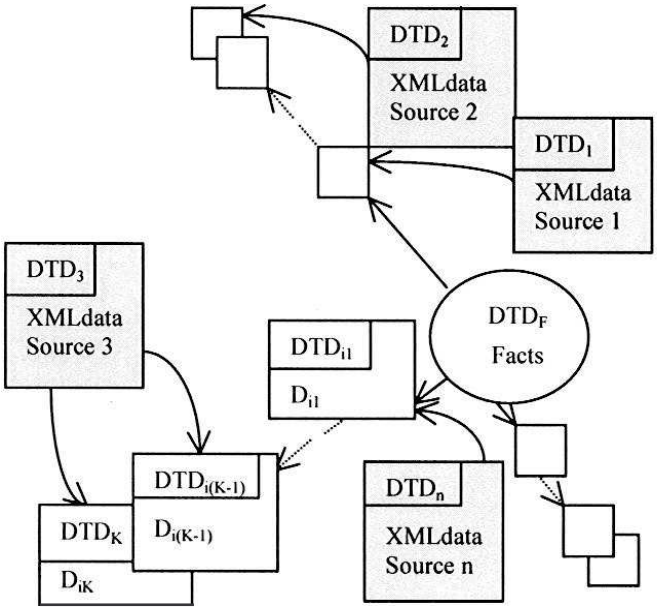

FIG. 7.4 – Schéma en étoile d'un entrepôt de données XML proposé par Pokorný

D.

Les vues XML sont utilisées pour modéliser les associations qui peuvent exister entre deux documents XML décrits par deux DTD différentes. Ces vues aident à vérifier l'intégrité référentielle et accélèrent l'exécution des requêtes.

Le détail de la construction des vues XML et des hiérarchies des dimensions est publié dans [Pok01, Pok02]. En revanche, aucun détail sur la construction des faits n'est donnée, mis à part que les faits sont aussi modélisés sous forme d'un document XML.

7.2.1.4 Discussion

XCube permet, à la différence des deux autres approches que nous avons recensées, une modélisation en étoile ou en constellation simple d'un entrepôt de données XML. XCube permet en effet de spécifier le schéma (métadonnées), les dimensions et les faits par des documents XML. Ces documents permettent une interrogation à l'aide de requêtes décisionnelles. La méthodologie utilisée pour la construction de DAWAX, qui correspond plutôt à un entreposage de *documents* XML qu'à un entrepôt de *données* XML, ne prend pas en charge les concepts de la modélisation multidimensionnelle. En outre, DAWAX utilise un

mapping avec une base de données relationnelles pour le stockage des données XML ; ce qui écarte la possibilité d'interrogation avec un langage XML. Or, ce dernier point est le propos de notre étude. Bien que l'approche de Pokorný penche vers une modélisation en étoile, elle reste dépendante de la DTD des données sources à entreposer et se révèle difficile à mettre en œuvre. XCube semble donc offrir une bonne modélisation de l'entrepôt de données afin d'adapter notre stratégie d'optimisation de performance dans l'environnement d'entrepôts de données XML.

Dans la suite de chapitre, nous choisissons de modéliser notre entrepôt de données selon la spécification XCube. Notre entrepôt de données XML de référence (Figure 7.5) est composé de trois documents XML :

1. `schema.xml` où est spécifie schéma de l'entrepôt de données (métadonnées) ;
2. `dimensions.xml` où sont définies les dimensions ;
3. `facts.xml` où sont spécifiés les faits.

FIG. 7.5 – Extrait d'entrepôt de données XML selon la spécification XCube

7.2.2 Stockage et interrogation des données XML

L'une des techniques généralement mises en œuvre pour la gestion de données XML est de réaliser un *mapping* vers une base de données relationnelle. Ce *mapping* produit une représentation non normalisée des données qui peut provoquer une explosion du nombre de tables à cause de la flexibilité des données XML. De plus, les requêtes définies sur les données XML formalisées dans un langage d'interrogation de données XML doivent être réécrites en SQL pour être supportées par le système de gestion de base de données relationnelle.

Dans ce cadre, des travaux ont été entrepris pour le stockage et la gestion des données XML dans leur format d'origine. Les bases de données natives XML sont plus particulièrement conçues pour stocker des documents XML. Le terme "native" signifie essentiellement que les documents XML sont enregistrés, indexés et accédés dans leurs formats d'origine tout en préservant leur contenu, balises, références et ordres d'imbrication. Il existe plusieurs produits de bases de données natives XML. Nous pouvons citer à titre d'exemple TIMBER [JAKC+02] et eXist [Mei05].

Concevoir un entrepôt de données XML revient à utiliser une base de données native pour le stockage des documents XML constituant l'entrepôt de données. Il est donc nécessaire d'envisager une base de données native XML qui permet de gérer des collections de documents XML et d'interroger les données par un langage de requête XML. Notre choix se porte sur eXist pour stocker notre entrepôt de données de référence, car il est libre et fournit un mécanisme pour le stockage des documents XML hiérarchisés en collections. Toutefois, d'autres systèmes ayant les mêmes spécificités peuvent remplir ce rôle. Notre souci majeur est de montrer la faisabilité et l'adaptabilité de notre stratégie d'optimisation dans un contexte d'entreposage de données XML.

7.2.2.1 Bases de données natives XML

Une base de données native XML définit un modèle logique comme DOM (*Document Object Model*) [Heg05] pour stocker et interroger ces données. C'est une base de données orientée modèle. Le modèle doit contenir au moins les éléments, les attributs et l'ordre d'imbrication du document XML.

Par conséquent, les données XML peuvent être représentées par une structure en arbre

ou en graphe dans le cas où les données présentent des références. La structure a pour nœuds des éléments, dits balises, et parfois, des attributs de ces nœuds. Le contenu des nœuds est représenté par des nœuds feuilles dans la structure.

Un schéma d'étiquetage est aussi utilisé pour coder les nœuds de la structure. Ce schéma assigne à chaque nœud un identifiant. Le schéma est généré soit par un parcours descendant, soit par un parcours ascendant de la structure du modèle [LM01]. Le système TIMBER associe à chaque nœud de la structure une étiquette `(Start, End, Level)`. Ces étiquettes permettent de mieux déterminer les relations père-fils dans la structure :

1. un nœud (S_1, E_1, L_1) est l'ancêtre du nœud (S_2, E_2, L_2) si et seulement si $S_1 < S_2$ et $E_1 > E_2$;
2. un nœud (S_1, E_1, L_1) est un parent du nœud (S_2, E_2, L_2) si et seulement si $S_1 < S_2$ et $E_1 > E_2$ et $L_1 = L2 - 1$, où S_1 et S_2 sont des étiquettes `Start`, E_1 et E_2 des étiquettes `End`, et L_1 et L_2 des étiquettes `Level`.

