ULB, Université Libre de Bruxelles
Faculté des Sciences Appliquées
Service Ingénierie Informatique
et de la Décision (CoDE)

Modélisation et interrogation de données multidimensionnelles en XML

Boris Verhaegen

Mémoire présenté sous la direction du **Prof. Esteban Zimányi**
en vue de l'obtention du Diplôme de DEA en Sciences Appliquées
Année académique 2006–2007

à Lili

Remerciements

Je tiens à remercier tout particulièrement Esteban Zimányi, mon promoteur, pour m'avoir aidé dans ce travail via de nombreux encouragements, impulsions, relectures et corrections. Je remercie également l'ensemble du Service Ingénierie Informatique et de la Décision pour leur sympathie et leur accueil.

Je voulais également exprimer ma gratitude envers ma famille et mes amis pour leur confiance et leur soutien, tout particulièrement envers Jérôme Vos et Xavier Devos pour leurs relectures et corrections.

Enfin, je tiens à exprimer ma reconnaissance aux développeurs du logiciel libre eXist, et en particulier Pierrick Brihaye, pour leur sympathie et leurs explications.

Table des matières

Chapitre 1

Introduction

Le but de ce travail est d'étudier diverses façons de modéliser des données multidimensionnelles en XML et de les interroger à l'aide de langages adaptés.

Pour ce faire, nous découperons notre travail en plusieurs parties. Tout d'abord, nous présenterons le langage XML ainsi que les outils permettant de gérer et d'interroger des documents écrits dans ce langage. Ensuite, nous introduirons les concepts nécessaires à la compréhension des systèmes d'analyse multidimensionnelle. Nous joindrons ces deux grands sujets en analysant différents modèles multidimensionnels pour XML et en étudiant leur interrogation. Pour finir, nous réaliserons une évaluation des performances d'interrogation des divers modèles afin de pouvoir les comparer.

Le second chapitre présente le langage XML. Après une brève introduction aux documents XML, nous étudierons les langages permettant de les interroger et en particulier XPath et XQuery, deux langages très prometteurs dans ce domaine.

Dans le troisième chapitre, nous traiterons des systèmes de bases de données permettant de stocker et de gérer des documents XML. Nous nous attarderons principalement sur les bases de données XML natives qui introduisent de nouveaux défis en ce qui concerne le stockage et l'indexation de données arborescentes. Pour finir, nous analyserons le logiciel eXist, une base de données XML native libre que nous utiliserons pour notre évaluation de performances.

Le quatrième chapitre introduira brièvement les concepts nécessaires à la compréhension des systèmes d'analyse multidimensionnelle. Nous présenterons les entrepôts de données ainsi que les techniques de modélisation couramment utilisées dans ce domaine.

Ensuite, dans le cinquième chapitre, nous présenterons les modèles de données multidimensionnelles existants et nous en proposerons un nouveau. Pour chacun de ces modèles, nous analyserons intuitivement leurs avantages et inconvénients.

Dans le sixième chapitre, nous traduirons des requêtes typiques d'analyse pour ces différents modèles à l'aide du langage d'interrogation pour XML XQuery. Nous tenterons pour chaque modèle d'évaluer la facilité d'écriture et de compréhension des requêtes et d'approcher leur complexité.

Le septième chapitre présente une extension à XQuery facilitant l'écriture des requêtes d'analyses tout en diminuant leur complexité. Cette extension est une clause de groupement inspirée de ce qui se fait en SQL, un langage d'interrogation relationnel bien connu. Nous implémenterons cette extension dans le logiciel eXist afin de pouvoir analyser ses avantages.

Dans le but de pouvoir comparer les modèles multidimensionnels XML de façon objective, nous réaliserons dans le huitième chapitre une analyse de performances. Pour ce faire, nous exécuterons un ensemble de requêtes identiques sur les divers modèles et nous étudierons l'évolution de leur temps de calcul.

Le neuvième et dernier chapitre commencera par une comparaison générale des modèles multidimensionnels XML prenant en compte un maximum de critères. Nous terminerons ce mémoire par une conclusion et des pistes de recherches futures.

Chapitre 2

XML et ses outils

Ce chapitre présente les fondements du langage de balisage extensible XML et les technologies qui lui sont associées comme les langages XPath et XQuery permettant d'interroger des données modélisées à l'aide de XML.

Contenu

XML (*eXtensible Markup Language*) a été défini en 1998 par le W3C (*World Wide Web Consortium*) [1]. Il s'agit d'une notation permettant de décrire un langage sous la forme d'une grammaire d'arbre. Il permet de stocker des données structurées à l'aide de balises dans un fichier texte. Ces balises peuvent être définies par l'utilisateur au contraire de l'HTML. Ce formalisme regroupe sous le nom d'XML une boîte à outils extensible et évolutive dont le but est de simplifier l'organisation et l'échange des données.

XML n'est donc pas seulement un méta-langage. Autour de la recommandation XML 1.0 du W3C qui définit principalement ce que sont les balises et les attributs, un nombre croissant de modules ont été définis. Ce sont des modules facultatifs qui fournissent des ensembles de balises et d'attributs, des règles pour certaines tâches particulières, des méthodes pour les exploiter, ... Citons en quelques uns comme XLink qui décrit une méthode pour ajouter des liens hyper-textes à un fichier XML. XPointer permet de se référer à des parties de document XML. XPath est un outil pour accéder aux données d'un fichier XML à l'aide d'expressions de chemin. XQuery est un langage d'interrogation de type SQL. De nombreux autres modules existent et leur nombre est en constante évolution. Dans les chapitres suivants, nous détaillerons certains de ces concepts, jugés intéressants dans le contexte de ce mémoire.

Les avantages principaux d'XML sont sa simplicité, sa modularité, sa stabilité (le format a peu évolué depuis son arrivée en 1998) et son universalité. De plus, il est libre de droit et possède un nombre grandissant d'outils et d'utilisateurs. Cela facilite grandement les recherches, la réutilisation et diminue la dépendance vis-à-vis des fournisseurs et applications. Enfin, XML est beaucoup plus souple que le relationnel pour modéliser le monde réel, ce qui nous intéresse plus particulièrement dans ce mémoire.

2.1 Document XML

Un document XML étant avant tout un fichier texte, il peut être édité avec n'importe quel éditeur de texte. De nombreux outils permettent d'afficher les documents XML de façon hiérarchique de telle sorte que l'on peut se déplacer dans le fichier comme dans un arbre. Ceux-ci permettent aussi de vérifier la validité d'un fichier XML.

2.1.1 Notions générales

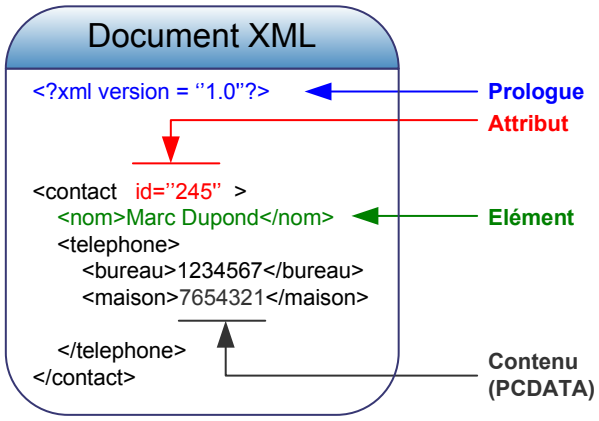


FIG. 2.1 – Exemple de document XML.

Un exemple de document XML très simple est illustré à la figure 2.1. Comme on peut le remarquer, chaque section est marquée à l'aide de balises descriptives permettant d'identifier le type de données qu'elle contient.

Un document XML est soit bien formé, soit valide. Il peut en outre être illégal mais, dans ce cas, nous ne pouvons le considérer en tant que document XML. Introduisons maintenant ces deux concepts et les notions nécessaires à leur compréhension : Un document XML bien formé est un document qui suit la recommandation XML, c'est-à-dire qu'il est conforme aux règles en ce qui concerne les éléments et les attributs, qu'il contient les déclarations essentielles et qu'il respecte parfaitement une structure arborescente. Un document XML est valide s'il est bien formé et qu'il suit les règles d'une définition de type de document (DTD) ou d'un Schéma XML. Une DTD permet d'imposer une structure à un document XML : un document XML peut donc se conformer à un ensemble de règles afin d'éviter des erreurs de forme. Un Schéma XML permet en plus de typer les données se trouvant dans le document auquel il est associé alors que la DTD ne se préoccupe pas du typage.

2.1.2 Eléments et attributs

Dans le document XML, après le prologue, se trouve une série d'éléments. Un élément est l'unité de base d'un document XML, composé de données textuelles et de balises. Les frontières

d'un élément sont définies par une balise de début et une balise de fin. Une balise est un ensemble de mots encadrés par les symboles `<` et `>`. Ces balises ne sont pas fixées : l'utilisateur peut définir ses propres balises, son propre format XML. Les balises en XML, au contraire de l'HTML, n'ont pas de signification précise : elles sont présentes pour délimiter les données et XML laisse l'entière interprétation des données à l'application qui les lit. Un document XML bien formé doit contenir exactement un élément non vide, appelé élément racine. Celui-ci peut contenir d'autres éléments définissant ainsi un arbre.

Les éléments peuvent contenir des attributs qui fournissent des informations supplémentaires. Ces attributs sont des couples de la forme `nom="valeur"`. Ils doivent se trouver dans la balise ouvrante après le nom de l'élément. Notons que les noms d'attributs sont sensibles à la casse, qu'ils doivent être uniques au sein d'un même élément et que les valeurs des attributs se trouvent obligatoirement entre guillemets. Comme le précise la spécification XML, tout texte qui n'est pas du balisage est une donnée textuelle du document. Précisons toutefois que certains symboles sont interdits comme `<`, `>` et `&`. Pour utiliser ces caractères, il faut les remplacer par leur représentation hexadécimale.

2.1.3 Types de documents XML

Les documents XML peuvent se classer en deux grandes catégories : orientés données et orientés présentation.

Documents orientés données

Les documents orientés données (*data-centric*) sont ceux pour lesquels XML est utilisé pour stocker et transporter des données. Ils incluent les ordres de ventes, les enregistrements de patients et les données scientifiques. Leur structure physique (l'ordre des éléments, l'utilisation d'éléments ou d'attributs) n'est pas importante dans la majorité des cas sauf si les données sont représentées en utilisant une structure arborescente, ce qui nous intéressera dans ce mémoire.

Documents orientés présentation

Les documents orientés présentation (*document-centric*) sont ceux dans lesquels XML est utilisé pour ses capacités de description, comme dans un manuel d'utilisateur, une page web statique en XHTML ou des brochures. Ils sont caractérisés par leur structure irrégulière et leur contenu mixte. A l'opposé des documents orientés données, leur structure physique est importante. Par exemple, pour un manuel d'utilisateur, l'ordre des chapitres est important. Par contre, pour une facture, l'ordre des articles ne l'est souvent pas.

2.2 Langages d'interrogation pour XML

2.2.1 Introduction

Il existe différents types de langages afin d'interroger des données XML :

- ▷ **Les langages d'adressage** : Ces langages permettent d'écrire des requêtes afin de naviguer à l'intérieur d'un ensemble de données, à l'aide d'expressions de chemin. Le standard des langages d'adressage est XPath [2] qui sert de fondation pour d'autres langages d'interrogation de documents XML.
- ▷ **Les langages de transformations** : Ces langages permettent de restructurer et de reformater des documents XML à l'aide de règles. Le standard actuel est XSLT [3]. Il en existe d'autres comme par exemple CDuce [4].
- ▷ **Les langages de requêtes** : Ces langages permettent d'émettre des requêtes complexes sur des documents XML, comme SQL pour les bases de données relationnelles. XQuery [5] et Lorel [6] sont des exemples de tels langages, le premier étant défini par le W3C.

Par la suite, nous nous concentrerons sur les langages d'adressage et de requêtes et d'adressage XPath et XQuery.

2.2.2 XPath

Le langage d'adressage XPath, a été publié par le W3C en 1999 [2]. Jim Melton en fait une description complète dans son livre *Querying XML* [7]. Selon sa spécification, XPath a été créé pour fournir une syntaxe commune pour les fonctionnalités partagées entre XSLT et XPointer. Son but est de pouvoir adresser des parties d'un document XML. Autrement dit, XPath permet de sélectionner un ensemble de nœuds de l'arbre XML à l'aide d'une expression de chemin.

La notation choisie pour XPath ressemble délibérément à la notation utilisée pour parcourir les systèmes de fichiers. Voici un exemple d'expression de chemin :

```
/clients/client[@id eq "1548"]/nom
```

Si nous ne tenons pas compte de l'expression entre crochets, cette expression permet de sélectionner les éléments `nom` des éléments `client` contenus dans l'élément `clients`. L'expression entre crochets est un filtre. Celui-ci est constitué d'un prédicat. Ce filtre permet de sélectionner les éléments `client` dont l'attribut `id` est égal à 1548.

Les expressions de chemin sont composées d'étapes. Les étapes sont représentées à l'aide d'un axe, d'un nom d'élément et d'un prédicat éventuel. Les axes permettent de définir le sens de parcours du document XML. XPath définit un grand nombre d'axes qui sont présentés au tableau 2.1. Les noms d'éléments sont similaires aux noms de dossiers ou de répertoires des systèmes de fichiers. Ils permettent de définir les éléments XML par lesquels le chemin doit passer. Les prédicats sont des expressions logiques relatives à une étape. Ils doivent être écrits entre crochets et permettent de filtrer les éléments lors du parcours.

XPath n'est pas exactement un langage de requêtes, car, bien qu'il permette de poser des conditions et d'obtenir des informations sur un document, il ne permet pas de restructurer les résultats des requêtes. Malgré cela, XPath est très utile et simple à comprendre et à assimiler. Cela en fait le standard pour l'expression de chemin sur un arbre XML. Il est utilisé à la fois pour XSLT et pour XQuery, le langage de requêtes XML défini par le W3C.

Axes	**Ensemble de nœuds résultant**
`parent (../)`	Le premier nœud sur le chemin de u vers le nœud racine
`ancestor`	Tous les nœuds sur le chemin de u vers la racine
`ancestor-or-self`	u et tous les nœuds sur le chemin vers la racine
`child (/)`	Les descendants directs du nœud u
`descendant`	Tous les nœuds dont u est l'ancêtre
`descendant-or-self (//)`	u et tous les nœuds dont u est l'ancêtre
`preceding`	Les nœuds précédents le nœud u (exceptés les ancêtres) dans l'ordre du document
`following`	Les nœuds suivants le nœud u (exceptés les descendants) dans l'ordre du document
`preceding-sibling`	Les frères précédents du nœud u dans l'ordre du document
`following-sibling`	Les frères suivants du nœud u dans l'ordre du document
`attribute`	Les attributs du nœud u
`self(./)`	u

TAB. 2.1 – Sémantique des axes XPath pour un nœud u de référence.

Expressions de comparaison

Le langage XPath permet d'exécuter trois types de comparaisons : la comparaison de valeurs, la comparaison générale et la comparaison de nœuds. Dans ce mémoire, nous serons amenés à analyser des expressions XPath complexes utilisant les deux premiers types de comparaisons. C'est pourquoi nous les décrivons ici.

Le type de comparaison le plus classique est la comparaison de valeurs qui est utilisée pour comparer des valeurs simples, c'est-à-dire comparer une valeur avec une autre. Les opérateurs de comparaison de valeurs XPath sont `eq`, `ne`, `lt`, `le`, `gt` et `ge` qui signifient respectivement $=$, $!=$, $<$, $<=$, $>$ et $>=$. Les deux opérandes de l'expression de comparaison, deux nœuds, sont atomisées et ensuite comparées de manière classique.

Le second type de comparaison est la comparaison générale. Ses opérateurs sont `=`, `!=`, `<`, `<=`, `>` et `>=`. Les opérandes de ces expressions de comparaison ne sont plus des nœuds mais des séquences de nœuds. Les opérateurs de comparaison générale ont une sémantique existentielle. En effet, l'expression de comparaison `(1,2)=(2,3)` renvoie une réponse positive car un élément de l'opérande de gauche (`2`) est égal (`eq`) à un élément de l'opérande de droite. Il s'agit donc d'une disjonction de comparaisons de valeurs.

La comparaison de nœuds, le dernier type de comparaison, n'implique pas leur atomisation. Les nœuds sont comparés en fonction de leur identité et non de leur valeur. Les trois opérateurs disponibles sont `is`, `<<` et `>>`. Le premier représente l'égalité d'identité : Deux nœuds de valeurs identiques mais placés à des endroits différents d'un document XML ne sont pas égaux du point de vue de leur identité. Les deux opérateurs suivants signifient respectivement avant et après. Cette comparaison compare donc les positions des nœuds dans le document XML sans se soucier de leurs valeurs.

2.2.3 XQuery

En 1998, le W3C a commencé à étudier un langage de requêtes pour XML. Un groupe de travail a ainsi été constitué et une étude comparative de nombreux langages a été réalisée [8]. Celle-ci a mis en évidence l'importance de décomposer une requête en trois parties :

- ▷ **un motif** qui associe des variables à des portions de documents,
- ▷ **un filtre** optionnel qui sélectionne une partie des résultats obtenus grâce au motif et
- ▷ **un constructeur** qui définit la structure de chaque résultat de la requête.

Un peu plus tard, un document du W3C [9] posera le cadre du futur XQuery. Les objectifs principaux y sont décrits et une notion importante y est définie : les requêtes devront pouvoir porter sur un simple document XML ou sur une collection de documents. Elles devront pouvoir sélectionner des documents entiers ou ne garder que des sous-arbres qui remplissent certaines conditions de contenu ou de structure.

Les objectifs principaux repris dans ce document sont les suivants :

- ▷ l'importance de la déclarativité pour une requête, avec le découpage de la requête en trois parties,
- ▷ la possibilité de porter une condition sur du texte,
- ▷ la présence de quantificateurs existentiels et universels,
- ▷ la combinaison d'informations de différentes parties d'un ou de plusieurs documents,
- ▷ l'agrégation d'informations à partir de documents proches,
- ▷ le tri sur les éléments,
- ▷ l'imbrication de requêtes (*nesting* en anglais),
- ▷ et l'opération sur les noms des éléments XML.

XQuery [5] est donc né à partir de ces recommandations et objectifs. Le langage a été conçu pour permettre des requêtes précises et facilement compréhensibles. Il introduit un type de requête nouveau, basé sur des expressions FLWOR (en anglais, prononcez *flower*) comparables aux expressions *select-from-where* de SQL. Ce type d'expression sera détaillé dans la suite de ce chapitre. XQuery est un langage basé sur Quilt [10] et emprunte de nombreuses idées à XQL [11], XML-QL [12], SQL et OQL [13].

XQuery est un langage fonctionnel dont chaque requête est une expression de différents types comme des expressions de chemin, des expressions FLWOR, des expressions conditionnelles ou des fonctions. Les expressions de chemin sont basées sur la syntaxe XPath. La recommandation XPath 2.0 est entièrement liée à XQuery.

Expressions FLWOR

La caractéristique principale de XQuery est qu'il utilise des expressions FLWOR pour l'écriture de requêtes. Son nom vient des 5 clauses principales suivantes :

- ▷ `for` : mécanisme d'itération sur un ensemble de nœuds.
- ▷ `let` : permet l'assignation de variables.
- ▷ `where` : les clauses `for` et `let` génèrent un ensemble de nœuds XML qui peuvent être filtrés par des prédicats de la clause `where`.
- ▷ `order by` : ordonne les résultats de façon alphabétique ou numérique.
- ▷ `return` : construit le résultat pour chaque nœud vérifiant la clause `where`.

Ce type d'expression peut se comparer à une expression SQL `select-from-where` mais leurs fonctionnements respectifs sont assez différents. En effet, le principe est inversé d'un type d'expression à l'autre. Par exemple, en SQL, la construction des résultats se fait à l'aide de la clause `select`, au début de la requête. Tandis qu'en XQuery, elle se fait à l'aide de la clause `return` en fin de requête. Le fonctionnement et la syntaxe principale de ces expressions est illustré aux figures 2.2 et 2.3.

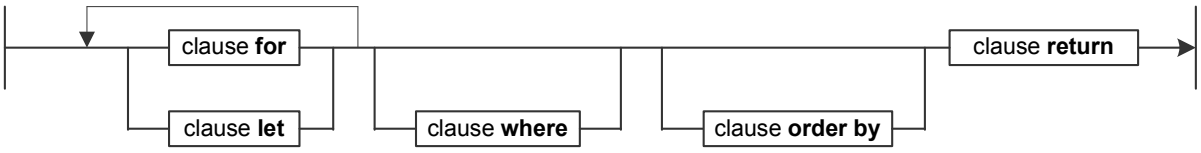


FIG. 2.2 – Expression FLWOR.

Quantificateurs

Tout comme SQL, XQuery possède un quantificateur existentiel : `some`. XQuery définit aussi un quantificateur universel `every`. Ce quantificateur manque cruellement à SQL. En effet, comme il n'y a pas de quantificateur universel dans SQL, il est nécessaire de s'aider de la logique pour transformer les requêtes universelles en requêtes utilisant des quantificateurs existentiels niés. Cela rend ces requêtes SQL particulièrement difficiles à lire et à comprendre et donc compliquées à maintenir. Un tel quantificateur universel est donc plus que bienvenu dans XQuery.

Fonctions

XQuery permet de définir et d'utiliser des fonctions. Les fonctions XQuery doivent respecter certaines règles comme le fait que tous les types utilisés doivent l'être fortement, qu'ils soient des types d'arguments ou le type de sortie. Les fonctions peuvent s'appeler elles-mêmes ou en appeler d'autres.