En plus du stockage des données, les systèmes de gestion de bases de données natifs XML stockent des métadonnées sur les données XML. Ces informations concernent le type des données, la taille des données, l'ordre des éléments, etc. Certaines d'entre elles peuvent être générées à partir de la DTD ou du schéma XML.

7.2.2.2 Interrogation de l'entrepôt de données XML

La majorité des SGBD natifs XML supportent un ou plusieurs langages de requêtes. Les langages de requêtes les plus répandus sont XPath [CD05] et XQuery [BCF+05]. XPath présente certaines limites parce qu'il n'a pas vraiment été conçu comme un langage d'interrogation de base de données. XPath ne peut ni regrouper des données, ni les trier, ni effectuer de jointures de documents. Il ne supporte pas non plus le typage de données. XQuery est un langage de requêtes plus complet qui a été implémenté par plusieurs constructeurs de base de données natives XML. Il se base sur XPath pour extraire et manipuler des fragments de documents XML. Les requêtes basiques de XQuery sont identiques à celles définies dans XPath. XQuery est intéressant dès lors que l'on désire définir des requêtes complexes avec des expressions du type FLWR (*For Let Where Return*) ou encore en faisant appel à la

récursivité. Plus encore, XQuery peut également exprimer des requêtes sur plusieurs documents à la fois, c'est-à-dire des opérations de jointure qui sont communément utilisées pour définir des requêtes décisionnelles.

La Figure 7.6 donne un exemple de requête XQuery de type FLWR. Cette requête retourne la somme des quantités de ventes des clients demeurant à la ville de Lyon. Elle réalise une jointure entre les documents `dimensions.xml` et `facts.xml`, une sélection et une opération d'agrégation `sum` sur la mesure quantité.

```
for $a in //dimensionData/classification/Level[@node='CUSTOMERS']/node,
      $x in //CubeFacts/cube/Cell
let $y:=$x/fact[@id='quantity']/@value
      where $a/attribute/@name='CUST_CITY'
            and $a/attribute/@value='Lyon'
            and $x/dimension/@node=$a/@id
            and $x/dimension/@id='CUSTOMERS'
return sum($y)
```

FIG. 7.6 – Exemple de requête FLWR

7.3 Optimisation de performance des entrepôts de données XML

Les deux documents XML `facts.xml` et `dimensions.xml` utilisés pour stocker les faits et les dimensions de notre entrepôt de données XML de référence sont volumineux. L'interrogation de ces données en XQuery est très coûteuse en terme de temps de calcul. Ce coût devient prohibitif lorsque les requêtes effectuent plus de deux opérations de jointure. Il est alors crucial de le réduire.

Nous proposons deux apports pour l'optimisation des performances d'un entrepôt XML : une technique d'indexation pour réduire le coût des jointures et l'adaptation de notre stratégie de sélection de vues matérialisées au cas des entrepôts de données XML.

7.3.1 Index de jointure pour les entrepôts de données XML

Les techniques d'indexation des données XML telles que le guide de données, le 1-index et APEX, présentées au Chapitre 2, Section 2.2, s'avèrent inadaptées pour les requêtes décisionnelles. En effet, ces structures ne sont applicables que sur des données XML ciblées par des requêtes définies en expressions de chemin simples. Or, dans le contexte des entrepôts de données XML, les requêtes sont complexes et comportent plusieurs expressions de chemin. De plus, ces index opèrent sur un seul document. Cet aspect est pénalisant car les requêtes définies sur notre entrepôt de données XML opèrent sur plusieurs documents XML afin de réaliser les jointures entre les faits du document `facts.xml` et les dimensions du document `dimensions.xml`. FABRIC indexe plusieurs documents XML à la fois. Cependant, il n'est pas non plus adapté pour l'indexation des entrepôts de données XML. En effet, FABRIC intègre dans sa structure plusieurs documents sans prendre en compte les relations, telles que les jointures, qui peuvent exister entre ces documents. Cet index n'est donc pas bénéfique aux requêtes décisionnelles. C'est pourquoi nous proposons un index de jointure XML dans lequel nous précalculons les jointures entre les faits et les dimensions. Notre index, de part son principe, ressemble aux index de jointure proposés dans les entrepôts de données relationnels.

7.3.1.1 Structure de notre index de jointure

Notre index (Figure 7.7) présente une structure similaire à celle du document `facts.xml`, à l'exception de l'élément *attribute*. Les étiquettes qui commencent par le caractère @ représentent les attributs et les autres représentent les éléments. Chaque cellule est identifiée par des dimensions et un ou plusieurs faits. Un fait (élément *Fact*) possède deux attributs, @*id* et @*value*, qui indiquent respectivement son nom et sa valeur. Chaque dimension (élément *dimension*) est identifiée par deux attributs : @*id* qui donne le nom de la dimension et @*node* qui donne la valeur de l'identifiant de la dimension. De plus, l'élément dimension possède un certain nombre d'éléments fils (éléments *attribute*). Ces éléments servent à stocker les noms et les valeurs des attributs de chaque dimension. Ils sont obtenus depuis le document des dimensions `dimensions.xml`. Un élément *attribute* est caractérisé par deux attributs, @*nom* et @*value*, qui indiquent respectivement le nom et la valeur de chaque attribut. Le nombre d'éléments *attribute* dépend du nombre d'attributs de chaque dimension.

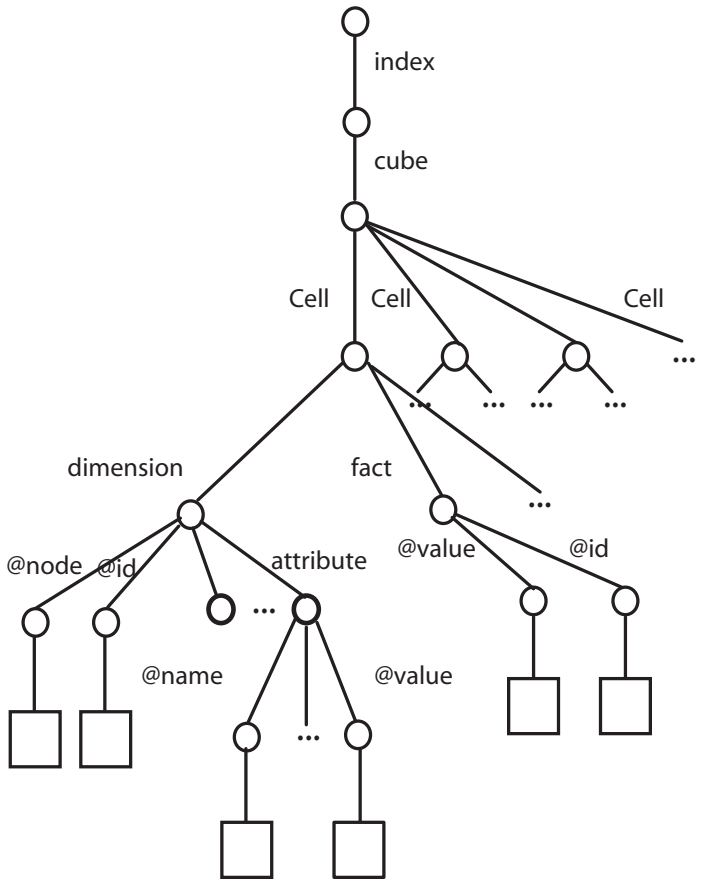

FIG. 7.7 – Structure de notre index de jointure XML

La migration des données du document `dimensions.xml` et du document `facts.xml` vers la structure d'index, et le fait de stocker pour une même cellule les faits, les dimensions et leurs attributs, nous permet d'éliminer les opérations de jointure. Toutes les informations nécessaires pour une jointure sont en effet stockées dans la même cellule.

7.3.1.2 Exécution de requête avec et sans utilisation de notre structure d'index

Une requête type définie sur un entrepôt de données XML modélisé selon la spécification XCube réalise plusieurs jointures entre les faits stockés dans `facts.xml` et les dimensions de `dimensions.xml`. La Figure 7.8 représente ces deux documents sous forme d'un graphe XML. Les éléments @*id* et @*node* de la Figure 7.8 (a) correspondent respectivement aux

identifiants des dimensions et aux noms de ces dimensions. Dans un contexte relationnel, @*node* correspondrait à une dimension et @*id* à un identifiant d'une dimension et *node* à un n-uplet d'une dimension. Les @*id* et @*node* de la Figure 7.8 (b) correspondent respectivement aux noms des dimensions et les références de ces dimensions correspondent à chaque cellule du cube. Dans un contexte relationnel, @*id* correspondrait à une dimension et @*node* à une clé étrangère de la table de faits.

Une jointure doit alors s'assurer que les éléments @*node* et @*id* de `dimensions.xml` sont respectivement égaux à @*id* et @*node* de `facts.xml`. Si plusieurs jointures sont définies, il faut alors vérifier les contraintes comme suit : `facts.@id = dimensions.@node` et `facts.@node = dimensions.@id`. La première égalité vérifie que la dimension composant une cellule est bel et bien la dimension exprimée dans la requête. La seconde vérifie que le nœud d'une dimension (clé primaire) correspond (peut être joint) au nœud, de la même dimension, défini dans une cellule (clé étrangère de la table de faits).

L'exécution d'une requête sans utilisation de notre index, peut se dérouler comme suit. Pour chaque dimension définie par `@node = 'nom de la dimension'`, les identifiants @*id* vérifiant la clause `Where` sont recherchés. Le document `dimensions.xml` est parcouru en profondeur, jusqu'au nœud *Level*. Les fils *node* du nœud *Level* sont ensuite parcourus en largeur de manière à trouver le nœud dont la valeur de @*node* est égale au nom de la dimension spécifié dans la requête. Le coût de ce parcours est égal au nombre des noeuds *Level* du document `dimensions.xml` ; c'est-à-dire, le nombre de de dimensions dans le schéma, dénoté $|dimension|$. Si plusieurs dimensions sont définies, alors le parcours donne lieu à autant de nœuds que de dimensions. En revanche, le coût de ce parcours reste invariant car tous les nœuds *Level* sont parcourus. Pour chaque nœud trouvé, ses fils sont parcourus en profondeur de manière à trouver la liste des @*id* vérifiant les conditions `@name = 'nom de l'attribut'` et `@value = 'valeur de l'attribut'`. Le coût de ce parcours est égal au nombre de fils `attribute`. En résumé, le coût de traitement des dimensions est égal à $|a_i| * |d_i|$, où $|a_i|$ est le nombre d'attributs de chaque dimension, $|d_i|$ le nombre d'éléments *node*, c'est-à-dire le nombre de fils de chaque dimension, et i une dimension donnée.

Pour réaliser une jointure entre les dimensions du document `dimensions.xml` et les faits du document `facts.xml`, les @*id* retrouvés dans le traitement des dimensions sont recherchés dans les faits. Rappelons que ces @*id* correspondent aux éléments @*node* des