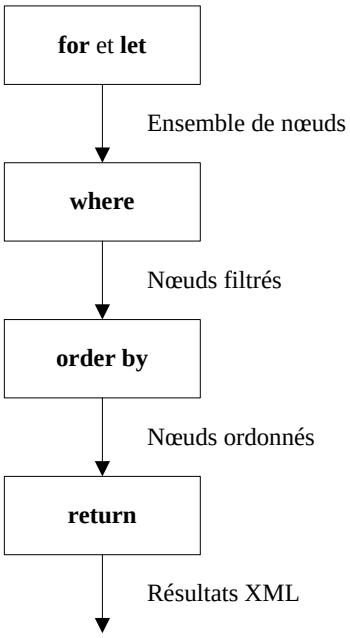


FIG. 2.3 – Représentation schématique du fonctionnement d'une expression FLOWR.

Limitations

De nombreuses discussions à propos d'XQuery dans la littérature scientifique [7, 14, 15, 16, 17] et dans les listes de discussions mettent en évidence les limitations d'XQuery et proposent des solutions.

Pour commencer, XQuery, bien qu'il soit un langage d'interrogation pour XML, ne respecte pas la syntaxe XML. XQueryX [16] a donc été proposé. XQueryX est une syntaxe alternative pour XQuery, où une requête est représentée comme un document XML bien formé.

Deuxièmement, XPath et XQuery ne disposent pas d'opérateurs et de fonctions adaptées à la recherche textuelle (*full-text query* en anglais) dans un document XML. Par exemple, il n'existe ni de fonction de recherche insensible à la casse, ni de fonction de recherche de texte à la structure du document. Or, XML étant énormément utilisé dans les sites Web, de telles fonctionnalités seraient les bienvenues pour fabriquer aisément un moteur de recherche efficace. eXist, une base de données XML native, implémente une série d'opérateurs et d'indexes pour pallier à ce manque. Nous en parlerons en détail à la section 3.3 consacrée à ce logiciel.

Un autre manque de XQuery est qu'il ne possède pas d'opérateur permettant d'ajouter des données ou de les mettre à jour. XQuery permet uniquement de consulter des informations sans pouvoir les modifier. Dans le contexte des bases de données, il s'agit d'un défaut important. Par contre, pour les processeurs XQuery qui travaillent en lecture séquentielle, cela ne pose pas de problème car leur but n'est jamais de modifier les données sources. Les moteurs commerciaux qui ont implémenté XQuery proposent des extensions permettant d'éviter ce problème, mais rien n'est standardisé. XML:DB, la communauté du logiciel libre autour des bases de données XML, propose un langage de mise à jour : XUpdate [18]. Le W3C travaille actuellement sur

XQuery Update Facility [19], une extension au langage XQuery permettant de mettre à jour des documents XML.

2.3 Conclusion

Dans ce chapitre, nous avons décrit le langage XML et les outils qui lui sont associés comme les langages d'interrogation. Nous avons vu que XML permet de décrire très facilement le monde réel avec une plus grande flexibilité que la représentation relationnelle et que ce format est aisément interrogeable.

Dans le chapitre suivant, nous allons introduire les systèmes permettant de stocker et d'interroger ces documents XML de façon stable et performante à l'aide notamment des langages d'interrogation présentés dans ce chapitre.

Chapitre 3

Bases de données XML

Dans ce chapitre, nous allons analyser les systèmes permettant de stocker et d'interroger des données XML. Nous nous concentrerons principalement sur les bases de données XML natives et nous en présenterons une en détail, le logiciel libre eXist.

Contenu

XML est un format d'échange de données de plus en plus utilisé. Il convient alors de posséder des outils permettant de les stocker, de les interroger, de garantir leur intégrité, etc. C'est le rôle d'un système de gestion de base de données (SGDB). Généralement, on attend d'un tel système les caractéristiques suivantes :

- ▷ Des outils d'interrogation, tels que les langages SQL ou XQuery.
- ▷ Des capacités de transactions respectant les propriétés ACID. C'est-à-dire qu'un tel système doit offrir l'atomicité des opérations, la consistance de la base de données dans son ensemble, l'isolation des opérations des différents utilisateurs et la résistance des opérations.
- ▷ L'échelonnabilité (*scalability* en anglais) et la robustesse.
- ▷ Une gestion de la sécurité et des performances. Par exemple, la gestion des utilisateurs, des indexes, de l'optimisation, etc.

Plusieurs sortes d'outils permettent de stocker et gérer des documents XML. Les principaux outils existants sont les bases de données relationnelles avec support XML et les bases de données XML natives.

Il est important de remarquer que des données XML peuvent être très différentes des données relationnelles classiques. Remarquons les différences suivantes :

- ▷ L'ordre d'un document XML peut être important.

- ▷ Les documents XML peuvent contenir des données non numériques.
- ▷ La structure d'un document XML est arborescente.
- ▷ Un document XML ne respecte pas nécessairement un schéma strict comme les tables relationnelles.
- ▷ La structure d'un document XML contient une sémantique intrinsèque.

En considérant les points soulevés ici, nous introduirons les deux types de bases de données XML en nous concentrant tout particulièrement sur les bases de données XML natives, systèmes particulièrement flexibles et prometteurs.

3.1 Bases de données relationnelles avec support XML

Depuis les années 1980, les bases de données relationnelles (SGDBR) sont les plus utilisées pour stocker et interroger des informations. La plupart des entreprises dépendent d'un tel système pour stocker et protéger leurs données. Les milliards de dollars investis dans les systèmes commerciaux comme Oracle et IBM DB2 leur ont donné une énorme force dans le domaine de la gestion de données. De tels systèmes sont aujourd'hui très performants, échelonnables (*scalable* en anglais) et fiables.

Au début des années 2000, la plupart des moteurs relationnels commerciaux ont intégré le support de l'XML. Au départ, il ne s'agissait que de stocker et de récupérer l'entièreté d'un document XML, sans traitement particulier des données contenues dans le document. Certains systèmes stockaient les documents XML à la façon d'une longue chaîne de caractères dans des colonnes CLOB[1] tandis que d'autres découpaient les données XML comme les éléments et les attributs en plusieurs tables. Ce mécanisme est appelé mise en lambeau (*shredding* en anglais) d'un document XML.

L'expérience aidant et les besoins d'utiliser XML grandissant, les moteurs relationnels ont intégré un support direct de XML, à l'aide d'un type de données spécialisé. Un type natif XML a été défini et de nouvelles fonctions ont été développées comme, par exemple, transformer des données relationnelles en XML. De plus, une multitude d'outils ont été inventés pour interroger le contenu des documents XML stockés dans ce type natif XML. Les principaux sont XPath, XQuery et SQL/XML. Enfin, ces systèmes offrent un support des méta-données, bien souvent à l'aide d'XML Schéma.

De par leur ancienneté et leur fiabilité, les bases de données relationnelles sont une bonne solution pour gérer des données XML mais sont malheureusement conceptuellement mal adaptées. En effet, un document XML représente un arbre et n'est pas nécessairement aussi bien structuré qu'une table relationnelle.

[1] CLOB, *Character Large Object*, objet pouvant contenir une longue chaîne de caractères.

3.2 Bases de données XML natives

3.2.1 Définition

Le terme *Native XML Database (NXD)*, ou base de données XML native, est apparu pour la première fois dans la campagne de publicité pour Tamino, une base de données XML native de Software AG [20]. Depuis, grâce au succès de cette campagne, le terme est arrivé dans l'usage courant par différentes entreprises développant des produits similaires. Etant devenu un terme publicitaire, il n'a jamais eu de définition technique formelle. Une définition possible de ce qu'est une base de données XML native serait la suivante :

- ▷ Une base de données XML native définit un modèle logique pour un document XML. Elle stocke et récupère les documents suivant ce modèle de données. Au minimum, il doit inclure les éléments, les attributs, les données et l'ordre du document. Des exemples de tels modèles sont le modèle XPath, le XML Infoset et les modèles utilisés par DOM.
- ▷ Une base de données XML native gère le document XML comme une unité fondamentale de stockage, comme une ligne dans une table relationnelle.
- ▷ Les bases de données XML natives n'ont pas un modèle physique sous-jacent particulier. Par exemple, le modèle physique peut être relationnel, hiérarchique, orienté objet ou utiliser un format de stockage propriétaire comme des fichiers compressés indexés.

La première partie de cette définition est similaire à celle des autres types de bases de données, définissant le modèle utilisé pour le stockage et l'interrogation. Il existe un certain nombre de modèles pour XML comme Infoset et DOM. Le modèle choisi pour faire une base de données XML native doit être conçu pour supporter arbitrairement la profondeur de l'imbrication des nœuds, la complexité de leurs relations, leur ordre, leur identité, etc.

La seconde partie de cette définition explique que l'unité de stockage fondamentale dans une base de données native XML est le document XML. Bien qu'il semble possible qu'une base de données XML native puisse assigner ce rôle à des fragments de documents, l'unité de stockage fondamentale reste effectivement le document XML dans la plupart des bases de données XML actuelles.

La troisième partie de la définition pose que le modèle physique sous-jacent n'est pas important. C'est exact et c'est certainement le cas pour toutes les sortes de base de données. Le format de stockage physique utilisé par une base de données relationnelle n'est pas une condition nécessaire au caractère relationnel de la base. De plus, il est tout à fait envisageable d'utiliser un support relationnel pour fabriquer un moteur de base de données XML native comme eXist l'a fait à ses débuts.

Les bases de données XML natives sont donc des bases données conçues spécialement pour stocker des documents XML et comme les autres bases de données, elles gèrent les transactions, la sécurité, l'accès multi-utilisateurs, offrent des API de programmation, des langages de requêtes, etc. Les bases de données XML natives s'inscrivent donc parfaitement dans notre approche entièrement basée sur XML.

3.2.2 Indexation des documents XML

Dans le monde des bases de données, le format XML est une nouvelle approche pour modéliser l'information. L'implémentation d'un système permettant de stocker et d'interroger efficacement des documents requiert le développement de nouvelles techniques d'indexation.

Indexation structurelle

Un nombre important de recherches ont été faites récemment afin de concevoir des structures d'index correspondants aux besoins spécifiques de XML. Plusieurs plans de numérotation pour les documents XML ont été proposés [21, 22, 23, 24, 25, 26]. Un plan de numérotation assigne un identificateur unique à chaque nœud dans l'arbre logique du document, comme en traversant l'arbre en pré-ordre ou par niveau. Les identificateurs ainsi générés sont utilisés dans le plan d'indexation comme référence au nœud actuel. Un plan de numérotation doit fournir des mécanismes pour déterminer rapidement les relations structurelles entre une paire de nœuds et identifier toutes les occurrences d'une telle relation dans un document ou dans une collection de documents XML.

En résumé, un plan de numérotation doit supporter deux opérations basiques :

- ▷ **La décision :** Pour deux nœuds donnés, décider s'ils ont une relation spécifique, comme parent-enfant, ancêtre-descendant, frère-suivant, frère-précédent.
- ▷ **La reconstruction :** Pour un nœud donné, déterminer les identificateurs des nœuds de son voisinage comme, par exemple, le père, le frère suivant, le premier enfant, ...

Nous ne nous étendrons pas sur les différents plans de numérotation existant mais nous présenterons celui utilisé dans la base de données XML native que nous utilisons, eXist, dans la section 3.3.

Indexation basée sur les valeurs

Pour indexer des données selon leurs valeurs, les structures de données communément utilisées dans les bases de données sont les arbres B ou *B-tree* en anglais et leurs variantes, comme le *B+-tree* ou arbre B+. Ce type d'indexation utilisé dans le monde relationnel peut donc être réutilisé dans les bases de données XML natives.

Les arbres B [27] sont des arbres balancés et triés qui permettent l'insertion et la suppression de nœuds en complexité amortie logarithmique. La recherche dans un tel arbre est similaire à celle effectuée dans un arbre binaire de recherche, c'est-à-dire en parcourant l'arbre de haut en bas et en choisissant à chaque fois le fils correspondant à la fourchette de valeur que l'on recherche.

L'idée principale des arbres B est que ses nœuds peuvent avoir un nombre variable de fils dans une plage de valeurs déterminée. Quand un nœud est inséré ou supprimé de l'arbre, le nombre de fils d'un nœud varie et les nœuds sont restructurés afin de garder la structure définie. Ce type d'arbre ne doit pas être re-balancé aussi fréquemment qu'un arbre binaire de recherche auto-balancé classique mais peut prendre plus de place en mémoire car certains nœuds peuvent ne pas être complètement remplis.

3.3 eXist, une base de données XML native libre

3.3.1 Introduction

Le projet eXist est une implémentation libre (LGPL) d'un système de gestion de base de données XML native, interfaçable entre autres à l'aide de XPath, de XQuery et de XUpdate. Le projet a été entamé en 2000 par Wolfgang Meier, un développeur allemand. Il s'est basé sur les travaux de Shin, Jang et Jin [24] qui proposaient un système efficace d'indexation des documents structurés. Ce fut tout d'abord une expérience d'implémentation d'une indexation de documents XML à l'aide d'un système relationnel. Aujourd'hui, eXist n'utilise plus de relationnel et fonctionne sur un système de stockage propre. La communauté autour d'eXist ne cessant de croître et les développeurs étant très actifs, eXist est devenu un SGDB XML natif complet. La base de données est complètement écrite en Java et peut être déployée de multiple façons, aussi bien comme un processus serveur que dans un moteur de *servlet* ou encore directement intégré dans une application.

eXist fournit un stockage sans schéma des documents XML dans des collections hiérarchiques. Une collection est un ensemble qui peut contenir d'autres collections ou des documents XML. En utilisant une syntaxe étendue d'XPath et d'XQuery, les utilisateurs peuvent interroger différentes parties de la hiérarchie de collections, ou tous les documents contenus dans la base de données. Le moteur de requêtes d'eXist implémente un traitement de requête efficace et basé sur les indexes. Le plan d'indexation permet une identification rapide des relations structurelles entre les nœuds, comme la relation parent-enfant, ancêtre-descendant et frère-suivant, frère-précédent. Basée sur des algorithmes de jointures de chemins, une large fourchette d'expressions de chemin est traitée en utilisant uniquement les informations d'index. L'accès aux nœuds courants, stockés dans le magasin central de documents XML, n'est pas nécessaire pour ce type d'expressions.

La base de données convient bien aux applications manipulant des petites ou larges collections de documents XML qui sont occasionnellement mises à jour. Le logiciel a été conçu de sorte qu'il supporte les documents orientés données ou présentation. Cependant, l'interrogation de ces derniers n'est pas très bien supporté par les langages de requêtes XML comme XPath. eXist fournit donc un certain nombre d'extensions au standard XPath et XQuery pour traiter efficacement des requêtes de recherche textuelle, incluant entre autres la recherche par mot clé ou via des expressions régulières.

3.3.2 Architecture

eXist est bel et bien un système de gestion de base de données XML natif, conformément à notre définition vue à la section 3.2.1. En effet, un modèle logique pour les documents XML est défini et le document XML est son unité de stockage fondamentale.

Les détails d'implémentation concernant le stockage des données sont totalement séparés du cœur d'eXist. Tous les appels au système de stockage se font par des courtiers (*brokers* en anglais). Un courtier peut être vu comme une interface entre le cœur d'eXist et les systèmes de stockages. Ces classes courtiers fournissent un set d'instructions basiques comme ajouter, supprimer ou

récupérer des documents ou des fragments. De plus, elles possèdent des méthodes pour utiliser les indexes, comme par exemple récupérer un ensemble de nœuds correspondant à un certain nom. Les moteurs de requête XPath et XQuery sont implémentés de la même manière, comme des modules gravitant autour du cœur d'eXist. Nous remarquons ici un excellent découpage objet du logiciel et ce, dès le début, dans le respect des principes du génie logiciel. Il devrait donc être aisé de modifier ou d'ajouter certaines parties comme un nouveau langage ou un nouveau système de stockage. Une illustration de l'architecture d'eXist est proposée à la figure 3.1.

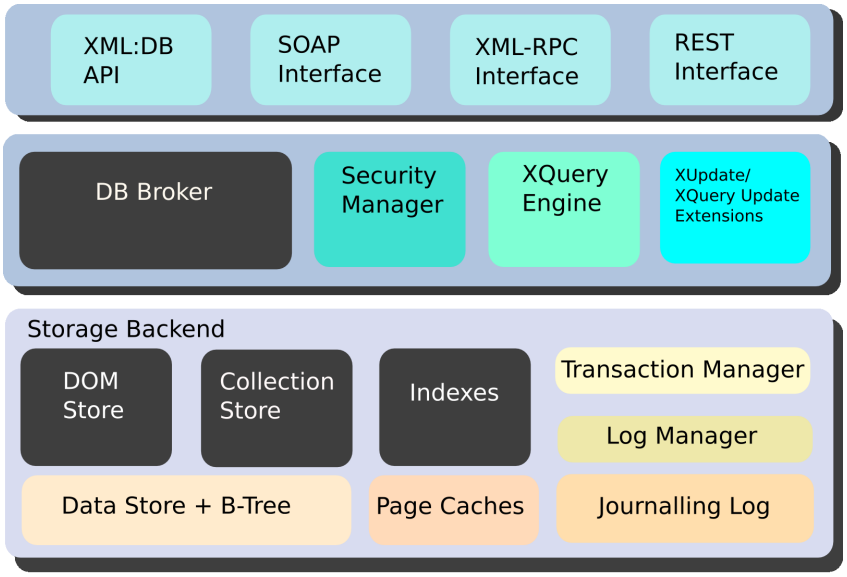


FIG. 3.1 – Architecture d'eXist ©Wolfgang Meier.

Système de stockage

eXist, au départ, utilisait uniquement un système relationnel. En effet, Wolfgang Meier voulait se concentrer sur la partie indexation des données. Les premières versions du logiciel ont servi à tester le système d'indexation pour vérifier son efficacité et évaluer ses performances. Depuis la version 0.6 en 2002, le système relationnel a été remplacé par un système maison. Pendant la période de transition, l'utilisateur avait le choix entre les systèmes de stockage, via un fichier de configuration. Cette option a maintenant disparue, depuis que de nombreuses recherches aient été effectuées sur le stockage efficace d'XML [28, 29, 30, 25]. Le test de ces différentes méthodes a pu se faire facilement, grâce à l'architecture modulaire bien pensée d'eXist.

Déploiement

eXist offre plusieurs modes de déploiement. Le moteur de base de données peut :

- s'exécuter comme un processus serveur, fournissant des interfaces HTTP, RESP et XML-RPC pour l'accès distant.

- être intégré dans d'autres applications, qui auront un accès direct à la base de données via l'API XML:DB
- s'exécuter dans un moteur de servlet comme Apache Tomcat. Les Servlets ou les Java Server Pages (JSP) s'exécutant dans la même contexte d'application pourront accéder directement à la base. L'accès distant est assuré par XML-RPC, SOAP, REST et WebDAV.

Il est donc très facile d'utiliser eXist dans un grand nombre d'applications. Par exemple, pour faire un service web, on choisira l'interface SOAP. Pour l'intégrer dans une application, on utilisera l'API XML:DB, etc. XML:DB est la méthode préférée pour accéder à eXist depuis une application Java. XML:DB est une initiative indépendante qui propose une interface commune pour l'accès aux bases de données XML natives ou toute autre base de données supportant XML. Cela permet aux développeurs d'écrire des applications portables, en travaillant avec différents produits implémentant l'interface XML:DB. Un client graphique Java est également fourni afin d'interagir rapidement avec le serveur d'eXist.

3.3.3 Indexation

Depuis juillet 2006, eXist est disponible en deux versions. La première, la branche 1.0, utilise le plan de numérotation virtuelle par niveau [31]. Beaucoup d'utilisateurs se plaignaient de ne pas pouvoir stocker certains documents XML, car ils étaient composés d'un nombre trop important de nœuds ou possédaient une structure trop complexe pour être indexés avec ce plan de numérotation. Fin 2005, les développeurs d'eXist ont donc décidé d'implémenter le plan d'indexation DLN de Boëhme et Rahm que nous présenterons plus loin dans cette section. Ce plan de numérotation permet d'indexer n'importe quel document XML, quelle que soit sa taille ou sa complexité. Il permet également l'insertion et la suppression d'éléments sans ré-indexation complète de l'arbre. La branche 1.1 sortie fin 2006 utilise ce plan de numérotation des nœuds. C'est maintenant la branche principale de développement.

Un point important à noter est qu'eXist gère ses indexes au niveau d'une collection et non au niveau d'un document lui-même. Dans la suite de cette section, nous présenterons les différents indexes disponibles dans eXist.

Index structurel

En 2004, Boëhme et Rahm ont proposé un plan de numérotation intéressant permettant de supprimer la limite de la taille des documents indexables et de mettre à jour les nœuds sans ré-indexation complète [32]. Ce plan est nommé numérotation dynamique par niveau ou DLN (*Dynamic Level Numbering* en anglais). Il est basé sur des identificateurs à longueur variable. C'est ce plan qui a été choisi en 2006 pour indexer structurellement les collections dans eXist.

Les identificateurs de ce plan de numérotation, appelés nombres dynamiques de niveau, sont inspirés de la classification décimale de Dewey, numérotation utilisée pour classer les livres dans les bibliothèques. Les identificateurs de Dewey sont une séquence de valeurs numériques séparées par des caractères spéciaux. Ils sont hiérarchiques. La racine du document possède

l'identificateur 1 tandis que tous les autres nœuds sont numérotés à l'aide de l'identificateur de leur nœud parent comme préfixe, suivit d'une valeur de niveau. Par exemple, pour un arbre simple : 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3, etc. Dans ce cas, 1 représente le nœud racine, 1.1 est le premier nœud du second niveau, 1.2 le second et ainsi de suite.