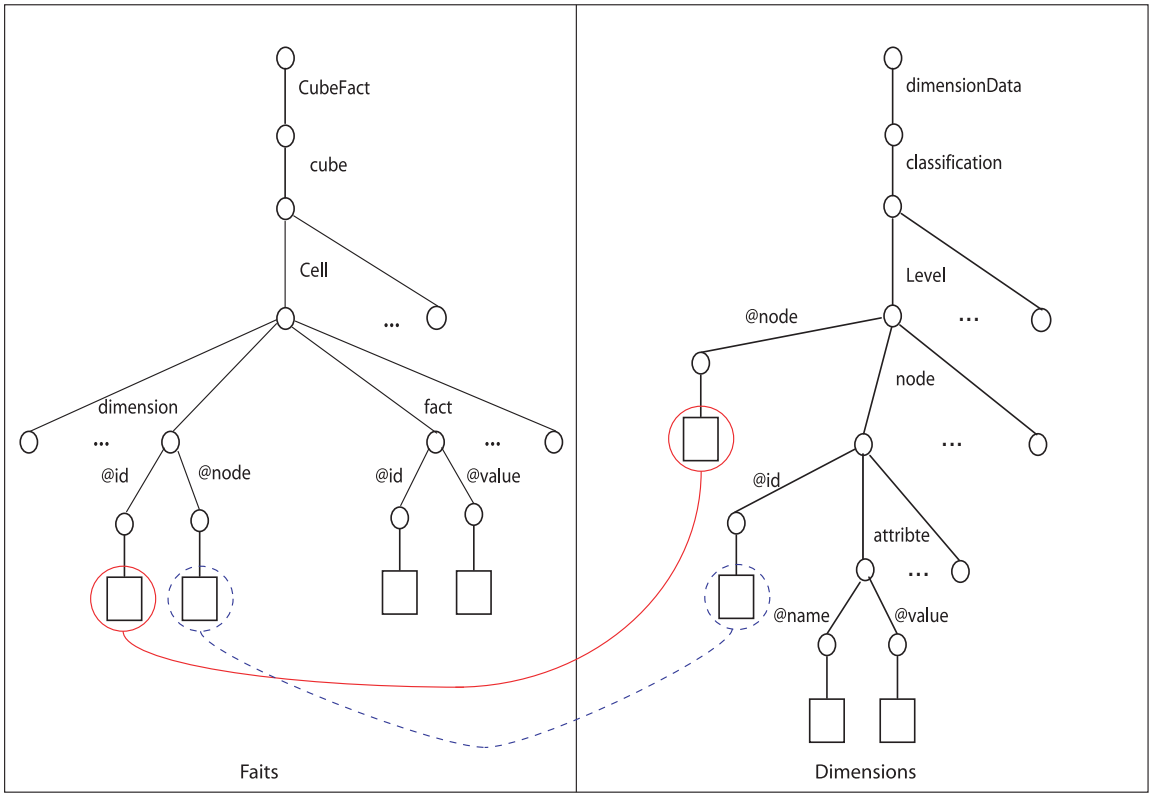

FIG. 7.8 – Graphe XML des documents `facts.xml` et `dimensions.xml`

faits. Le traitement des faits dans le but de réaliser des jointures se déroule comme suit. Le document `facts.xml` est parcouru en profondeur jusqu'au niveau *Cell*. Les cellules sont ensuite parcourues en largeur afin de trouver les dimensions dont le fils @*id* est égal à @*node* de `dimensions.xml` et @*node* est égal à @*id* de `dimensions.xml`. En résumé, le coût de traitement du document `facts.xml` est $|Cell|$, où $|Cell|$ est le nombre de cellules du documents `facts.xml`. Le coût de traitement d'une requête est donné par la formule suivante.

$$C_{sans\_index} = ((|Cell|) * |Dimension|) * (|Dimension| + (|d_i| * |a_i|))$$

Il est nécessaire de réécrire les requêtes afin qu'elles puissent exploiter notre index. La réécriture de ces requêtes consiste à conserver les expressions de sélection et les opérations d'agrégation. Nous illustrons l'exécution d'une requête par l'exemple de la Figure 7.9. Notre

index est stocké dans le document XML `index.xml` avec l'élément *Index* comme racine.

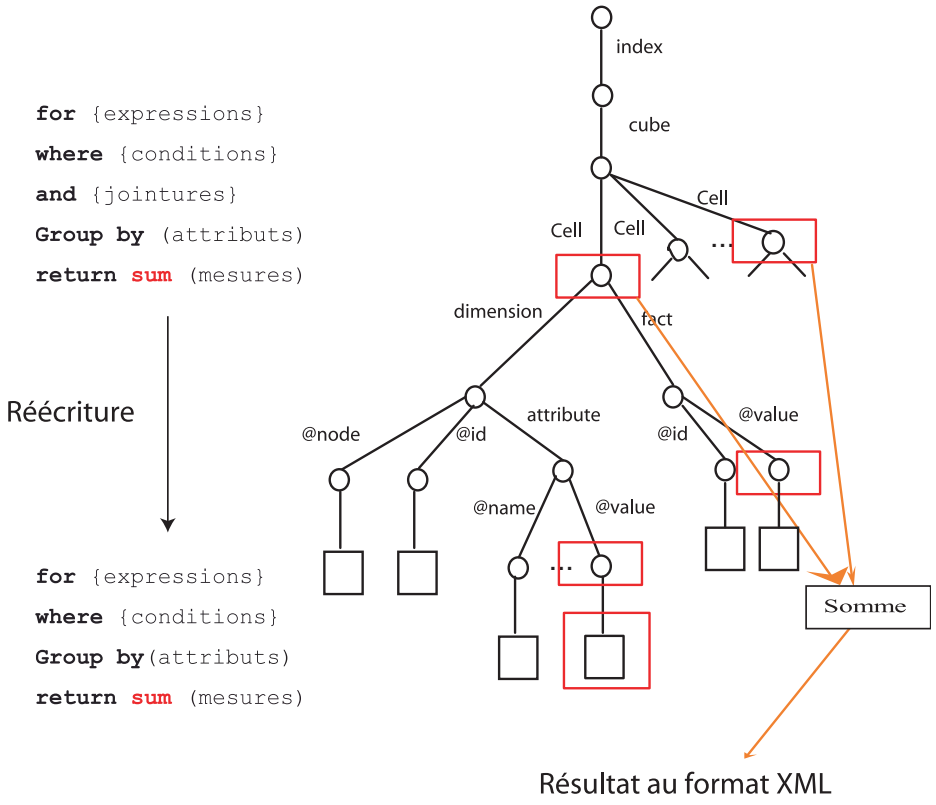

FIG. 7.9 – Réécriture de requête exploitant notre index de jointure XML

L'exécution d'une requête, avec utilisation de notre index, peut se dérouler comme suit. Pour chaque dimension définie par `@node = 'nom de la dimension'`, les identifiants @*id* vérifiant la clause *Where* sont recherchés. Le document `index.xml` est parcouru en profondeur, jusqu'au nœud *Cell*. Le coût de ce parcours est égal au nombre de cellules dans le document `index.xml`. Les fils *dimension* du nœud *Cell* sont ensuite parcourus en largeur de manière à trouver le nœud dont la valeur de @*id* est égale au nom de la dimension spécifiée dans la requête. Le coût de ce parcours est égal au nombre de nœuds *dimension* du document `index.xml` ; c'est-à-dire, le nombre de de dimensions dans le schéma, |*dimension*|. Pour chaque nœud trouvé, ses fils sont parcourus en profondeur de manière à trouver le nœud *attribute* vérifiant les conditions `@name = 'nom de l'attribut'` et `@value = 'valeur de`

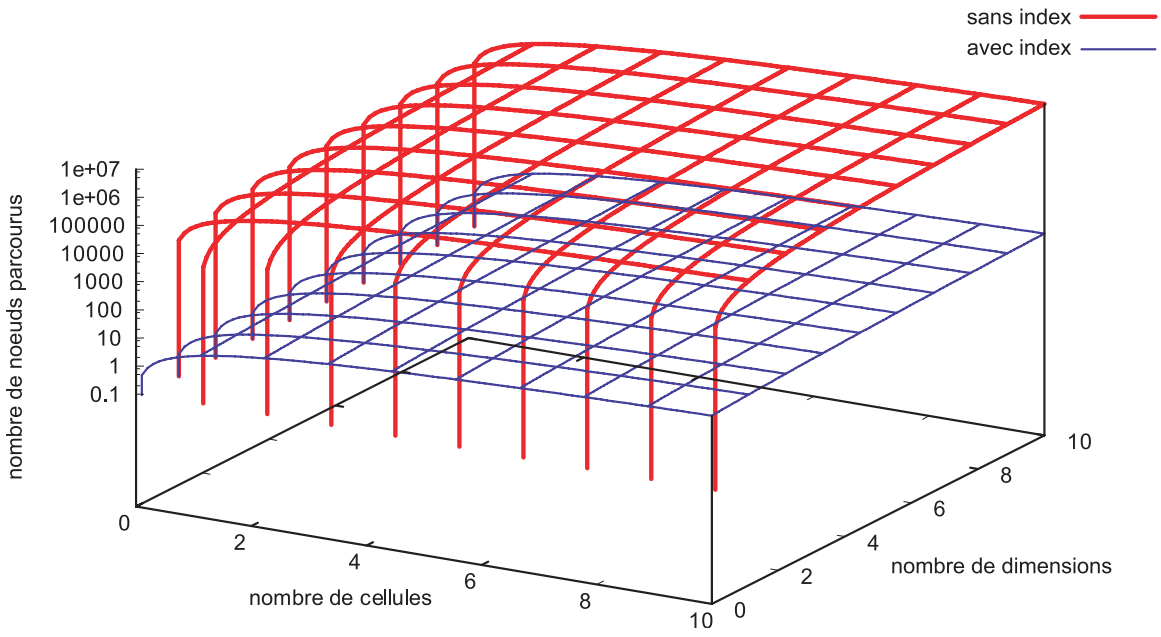

FIG. 7.10 – Variation des coûts C_{sans_index} et C_{index} en fonction du nombre de cellules et de dimensions

`l'attribut'`. Le coût de ce parcours est égal au nombre de fils *attribute*, dénoté $|a_i|$. En résumé, le coût de traitement d'une requête qui exploite notre structure d'index est donné par la formule suivante.

$$C_{index} = |Cell| * (|Dimension| + |a_i|)$$

La Figure 7.10 représente la variation des coûts (nombre de nœuds parcourus) C_{sans_index} et C_{index} en fonction du nombre de cellules et de dimensions dans l'entrepôt de données XML test (Section 7.3.1.3). Nous avons fixé la cardinalité moyenne des dimensions à 12394, car cette valeur correspond à la valeur moyenne des cardinalités des dimensions de notre entrepôt test. Nous représentons également à la Figure 7.11 ces coûts en fonction du nombre de cellules et de la cardinalité moyenne des dimensions. Nous avons fixé le nombre de dimensions à cinq, car ce nombre correspond au nombre de dimensions de notre entrepôt test. Nous utilisons une échelle logarithmique pour représenter le coût afin de mieux visualiser la différence entre ces coûts. Dans les deux cas, nous constatons que le coût avec index C_{index} est plus petit que le coût sans index C_{sans_index}. Cela montre qu'en théorie notre index est efficace pour

réduire le coût des jointures.

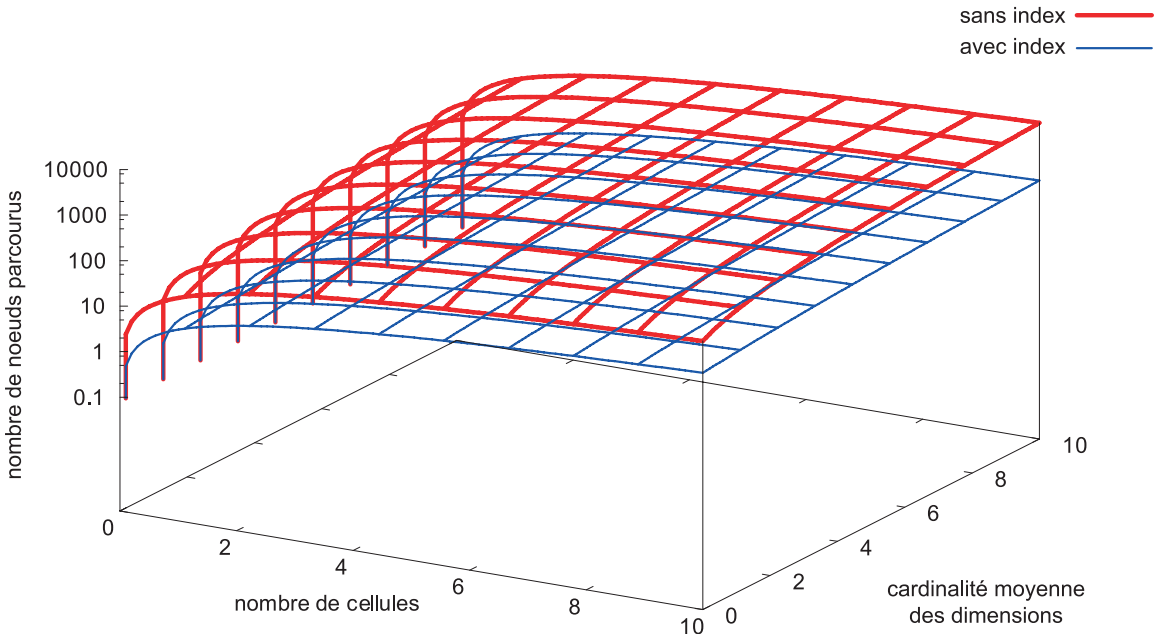

FIG. 7.11 – Variation des coûts C_{sans_index} et C_{index} en fonction du nombre de cellules et de la cardinalité des dimensions dans l'entrepôts de données XML

La Figure 7.12 représente une coupe verticale de la courbe représentée à la Figure 7.10 pour la valeur cinq (nombre de dimensions de notre entrepôt test) de l'axe `nombre dimension`. Le coût avec index C_{index} est en moyenne 140000 fois plus petit que le coût sans index C_{sans_index}.