A l'aide de ce plan de numérotation, déterminer la relation entre deux nœuds donnés est une opération triviale et fonctionne aussi bien pour l'axe ancêtre-descendant que pour les relations frère-suivant et frère-précédent. Tous les axes de navigation XPath peuvent donc être gérés efficacement. Le problème principal est que ces identificateurs risquent d'avoir besoin d'un nombre plus important de bits pour leur encodage que dans les plans précédents. Il a donc fallu trouver un encodage efficace qui

- ▷ restreint l'espace de stockage nécessaire pour un identificateur et
- ▷ garantit une comparaison binaire correcte des identificateurs, en respectant l'ordre du document.

En effet, en fonction du niveau d'encapsulation des éléments dans le document XML, les identificateurs peuvent devenir très longs. Il est à noter qu'il n'est pas rare qu'un document XML contienne plus de 15 niveaux. Cet encodage est décrit dans [31].

En plus de supprimer la limite de taille du document XML indexable grâce à un encodage binaire bien pensé, ce plan de numérotation permet d'insérer facilement des nœuds dans l'arbre XML. En effet, pour éviter de devoir re-numéroter les nœuds après chaque insertion, suppression ou mise à jour, Boëhme et Rahm proposent l'idée de sous-niveaux. Entre deux nœuds 1.1 et 1.2, un nouveau nœud peut être inséré en 1.1/1, où / est le séparateur de sous-niveaux. 1.1 et 1.1/1 sont ainsi tous les deux sur le même niveau de l'arbre, dans le bon ordre. Pour éviter de perdre des performances à cause de ces sous-niveaux, une défragmentation de l'arbre DLN est prévue par eXist après un nombre donné d'insertions.

Indexes basés sur des valeurs

eXist possède plusieurs indexes basés sur les valeurs. A l'inverse de l'index structurel, ces indexes sont optionnels et leur configuration est laissée à l'utilisateur. Dans les lignes suivantes, nous allons présenter les différents indexes disponible dans eXist.

Index inversé. eXist possède un index inversé. Ce type d'index existe dans la plupart des systèmes de gestion de bases de données. L'index inversé est utilisé spécifiquement pour associer un mot à l'ensemble des positions dans lesquelles il apparaît dans ces documents de la base de données.

Index de portée. Un troisième type d'index est disponible dans eXist, l'index de portée (*range index* en anglais). Il s'agit d'un index basé sur les valeurs utilisant des arbres B comme présenté à la section 3.2.2. Cet index est spécifique aux types de données des valeurs des nœuds du document. Ces indexes fournissent un raccourci à la base de données afin de sélectionner les nœuds directement selon leurs valeurs. eXist, par défaut, n'utilise pas cet index car il est

incapable de déterminer le type des valeurs des nœuds de l'arbre XML. Il le pourrait à l'aide d'un XML Schéma mais cette fonctionnalité n'est pas encore implémentée et n'est pas prévue à court terme. De plus, il pourrait ne pas être efficace d'indexer tous les champs de la base. Cependant, si des indexes de portée sont utilisés, ils sont créés par eXist lors du chargement du document et sont actualisés automatiquement lors de ses mises à jour.

Les indexes de portée sont utilisés au besoin lors des comparaisons explicitement demandées via les opérateurs et fonctions XPath standards, si, bien sûr, ces indexes sont définis par l'utilisateur. Par défaut, si un tel index n'est pas défini, eXist procède à une inspection en force brute (*brute-force* en anglais) du magasin de données, ce qui est très coûteux en temps de calcul. Il s'agit ici donc d'un point important si l'on compte gérer une base de données importante avec eXist et que l'on a donc besoin de performance.

L'index de portée fournit au moteur de requête d'eXist une méthode plus efficace de conversion de données : au lieu de récupérer la valeur de chaque élément sélectionné et de la transformer en nombre réel afin de la comparer, eXist peut évaluer l'expression en utilisant l'index de portée par un simple parcours.

Autres indexes. D'autres indexes basés sur les valeurs sont disponibles mais sont en cours de développement et ne sont pas encore assez stables pour que nous les détaillions ici. Il s'agit notamment d'un index basé sur les noms des éléments permettant de retrouver facilement tous les éléments d'un certain nom quelle que soit leur imbrication. Depuis quelques mois, un grand travail de refactorisation de l'architecture d'indexation est entamé dans le but d'offrir à l'utilisateur la possibilité d'implémenter ses propres structures d'indexation. Un exemple d'utilisation serait un index pour des données géographiques représentées en GML, un modèle de données géographiques basé sur XML.

3.3.4 Evaluation des expressions

eXist va utiliser activement ses structures d'indexes pour traiter les requêtes XPath et XQuery. En effet, à l'aide de ces indexes, eXist est capable d'accéder aux nœuds par leur identificateur unique, de récupérer un ensemble d'identificateurs de nœuds correspondant à un certain nom d'élément ou à des mots clés. D'un point de vue implémentation, les courtiers ont des méthodes pour chacune de ces opérations simples.

En utilisant les fonctionnalités fournies par le plan d'indexation, le moteur de requête d'eXist est capable d'utiliser des algorithmes rapides de jointure de chemins pour traiter efficacement les expressions de chemin. Un certain nombre de tels algorithmes ont été proposés récemment. Celui utilisé dans eXist est basé sur l'algorithme proposé par Li et Moon [23].

Le processeur de requête d'eXist va d'abord décomposer l'expression de chemin donnée en une série d'étapes simples. Considérons l'expression XPath suivante :

```
/commande//client[Nom="Dupond"]
```

Cette expression est décomposée en 3 sous-expressions :

1. `commande//client`
2. `client[Nom]`
3. `Nom="Dupond"`

Notons l'ordre des sous-expressions : celles nécessitant un accès éventuel au magasin de données sont mises en fin de chaîne. En effet, sans index de portée configuré correctement, la troisième sous-expression impliquera un parcours du magasin de données, stocké sur disque, avec toutes les conséquences que cela peut avoir en terme de performances. L'évaluation de cette sous-expression est différée pour pouvoir filtrer les nœuds à récupérer dans le magasin de données afin de les comparer, de sorte qu'il y ait le moins de nœuds possibles à aller chercher sur disque. Des détails sur cet algorithmes sont disponibles dans [31].

Depuis quelques mois, le développeur principal d'eXist, Wolfgang Meier, travaille activement sur un nouveau moteur d'optimisation des expressions de chemin. Il s'agit ici d'un gros chantier qui devrait permettre un usage encore plus optimal des indexes disponibles ainsi qu'un meilleur ordonnancement dans l'évaluation des sous-expressions. Le but est évidemment de gagner du temps de calcul lors de l'évaluation de requêtes complexes. Ce moteur est loin d'être terminé et n'est pas encore stable. Nous n'avons donc pas pu le tester entièrement, celui-ci renvoyant encore de faux résultats.

3.3.5 Extensions textuelles

La spécification XPath ne définit qu'un ensemble limité de fonctions pour rechercher une chaîne de caractères dans le contenu d'un nœud. Cela peut être gênant si l'on veut traiter des documents XML contenant beaucoup de texte comme les documents XML orientés présentation. Pour rechercher des chapitres à propos des bases de données XML en respectant la norme XPath, il faudrait écrire une expression du type de la suivante, tirée du site web d'eXist :

```
//chapitre[contains(.,XML) and contains(.,databases)]
```

L'exécution de cette requête peut être assez lente car le moteur XPath va scanner entièrement le contenu des nœuds `chapitre` et de leurs descendants. De plus, il est possible que le mot `database` soit écrit avec une majuscule en début de phrase, de même si ce mot est utilisé au singulier. Dans ces cas, l'expression précédente n'est pas suffisante.

Pour résoudre ce problème, eXist fournit des opérateurs et des fonctions pour accéder efficacement au contenu textuel des nœuds en utilisant l'index inversé dont nous avons parlé à la page 19. Par exemple, eXist permet d'écrire la requête précédente de la façon suivante :

```
//chapitre[near(.,'XML database?',50)]
```

Cette expression va renvoyer tous les chapitres contenant les deux mots `XML` et `database` et dont la distance entre ces deux mots est inférieure à 50 mots. De plus, le caractère `?` dans `database?` permet de trouver les occurrences du mot qu'il soit au pluriel ou au singulier. Il est également intéressant de constater que ces recherches ne sont pas sensibles à la casse. Cela résout donc

entièrement le problème soulevé plus haut. Pour conclure sur cet exemple, comme la requête est basée sur les indexes, celle-ci va s'exécuter plus rapidement que son équivalent XPath standard.

eXist fournit deux opérateurs supplémentaires de comparaison afin d'améliorer la recherche textuelle dans des documents XML. Ces deux opérateurs sont `&=` et `|=`. Le premier opérateur permet de retrouver les nœuds contenant tous les mots d'une chaîne de caractères tandis que le second permet de déterminer les nœuds contenant au moins un mot d'une chaîne de caractères. Ces deux opérateurs utilisent bien sûr l'index inversé pour améliorer l'efficacité des requêtes. De plus, cette recherche n'est pas sensible à la casse. Tous ces opérateurs et extensions sont utilisables également avec des expressions régulières.

3.4 Conclusion

Dans ce chapitre, nous avons analysé en détail les bases de données XML natives et nous en avons présenté un exemple, eXist, une base de données XML native libre très prometteuse.

Dans le chapitre suivant, nous allons présenter l'analyse de données multidimensionnelles et les notions nécessaires à sa compréhension comme les entrepôts de données et les différents modèles multidimensionnels relationnels.

Chapitre 4

Analyse de données

Dans ce chapitre, nous allons présenter les notions nécessaires à la compréhension des entrepôts de données et de l'analyse multidimensionnelle.

Contenu

4.1 Motivations

Les entreprises, de par leur taille importante et leur ancienneté ont de plus en plus de difficultés à accéder à l'ensemble de leurs données, à regrouper leurs différentes bases de données disséminées dans leurs différents services afin de les analyser et de prendre des décisions rapidement.

Dans la grande distribution, par exemple, on aimerait déterminer les produits à succès, les modes, les habitudes d'achat des consommateurs globalement ou par secteur géographique. Dans le domaine des services tels que les banques, les entreprises de télécommunication, les assurances et mutualités, on aimerait pouvoir classifier les clients, détecter des fraudes ou les clients qui risquent d'être infidèles, etc.

Pour réaliser ces analyses facilement et rapidement, il convient de rassembler toutes les informations de l'entreprise dans une base unique, spécialement conçue pour l'analyse : un entrepôt de données.

4.2 Entrepôts de données

Un entrepôt de données (*data warehouse* en anglais) est un lieu de stockage intermédiaire de différentes données en vue de la constitution d'un système d'informations décisionnelles. Un des précurseurs du concept d'entrepôt de données, Bill Inmon [33], le définit comme suit :

> « Un entrepôt est une collection de données orientées sujet, intégrées, non volatiles et historisées, organisées pour le support d'un processus d'aide à la décision. »

Un entrepôt est donc une sorte de point focal stockant en un point unique toute l'information utile provenant des systèmes de production et des sources externes. Ralph Kimball, l'auteur de *The Data Warehouse Toolkit* [34], propose une autre définition :

> « Un entrepôt de données est une copie de données transactionnelles spécifiquement structurée pour l'interrogation et l'analyse. »

Trois fonctions essentielles sont nécessaires pour créer et utiliser un entrepôt :

- ▷ la collecte de données des différentes sources d'informations et leur intégration
- ▷ l'organisation des données dans l'entrepôt
- ▷ l'analyse de données pour la prise de décision en interaction avec les analystes.

Dans ce mémoire, nous nous concentrerons principalement sur ce dernier point, c'est-à-dire l'analyse de données.

4.3 Modélisation multidimensionnelle

Comme nous l'avons vu en début de chapitre, un entrepôt de données est orienté sujet et doit permettre d'analyser des données suivant plusieurs dimensions. Les bases de données multidimensionnelles, à la différence des bases de données relationnelles classiques permettent d'effectuer des traitements sur des données en prenant en compte plus de deux axes : les données sont modélisées à un certain niveau d'abstraction sous forme d'hyper-cubes (ou cubes de données, *data cubes* en anglais).

Ces structures sont essentiellement utilisées par des analystes qui cherchent à trouver des tendances dans de grandes quantités de données. Par exemple, rechercher les facteurs de risques (âge, sexe, traitement, ...) de maladies pour une étude médicale, caractériser des données (type d'objet, siècle, ...) dans le cadre de fouilles archéologiques ou déterminer des corrélations entre produits-magasins-vendeurs pour le directeur de ressources humaines de chaînes de distribution.

4.3.1 Notions

Un sujet est défini par un ensemble de mesures et un ensemble de dimensions.

Une dimension est une liste d'éléments organisés de façon hiérarchique. Les granularités d'une dimension sont ses différents niveaux hiérarchiques. Certains cas spéciaux sont à gérer si les dimensions sont organisées en hiérarchies multiples : les magasins d'une entreprise peuvent

être organisées par ville, par province, par secteur ou par région de vente. Une région de vente peut regrouper certaines villes de provinces différentes. Malinowski *et al* ont fait une étude exhaustive des différents types de hiérarchies de dimensions dans [35].

Un fait est composé des valeurs des ses mesures, mesurées ou calculées selon un membre de chacune des dimensions. Les faits contiennent également les clés vers les membres de chacune de leurs dimensions.

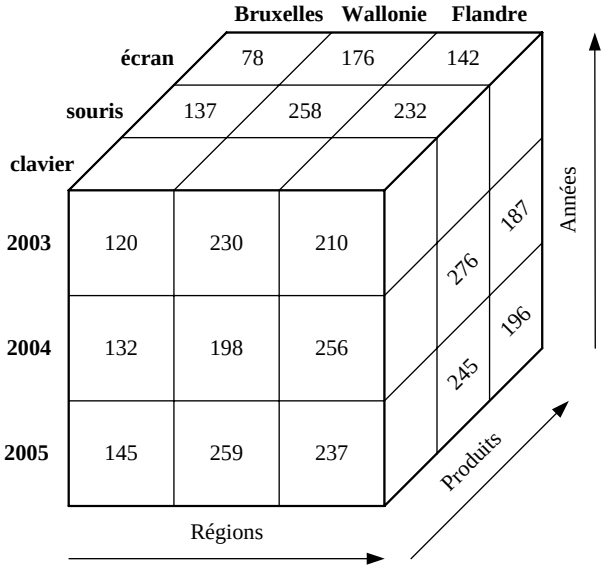


FIG. 4.1 – Représentation d'un cube de données.

Tout cela permet de placer les données dans des matrices multidimensionnelles appelées cubes. Un exemple d'un tel cube est présenté à la figure 4.1 Ces cubes sont parfois appelés hyper-cubes s'il y a plus de 3 dimensions. Les données pourront être interrogées directement et facilement sur n'importe quelle combinaison de dimensions, sans utiliser de requêtes trop complexes.

Les opérations *slice* et *dice* permettent respectivement d'extraire une tranche du cube et un sous-cube selon des prédicats sur les dimensions. Deux autres opérations importantes sont possibles sur les bases de données multidimensionnelles : *roll-up* et *drill-down*. Ces deux opérations permettent de naviguer dans les données en suivant les hiérarchies de dimensions. La première, *roll-up*, permet d'agréger les données en suivant une dimension. La deuxième, *drill-down*, permet de faire le contraire, c'est-à-dire de détailler les données. Par exemple, un *roll-up* sur la dimension temporelle nous permet de passer d'une vue par trimestre à une vue par année. Un *drill-down* de passer d'une vue par année à une vue par trimestre.

Dans ce mémoire, nous nous concentrerons principalement sur l'extraction de cubes aux dimensions choisies à partir d'une base de données multidimensionnelles en XML.

4.3.2 Modèles relationels de données

Modèle en étoile

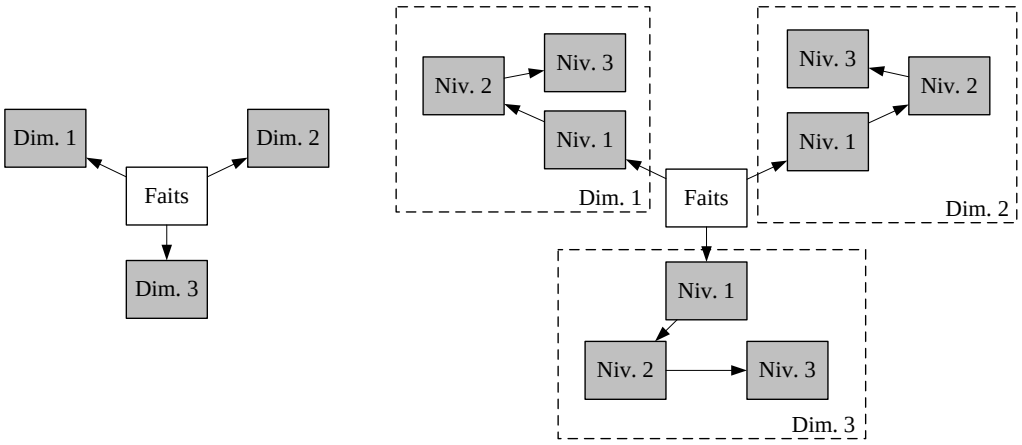


FIG. 4.2 – Illustration d'un schéma en étoile et d'un schéma en flocons.

La modélisation en étoile (*star schema* en anglais) agence les données d'une façon très différente de la structure en troisième forme normale (3FN) fréquemment utilisée par les modélisateurs des systèmes OLTP. Une illustration d'un tel schéma est présentée à la gauche de la figure 4.2. Ce modèle consiste en une grande table de faits et un cercle d'autres tables qui contiennent les dimensions. Les tables représentant les dimensions contiennent toutes les les informations à propos de la dimension, c'est-à-dire tous ses niveaux de granularité. Ces tables contiennent donc beaucoup de redondance.

On peut citer, comme avantages, la facilité de lecture et la faible complexité des requêtes. En effet, peu de jointures sont nécessaires car le nombre de tables reste faible. Par contre, il y a beaucoup de redondance dans les tables de dimensions, ce qui entraîne un stockage lourd et une alimentation complexe de la base de données.

Modèle en flocons de neige

Le schéma en flocons de neige (*snowflake schema* en anglais) est une variante du schéma en étoile. Une illustration d'un tel schéma est présentée à la droite de la figure 4.2. Dans la théorie, la différence réside dans la simple normalisation des tables de dimensions en 3FN. Il est donc tout simplement question de mettre les attributs de chaque niveau hiérarchique dans une table de dimension distincte pour éviter la redondance. Bien que ce modèle permet de gagner de l'espace disque et facilite l'alimentation, il implique de nombreuses jointures dans les requêtes et donc une difficulté d'écriture de ces dernières.

4.4 Analyse de données

Il faut différencier deux types d'analyse de données : le *data mining* ou forage de données et l'analyse multidimensionnelle (OLAP). Dans ce mémoire, nous nous concentrerons principalement sur l'analyse multidimensionnelle.

Le forage de données (*data mining*) a pour but de mettre en évidence des corrélations éventuelles dans un volume important de données afin de dégager des tendances. Il s'appuie sur des techniques d'intelligence artificielle comme des réseaux de neurones ou sur des techniques statistiques afin de mettre en évidence des liens cachés entre les données et ainsi prévoir des tendances.

Online Analytical Processing (OLAP) est un terme commercial qui désigne les bases de données multidimensionnelles (aussi appelées cubes ou hyper-cubes) destinées à l'analyse et il s'oppose au terme OLTP qui désigne les systèmes transactionnels. Ce terme a été défini par E. F. Codd [36] il y a plus de 10 ans au travers de 12 règles que doit respecter une base de données si elle veut adhérer au concept OLAP. Les 12 règles de Codd sont :

- ▷ Vue conceptuelle multidimensionnelle
- ▷ Transparence
- ▷ Accessibilité
- ▷ Constance des temps de réponses
- ▷ Architecture Client/Serveur
- ▷ Indépendance des dimensions
- ▷ Gestion des matrices creuses
- ▷ Accès multi-utilisateurs
- ▷ Pas de restrictions sur les opérations inter et intra dimensions
- ▷ Manipulation des données aisée
- ▷ Simplicité des rapports
- ▷ Nombre illimité de dimensions et nombre illimité d'éléments sur les dimensions

Codd a défini ces règles sur demande de la compagnie Arbor Software, devenue aujourd'hui Hyperion, un grand nom dans les entrepôts de données. C'est pourquoi ces règles sont controversées de par leur origine commerciale. De plus, ce terme ne donne ni une définition ni une description claire de ce que signifie OLAP. Il ne donne non plus aucune indication des motifs d'utilisation d'outils OLAP, ni de leurs particularités. Nigel Pendse [37] a redéfini le terme OLAP comme un système d'analyse rapide d'informations multidimensionnelles partagées (FASMI, *Fast Analysis of Shared Multidimensional Information*), ce qui nous paraît être une définition plus appropriée.

Les opérations que l'on souhaite pouvoir faire grâce à ce genre d'outils sont les opérations classiques que l'on peut effectuer sur un cube ou un hyper-cube. Nous avons détaillé ces différentes opérations un peu plus haut dans ce chapitre, à la page 25. Bien souvent, ces opérations

demandent l'agrégation de mesures suivant plusieurs dimensions et les outils OLAP procèdent généralement à des pré-agrégations qui permettent d'accélérer les requêtes.

Il existe des langages spécifiques, comme Microsoft MDX, qui permettent d'écrire ces opérations de manière simplifiée. Ces langages combinés à des outils graphiques permettent aux décideurs d'utiliser ces outils sans assistance. Il faut bien se rendre compte que ce genre de système est destiné à être utilisé par des personnes non informaticiennes et leur accessibilité est donc importante.

Les produits les plus utilisés actuellement, selon une étude de *The OLAP Report*[1], un site influent dans le domaine, sont SQL Server (28% du marché), Hyperion (21%) et Cognos (14%). Le marché paraît donc encore assez ouvert, même si Microsoft semble le dominer.SAP et Oracle se placent respectivement à la 5^e et 10^e place. Il existe aussi quelques systèmes libres comme Mondrian et Palo.

4.5 Conclusion

Dans ce chapitre nous avons introduit les entrepôts de données et l'analyse que l'on peut faire sur de tels systèmes. Nous avons également présenté les différents modèles de données multidimensionnels existant pour les systèmes relationnels.

Dans le chapitre suivant, nous allons présenter différentes façons de modéliser en XML des données multidimensionnelles.

[1] `http://www.olapreport.com`

Chapitre 5

Modélisation multidimensionnelle en XML

Jusqu'ici, nous avons essayé de donner tous les pré-requis nécessaires à la compréhension de notre travail. Nous avons présenté le méta-langage XML ainsi que les outils permettant d'interroger des données représentées dans ce langage. Nous avons également introduit l'analyse de données multidimensionnelles. Dans ce chapitre, nous allons joindre ces deux grands sujets en présentant différentes manières de modéliser des données multidimensionnelles grâce au langage XML. Pour ce faire, nous commencerons par décrire un exemple de travail et nous détaillerons quatre façons le modéliser à l'aide du langage XML. Enfin, pour chaque modèle, nous tenterons d'analyser intuitivement leurs avantages et inconvénients.

Contenu

5.1 Exemple de travail

Dans le but de faciliter la compréhension et de permettre une comparaison, nous utiliserons le même exemple tout au long de ce mémoire. Il s'agit d'un scénario d'entrepôt de données comprenant un ensemble de faits et deux dimensions. Celui-ci est délibérément très simple afin de rendre plus agréable la lecture.

Les faits sont les commandes de produits (`order`) des clients d'une multinationale fictive de vente de meubles. Les mesures des faits sont sont le prix (`price`), la devise (`currency`) et le nombre de produits commandés (`quantity`). Chaque commande possède deux dimensions : le produit commandé (`product`) et le client (`customer`). Ces deux hiérarchies de dimensions

possèdent trois niveaux de granularité et sont strictes, c'est-à-dire qu'il existe une relation 1-n entre un niveau de granularité et le niveau inférieur. Dans cet exemple, un produit fera donc partie d'une et une seule catégorie (`category`). Un schéma relationnel en flocon de ce scénario est illustré à la figure 5.1 et des tables d'exemple à la figure 5.2.

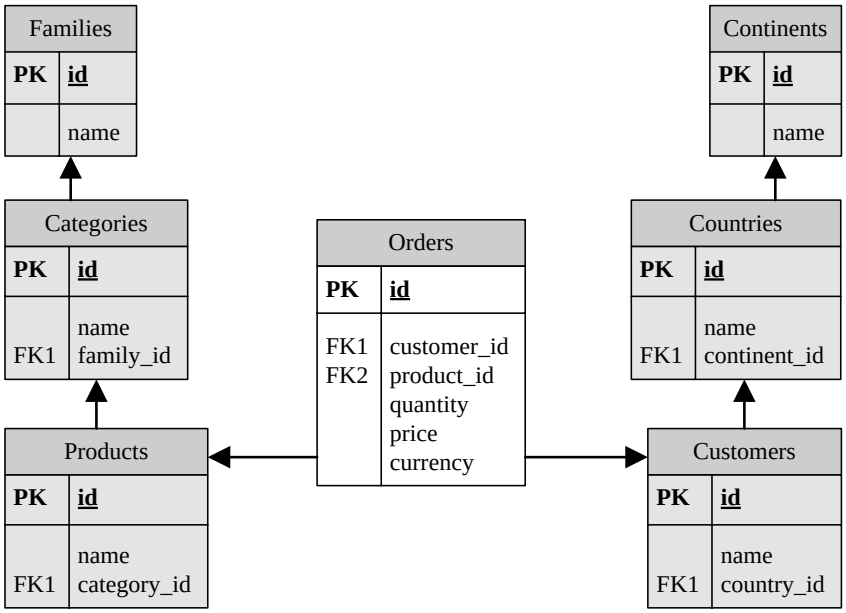


FIG. 5.1 – Schéma en flocons de l'exemple de travail.

Orders

id	customer_id	product_id	quantity	price	currency
ORD19	CUS98	PRO3	3	125,67	EUR

Customers

id	name	country_id
CUS98	Jean Dupond	COU77

Countries

id	name	continent_id
COU77	Belgium	CON1

Continents

id	name
CON1	Europe

Products

id	name	category_id
PRO3	Table	CAT58

Categories

id	name	family_id
CAT58	Kitchen	FAM6

Families

id	name
FAM6	Furniture

FIG. 5.2 – Version relationnelle de la base de données d'exemple.

5.2 Modélisation plate

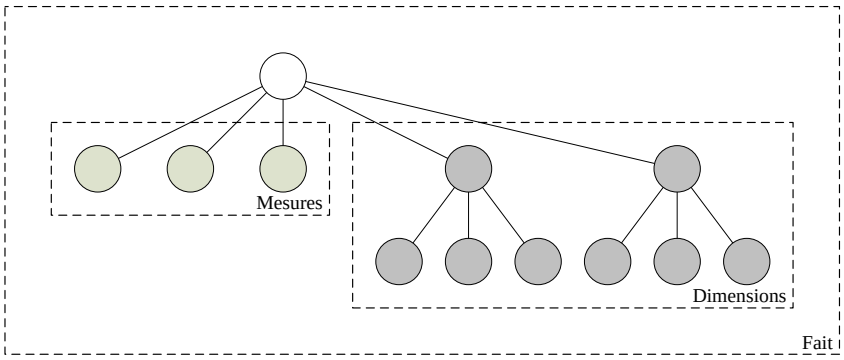


FIG. 5.3 – Illustration de la modélisation plate.

La façon la plus simple de construire un cube de données en XML est de représenter chaque fait par un élément XML englobant toutes les informations possibles. C'est intuitivement la solution la plus facile à appliquer et c'est celle qui est communément utilisée, par exemple dans les travaux de Beyer *et al.* [38]. Cette représentation n'a pas besoin de mécanisme de relation puisque toutes les informations nécessaires sont inclues dans l'élément représentant le fait. Ces informations, c'est-à-dire les dimensions et les mesures, sont exprimées de manière plate à l'image d'un schéma en étoile dénormalisé, d'où le nom de modélisation plate. Un schéma d'un fait du modèle plat est disponible à la figure 5.3. Un fait de notre exemple de travail suivant ce modèle est illustré à la figure 5.4.

Plus formellement, une instance d'un niveau de granularité est représentée par un élément dont le nom est le nom du niveau et dont le contenu textuel représente le nom de l'instance. Une dimension est représentée par un élément dont le nom correspond au nom de la dimension et qui contient les différentes instances des niveaux de sa hiérarchie. Une mesure d'un fait est représentée par un élément du nom de la mesure et la valeur de la mesure est contenue dans cet élément. Enfin, un fait est modélisé par un élément qui porte le nom du fait et qui contient toutes ses mesures ainsi que les instances de ses dimensions.

Cette modélisation offre plusieurs avantages. Tout d'abord, elle permet de traduire facilement des données relationnelles suivant un schéma en étoile vers le format XML, ce qui peut être intéressant pour fédérer des données relationnelles avec des données XML ou pour transmettre ces données d'une application à une autre. Ensuite, les documents obtenus peuvent peuvent être lus aisément par l'utilisateur. En effet, toutes les informations nécessaires à propos d'un fait se retrouvent dans le même document. Ceci devrait également faciliter l'interrogation puisqu'il ne faudra pas faire de jointures avec d'autres documents XML pour obtenir les informations de dimensions.

Intuitivement, nous pouvons trouver quelques désavantages à cette modélisation. En effet, ce modèle implique énormément de redondance dans les données des dimensions. Une telle redondance peut impliquer des difficultés de mise à jour et de maintenance de la base de données. Pour la même raison, la consommation en espace disque de ce modèle risque d'être également un désavantage.

```
<order>
  <price>125,67</price>
  <currency>EUR</currency>
  <quantity>3</quantity>
  <customer_dimension>
    <customer>Jean Dupond</customer>
    <country>Belgium</country>
    <continent>Europe</continent>
  </customer_dimension>
  <product_dimension>
    <product>Table</product>
    <category>Kitchen</category>
    <family>Furniture</family>
  </product_dimension>
</order>
```

FIG. 5.4 – Modélisation plate d'un fait.

5.2.1 Variante X-Warehousing

Boussaïd *et al.* proposent dans [39] une autre forme de modélisation multidimensionnelle en XML très proche de la modélisation plate présentée plus haut dans cette section. En effet, chaque fait est également représenté par un élément XML (*XML fact*) différent et celui-ci contient toutes les informations concernant les dimensions. Deux formes sont proposées. La première est basée sur un schéma en étoile et la seconde sur un schéma en constellation. Dans cet article, nous ne considérerons que la deuxième forme car la première est presque identique à notre modélisation plate. Un fait de notre exemple de travail traduit suivant ce modèle est illustré à la figure 5.5.

Par rapport à la modélisation plate, nous constatons deux différences. Tout d'abord, les valeurs des mesures et les noms des niveaux de dimensions sont exprimés à l'aide d'attributs plutôt qu'à l'aide de valeurs d'éléments. Cette différence est minime et ne devrait pas impliquer de grands changements dans l'interrogation. Deuxièmement, les hiérarchies de dimensions ne sont plus représentées de manière plate mais de manière arborescente, du niveau le plus fin au niveau le plus grossier.

Dans le modèle X-Warehousing, une instance de niveau de granularité est représentée par un élément dont le nom est le nom du niveau et qui possède un attribut `name` dont la valeur est le nom de l'instance. Cet élément contient récursivement son instance de niveau supérieur s'il existe un tel niveau. Une instance d'une dimension est représentée par l'instance de son niveau le plus haut dans la hiérarchie. Une mesure d'un fait est représenté par un attribut du nom de la mesure et la valeur de la mesure est assignée à cet attribut. Enfin, un fait est modélisé par un élément qui porte le nom du fait et qui contient toutes ses mesures en attributs et les instances de ses dimensions.

Intuitivement, les avantages et inconvénients semblent similaires à ceux de la modélisation plate. En effet, les documents ainsi constitués sont aisés à lire et contiennent toute l'information

```
<order price="125,67" currency="EUR" quantity="3">
  <customer name="Jean Dupond">
    <country name="Belgium">
      <continent name="Europe"/>
    </country>
  </customer>
  <product name="Table">
    <category name="Kitchen">
      <family name="Furniture"/>
    </category>
  </product>
</order>
```

FIG. 5.5 – Modélisation X-Warehousing d'un fait de l'exemple de travail.

nécessaire mais une redondance importante est présente ce qui devrait impliquer des difficultés en ce qui concerne les mises à jour et le stockage.

Notons cependant qu'un grand avantage de la modélisation X-Warehousing par rapport à la modélisation plate est qu'elle utilise adéquatement les imbrications d'éléments XML pour modéliser les dimensions. Grâce à cela, on peut connaître la profondeur des niveaux des hiérarchies et leur imbrication alors que cette information n'était pas contenue dans le modèle plat. On gagne donc en sémantique en utilisant ce modèle.

5.3 Modélisation hiérarchique

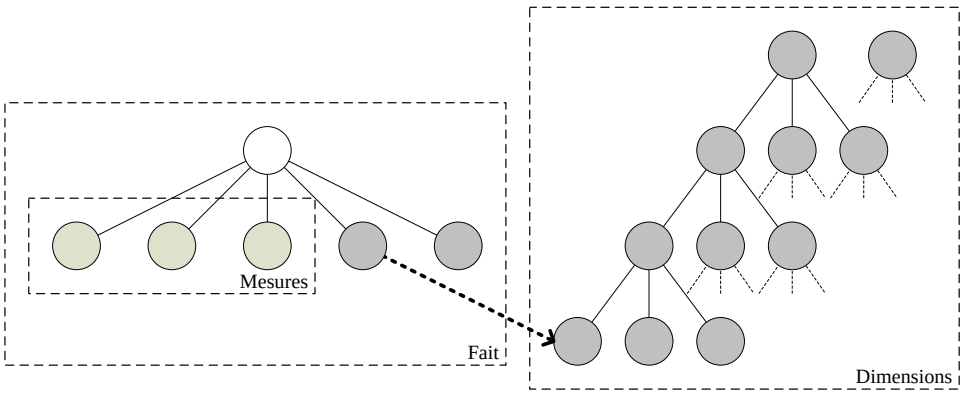


FIG. 5.6 – Illustration de la modélisation hiérarchique.

Le format XML, par définition, permet de modéliser aisément des structures arborescentes. Or les dimensions dans les systèmes d'analyse sont bien souvent hiérarchiques. Pourquoi donc ne pas utiliser la structure arborescente inhérente au format XML pour modéliser les dimensions

comme le suggèrent Bordawekar *et al.* dans [40] ? C'est ce à quoi nous allons tenter de répondre en proposant un modèle utilisant cette structure.

Dans le modèle arborescent que nous proposons, chaque hiérarchie de dimension est représentée par un fichier XML. La relation père-fils entre éléments XML représente la relation d'appartenance des niveaux de la hiérarchie de dimensions. Les feuilles de l'arbre représentent le plus fin niveau de granularité de la hiérarchie de dimension et possèdent un identificateur. Comme dans les autres modèles, chaque fait est contenu dans un élément XML différent et possède les clés d'identification des niveaux les plus fins de ses dimensions. Ce modèle emprunte donc au monde relationnel son mécanisme de clés étrangères et utilise la structure arborescente de XML pour modéliser les hiérarchies. Une illustration schématique de cet exemple est disponible à la figure 5.6. Une traduction d'une partie de la dimension `products` ainsi que d'un fait sont présentés respectivement aux figures 5.7 et 5.8.

De manière plus formelle, une instance d'un niveau de granularité est représentée par un élément dont le nom est le nom du niveau et qui possède un attribut `name` qui correspond au nom de l'instance. Cet élément contient récursivement son instance de niveau inférieure s'il existe un tel niveau. L'élément représentant l'instance du niveau le plus bas possède un attribut supplémentaire `id` unique servant, comme son nom l'indique, d'identificateur pour cette instance. Une instance d'une dimension est représentée par un élément du nom de la dimension contenant un attribut `ref` correspondant à l'identificateur de l'instance de son niveau de granularité le plus bas. Une mesure d'un fait est représenté par un attribut du nom de la mesure dont la valeur est la valeur de la mesure. Enfin, un fait est modélisé par un élément qui porte le nom du fait et qui contient toutes ses mesures ainsi les instances de ses dimensions.

Il est à noter que dans les exemples de cette section, nous avons utilisé le mécanisme de liaison ID/IDREF de XML mais dans la pratique, n'importe quel mécanisme de liaison pourrait être utilisé en fonction des besoins comme XInclude, XPointer ou toute autre système relationnel.

```
<products>
  <family name="Furniture">
    <category name="Kitchen">
      <product name="Table" id="PRO1"/>
      <product name="Cupboard" id="PRO2"/>
    </category>
    <category name="Bathroom">
      <!- ... ->
    </category>
  </family>
  <!- ... ->
</products>
```

FIG. 5.7 – Modélisation arborescente d'une partie de la dimension `products`.

Un avantage de modéliser les dimensions de façon arborescente est que celles-ci seront en théorie adaptées aux langages d'interrogations XML comme XPath et XQuery, langages spé-

```
<order price="125,67" currency="EUR" quantity="3">
  <product ref="PRO1"/>
  <customer ref="CUS98"/>
</order>
```

FIG. 5.8 – Exemple d'un fait dans la modélisation arborescente.

cialement conçus pour le parcours de structures arborescentes. Un autre avantage par rapport aux modèles présentés ci-dessus est qu'il n'y a plus aucune redondance dans les informations ce qui devrait impliquer une consommation disque moindre. Ce dernier avantage devrait également faciliter les modifications et les mises à jour. Par exemple, si l'on veut changer un produit de famille, il ne faudra réaliser qu'une modification minime au document représentant la dimension des produits.

Du côté des désavantages, lors de l'interrogation, il faudra prêter une attention particulière aux coûts des jointures requises par le mécanisme de liaison entre les dimensions et les faits. Un autre désavantage est que ce format est moins lisible puisque peu d'informations sont contenues dans les faits. Cependant, c'est le prix à payer pour meilleure flexibilité et pour moins de redondance.

5.4 Modélisation XCube

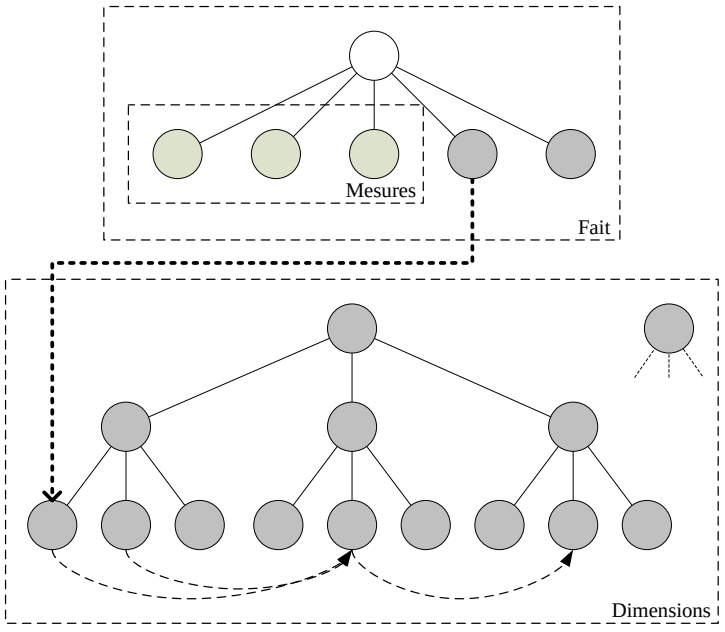


FIG. 5.9 – Illustration de la modélisation XCube.

XCube [41] est un format d'échange de données multidimensionnelles qui permet de représenter un cube de données dans plusieurs fichiers XML :

- ▷ un *XCubeSchema* qui décrit la structure multidimensionnelle du cube,
- ▷ plusieurs *XCubeDimension* qui représentent les différentes hiérarchies de dimensions et
- ▷ un *XCubeFact* qui définit les cases du cube de données, c'est-à-dire les faits et mesures.

XCube fournit également un protocole d'interrogation basé sur XML, *XCubeQuery*, pour interroger ces données dans une architecture OLAP client-serveur.

Ce modèle ressemble au modèle hiérarchique dans le sens qu'il utilise un mécanisme de liaison relationnel entre ses éléments. En effet, les faits et les dimensions sont représentés dans des documents séparés et sont liés via des identificateurs. Par contre, les dimensions ne sont pas représentées de façon arborescente comme dans le modèle hiérarchique, les instances des niveaux étant liées entre elles par des identificateurs et non pas par la relation père-fils XML. Les faits sont représentés d'une manière très similaire au modèle hiérarchique : ceux-ci sont composés de leurs mesures ainsi que des références vers les instances de leurs dimensions.

Une illustration schématique de ce modèle est disponible à la figure 5.9. Une traduction de notre exemple de travail dans ce modèle est présentée aux figures 5.10, 5.11 et 5.12. La figure 5.10, un *XCubeSchema*, représente la structure de notre cube. La figure 5.11, un *XCube-Dimension*, décrit les instances de chaque niveau de la hiérarchie de dimension `customers` ainsi qu'un lien vers l'instance de niveau supérieur. Enfin la figure 5.12, un *XCubeFact*, décrit les cases du cube de données multidimensionnelles, c'est-à-dire les mesures des faits et les références à leurs dimensions.

```
<multidimensionalSchema version="0.4">
  <cubeSchema id="order">
    <fact id="price"/>
    <fact id="quantity"/>
    <dimension id="products" granularity="product"/>
    <dimension id="customers" granularity="customer"/>
  </cubeSchema>
  <classSchema>
    <!- customers ->
    <classLevel id="customer">
      <attribute id="name"/>
      <rollUp toLevel="country"/>
    </classLevel>
    <classLevel id="country">
      <attribute id="name"/>
      <rollUp toLevel="continent"/>
    </classLevel>
    <!- ... ->
  </classSchema>
</multidimensionalSchema>
```

FIG. 5.10 – *XCubeSchema* de notre exemple de travail.

```
<dimensionData version="0.4">
  <units>
    <entry unitType="currency" unit="EUR"/>
  </units>
  <classification>
    <level id="continent">
      <node id="CON1" name="Europe"/>
      <!- ... ->
    </level>
    <level id="country">
      <node id="COU77" name="Belgium">
        <rollUp toNode="CON1" level="continent"/>
      </node>
      <!- ... ->
    </level>
    <level id="customer">
      <node id="CUS98" name="Jean Dupond">
        <rollUp toNode="COU77" level="country"/>
      </node>
      <!- ... ->
    </level>
  </classification>
</dimensionData>
```

FIG. 5.11 – *XCubeDimension* de la dimension `customers`.

Le modèle XCube possède des avantages similaires au modèle hiérarchique en ce qui concerne la flexibilité et l'absence de redondance dans les données. Ce dernier point facilite grandement les mises à jour et la maintenance de la base de données. Bien que ce modèle soit très verbeux, celui-ci devrait consommer moins d'espace disque que le modèle plat une fois que le nombre de faits devient suffisamment important.

Ces avantages en font un bon format d'échange de données multidimensionnelles mais le fait de l'utilisation abusive de références pourrait poser des problèmes de performance et d'écritures de requêtes si ce modèle est utilisé pour stocker physiquement les données. En effet, un nombre important de jointures seront nécessaires pour naviguer dans les hiérarchies de dimensions ce qui aura un impact certain sur les performances d'interrogation.