7.3.1.3 Expérimentation

En complément de notre étude théorique, nous avons effectué des expérimentations afin de tester l'efficacité de notre proposition de structure d'index. Nous avons généré notre entrepôt de données XML de référence, modélisé selon la spécification XCube, à partir de notre entrepôt de données relationnel test (cf. Chapitre 4, Section 4.6). Il est constitué d'une table de faits `Sales` stockée dans le document `facts.xml` (4,92 Mo) et de cinq dimensions `Channels`, `Promotions`, `Customers`, `Products` et `Times` stockées dans le document `facts.xml` (3,77 Mo). Cet entrepôt de données a été implanté au sein du SGBD XML natif

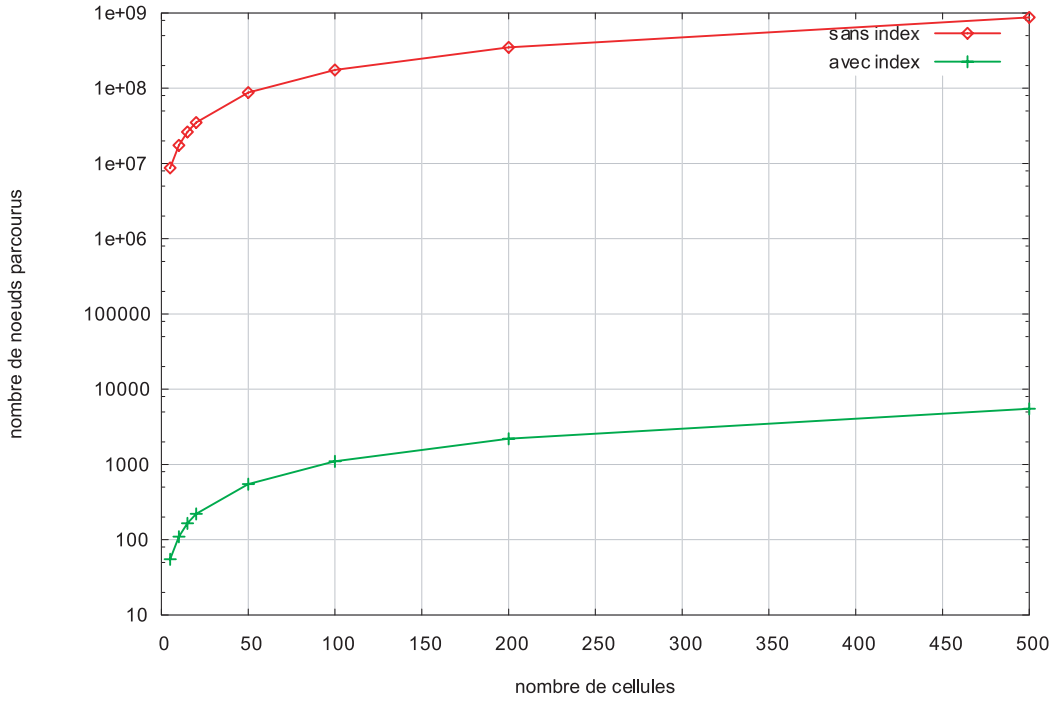

FIG. 7.12 – Résultats théoriques de notre index de jointure XML

eXist [Mei05], qui est un outil gratuit, qui permet le stockage de documents volumineux et qui supporte le langage XQuery (alors que de nombreux outils ne supportent que XPath). Nous avons effectué nos tests sur une machine dotée d'un processeur Intel Pentium 2 GHz avec 1 Go de mémoire et un disque dur IDE.

Nous avons exécuté la requête décisionnelle XQuery de la Figure 7.6 avec et sans utilisation de l'index et en faisant varier la taille de l'entrepôt. La Figure 7.13 présente les résultats obtenus exprimés en temps d'exécution par rapport au nombre de cellules (faits) dans le document `facts.xml`. Pour mieux visualiser les résultats, nous avons utilisé une échelle logarithmique sur l'axe des ordonnées.

La Figure 7.13 montre qu'avec utilisation de notre index, nous obtenons des temps de traitement nettement inférieurs par rapport aux temps des requêtes n'utilisant pas notre index. Cette figure est semblable à celle qui représente nos estimations (Figure 7.12), ce qui valide notre étude théorique.