5.5 Conclusion

Dans ce chapitre, nous avons présenté différents modèles permettant de représenter des données multidimensionnelles à l'aide du langage XML. Il s'agit des modèles plats, hiérarchique et XCube.

```
<cubeFacts version="0.4">
  <cube id="order">
    <cell>
      <dimension id="customers" node="COU77"/>
      <dimension id="products" node="PRO3"/>
      <fact id="price" value="125,67"/>
      <fact id="quantity" value="3"/>
    </cell>
    <!- ... ->
  </cube>
</cubeFacts>
```

FIG. 5.12 – *XCubeFact* de notre exemple de travail.

Dans les chapitres suivants, nous allons analyser les moyens d'interroger ces données à l'aide du langage de requêtes XQuery.

Chapitre 6

Interrogation multidimensionnelle avec XQuery

Dans le chapitre précédent, nous avons présenté différentes façons de modéliser des données multidimensionnelles dans le format XML.

Dans ce chapitre et dans le chapitre suivant, nous allons tenter d'interroger de façon multidimensionnelle des données modélisées dans ces différents modèles. Plusieurs critères entreront en compte dans notre analyse comme par exemple, la quantité de jointures nécessaires, la complexité en grand O ou encore la facilité d'écriture et de lecture des requêtes.

Contenu

6.1 Introduction

Les outils OLAP existants permettent d'afficher les données sous la forme d'un tableau multidimensionnel dynamique dans lequel les mesures des faits sont agrégées selon différents critères et dimensions. Un exemple bien connu d'un tel outil est la fonctionnalité de tableaux croisés dynamiques de Microsoft Excel.

Pour obtenir un tel tableau, il faut réaliser une extraction de sous-cube, opération très classique dans le domaine de l'analyse. Une extraction de sous-cube consiste à choisir des dimensions d'analyse ainsi que leur niveau de granularité, les mesures à observer et une fonction d'agrégation. Il est également possible d'appliquer des filtres sur les données et les dimensions. Il s'agit de grouper les faits selon les niveaux de granularité choisis et d'agréger les mesures des faits à l'aide

la fonction spécifiée. Une fois le sous-cube extrait, les interfaces utilisateur OLAP permettent de le modifier dynamiquement, par exemple en ajoutant des filtres ou en changeant le niveau d'une dimension.

Les requêtes d'extraction de sous-cubes sont très gourmandes en temps de calcul car elles impliquent le parcours et l'agrégation d'un important ensemble de données. Pour pallier ce problème, les moteurs OLAP utilisent bien souvent des tables d'agrégation contenant des données pré-agrégées afin d'accélérer le traitement de ces requêtes. Nous ne prendrons pas en compte cet aspect dans ce mémoire.

6.2 Requêtes de travail

Tout au long de ce chapitre ainsi que du chapitre suivant, nous allons utiliser les mêmes requêtes afin de pouvoir comparer l'interrogation des modèles présentés au chapitre précédent. Il s'agit de requêtes d'extraction de sous-cubes à une et à deux dimensions.

Requête 1 *Pour chaque catégorie de produit, donner la somme des montants des commandes correspondantes.*

La première requête est très simple. Il s'agit d'une extraction de sous-cube à une dimension. On interroge l'ensemble de la base de données pour connaître le chiffre d'affaires de chaque catégorie de produit. Il n'y a qu'une dimension – les catégories de produits – et aucun filtre particulier n'est appliqué. Le nombre attendu de résultats est au maximum le nombre de catégories de produits. En effet, certaines catégories pourraient être vides ou ne contenir aucun produit commandé.

Requête 2 *Pour chaque catégorie de produit et pour chaque pays, donner la somme des montants des commandes correspondantes.*

La seconde requête est similaire à la première mais extrait un sous-cube à deux dimensions – les catégories de produits et le pays du client – plutôt qu'une seule. Pour les mêmes raisons qu'énoncées précédemment, le nombre de résultats attendu est au plus le produit du nombre de catégories par le nombre de pays.

6.3 Interrogation du modèle plat

6.3.1 Traduction des requêtes

La requête 1 traduite en XQuery pour des données modélisées de façon plate est présentée à la figure 6.1. Comme nous le constaterons, le principe de base des requêtes sur les différents modèles est comparable. C'est pourquoi, nous allons détailler les différentes étapes de cette première requête.

```
for $cat in distinct-values(/orders/order/product_dimension/category)
let $facts := /orders/order[product_dimension/category eq $cat]
where exists($facts)
return
<group>
  <product_category>{$category}</product_category>
  <sum>{sum($facts/price)}</sum>
</group>
```

FIG. 6.1 – Traduction de la requête 1 pour le modèle plat.

1. Tout d'abord, il faut déterminer les valeurs distinctes du niveau choisi de la dimension. Pour ce faire, XQuery possède une fonction `distinct-values()` qui renvoie une collection contenant les valeurs différentes des nœuds donnés en argument (ligne 1). Il faut par la suite itérer sur cette collection de valeurs à l'aide d'une clause `for` d'XQuery.
2. Ensuite, à chaque itération de la clause `for`, les faits correspondants à l'instance du niveau courant sont rapatriés via la clause `let` assignant à la variable `$facts` le résultat d'une expression XPath. Cette expression réalise une auto-jointure (*self join*, jointure d'un document sur lui même) à l'aide d'un prédicat vérifiant l'égalité des catégories (ligne 2).
3. Afin de ne pas renvoyer de groupes vides, un test est réalisé grâce à la clause `where` (ligne 3).
4. Pour finir, l'agrégation est calculée et les résultats sont construits à l'aide de la clause `return` (lignes 4-8)

```
for $cat in distinct-values(/orders/order/product_dimension/category)
for $cou in distinct-values(/orders/order/customer_dimension/country)
let $facts := /orders/order[(product_dimension/category eq $cat) and
                            (customer_dimension/country eq $cou)]
where exists($facts)
return
<group>
  <product_category>{$cat}</product_category>
  <customer_country>{$cou}</customer_country>
  <sum>{sum($facts/price)}</sum>
</group>
```

FIG. 6.2 – Traduction de la requête 2 pour le modèle plat.

La seconde requête traduite à la figure 6.2 s'écrit de la même manière que la requête 1 si ce n'est que nous devons itérer sur les instances des niveaux de deux dimensions plutôt que d'une seule. Il y aura donc deux clauses `for` qui itéreront chacune sur une collection renvoyée par `distinct-values()` afin de parcourir toutes les combinaisons possibles des instances des niveaux pour chaque dimension. La jointure exprimée par le prédicat XPath de la clause `let`

contiendra également deux conditions, chacune relative à une dimension. Il y aura donc autant de clauses `for` et de jointures qu'il y a de dimensions à analyser.

Nous n'entrerons pas dans les détails de l'interrogation de la variante X-Warehousing du modèle physique plat car celle-ci se fait exactement de la même manière.

6.3.2 Analyse

Ces requêtes XQuery ne sont pas facile à écrire, à comprendre et donc à maintenir. Effectivement, il semble illogique de devoir déterminer toutes les combinaisons différentes des dimensions choisies et ensuite, pour chaque combinaison, rapatrier les faits correspondants. Dans le monde relationnel, avec SQL, on aurait procédé différemment car il existe une clause de groupement, `GROUP BY`, permettant de réaliser exactement la même chose de façon plus lisible. En effet, on peut voir l'extraction de sous-cubes comme un problème de groupement où les clés de groupements seraient les niveaux choisis des dimensions. Nous en reparlerons plus en détail dans le chapitre 7.

Analysons maintenant intuitivement la complexité en grand O de ces requêtes en faisant abstraction d'un éventuel moteur d'optimisation pour XQuery. Le but recherché ici n'est pas de faire une analyse complète de la complexité des requêtes mais de l'approcher intuitivement afin de comparer les requêtes sur les différents modèles. Soit n le nombre de faits de la base de données, m le nombre d'instances du ou des niveaux choisis et d le nombre de dimensions à analyser. Considérons également que les données sont indexées correctement et que donc une jointure à une condition se réalise via une recherche de l'ordre de $O(\log(n))$. Une jointure à c conditions consiste en c jointures simples et en l'intersection des c ensembles résultants de ces jointures. Supposons que l'intersection de c ensembles de taille moyenne e se réalise en maximum $O((c-1)e^2)$. Nous considérons donc qu'une jointure à c conditions est au maximum de l'ordre de $O((c\log(n))+(c-1)l^2)$ si l est la taille moyenne des résultats des jointures. Avec ces hypothèses, la complexité des requêtes d'extraction de sous-cubes sur le modèle plat approche $O(m^d(d\log(n) + (d-1)l^2))$.

Comme on peut le constater, cette complexité est quadratique et augmente exponentiellement avec le nombre d de dimensions à analyser. Elle augmente également linéairement avec le nombre m^d de cases du sous-cube à extraire et avec la taille n des données. Il est à noter que nous avons délibérément oublié le coût de la fonction `distinct-values()` car celle-ci ne sera exécutée qu'un nombre très limité de fois (d fois).

Des requêtes dont la complexité est quadratique ne sont pas appropriées aux systèmes contenant beaucoup de données et en particulier les systèmes d'analyse multidimensionnelle OLAP. En effet, une telle complexité introduit inévitablement des problèmes de performance. Nous étudierons un moyen d'améliorer cette complexité et donc les performances dans le chapitre 7.

```
for $category in /products/family/category
let $facts := /orders/order[product/@ref = $category/product/@prodId]
where exists($facts)
return
<group>
  <product_category>{$category/@name}</product_category>
  <sum>{sum($facts/@price)}</sum>
</group>
```

FIG. 6.3 – Traduction de la requête 1 pour le modèle hiérarchique.

6.4 Interrogation du modèle hiérarchique

6.4.1 Traduction des requêtes

La requête 1 traduite en XQuery pour notre modèle hiérarchique est décrite à la figure 6.3. Son principe est le même que celui utilisé pour interroger le modèle plat. C'est-à-dire que nous devons itérer sur les différentes instances du niveau choisi et à chaque itération, récupérer les faits correspondants.

Une différence par rapport au modèle plat est que les instances distinctes des niveaux choisis sont déjà explicitement définies dans le document XML correspondant à la dimension. En effet, dans le modèle plat, ces instances distinctes n'étaient pas définies et nous devions utiliser la fonction `distinct-values()` de XQuery pour les obtenir.

Une deuxième différence est que les jointures sont plus complexes pour ce modèle. En effet, dans cette représentation, les faits contiennent uniquement les identificateurs des produits vendus et non de la catégorie du produit. Il faut donc à partir d'une catégorie déterminer l'ensemble de ses produits pour pouvoir rapatrier les commandes correspondantes à ces produits. Il ne s'agit donc plus d'une jointure simple (représentée par `eq` dans le prédicat XPath) mais d'une jointure multiple (représentée par `=`). Cette dernière est bien sûr plus coûteuse en temps de calcul qu'une jointure simple. Nous tenterons d'approcher sa complexité dans la section suivante.

La seconde requête (figure 6.4) se traduit de la même manière si ce n'est qu'il faut itérer sur deux dimensions plutôt qu'une seule et que la jointure multiple exprimée dans la clause `let` contient deux conditions plutôt qu'une seule. Il s'agit en fait de deux jointures multiples suivies d'une intersection entre celles-ci.

6.4.2 Analyse

Les requêtes d'interrogation XQuery traduite pour le modèle hiérarchique sont un peu plus lisibles et compréhensibles que les requêtes du modèle plat. En effet, parcourir un ensemble déterminé d'instances de niveau paraît plus intuitif que de devoir déterminer au préalable leurs valeurs distinctes. Quoi qu'il en soit, ces requêtes ne sont pas encore aussi lisibles que des

```
for $category in /products/family/category
for $country in /customers/continent/country
let $facts := /orders/order[(product/@ref = $category/product/@prodId)
                        and (customer/@ref = $country/customer/@cusId)]
where exists($facts)
return
<group>
  <customer_country>{$country/@name}</customer_country>
  <product_category>{$category/@name}</product_category>
  <sum>{sum($facts/@price)}</sum>
</group>
```

FIG. 6.4 – Traduction de la requête 2 pour le modèle hiérarchique.

requêtes similaires écrites en SQL à l'aide de la clause de groupement. Nous en reparlerons dans le chapitre 7.

En ce qui concerne la complexité de ces requêtes, nous pouvons observer qu'elle semble très proche de celle de l'interrogation du modèle plat car elles sont constituées du même nombre d'itérations. Par contre, les jointures nécessaires pour interroger ce modèle sont beaucoup plus complexes que les jointures utilisées dans l'interrogation du modèle plat. En effet, on peut remarquer qu'à la ligne 2 de la figure 6.3, l'on rapatrie toutes les commandes dont le produit *fait partie* de l'ensemble des produits de la catégorie courante. Il s'agit ici d'une jointure multiple. Pour interroger le modèle plat, la jointure rapatriait toutes les commandes dont la catégorie *est égale* à la catégorie courante, il s'agissait là d'une jointure simple, remarquez pour cela l'utilisation respective des signes = et eq dans les prédicat XPath. Nous considérerons que les jointures multiples consistent en un certain nombre de jointures simples et suivent donc une complexité de $O(crlog(n) + (c-1)l^2)$, r étant la cardinalité de la partie droite de la comparaison, c le nombre de critères et l la taille moyenne des résultats des jointures simples. Avec les mêmes hypothèses que pour le modèle plat, nous arrivons donc à une complexité de $O(m^d(rdlog(n) + (d-1)l^2))$.

Il est important de noter que certaines variables sont liées. Par exemple, quand m augmente, r diminue. En d'autres mots, plus il y a d'instances de niveau à analyser, moins la jointure multiple est coûteuse. Effectivement, nos hiérarchies de dimensions sont strictes, c'est-à-dire que dans le cas présent, un produit sera contenu dans une et une seule catégorie. Un produit ne sera donc compté qu'une fois dans les requêtes d'extraction de sous-cubes. Cela implique que si m est petit, les m^d jointures multiples seront très coûteuses mais si m est grand, les jointures, plus nombreuses, seront moins coûteuses.

Pour conclure, la complexité de ces requêtes est quadratique et donc certainement mal adaptée aux systèmes OLAP qui doivent en général gérer de grands ensembles de données. Cette complexité est très proche de celle de l'interrogation du modèle plat mais la relation entre le nombre de cases du sous-cube à extraire et le coût des jointures pourrait nous mener à des résultats différents. Nous tenterons d'analyser cela dans le chapitre 8 dans lequel nous calculerons le temps de calcul de ces requêtes selon différents paramètres.

6.5 Interrogation du modèle XCube

6.5.1 Traduction des requêtes

```
for $category in doc("XCubeDimension_Product.xml")/dimensionData
                 /classification/level[@id eq "category"]/node
let $prodId :=   doc("XCubeDimension_Product.xml")/dimensionData
                 /classification/level[@id eq "product"]
                 /node[rollUp/@toNode eq $category/@id]/@id
let $facts :=    doc("XCubeFact_Orders.xml")/cubeFacts/cube
                 /cell[dimension[@id="products"]/@node = $prodId]
                 /fact[@id="price"]
return
<group>
  <category>{$category/@id}</category>
  <sum>{sum($facts/@value)}</sum>
</group>
```

FIG. 6.5 – Traduction de la requête 1 pour XCube.

La traduction de la requête 1 pour le modèle XCube est présentée à la figure 6.5 et celle la requête 2 à la figure 6.6. De la même manière que pour les deux modèles précédents, nous devons itérer sur les instances distinctes du niveau choisi afin de récupérer les faits correspondants. Dans ce modèle, les différentes instances sont déjà explicitement définies et nous ne devons donc pas éliminer les doublons, ce que nous avons dû faire avec le modèle plat.

Bien que le principe de base soit similaire à ceux des deux précédents modèles, un différence importante est à observer. En effet, ce modèle ne représentant pas les hiérarchies de dimensions en utilisant les relations père-fils de XML mais à l'aide d'un mécanisme de liaison relationnel (`rollUp/@toNode`), il faut réaliser des jointures entre les différents niveaux des dimensions pour les parcourir. C'est ce que nous avons fait dans la première clause `let` de la requête de la figure 6.5. Il est à noter que le nombre de jointures nécessaires pour traverser cette hiérarchie de dimension est proportionnel à la distance entre le niveau choisi et le niveau le plus fin de la dimension. En effet, les faits sont liées au niveau le plus fin de leurs dimensions. Ces faits doivent être récupérés à l'aide de la deuxième clause `let`. De plus, il n'y a pas que le nombre de jointures qui augmente : la complexité de ces jointures est également en hausse. En effet, dans ces requêtes se trouvent un nombre impressionnant de prédicats XPath, parfois même imbriqués. Nous pouvons donc observer ici qu'un nombre très important de jointures complexes est nécessaire pour écrire ces requêtes et que donc les performances risquent d'être moins bonnes que pour les autres modèles présentés ci-dessus.

```
for $category in doc("XCubeDimension_Product.xml")/dimensionData
                 /classification/level[@id eq "category"]/node
for $country in  doc("XCubeDimension_Customer.xml")/dimensionData
                 /classification/level[@id eq "country"]/node
let $prodId :=   doc("XCubeDimension_Product.xml")/dimensionData
                 /classification/level[@id eq "product"]
                 /node[rollUp/@toNode eq $category/@id]/@id
let $cusId :=    doc("XCubeDimension_Customer.xml")/dimensionData
                 /classification/level[@id eq "customer"]
                 /node[rollUp/@toNode eq $country/@id]/@id
let $facts :=    doc("XCubeFact_Orders.xml")/cubeFacts/cube
                 /cell[dimension[@id="products"]/@node = $prodId
                       and dimension[@id="customers"]/@node = $cusId]
                 /fact[@id="price"]
return
<group>
  <category>{$category/@id}</category>
  <country>{$country/@id}</country>
  <sum>{sum($facts/@value)}</sum>
</group>
```

FIG. 6.6 – Traduction de la requête 2 pour XCube.

6.5.2 Analyse

Du point de vue de la facilité d'écriture, le nombre de jointures nécessaires pour parcourir les hiérarchies rendent les requêtes compliquées à écrire. La même raison implique que ces requêtes sont difficiles à lire et donc à maintenir.

La complexité en grand O de ces requêtes devrait être très proche de celle de l'interrogation du modèle hiérarchique si ce n'est qu'il faut ajouter un terme j représentant le nombre de jointures nécessaires pour traverser la ou les hiérarchie(s) de dimension. Ce terme j est inversement proportionnel à la profondeur du niveau que l'on veut analyser. En effet, si l'on veut analyser les faits selon le niveau le plus bas d'une dimension ($p = 2$ dans notre exemple), il n'y aura pas de jointure nécessaire pour traverser la hiérarchie. La complexité sera donc comparable à celle de la même requête exprimée pour le modèle hiérarchique. Par contre, si l'on veut analyser les faits selon le niveau le plus haut d'une dimension ($p = 0$), il y aura un nombre important de jointures à exécuter car il faudra traverser toute la hiérarchie de haut en bas afin de rapatrier les instances du niveau le plus bas qui possèdent les références nécessaires pour récupérer les faits correspondants. Le terme j est évidement également proportionnel au nombre de dimensions (d) à analyser. Quoi qu'il en soit, ces j jointures seront bien moins coûteuses que la jointure principale servant à rapatrier les faits. En effet, les j jointures pour traverser les dimensions sont exécutées sur les documents représentant les hiérarchies qui sont en général de taille très inférieure à la taille des faits.

Nous n'analyserons pas cette complexité plus longuement mais nous nous contenterons d'attirer l'attention sur le fait que la profondeur des niveaux choisis pour l'analyse aura une influence, quoique minime, sur les performances des requêtes contrairement aux autres modèles présentés dans ce mémoire.

6.6 Conclusion

Dans ce chapitre, nous avons écrit des requêtes XQuery afin d'interroger de façon multidimensionnelle les divers modèles que nous avons présentés. Nous avons constaté via une analyse de complexité intuitive que ces requêtes risquaient de poser des problèmes de performances sur de grands ensembles de données typiques des systèmes d'analyse. Nous avons également remarqué que ces requêtes sont difficiles à écrire et à comprendre à première vue.

Dans le chapitre suivant, nous analyserons un moyen de réduire la complexité de ce type de requêtes tout en les rendant plus facile à écrire et à comprendre.

Chapitre 7

Clause de groupement pour XQuery

Dans le chapitre précédent, nous avons traduit des requêtes d'extraction de sous-cubes en XQuery pour les modèles présentés au chapitre 5. Nous en avons également fait une brève analyse. Comme nous l'avons constaté, ces requêtes soulèvent deux problèmes importants : la facilité de lecture et d'écriture ainsi que leur complexité quadratique.

Dans ce chapitre, nous allons tenter d'améliorer ces deux points en proposant une extension au langage XQuery basé sur une fonctionnalité du langage d'interrogation relationnel bien connu SQL, la clause de groupement.

Contenu

7.1 Groupements en SQL

Le langage SQL possède une clause de groupement `GROUP BY` qui permet de grouper des tuples par valeurs. Cette clause est parfaitement adaptée à l'extraction de sous-cubes car on peut voir le problème de l'extraction comme un problème de groupement de données par valeurs. Une telle clause n'existe pas dans la recommandation XQuery 1.0 du W3C [5] et cela implique que nous devons interroger les cubes XML et définissant et en itérant sur les différences instances des niveaux des dimensions pour extraire un sous-cube. En XQuery, nous devons donc faire un groupement *à priori* alors qu'un groupement en SQL se fait *à posteriori* c'est-à-dire à partir d'un ensemble de tuples donnés.

A titre d'exemple, la figure 7.1 présente notre requête 1 traduite en SQL pour le modèle relationnel présenté au chapitre 5. Comme on peut le constater, la clause de groupement `GROUP BY`

permet d'écrire ce genre de requêtes facilement et de façon concise. Les requêtes ainsi écrites sont plus faciles à lire et donc à maintenir.

```
SELECT   Categories.name, SUM(Orders.price)
FROM     Categories, Products, Orders
WHERE    Products.category_id = Categories.id and
         Orders.product_id = Products.id
GROUP BY Categories.name
```

FIG. 7.1 – Traduction de la requête 1 en SQL.

7.2 Groupements en XQuery

Avant d'aller plus loin, il est important de noter qu'il existe différents types de groupements dans le domaine des documents XML : le groupement par valeur et le groupement positionnel. Le but du groupement par valeur est de regrouper des données en fonction de la valeur d'une ou plusieurs de ses dimensions (appelées clés de groupement). Nous prenons le terme valeur dans le sens défini par la spécification XML, c'est-à-dire les valeurs atomiques d'un attribut ou d'un élément XML. Le groupement par valeur sert donc, par exemple, à regrouper une collection de livres par auteur et par année. Ce groupement, pour XQuery, a fait l'objet de plusieurs recherches scientifiques, que nous présenterons un peu plus loin dans cette section.

Par ailleurs, si nous voulons, par exemple, grouper un document XML contenant une suite d'éléments H, P, P, P, H, P, P en deux sections comprenant chacune un élément H et les éléments P qui le suivent jusqu'au prochain élément H, il s'agit alors d'un groupement positionnel. Ce type de groupement est possible en XQuery mais demande l'écriture de fonctions récursives et de requêtes trop compliquées pour un utilisateur classique d'XQuery. Michael Kay, le développeur de SAXON, a identifié les cas d'utilisations d'un tel groupement et a proposé très récemment une syntaxe ad hoc dans [42].

Ces deux types de groupement sont très différents et ne peuvent donc pas être traités de la même manière. Dans ce mémoire, nous nous focalisons sur le groupement par valeurs, parfaitement adapté, comme nous venons de le voir, aux requêtes d'extraction de sous-cubes.

Une clause de groupement par valeurs a déjà été proposé par Beyer *et al.* [38] et par Borkar [43]. Une autre étude à ce sujet a été menée par Deutsch *et al.* [44]. Ils proposent également un opérateur de groupement pour XQuery ainsi qu'un ensemble de règles permettant de traduire les requêtes utilisant la fonction `distinct-values()` en une forme minimisée et optimisée. Cette minimisation n'est possible que si un tel opérateur est implémenté dans XQuery. Ces règles peuvent être réutilisées, par exemple, à des fins d'optimisations interne.

Il est bien entendu que cette clause n'ajoute rien au pouvoir d'expressivité du langage, comme l'ont démontré Deutsch *et al.* [44]. Cependant, nous pensons que cet ajout permet d'écrire des requêtes d'extraction de sous-cubes beaucoup plus facilement. En effet, il semble plus logique d'utiliser explicitement un opérateur de groupement plutôt que de devoir cycler sur les valeurs

distinctes des clés de groupement. En plus d'être plus faciles à écrire, les requêtes seraient plus aisées à lire et donc à maintenir. Nous avons alors décidé d'implémenter et de tester une telle clause de groupement à l'aide d'un moteur XQuery existant.

7.3 Syntaxe

Une syntaxe très complète d'une clause de groupement pour XQuery a été proposée dans [38] mais n'a pas encore été implémentée publiquement. Pour les besoins de cette étude, nous avons décidé d'utiliser une syntaxe un peu plus simple dans un soucis de lisibilité et de rapidité d'implémentation.

L'ajout d'une clause de groupement modifie la syntaxe de l'expression FLWOR propre à XQuery. La clause `group by` se place entre la clause `where` et la clause `order by` de telle sorte qu'il soit possible de trier les résultats du groupement de la même manière qu'en SQL. Elle est bien sur optionnelle. La syntaxe simplifiée de cette clause est la suivante :

```
GroupByClause =  "group" toGroupVarName "as" groupedVar
                 "by" groupKeyExpr "as" groupKeyVar
                 ("," groupKeyExpr "as" groupKeyVar)*
```

`toGroupVarName` est la variable à grouper, déclarée précédemment à l'aide d'une clause `for` ;

`groupedVar` est une nouvelle variable qui contiendra, pour chaque groupe, une séquence comprenant l'ensemble des nœuds `toGroupVarName` correspondants au groupe courant ;

`groupKeyExpr` est l'expression représentant une clé de groupement ;

`groupKeyVar` est une nouvelle variable qui contiendra, après la clause de groupement, le résultat de l'évaluation de `groupKeyExpr` pour chaque groupe, c'est-à-dire la valeur de la clé de groupement pour le groupe courant.

Un groupe est composé d'une séquence de nœuds groupés, `groupedVar`, et des valeurs des clés de groupement correspondantes, `groupKeyVar`.

Il est important de préciser que la cardinalité du flux de nœuds change au niveau de la clause de groupement. En effet, en théorie, un résultat est renvoyé par itération de la clause `for`. Les résultats peuvent bien sûr être filtrés à l'aide d'une clause `where` et dans ce cas on dit que la cardinalité du flux de nœuds change. Quand une clause de groupement est utilisée, un résultat est renvoyé par groupe. La cardinalité du flux change donc au niveau de cette clause car il y aura en général un plus petit nombre de groupes que de nœuds parcourus par la clause `for`.

Du fait de ce changement de cardinalité, les variables assignées avant la clause de groupement ne seront plus accessibles après celle-ci car elles n'auront plus de sens. La variable `toGroupVarName` aurait pu être réutilisée pour contenir la séquence de nœuds groupés mais nous avons préféré utiliser une nouvelle variable afin d'éviter la confusion, comme cela se fait en OQL [13]. Après cette clause, seules les variables `groupedVar` et `groupKeyVar` seront disponibles et pourront être utilisées dans les clauses postérieures, c'est-à-dire les clauses d'ordonnancement et de construction des résultats.

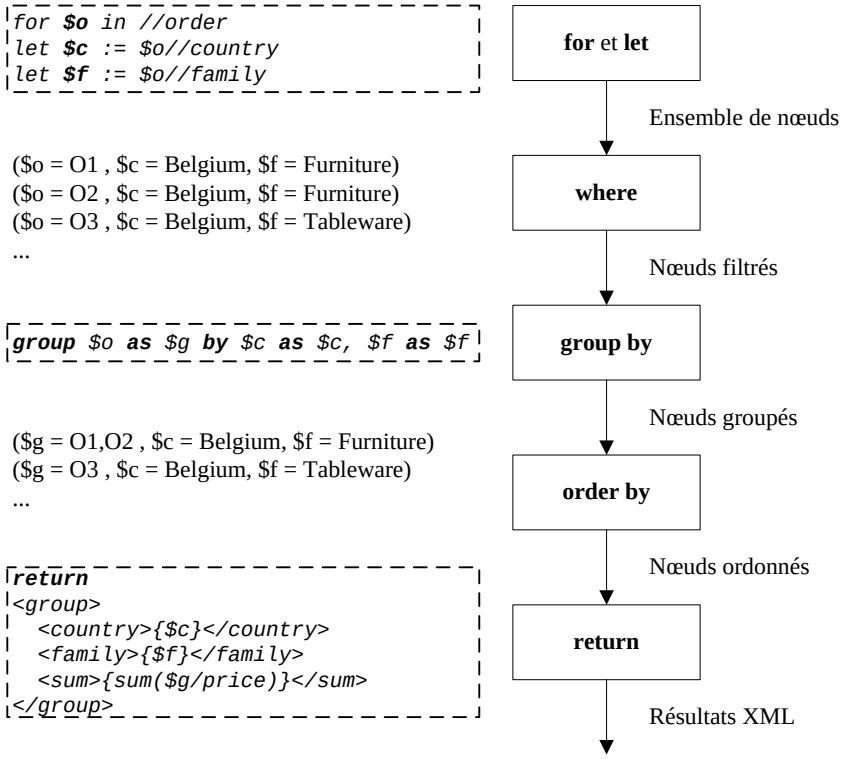


FIG. 7.2 – Illustration du fonctionnement de la clause de groupement pour XQuery.

7.4 Algorithme

La façon dont nous avons implémenté cette clause de groupement est basée sur un algorithme itératif simple. On peut considérer cette clause comme une boîte prenant en entrée un flux de nœuds générés par les clauses antérieures et qui renvoie en sortie une flux de groupes comme illustré à la figure 7.2. Pour chaque nœud du flux d'entrée, l'algorithme évalue la ou les clés de groupement. Ensuite, le nœud courant est ajouté au groupe correspondant, créé dynamiquement si nécessaire. Une fois tous les nœuds traités, il renvoie un flux de groupes aux clauses postérieures. Un algorithme en pseudo-code est présenté à la figure 7.3.

7.4.1 Complexité

Soit n le nombre de faits à grouper, cet algorithme de complexité $O(n)$ puisqu'il traite les nœuds un par un, ce qui est acceptable pour un ensemble important de données. La complexité de l'accès aux groupes est volontairement oubliée car le nombre de groupes est en général minime par rapport au nombre de données et donc le temps l'accès à un groupe donné est négligeable.

Cet algorithme peut être exécuté en parallèle ce qui est intéressant de nos jours où la puissance de calcul est de plus en plus distribuée. En effet, il ne faut pas nécessairement attendre que la partie de la requête antérieure à la clause de groupement soit entièrement terminée pour commencer le groupement des nœuds.

```
POUR TOUT noeud
  évaluation des clés de groupement
  SI un groupe correspond à ces clés ALORS
    ajout du noeud au groupe
  SINON
    création du groupe
    ajout du noeud au groupe
  FIN SI
FIN POUR

POUR TOUT groupe
  renvoi du groupe
FIN POUR
```

FIG. 7.3 – Algorithme en pseudo-code de la clause de groupement.

7.5 Interrogation

Dans cette section, nous allons traduire notre requête 1 pour les modèles présentés au chapitre 5 en utilisant notre clause de groupement XQuery précédemment introduite. Nous analyserons ensuite brièvement ces requêtes.

7.5.1 Modèle plat

La figure 7.4 présente la traduction de la requête 1 pour le modèle plat à l'aide d'XQuery et de la clause de groupement. Par rapport à la même requête exprimée sans clause de groupement, cette requête est plus concise et est compréhensible dès la première lecture.

```
for $fact in /orders/order
group $fact as $group by $fact/product_dimension/category as $catKey
return
<group>
  <product_category>{$catKey}</product_category>
  <sum>{sum(group/@price)}</sum>
</group>
```

FIG. 7.4 – Traduction de la requête 1 pour le modèle plat.

D'un point de vue complexité, nous pouvons constater que nous n'avons plus besoin d'itérer sur les instances différentes du niveau choisi et que par conséquent nous n'avons plus de jointure à faire. Le nombre n d'itérations de cette requête est différent du nombre m^d d'itérations de la même requête sans clause de groupement. En effet, dans le cas présent, le nombre d'itérations correspond au nombre de faits de la base de données. Ce nombre est donc stable quel que soit le

nombre m d'instances de niveau. Le nombre d'itérations n'augmente donc plus exponentiellement avec le nombre de clés de groupement. Cela devrait impliquer de meilleurs performances quand le nombre de groupes ou le nombre de dimensions à analyser augmentent.

7.5.2 Modèle hiérarchique

La figure 7.5 présente la traduction de la requête 1 de l'exemple de travail pour le modèle hiérarchique en utilisant la clause de groupement que nous venons d'introduire. La différence principale par rapport au modèle plat est qu'une jointure (ligne 2) est nécessaire pour récupérer la catégorie du produit contenu dans le fait car, comme nous l'avons déjà écrit, un fait ne contient que l'identifiant du produit et non pas toute la hiérarchie de dimension.

Comme nous l'avons déjà constaté, la clause de groupement permet d'écrire les requêtes d'extraction de sous-cubes de façon plus lisible et concise.

```
for $fact  in /orders/order
let $cat := /products/family/category[product/@prodId eq
                                      $fact/product/@ref]
group $fact as $group by $cat as $catKey
return
<group>
  <product_category>{$catKey}</product_category>
  <sum>{sum(group/@price)}</sum>
</group>
```

FIG. 7.5 – Traduction de la requête 1 pour le modèle hiérarchique.

En ce qui concerne la complexité, nous pouvons émettre les mêmes constatations que pour le modèle plat. C'est-à-dire que le nombre d'itérations est relatif au nombre de faits de la base de données et non plus au nombre d'instances de niveau à analyser. Le temps d'exécution de ces requêtes devrait donc être constant quel que soit le nombre de groupes résultants.

7.5.3 Modèle XCube

La traduction de la requête 1 est présente à la figure 7.6. Comme nous pouvons le constater, les requêtes portant sur le modèle XCube nécessitant beaucoup de jointures, la facilité de compréhension de ces requêtes n'est que peu améliorée grâce à la clause de groupement.

Il est à noter que dans le cas de l'utilisation de la clause de groupement, il ne faut plus *descendre* la hiérarchie de dimension mais la *remonter*. En effet, on itère sur chaque fait et à chaque itération, on doit déterminer la catégorie du produit commandé. On parcourt donc la hiérarchie dans le sens contraire par rapport à la même requête sans cette nouvelle clause. Quoi qu'il en soit, le nombre de jointures nécessaires pour traverser la hiérarchie ne varie pas d'une méthode à l'autre, ce nombre reste inversement proportionnel à la profondeur du niveau de la dimension à analyser. Par contre, ces jointures ne seront plus exécutées autant de fois

```
for $fact in   doc("XCubeFact_Orders.xml")//cubeFacts/cube/cell
let $prodId := $fact/dimension[@id="products"]/@node
let $catId  := doc("XCubeDimension_Product.xml")//dimensionData
               /classification/level[@id="product"]/node[@id = prodId]
               /rollup/@toNode
group $fact as $group by $catId as $category
return
<group>
  <category>{$category}</category>
  <sum>{sum($group/fact[@id="price"]/@value)}</sum>
</group>
```

FIG. 7.6 – Traduction de la requête 1 pour le modèle XCube.

qu'il y a d'instances de niveau à analyser mais seront exécutées autant de fois qu'il y a de faits dans la base de données. Nous pouvons donc émettre une conclusion similaire que pour le modèle hiérarchique, c'est-à-dire que l'intérêt de la clause de groupement ne devrait être sensible qu'à partir d'un nombre important de groupes ou de dimensions à analyser. Pour confirmer ces attentes, nous réaliserons un test de performances dans le chapitre 8.

7.6 Implémentation dans le moteur XQuery d'eXist

Pour mener à bien notre expérimentation, nous avons choisi d'implémenter la clause de groupement dans le moteur XQuery d'eXist en respectant la syntaxe et le fonctionnement présenté au début de ce chapitre.

Nous avons choisi ce logiciel pour plusieurs raisons. Tout d'abord parce que celui-ci est diffusé sous licence libre et qu'ainsi nous avons pu le modifier et partager notre travail avec la communauté ce qui nous tenait à cœur. Deuxièmement, eXist implémente la presque totalité du langage XQuery ce qui était important pour nos différentes expériences. Ensuite, ce logiciel est écrit de telle façon qu'il respecte les principes modernes du génie logiciel comme les tests unitaires, la modularité, une documentation complète, ce qui nous a permis de rapidement prendre en main le code source du logiciel. Enfin, les développeurs d'eXist sont très actifs et disponibles pour répondre à toute question ou réflexion sur leur logiciel. Les soumissions de bugs et de patchs sont pris très au sérieux et sont réglés très rapidement. Ce fut un réel plaisir de pouvoir travailler avec eux lors de notre implémentation. Cette expérience fut très enrichissante et nous tenons à remercier les développeurs d'eXist pour leur collaboration amicale.

Nous avons commencé l'implémentation tout au début de cette année académique, en octobre 2006. Il a d'abord fallu intégrer le code actuel d'eXist de manière à bien comprendre comment celui-ci fonctionnait et était structuré. Pendant cette période, nous avons pris contact avec les développeurs via les différents canaux mis à notre disposition comme leur salle IRC et leurs listes de discussions. Ce contact nous a grandement aidé à la compréhension du logiciel. Quelques

semaines plus tard, nous avons commencé l'implémentation de notre clause de groupement et une première version a été soumise aux développeurs en décembre 2006.

7.6.1 Détails techniques

Tout d'abord, ils nous a fallu modifier l'analyseur syntaxique XQuery[1] d'eXist ainsi que son arbre[2] pour y ajouter la syntaxe de notre clause de groupement. L'analyseur syntaxique d'eXist est développé à l'aide d'ANTLR, un outil de construction de compilateurs.

Une structure a été créée afin de contenir les propriétés d'une clé de groupement[3], c'est-à-dire l'expression `groupKeyExpr` et la variable `groupKeyVar`. Il y aura autant d'instances de cette structure qu'il y a de clés de groupement. Lors de l'analyse syntaxique de l'expression FLWOR XQuery, ces structures ainsi que les variables `toGroupVarName` et `groupedVar` sont instanciées.

Lors de l'évaluation de la première clause `for` de l'expression FLWOR, si une clause de groupement a été spécifiée, une structure de données permettant de contenir tous les groupes est instanciée[4]. Au départ, cette structure reposait sur un simple tableau ce qui nous a conduit à de graves problèmes de temps d'accès aux groupes lorsque ceux-ci devenaient nombreux. Nous avons donc décidé de baser cette structure sur une table de hashage dont la clé est basée sur la valeur des clés de groupement. Nous avons choisi cette structure de données afin de diminuer la complexité des fréquents accès aux groupes et de ce fait les accélérer. Nous avons ainsi pu obtenir les performances prévues.

Les éventuelles clauses ultérieures à la première clause `for` sont ensuite évaluées récursivement. Après cette récursion et pour chaque itération du premier `for` de la FLWOR, les clés de groupement du nœud courant sont évaluées et ce nœud est placé dans le groupe[5] correspondant via un accès à la table de hashage. Si le groupe correspondant n'existe pas encore, il sera crée. Un groupe est ainsi composé d'une séquence de nœuds groupés, accessible via la variable `groupedVar`, et des valeurs de ses clés de groupement, accessibles via les variables `groupKeyVar`.

Pour terminer, les variables qui n'ont plus de sens après le groupement sont retirées de la pile du contexte. La table de hashage est ensuite parcourue séquentiellement afin de renvoyer chaque groupe un à un pour le traitement des clauses `order by` et `return`. Une douzaine de tests unitaires ont également été développés. Certains de ces tests sont inspirés d'intéressantes requêtes provenant de l'article de K. Beyer [38]. Fin 2006, notre opérateur de groupement a été intégré à la branche de développement d'eXist[6] et devrait donc faire partie intégrante de la prochaine version de ce logiciel.

[1] `org.exist.xquery.parser.XQuery.g`
[2] `org.exist.xquery.parser.XQueryTree.g`
[3] `org.exist.xquery.GroupSpec`
[4] `org.exist.xquery.value.GroupedValueSequenceTable`
[5] `org.exist.xquery.value.GroupedValueSequence`
[6] Révision SVN 4881

7.7 Conclusion

Dans ce chapitre, nous avons introduit une clause de groupement pour le langage XQuery permettant de faciliter l'écriture des requêtes d'extraction de sous-cubes tout en améliorant leur complexité. Nous avons également implémenté cette clause de groupement dans le moteur XQuery d'eXist.

Dans le chapitre suivant, nous allons réaliser un test de performance calculant le temps de calcul d'un ensemble de requêtes d'extraction de sous-cubes sur tous les modèles présentés, avec et sans utilisation de la clause de groupement pour XQuery.

Chapitre 8

Analyse expérimentale

Dans ce chapitre, nous allons réaliser un banc d'essai des différents modèles multidimensionnels XML. Pour ce faire, nous exécuterons un ensemble de requêtes sur des données modélisées suivant chacun de ces modèles. Chaque requête sera traduite avec et sans utilisation de la clause de groupement pour XQuery que nous avons introduite au chapitre précédent.

Contenu

8.1 Scénario d'évaluation

Pour mener à bien cette évaluation, nous avons procédé en différentes étapes. Tout d'abord nous avons dû générer des données afin de pouvoir les analyser. Ensuite, nous avons exécuté un certain nombre de requêtes bien choisies sur ces données.

8.1.1 Génération des données

Pour les besoins de notre analyse, nous avons développé un générateur de données XML aléatoires en Python afin de remplir une base de données correspondante à notre exemple de travail dans le modèle XML hiérarchique.

Ce script génère les hiérarchies de dimensions de manière déterministe et balancée. Les faits sont dispersés de manière équiprobable dans les différentes dimensions. Il y aura donc, de façon probabiliste, le même nombre de commandes (`order`) pour chaque client (`customer`). Nous espérons ainsi tester un cas classique d'extraction de sous-cubes, celui où il y a peu de cases vides et où chaque case contient plus ou moins le même nombre de faits.

Ce script est entièrement paramétrable. En effet, il est possible entre autres de spécifier les paramètres suivants :

- ▷ le nombre de faits (`order`),
- ▷ le nombre de familles (`family`),
- ▷ le nombre de catégories (`category`) par famille,
- ▷ le nombre de produits (`product`) par catégorie,
- ▷ le nombre de continents (`continent`),
- ▷ le nombre de pays (`country`) par continent et
- ▷ le nombre de clients (`customer`) par pays.

Nous avons choisi de générer les données en suivant le modèle hiérarchique car celui-ci nous permet de les traduire dans les autres modèles très facilement. Si nous avions choisi le modèle plat pour générer les données, nous n'aurions pas pu les transformer dans le modèle hiérarchique ou XCube car l'ordre des niveaux des dimensions n'y est pas exprimé.

8.1.2 Méthode et configuration

Pour construire la hiérarchie de dimensions, notre générateur de données a été configuré avec les paramètres présentés à la Table 8.1. En ce qui concerne les faits, nous avons généré trois ensembles de données de tailles différentes (1000, 10 K et 100 K) afin de vérifier l'échelonnabilité des performances de nos requêtes. La hiérarchie de dimensions est identique pour les trois ensembles de données.