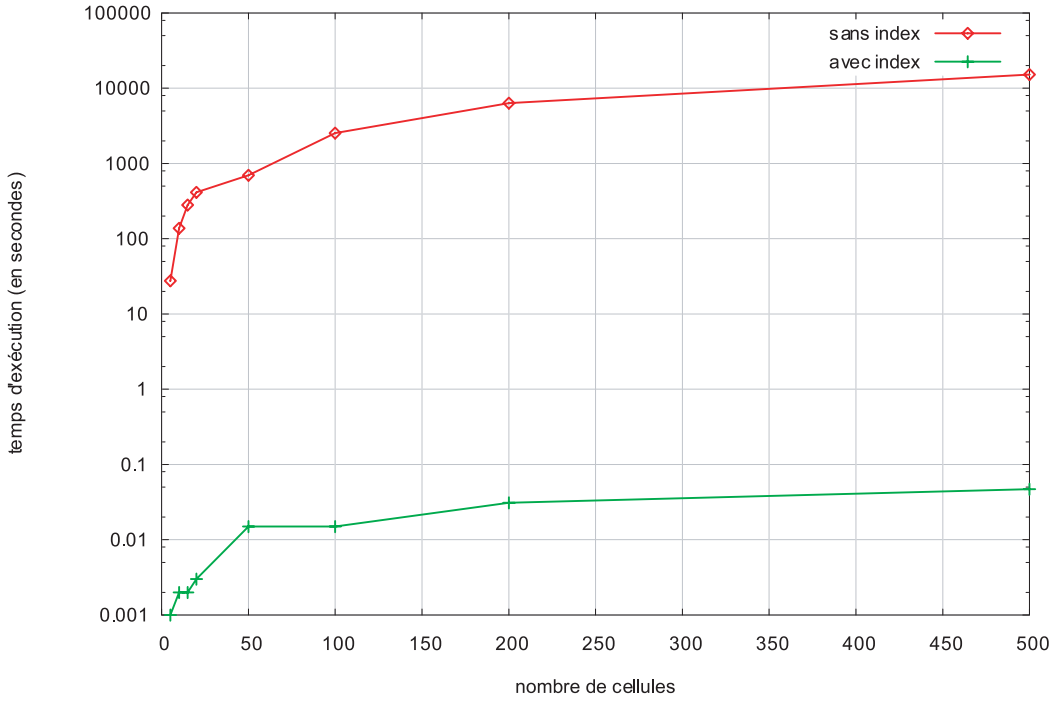

FIG. 7.13 – Résultats d'expérimentation de notre index de jointure XML

Pour finir, nous avons également utilisé notre structure d'index pour le traitement d'une requête sur la totalité des cellules du documents `facts.xml`. Nous avons obtenu un temps d'exécution de moins de deux secondes, alors que le système s'est avéré incapable de répondre à la requête lorsque nous n'utilisions pas notre index.

7.3.2 Matérialisation de vues basée sur la classification de requêtes XML

Dans cette section, nous présentons l'adaptation de notre stratégie de sélection de vues matérialisées (Chapitre 5) au contexte des entrepôts de données XML.

Dans ce contexte, nous cherchons à matérialiser les requêtes de type : sélection, jointure et agrégation. Une requête de ce type est exprimée, en XQuery, sous forme d'une expression FLWR étendue par la clause `Group by` définie précédemment.

Tout comme dans notre stratégie de sélection de vues matérialisées dans le contexte relationnel, l'obtention d'une configuration de vues XML à matérialiser consiste à :

– analyser la charge,
– construire un ensemble de vues XML candidates,
– sélectionner celles qui sont pertinentes à matérialiser.

La construction de l'ensemble de vues XML candidates est réalisée à l'aide de la classification non supervisée des requêtes XQuery de la charge. Dans la suite de cette section, nous ne présentons que les étapes adaptées au contexte XML.

7.3.2.1 Analyse de la charge

La charge extraite de l'entrepôt de données XML est constituée de requêtes exprimée en XQuery. Les attributs représentatifs de chaque requête sont les attributs des prédicats de sélection de la clause `Where` et de la clause `Group by`. L'analyse des requêtes de l'extrait de charge représenté à la Figure 7.14 donne le contexte de classification représenté au Tableau 7.1. Nous obtenons ainsi un contexte de classification similaire à celui présenté au Chapitre 5.

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	...
q_1	1	0	0	0	0	0	0	0	
q_2	0	0	1	1	0	0	0	0	
q_3	0	1	0	0	1	1	1	0	
...									

a_1	day_name	a_5	cust_first_name
a_2	fiscal_day	a_6	cust_marial_status
a_3	promo_name	a_7	prod_category
a_4	promo_category	a_8	prod_name

TAB. 7.1 – Contexte de classification non supervisée

7.3.2.2 Construction de l'ensemble de vues candidates

Tout comme dans le cas des entrepôts de données relationnels, il est difficile dans la pratique de matérialiser toutes les vues XML candidates. Pour pallier ces limitations, nous proposons des modèles de coût permettant de ne conserver que les vues XML les plus avantageuses. Les modèles que nous proposons sont adaptés du cas relationnel.

La Figure 7.15 montre une structure typique, dérivée de la spécification XCube, d'une

```
q1  for $a in //dimensionData/classification/Level[@node='times']/node,
    $x in //CubeFacts/cube/Cell
    let $q := $b/attribute[@name='day_name']/@value
    where $a/attribute/@name='fiscal_day'
    and $a/attribute/@value='2' and $x/dimension /@node=$a/@id
    and $x/dimension/@id='times'
    group by(@name='day_name')
    return @name='day_name', sum(quantity)

q2  for $a in //dimensionData/classification/Level[@node='promotions']/node,
    $b in //dimensionData/classification/Level[@node='products']/node,
    $x in //CubeFacts/cube/Cell
    let $q := $a/attribute[@name='promo_name']/@value
    where $a/attribute/@name='promo_category'
    and $a/attribute/@value='newspaper' and $x/dimension /@node=$a/@id
    and $x/dimension/@node=$b/@id and $x/dimension/@id='products'
    and $x/dimension/@id='promotions'
    group by(@name='promo_name')
    return @name='promo_name',sum(amount)

q3  for $a in //dimensionData/classification/Level[@node='products']/node,
    $b in //dimensionData/classification/Level[@node='customers']/node,
    $c in //dimensionData/classification/Level[@node='times']/node,
    $x in //CubeFacts/cube/Cell
    let $q := $b/attribute[@name='cust_first_name']/@value
    where $c/attribute/@name='fiscal_day'
    and $c/attribute/@value='3' and $b/attribute/@name='cust_marial_status'
    and $b/attribute/@value='single'
    where $a/attribute/@name='prod_category'
    and $a/attribute/@value='TV' and $x/dimension /@node=$a/@id
    and $x/dimension /@node=$b/@id and $x/dimension /@node=$c/@id
    and $x/dimension/@id='customers' and $x/dimension/@id='products'
    and $x/dimension/@id='times'
    group by(@name='cust_first_name')
    return @name='cust_first_name', sum(quantity)

...
```

FIG. 7.14 – Extrait de charge de requêtes XQuery

vue XML. La structure d'une vue est composée d'éléments *Cell*. Chaque élément *Cell* est composé à son tour d'éléments *dimension* qui contiennent les attributs de la clause `Group by`, leur valeurs et les éléments *fact* qui contiennent les résultats de l'opération d'agrégation. Nous présentons dans la suite le modèle de coût qui estime la taille d'une vue donnée, ainsi que son coût de stockage.

La taille d'une vue est estimée par le nombre de ses différents éléments *Cell*. Pour chaque élément *Cell*, le nombre d'occurrences des éléments *dimension* et *fact* est fixe. Celui de *dimension* est égal au nombre de dimensions de la clause `Group by` et celui de *fact* est égal au nombre d'opérations d'agrégation effectué sur les mesures. Pour estimer le nombre d'éléments *Cell*, nous estimons dans un premier temps le nombre maximal d'éléments *Cell*

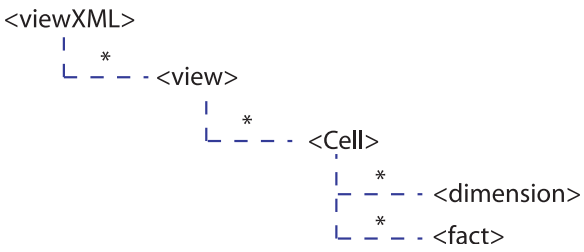

FIG. 7.15 – Structure d'une vue XML dérivée de la spécification XCube

par la formule suivante :

$$ms(Cell) = \prod_{i=1}^{d} |d_i|$$

où $|d_i|$ représente la cardinalité d'une dimension caractérisant une cellule de `facts.xml` et d le nombre de dimension du document `dimensions.xml`.