Paramètre	**Valeur**
nombre de familles	2
nombre de catégories par famille	10 (20 catégories)
nombre de produits par catégorie	50 (1000 produits)
nombre de continents	5
nombre de pays par continent	10 (50 pays)
nombre de clients par pays	20 (1000 clients)

Tab. 8.1 – Paramètres des hiérarchies de dimensions.

Une fois les données générées, celles-ci ont été transformées dans les différents modèles présentés au chapitre 5 à l'aide d'eXist et de requêtes XQuery. Chaque base de données ainsi créée a été placée dans une collection distincte d'eXist.

Nous avons configuré les indexes d'eXist de la manière suivante : tout élément ou attribut utilisé comme clé lors des jointures est indexé via un index de portée (*range-index*) basé sur des nombres entiers. L'index textuel inversé est désactivé car inutile dans notre cas et aucun autre index configurable n'a été utilisé dans cette évaluation.

Notre évaluation a été réalisée sur un PC Intel Pentium 4 2400 Mhz possédant 768 Mo de mémoire vive et tournant sur le noyau Linux 2.6.17.11. La version d'eXist utilisée est la révision Subversion 5743 et a été exécutée sur l'environnement SUN JRE 1.5 pour lequel nous avons réservé 512 Mo de mémoire.

Requête	Clé 1 (profondeur)	Clé 2 (profondeur)	Nb. groupes
R1	`continent` (0)	-	5
R2	`family` (0)	-	2
R3	`continent` (0)	`family` (0)	10
R4	`category` (1)	-	20
R5	`continent` (0)	`category` (1)	100
R6	`country` (1)	-	50
R7	`category` (1)	`country` (1)	1000
R8	`customer` (2)	-	1000

TAB. 8.2 – Paramètres des requêtes de l'évaluation.

Pour mener à bien notre évaluation de performances, nous avons choisi huit requêtes d'extraction de sous-cubes. Ces requêtes varient selon plusieurs paramètres : le nombre de clés de groupement ou de dimensions à analyser, la profondeur des niveaux des dimensions choisis et le nombre approximatif de groupes attendus. Ce dernier paramètre est approximatif car il se peut que malgré notre distribution équiprobable des faits dans les dimensions, certains groupes ou cases soient vides. Les requêtes effectuées ainsi que leurs paramètres sont illustrés à la Table 8.2.

Ces requêtes ont chacune été traduites en XQuery pour les trois modèles multidimensionnels présentés au chapitre 5. Chaque requête a été traduite de deux manière différentes : la première en utilisant la syntaxe normale d'XQuery et la seconde en utilisant notre clause de groupement fraîchement implémentée. Nous possédons donc un bouquet de 48 requêtes qui seront chacune exécutées sur les trois tailles de données choisies. Il est à noter que ces requêtes portent toutes sur l'ensemble complet des données sur lesquelles elles sont exécutées. En d'autres mots, nous n'utilisons aucun filtre et tous les faits de la base de données sont donc traités.

Afin d'automatiser notre test de performances, nous avons écrit un script Python exécutant nos différentes requêtes sur notre base de données. Ce script communique avec eXist via l'interface XML-RPC. Chaque requête est exécutée quatre fois de suite et le temps d'exécution pris en compte est la moyenne arithmétique des trois derniers temps d'exécutions. Nous ne prenons pas en compte le premier résultat afin de s'assurer que les données aient été mises en cache et ainsi approcher un temps de calcul moyen plus juste. Pour terminer, les résultats sont validés par la comparaison des réponses aux requêtes identiques exécutées sur les différents modèles et ensembles de données.

8.2 Résultats de l'évaluation

Dans cette section, nous allons présenter les résultats de notre évaluation de performances. Pour chaque modèle, nous exposerons deux graphiques représentant le temps de calcul des requêtes en fonction du nombre de résultats. Le nombre de résultats correspond au nombre de cases du sous-cube extraites ou encore au nombre de groupes résultants. Il est à noter que les graphiques présentés utilisent des échelles logarithmiques afin d'améliorer leur lisibilité. Pour chaque modèle, la première figure présentera les résultats pour les requêtes d'extraction à une

dimension et la seconde pour les requêtes à deux dimensions. Les requêtes choisies pour la première figure sont R1, R2, R4, R6 et R8 et pour la deuxième figure R3, R5 et R7. Nous ferons pour chaque modèle une brève analyse de ses résultats et enfin nous comparerons l'interrogation de tous les modèles.

8.2.1 Modèle plat

La figure 8.1 présente les résultats de notre évaluation pour le modèle plat. Analysons tout d'abord la courbe représentant le temps de calcul de nos requêtes écrites sans clause de groupement. Nous pouvons observer que celui-ci tend à augmenter en fonction du nombre de résultats. Cette observation confirme le fait que la complexité de ces requêtes augmente linéairement avec le nombre de cases du sous-cube à extraire.

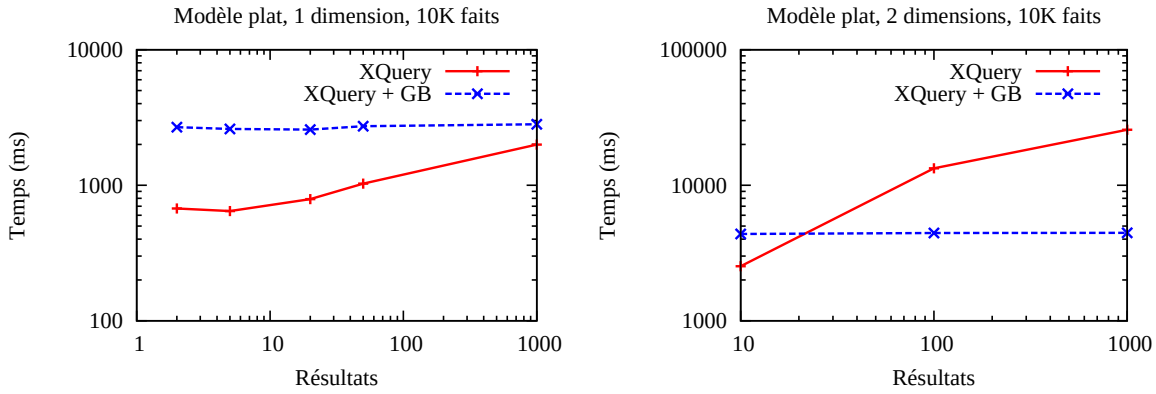


FIG. 8.1 – Temps d'exécution des requêtes en fonction du nombre de résultats pour le modèle plat.

Si on analyse le temps de calcul de la requête R8 et R7 qui retournent chacune le même nombre de résultats mais pour respectivement une et deux dimensions, on peut constater que le temps pris par la première est bien inférieur à celui pris par la seconde. Le rapport entre les deux est de l'ordre de 1500% ce qui tend à confirmer notre deuxième hypothèse spécifiant que le temps de calcul augmente exponentiellement avec le nombre de dimensions à analyser. Il nous faudrait bien sûr exécuter des requêtes à plus de deux dimensions pour nous en assurer mais nous pouvons observer que le temps de calcul augmente de manière bien plus significative en fonction du nombre de dimensions à analyser qu'en fonction du nombre de cases du sous-cube à extraire. Cela s'explique très simplement par le fait que dans le cas où il y a plusieurs dimensions à observer, les jointures à effectuer sont bien plus complexes comme nous l'avons constaté dans le chapitre 6.

Observons ensuite la courbe représentant le temps de calcul pour les requêtes utilisant la clause de groupement. La première constatation qui saute au yeux est que le temps de groupement est constant quel que soit le nombre de résultats. Cela confirme, s'il le fallait encore, le fait que la complexité de notre algorithme de groupement est uniquement relative à la taille de

l'ensemble de données à analyser et non au nombre de cases du sous-cube à extraire. Ce résultat est encourageant car nous pouvons facilement prévoir le temps pris par ce type de requêtes.

Une deuxième observation intéressante est que si nous comparons nos requêtes R8 et R7 portant sur le même nombre de résultats mais sur un nombre différent de dimensions, le rapport entre les deux temps d'exécution est de l'ordre de 200%. Nous pouvons donc constater que la complexité des requêtes utilisant la clause de groupement semble augmenter linéairement avec le nombre de dimensions à analyser, ce qui est plus raisonnable qu'exponentiellement surtout pour les quantités de données et de dimensions généralement utilisées dans les systèmes d'analyse. Bien sûr, il nous faudrait exécuter des requêtes sur plus de dimensions pour nous en assurer. Quoi qu'il en soit, cette augmentation s'explique simplement par le fait que pour chaque nœud à grouper, il faut évaluer autant de clés de groupement que de dimensions comme nous l'avons vu à la section 7.4.

Si nous observons les deux courbes, nous pouvons constater que la clause de groupement apporte un gain de temps considérable pour les requêtes d'extraction de sous-cubes mais seulement à partir d'un certain nombre de résultats ou de dimensions. Notons cependant que ce gain de temps est très vite sensible. En effet, pour les requêtes portant sur deux dimensions, les requêtes utilisant la clause de groupement sont déjà bien plus rapides si on les compare aux requêtes écrites en XQuery standard. Cette observation aurait certainement pu se faire pour des requêtes à une dimension si celles-ci renvoyaient plus de résultats. Nous pouvons donc conseiller d'utiliser la clause de groupement dès que le nombre de dimensions à analyser est supérieur ou égal à deux ou si le nombre de résultats attendus est supérieur à 1000.

8.2.2 Modèle hiérarchique

La figure 8.2 présente les résultats de notre analyse de performances pour l'interrogation du modèle hiérarchique. Les deux graphiques représentent respectivement dans le sens de la lecture les résultats pour les requêtes à une dimension et à deux dimensions.

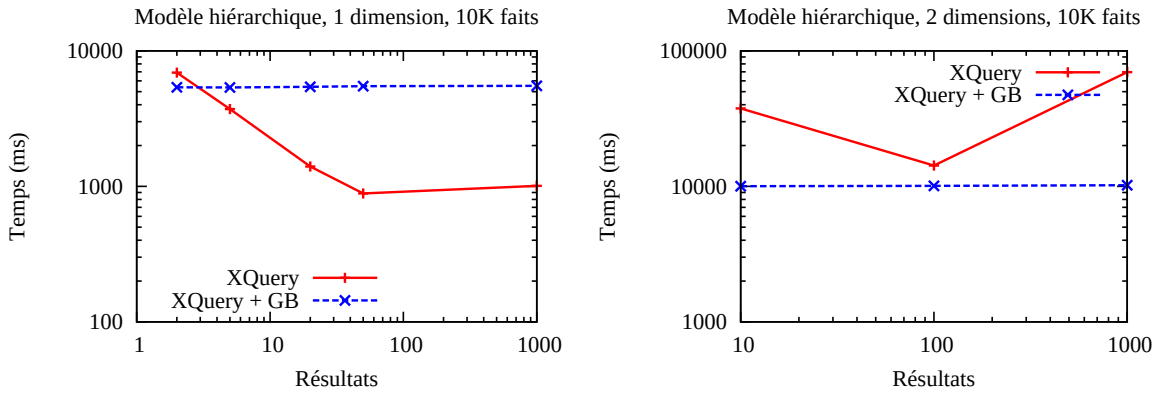


FIG. 8.2 – Temps d'exécution des requêtes en fonction du nombre de résultats pour le modèle hiérarchique.

Nous pouvons constater, en observant les courbes représentant le temps de calcul des requêtes écrites en XQuery standard, que ce temps a tendance à baisser jusqu'à un certain nombre de résultats (de l'ordre de 100) et puis augmente. Cette constatation peut être faite sur les deux figures. Ce phénomène est dû au fait que quand nous avons un petit nombre de résultats, il y a peu d'itérations et donc peu de jointures. Comme nous l'avons précisé dans la section 6.4.2, plus le nombre de résultats est petit, plus les jointures rapatrient de nœuds puisque nos requêtes portent sur l'ensemble complet des faits. Ces jointures retournant beaucoup de nœuds sont très coûteuses. Cela explique la diminution du temps de calcul jusqu'à un certain nombre de résultats. Le phénomène d'augmentation s'explique de manière similaire. Plus il y a de résultats, plus il y a d'itérations et donc plus il y a de jointures. Ces jointures sont très peu coûteuses mais comme leur nombre est important, le temps de calcul total augmente.

Nous pouvons constater un rapport de l'ordre de 7000% entre le temps d'exécution de nos requêtes R8 et R7 rapatriant le même nombre de résultat mais respectivement pour une et deux dimensions. Cela tend à confirmer notre hypothèse spécifiant que la complexité de ces requêtes est quadratique et augmente exponentiellement avec le nombre de dimensions à analyser. Il nous faudrait bien sûr exécuter des requêtes portant sur plus de dimensions pour nous en assurer mais nous pouvons observer que le temps de calcul augmente de manière beaucoup plus forte en fonction du nombre de dimensions qu'en fonction du nombre de cases du sous-cube à extraire. La raison de cette sensible augmentation est que le nombre de jointures nécessaires à chaque itération augmente avec le nombre de dimensions à analyser.

Observons maintenant les courbes représentant le temps d'exécution des requêtes utilisant la clause de groupement pour XQuery. La première constatation est que, pour un nombre de dimensions donné, ce temps est constant et ne varie donc pas en fonction du nombre de résultats. Cela s'explique simplement par le fait que l'algorithme utilisé par notre clause de groupement est de complexité relative au nombre de faits de la base de données.

Un rapport de l'ordre de 200% est observé entre les temps pris par les requêtes portant sur une dimension et celles portant sur deux dimensions. Cela est tout a fait normal car le nombre de clés de groupement à analyser à chacune des itérations de ces requêtes est égal au nombre de dimensions prises en compte. Une requête de groupement utilisant une dimension prendra donc deux fois moins de temps qu'une requête à deux dimensions sur le même ensemble de données. Il nous manque des résultats pour affirmer que cette augmentation est linéaire mais nous pouvons constater que celle-ci est bien plus forte que celle des requêtes classiques n'utilisant pas de clause de groupement.

Si nous comparons maintenant les temps de calcul des deux types de requêtes, nous pouvons constater que l'intérêt d'utiliser la clause de groupement ne se fait sentir qu'à partir d'une analyse à deux dimensions. Les requêtes d'analyse multidimensionnelles portant généralement sur deux dimensions ou plus, nous pouvons donc conseiller d'utiliser la méthode du groupement afin d'interroger des données XML modélisées de façon hiérarchique.

8.2.3 Modèle XCube

Les résultats de notre évaluation de performances pour le modèle XCube sont présentés à la figure 8.3. Le graphique de gauche représente les temps de calcul pour les requêtes à une dimension et celui de droite à deux dimensions.

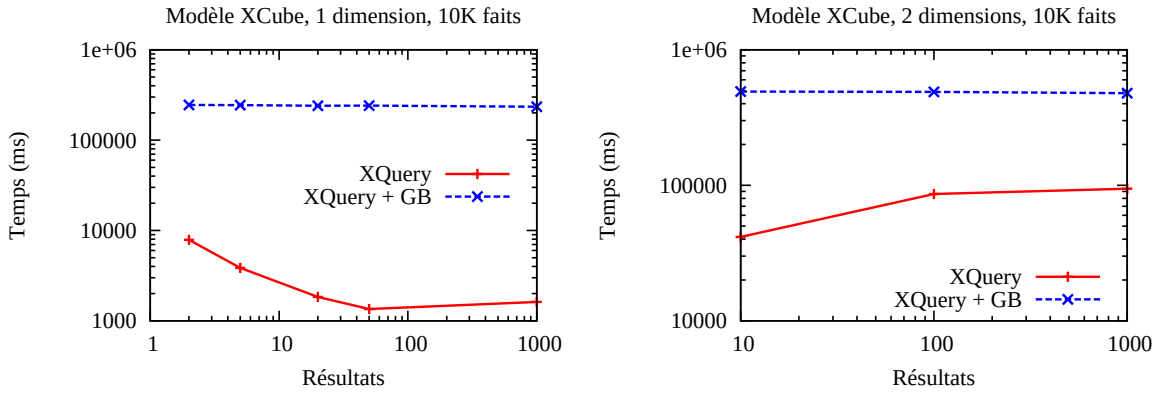


FIG. 8.3 – Temps d'exécution des requêtes en fonction du nombre de résultats pour le modèle XCube.

Si nous observons sur le premier graphique la courbe représentant les performances des requêtes en XQuery classique, nous pouvons constater que celle ci descend jusqu'à un certain nombre de résultats et puis remonte doucement. Ce même phénomène a été observé plus haut dans cette section pour le modèle hiérarchique et s'explique de la même manière.

Cependant, cette dernière observation ne peut se faire sur le deuxième graphique représentant les temps des requêtes à deux dimensions. En effet, cette courbe semble tout le temps augmenter. Nous aurions besoin de faire des requêtes portant sur un plus grand nombre de résultats pour nous en assurer mais nous pensons que cet effet est dû au fait que le nombre de jointures nécessaires à chaque itération est bien plus considérable qu'avec le modèle hiérarchique. En effet, comme nous l'avons vu à la section 6.5.2, le nombre de jointures nécessaires pour traverser les hiérarchies de dimensions XCube est proportionnel au nombre de dimensions à analyser et dans une moindre mesure à la profondeur de celles-ci, nous y reviendrons un peu plus loin dans ce même point.

Si l'on compare le temps de calcul de nos requêtes R8 et R7 portant sur le même nombre de résultats mais respectivement pour une et deux dimensions, nous constatons un rapport de l'ordre de 6000%. De la même manière que pour les deux autres modèles, cela tend à confirmer la complexité quadratique de nos requêtes en fonction du nombre de dimensions bien qu'il faudrait encore quelques requêtes pour nous en assurer. Cette forte augmentation s'explique par le fait que le nombre de jointures nécessaires à chaque itération est très fortement dépendant du nombre de dimensions à analyser comme c'est le cas pour tous les modèles que nous avons analysé dans ce mémoire.

Analysons à présent les courbes décrivant le temps de calcul des requêtes utilisant la clause de groupement pour XQuery. Nous constatons, comme pour les deux autres modèles, que cette

courbe est constante quel que soit le nombre de résultats renvoyés. Ce comportement est tout a fait logique quand on sait qu'une itération est faite par fait analysé. Si l'on compare les deux courbes, respectivement pour une et deux dimensions, nous observons un rapport de temps de l'ordre de 200% qui s'explique par le fait qu'à chaque itération, il faut analyser autant de clés de groupement qu'il y a de dimensions à prendre en compte. Il nous faudrait interroger les données en analysant plus de dimensions pour conclure que le temps de calcul augmente linéairement avec le nombre de dimensions mais nous pouvons constater que cette hausse est beaucoup moins sensible comparativement au temps pris par les requêtes écrites en XQuery standard.

En comparant les deux méthodes, c'est-à-dire sans et avec clause de groupement, nous pouvons constater que les requêtes n'utilisant pas cette clause sont toujours plus rapides que les requêtes l'utilisant. Cela s'explique par le nombre de jointures impressionnant nécessaires à chaque itération afin de parcourir les hiérarchies de dimensions. Souvenons nous que ces hiérarchies ne sont pas modélisées en utilisant le mécanisme arborescent de XML et que donc des jointures sont nécessaires pour les parcourir. Il semble cependant que le rapport de temps entre les deux méthodes tend à diminuer très fortement proportionnellement au nombre de dimensions prises en compte. Si nous observons les requêtes R8 et R7, ce rapport est de l'ordre de 15000% pour une dimension et n'est plus que de 500% pour deux dimensions. Nous pouvons donc nous attendre à ce que la méthode de groupement soit avantageuse à partir d'un nombre plus important de dimensions mais les résultats ne permettent malheureusement pas de l'affirmer. Nous ne pouvons donc pas nous prononcer sur l'avantage de notre clause de groupement pour ce modèle.

8.2.4 Comparaison des performances

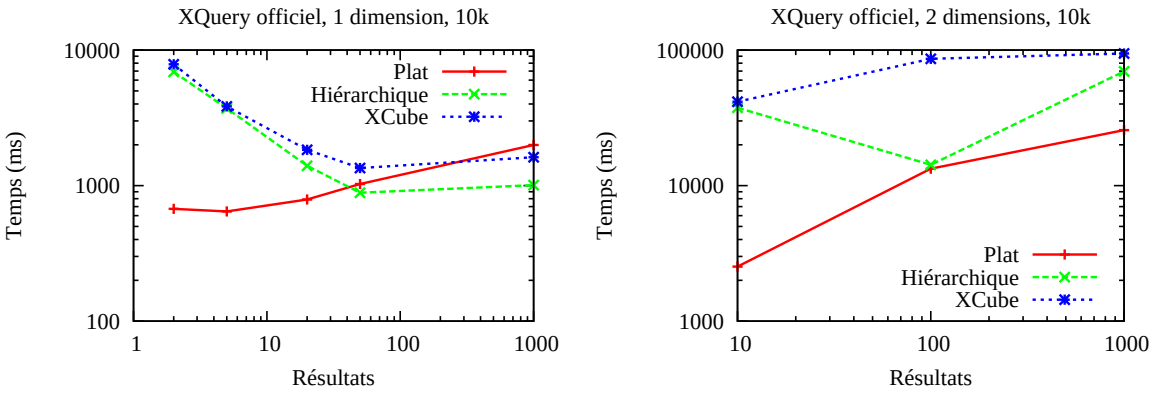
La figure 8.4 présent respectivement les résultats de toutes les requêtes sans (a) et avec utilisation de la clause de groupement (b). Pour chaque figure, le graphique de gauche représente les performances des requêtes à une dimension et celui de droite à deux dimensions.