Soit $ms(v)$ la taille maximale d'une vue XML v composée des dimensions $d_1, \ldots, d_k$, où k est le nombre de dimensions de v et $|d_i|$ la cardinalité de la dimension d_i. $ms(v)$ s'exprime comme suit :

$$ms(v) = \prod_{i=1}^{k} |d_i|.$$

La taille de la vue XML v peut être calculée à l'aide de la formule de Yao comme suit :

$$|v| = ms(v) \times \left[1 - \prod_{i=1}^{|Cell|} \frac{ms(Cell) \times c - i + 1}{ms(Cell) - i + 1}\right]$$

où $c = 1 - \frac{1}{ms(v)}$.

Lorsque $\frac{ms(Cell)}{ms(v)}$ est suffisamment élevé, la formule de Yao est bien approximée par la formule de Cardenas comme suit :

$$|v| = ms(v) \times \left(1 - \left(1 - \frac{1}{ms(v)}\right)^{|Cell|}\right).$$

La taille physique, en octets, d'une vue XML est égale au nombre de ses éléments multiplié par la taille moyenne de l'espace nécessaire pour stocker un seul élément de cette vue.

À partir du nombre d'éléments de v, nous estimons sa taille, en octets, comme suit :

$$taille(v) = |v| \times \sum_{i=1}^{k} taille(d_i)$$

où $taille(d_i)$ dénote la taille, en octets, de la dimension d_i de v, et k est le nombre de ses dimensions. La taille de chaque dimension peut être obtenue directement à partir des métadonnées de l'entrepôt XML.

7.3.2.3 Expérimentation

Nous avons construit une charge de dix requêtes décisionnelles sur laquelle nous avons appliqué notre stratégie de sélection de vues XML (cf. Annexe C). Nous n'avons mesuré les temps d'exécution des requêtes que sur des fragments des documents `dimensions.xml` et `facts.xml`. Le moteur d'exécution de requêtes de la base de données eXist s'est avéré peu performant pour l'exécution de requêtes contenant plusieurs opérations de jointures sur des documents volumineux.

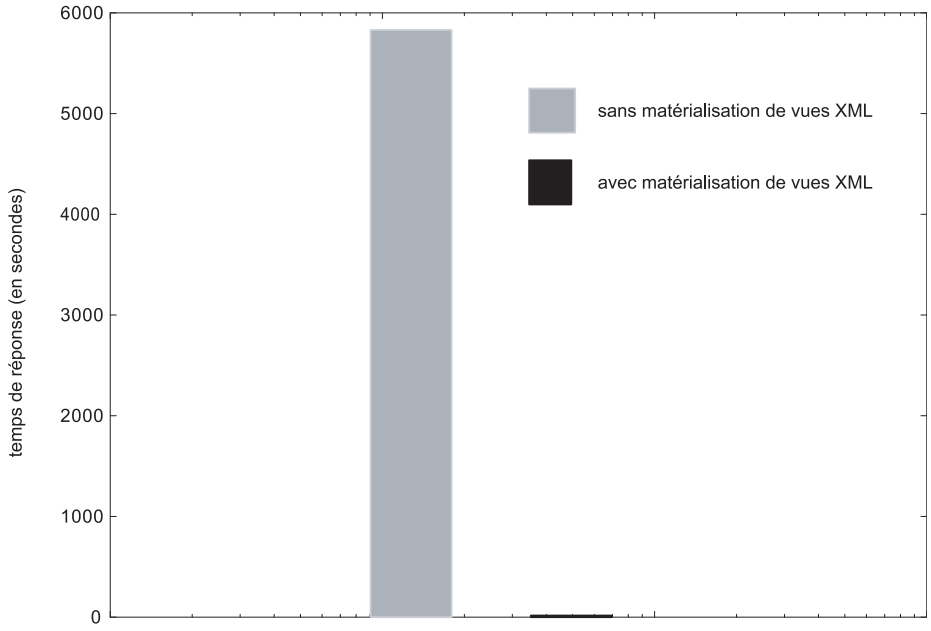

FIG. 7.16 – Résultats de sélection de vues XML

Nous avons mesuré le temps d'exécution dans les cas suivants : sans et avec utilisation des vues matérialisées. La Figure 7.16 représente le temps d'exécution des requêtes sans et avec matérialisation de vues XML. Nous notons que le temps d'exécution obtenu en utilisant la matérialisation des vues offre de meilleures performances, car les jointures présentes dans les requêtes de la charge ont été éliminées dans la phase de réécriture de ses requêtes en présence des vues XML sélectionnées. L'exécution des requêtes avec les vues XML est environ 24700 fois plus rapide que l'exécution de ces mêmes requêtes sans vues XML.

7.4 Conclusion

Le travail présenté dans ce chapitre nous a permis de proposer des stratégies pour l'amélioration du traitement des requêtes décisionnelles exprimées en XQuery. Pour cela, nous nous sommes basés sur un entrepôt de données de référence. Nous avons d'ailleurs dû étendre le langage XQuery pour qu'il supporte les regroupements multiples (`Group by`) communément utilisés pour exprimer des requêtes décisionnelles. Nous avons également proposé un index de jointure XML pour rendre efficace l'exécution des requêtes. De plus, nous avons mis en œuvre notre approche basée sur la classification non supervisée des requêtes pour la sélection de vues matérialisées XML. Les résultats expérimentaux montrent que notre index de jointure XML ou notre stratégie de sélection de vues XML sont efficaces pour améliorer les performances des systèmes natifs XML.

Le travail présenté dans ce chapitre est préliminaire et vise à prospecter des méthodes permettant d'optimiser les performances des entrepôts de données XML. À notre connaissance, ce domaine n'est pas étudié. Nous n'avons donc pas un élément de comparaison avec les proposions que nous apportons dans ce chapitre.