Si nous observons les temps des requêtes sans clause de groupement (figure 8.4 (a)), nous pouvons constater que le choix d'un modèle par rapport à un autre n'est pas aisé. Les performances des requêtes renvoyant peu de résultats sont assez différents et dans ce cas, on pourrait conseiller l'utilisation du modèle plat. Par contre, dès que le nombre de résultats devient important, les performances des différents modèles tendent à se rejoindre. Cependant, l'interrogation du modèle plat semble avoir les meilleurs performances moyennes bien qu'il faudrait étendre notre bouquet de requêtes à plus de dimensions pour nous en assurer.

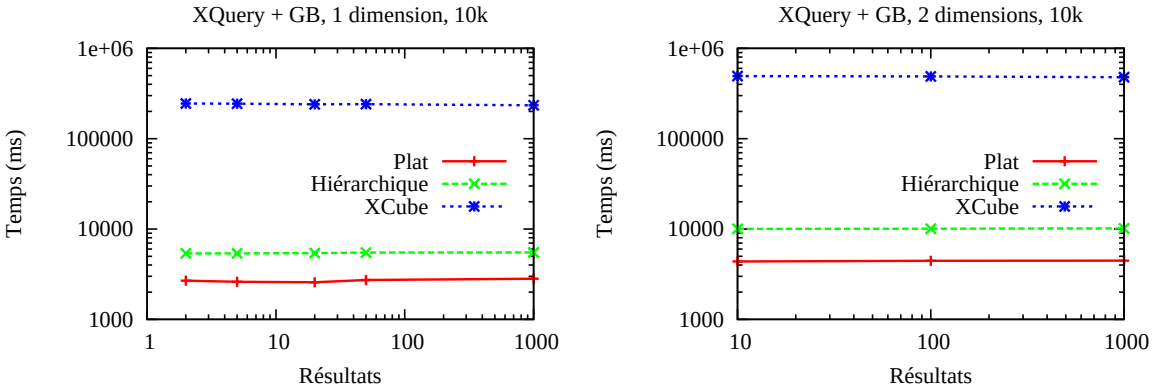
Une autre constatation que l'on peut faire est que les courbes des modèles hiérarchiques et XCube sont très similaires. Cela s'explique simplement par le fait que leur complexité est comparable.

Observons à présent la figure 8.4b représentant le temps de calcul des requêtes utilisant la clause de groupement. Nous avons déjà constaté que ces courbes sont constantes quelle que soit le nombre de résultats et que le rapport de temps entre les requêtes portant sur une et deux dimensions est de l'ordre de 200%.

Constatons que dans tous les cas, l'interrogation du modèle plat est la plus rapide. Les performances de l'interrogation du modèle hiérarchique semblent proches mais n'oublions pas



(a) sans clause de groupement



(b) avec clause de groupement

FIG. 8.4 – Temps d'exécution des requêtes en XQuery en fonction du nombre de résultats pour tous les modèles.

que nos graphiques utilisent des échelles logarithmiques. En effet, le rapport entre ces deux courbes est de l'ordre de 200% c'est-à-dire que les requêtes du modèle hiérarchique prennent deux fois plus de temps que celles du modèle plat. Remarquons également que la courbe du modèle XCube est très éloignée des deux autres. Le rapport moyen évolue entre 4000 et 8000% ce qui est prohibitif. Nous pouvons donc éliminer le modèle XCube si l'on recherche des performances d'interrogation.

Un dernier point à préciser est que le temps pris par toutes requêtes pour tous les modèles évolue linéairement avec le nombre de faits de la base de données.

En conclusion, il semble que le modèle plat conduit aux meilleures performances d'interrogation, que l'on utilise ou non la clause de groupement. Celui-ci est suivi d'assez prêt par le modèle hiérarchique lui même suivi par le modèle XCube. Nous pouvons donc conseiller l'utilisation du modèle plat et de la clause de groupement si le but recherché est d'exécuter ce type de requêtes le plus rapidement possible en dépit de la flexibilité apportée par les modèles hiérarchique et XCube. Le modèle XCube, comme on a pu le constater, n'est pas adapté à l'interrogation via le langage XQuery. En effet, ce modèle n'a pas été conçu dans le but d'être interrogé mais comme

un protocole de communication dans une architecture OLAP client-serveur utilisant une base de données relationnelle. Ce modèle est très flexible et sans redondance mais au vu de ses performances, nous ne pouvons pas le conseiller comme structure de données à interroger à l'aide XQuery. Cependant, les temps d'exécutions des requêtes du modèle hiérarchique ne sont pas beaucoup plus élevés que ceux du modèle plat. Nous conseillons donc l'utilisation du modèle hiérarchique si nous prenons en compte la flexibilité des données ainsi la rapidité d'interrogation. Ce modèle nous semble le meilleur compromis pour modéliser et interroger des données XML multidimensionnelles dans une approche entièrement basée sur les technologies XML.

8.3 Conclusion

Dans ce chapitre, nous avons réalisé un test de performances des requêtes d'extraction de sous-cubes sur les différents modèles présentés. Ensuite, nous avons effectué brièvement une analyse des résultats pour chaque modèle et nous avons ainsi pu comparer la rapidité d'interrogation des divers modèles.

Dans le chapitre suivant, nous allons comparer les modèles XML multidimensionnels en prenant en compte un maximum de critères.

Chapitre 9

Conclusions

9.1 Comparaison des modèles

Dans le chapitre 5, nous avons présenté différentes manières de modéliser des données multidimensionnelles à l'aide du langage XML. Dans les chapitres suivants, nous avons tenté d'interroger ces modèles via XQuery et nous avons évalué les performances d'interrogation de ces modèles. Dans cette section, nous allons comparer les divers modèles multidimensionnels XML en prenant en compte un maximum de facteurs comme leur taille, leur flexibilité, leur facilité et performance d'interrogation, etc. Pour ce faire, nous avons compilé ces différents résultats dans le tableau 9.1.

	Plat	**Plat XW**[1]	**Hiérarchique**	**XCube**
Taille de base de données (pour 100 K faits)	27088 Ko	25176 Ko	9148 Ko	14956 Ko
Temps total de l'évaluation des requêtes (sur 10 K faits) sans clause de groupement	46668 ms	46668 ms[2]	135435 ms	238901 ms
Temps total de l'évaluation des requêtes (sur 10 K faits) avec clause de groupement	26674 ms	26674 ms[2]	57463 ms	2663888 ms
Rapport du temps des requêtes d'une et de deux dimensions sans clause de groupement	1500%	1500%[2]	7000%	6000%
Rapport du temps des requêtes d'une et de deux dimensions avec clause de groupement	200%	200%[2]	200%	200%
Type de la jointure principale	simple	simple	multiple	multiple
Système de liens	✘	✘	✔	✔
Concision des requêtes	✔	✔	✔	✘
Flexibilité	✘	✘	✔	✔
Utilité de la clause de groupement	✔	✔	✔	?
Hiérarchie de dimension conservée	✘	✔	✔	✔
Utilisation de la relation père-fils XML pour modéliser les dimensions	✘	✔	✔	✘

Tab. 9.1 – Comparaison des modèles multidimensionnels XML.

[1] X-Warehousing
[2] Résultats non mesurés

Nous avons déjà comparé les performances d'interrogation de nos modèles à la section 8.2.4. Nous avons conclu que les modèles plats permettent d'obtenir les meilleurs performances d'interrogation. Ceux-ci sont suivis d'assez prêt par le modèle hiérarchique pour lequel l'interrogation ne prend en moyenne que deux fois plus de temps. Pour ces trois modèles, la clause de groupement que nous avons implémentée nous a permis de réduire le temps pris par les requêtes dès que celles-ci portaient sur deux dimensions ou plus. L'interrogation du modèle XCube est la moins rapide de notre analyse de performances et la clause de groupement ne semble pas pouvoir l'améliorer sauf peut-être pour un nombre important de dimensions. Nous pouvons donc conseiller l'utilisation des modèles plats ou du modèle hiérarchique si seul le critère de rapidité d'interrogation est pris en compte.

En ce qui concerne la flexibilité des modèles, nous avons constaté que dans les modèles plats, les faits contiennent physiquement toute leur hiérarchie de dimensions. Cela implique que si on veut changer quoi que ce soit dans la hiérarchie, il faut répercuter ces modifications dans tous les faits, ce qui est lourd et risqué. Par contre, avec le modèle hiérarchique ou le modèle XCube, il suffit pour ce faire d'opérer une mince modification dans les documents XML représentant les dimensions. Les deux modèles plats sont donc peu flexibles en ce qui concerne leurs dimensions, ce qui pourrait mener à des problèmes de maintenance. Dans cette optique nous conseillons l'utilisation du modèle hiérarchique ou XCube avec une préférence pour le premier car celui-ci utilise adéquatement les relations père-fils XML pour modéliser les hiérarchies de dimensions.

Le manque de flexibilité des modèles plats introduit beaucoup de redondance dans les données. Cela implique une consommation disque importante ce qui peut, par exemple, être un frein pour transporter ces données sur un réseau. Les modèles plus flexibles, hiérarchique et XCube, utilisent moins d'espace disque et sont donc plus adaptés pour le transport d'informations.

Discutons maintenant de la facilité d'écriture et de compréhension des requêtes XQuery traduites pour ces différents modèles. Nous en avons fait une analyse dans les chapitres 6 et 7. Nous avons conclu que les requêtes XQuery classiques étaient compliquées à écrire pour tous les modèles car, souvenons-nous, il fallait itérer sur toutes les instances des niveaux de dimension à analyser pour récupérer les différentes cases des sous-cubes. La clause de groupement que nous avons introduite a permis de faciliter grandement l'écriture de ce type de requêtes. Les requêtes ainsi écrites sont plus concises et semblent plus logiques. Nous conseillons donc l'utilisation de la clause de groupement couplée à la modélisation plate ou hiérarchique si le but recherché est la facilité d'écriture et de lecture. Pour ce point, nous ne pouvons que déconseiller l'utilisation du modèle XCube car le nombre de jointures nécessaires pour l'interroger est très important ce qui rend ces requêtes compliquées à comprendre et donc augmente le risque d'erreur, que l'on utilise la clause de groupement pour XQuery ou non.

En conclusion, le modèle hiérarchique et la clause de groupement nous semblent le meilleur compromis si l'on veut modéliser et interroger des données multidimensionnelles en XML. En effet, ce modèle a de bonnes performances d'interrogation, est flexible, léger du point de vue de l'espace disque et les requêtes pour l'interroger sont relativement faciles à écrire et à comprendre. Si nous ne prenons pas en compte la flexibilité et la consommation mémoire, les modèles plats semblent les plus intéressants car leurs performances d'interrogation sont les meilleures de notre analyse. Parmi les modèles plats, le modèle X-Warehousing est à conseiller car celui-ci permet

de conserver les hiérarchies de dimensions et utilise adéquatement les relations père-fils de XML pour les modéliser. Pour finir, malgré sa bonne flexibilité, nous ne pouvons conseiller le modèle XCube pour l'interrogation. En effet, ce modèle a été conçu comme un protocole de transport de données relationnelles multidimensionnelles ce en quoi il réussit plutôt bien car il est assez léger et très flexible. Par contre, il n'a pas été conçu pour être interrogé via des langages adaptés à XML ce qui se traduit par des requêtes compliquées et de mauvaises performances d'interrogation.

9.2 Conclusions générales

De nos jours, le format XML est devenu le format de choix pour représenter des données structurées. De nombreux outils permettant d'utiliser des documents XML ont vu le jour et sont en constante évolution. Les entreprises, de par leur taille importante et leur ancienneté, regroupent leurs informations dans des entrepôts de données afin de garantir leur pérennité et d'analyser ces données dans un but d'aide à la décision. Cependant, bien que XML soit de plus en plus utilisé, il n'existe pas encore de système d'analyse de données basé entièrement sur XML et seuls quelques articles scientifiques sont parus à ce sujet. Dans ce mémoire, nous avons tenté de joindre ces deux grands sujets, c'est-à-dire XML et les systèmes d'analyse.

Nous avons commencé par introduire les concepts nécessaires à la compréhension de notre travail. Nous avons présenté le langage XML et les outils qui lui sont associés comme les langages d'interrogation et les bases de données XML natives. Nous avons également introduit le domaine des entrepôts de données et de l'analyse multidimensionnelle.

Ensuite, nous avons présenté quatre façons de modéliser des données multidimensionnelles à l'aide du langage XML dont une est originale. Pour ce faire, nous avons traduit un exemple de travail dans chacun de ces modèles et nous en avons analysé intuitivement leurs avantages et inconvénients.

Par la suite, nous avons étudié la manière d'interroger des données ainsi modélisées en traduisant des requêtes typiques d'analyse à l'aide du langage d'interrogation XQuery. Nous avons remarqué que ces requêtes n'étaient pas faciles à comprendre et que leur complexité risquait de les rendre peu performantes. Pour pallier ce problème, nous avons présenté une extension au langage XQuery : une clause de groupement. Celle-ci nous a permis d'améliorer la lisibilité des requêtes ainsi que leur complexité.

Afin de pouvoir comparer objectivement l'interrogation des modèles multidimensionnels pour XML, nous avons réalisé un test de performance en exécutant une série de requêtes typiques d'analyse sur des données ainsi modélisées. Nous avons constaté pour celles-ci l'intérêt de la clause de groupement pour XQuery fraîchement implémentée. Nous ne pouvons donc que recommander qu'une version ultérieure des spécifications XQuery ajoute à sa syntaxe une telle clause.

Pour finir, nous avons réalisé une comparaison des modèles basée sur un ensemble important de critères. Le modèle hiérarchique que nous avons proposé couplé à la clause de groupement semble le meilleur compromis pour modéliser et interroger des données multidimensionnelles en XML. En effet, les performances de ce modèle sont très correctes et ce modèle est flexible de par l'utilisation adéquate de la structure XML pour modéliser les hiérarchies de dimensions.

9.3 Perspectives futures

Le sujet dans lequel nous nous sommes lancés dans ce mémoire est très vaste et est encore peu étudié. De nombreux aspects pourraient encore être analysés. En voici une liste non-exhaustive :

- ▷ La clause de groupement pourrait être étendue. En effet, il existe en SQL des extensions comme `ROLLUP` et `CUBE` permettant d'agréger les données de façon différente. Ces extensions étant très utilisées, il serait intéressant d'avoir un équivalent en XQuery.
- ▷ Les performances de groupement pourraient encore être améliorées via l'utilisation d'indexes spécifiques ou d'autres algorithmes. Il serait dès lors utile d'explorer ces voies.
- ▷ L'analyse de performance que nous avons réalisée pourrait être encore plus complète. En effet, celle-ci pourrait contenir plus de dimensions ainsi que des requêtes portant sur des sous-cubes plus grands.
- ▷ Il existe des langages simplifiés pour l'analyse comme MDX de Microsoft. Un tel langage compatible avec XML serait intéressant pour faciliter encore l'écriture des requêtes d'analyse multidimensionnelle.
- ▷ Pour notre analyse, nous avons supposé que les hiérarchies de dimensions étaient symétriques. Il faudrait analyser la possibilité de modéliser d'autres types de hiérarchies avec les modèles multidimensionnels présentés et les étendre au besoin. Une analyse de performances pourrait également être faite en prenant en compte plusieurs types de hiérarchies.

Bibliographie

[1] T. Bray, J. Paoli, and C. Michael. Extensible Markup Language (XML) Version 1.0. W3C Recommandation, 1998.

[2] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, 1999.

[3] J. Clark. XML Transformations (XSLT) Version 1.0. W3C Recommendation, 1999.

[4] V. Benzaken and A. Frisch. CDuce : an XML-centric general-purpose language. *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 51–63, 2003.

[5] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery 1.0 : A Query Language for XML. W3C Recommendation, 2007.

[6] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J.L. Wiener. The Lorel query language for semistructured data. *Proceedings of the International Journal on Digital Libraries*, 1(1) :68–88, 1997.

[7] J. Melton and S. Buxton. *Querying XML - XQuery, XPath, and SQL/XML in Context*. Morgan Kaufmann, 2006.

[8] M. Fernandez, J. Simeon, P. Wadler, et al. XML Query Languages : Experiences and Exemplars. *Communication to the XML Query W3C Working Group*, 256, 1999.

[9] J. Robie, D. Chamberlin, M. Marchiori, and P. Fankhauser. XML Query (XQuery) Requirements. W3C Working Draft, 2005.

[10] J. Robie, D. Chamberlin, and D. Florescu. Quilt : an XML Query Language. *XML 2000*, pages 316–329, 2000.

[11] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). W3C Proposal, 1998.

[12] A. Deutsch, M. Fernandez, and D. Florescu. XML-QL : A Query Language For XML. W3C Proposal, 1998.

[13] RGG Cattell, D. Wade, D.K. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, and H. Strickland. *The object database standard : ODMG 2.0*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1997.

[14] D. Chamberlin and J. Robie. XQuery Update Facility Requirements. W3C Working Draft, 2005.

[15] S. Buxton and M. Rys. XQuery and XPath Full-Text Requirements. W3C Working Draft, 2003.

[16] J. Melton and S. Muralidhar. XML Syntax for XQuery 1.0 (XQueryX). W3C Candidate Recommendation, 2006.

[17] K.S. Beyer, R.J. Cochrane, L.S. Colby, F. Özcan, and H. Pirahesh. XQuery for Analytics : Challenges and Requirements. *Proceedings of the 1st International Workshop on XQuery Implementations, Experiments and Perspectives (XIME-P 04)*, pages 3–8, 2004.

[18] A. Laux and L. Martin. XUpdate XML Update Language. XML :DB Working Draft, 2000.

[19] Chamberlin D., D. Florescu, and J. Robie. XQuery Update Facility. W3C Working Draft, 2006.

[20] H. Schöning. Tamino—A DBMS designed for XML. *Proceedings of the 17th International Conference on Data Engineering*, pages 149–154, 2001.

[21] S.Y. Chien, V.J. Tsotras, C. Zaniolo, and D. Zhang. Efficient Complex Query Support for Multiversion XML Documents. *Proceedings of the 9th International Conference on Extending Database Technology*, 2002.

[22] Y.K. Lee, S.J. Yoo, K. Yoon, and P.B. Berra. Index structures for structured documents. *Proceedings of the first ACM international conference on Digital libraries*, pages 91–99, 1996.

[23] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 361–370, 2001.

[24] D. Shin, H. Jang, and H. Jin. BUS : An Effective Indexing and Retrieval Scheme in Structured Documents. *Proceedings of the third ACM conference on Digital libraries*, pages 235–243, 1998.

[25] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *ACM SIGMOD Record*, 30(2) :425–436, 2001.

[26] S. Al-Khalifa, HV Jagadish, N. Koudas, JM Patel, and D. Srivastava. Structural joins : a primitive for efficient XML query patternmatching. *Proceedings of the 18th International Conference on Data Engineering*, pages 141–152, 2002.

[27] R. Bayer and EF Codd. Binary B-Trees for Virtual Memory. *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description*, pages 219–235, 1971.

[28] D. Florescu and D. Kossmann. A Performance Evaluation of Alternative Mapping Schemes for Storing XML Data in a Relational Database. Technical report, 1999.

[29] T. Grust. Accelerating XPath location steps. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 109–120, 2002.

[30] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D.J. DeWitt, and J.F. Naughton. Relational Databases for Querying XML Documents : Limitations and Opportunities. *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 302–314, 1999.

[31] W. Meier. eXist : An Open Source Native XML Database. *Proceedings of the 2002 Web, Web-Services, and Database Systems.*, pages 7–10, 2002.

[32] T. Böhme and E. Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. *Proceedings of the 3rd Int. Workshop Data Integration over the Web (DIWeb)*, 2004.

[33] B. Inmon. *Building the Data Warehouse*. John Wiley, 1992.

[34] K. Kimball. *The Data Warehouse Toolkit*. John Wiley, 1996.

[35] E. Malinowski and E. Zimányi. Hierarchies in a multidimensional model : From conceptual modeling to logical representation. *Proceedings of Data & Knowledge Engineering*, 2006.

[36] E.F. Codd, S.B. Codd, and C.T. Salley. Providing OLAP (Online Analytical Processing) to User-Analysts : An IT Mandate. Technical report, 1993.

[37] N. Pendse and C. Creeth. *The OLAP report*. Business Intelligence, 1997.

[38] K.S. Beyer, D. Chambérlin, L.S. Colby, F. Özcan, H. Pirahesh, and Y. Xu. Extending XQuery for analytics. *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 503–514, 2005.

[39] O. Boussaid, R.B. Messaoud, R. Choquet, and S. Anthoard. X-Warehousing : an XML-Based Approach for Warehousing Complex Data. *Proceedings of the 10th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 39–54, 2006.

[40] R.R. Bordawekar and C.A. Lang. Analytical processing of XML documents : opportunities and challenges. *ACM SIGMOD Record*, 34(2) :27–32, 2005.

[41] W. Hümmer, A. Bauer, and G. Harde. XCube : XML for data warehouses. *Proceedings of the 6th ACM international workshop on Data warehousing and OLAP*, pages 33–40, 2003.

[42] M. Kay. Positional Grouping in XQuery. *Proceedings of the XIME-P*, pages 22–28, 2006.

[43] V. Borkar and M. Carey. Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins. *Proceedings of the XML 2004 Conference and Exibition (Washington, DC)*, 2004.

[44] A. Deutsch, Y. Papakonstantinou, and Y. Xu. Minimization and Group-By Detection for Nested XQueries. *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, 1063(6382/04) :20–00.

Annexe A

Requêtes XQuery de l'évaluation de performance

A.1 Modèle plat

```
for $continent in distinct-values($factDoc//orders/order/customer_dimension/continent/
    @conId)
let $facts := $factDoc//order[customer_dimension/continent/@conId eq $continent]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.1 – Traduction de la requête R1 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $continent := $fact/customer_dimension/continent/@conId
group $fact as $group by $continent as $con
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.2 – Traduction de la requête R1 pour le modèle plat avec clause de groupement.

```
for $family in distinct-values($factDoc//orders/order/product_dimension/family/@famId)
let $facts := $factDoc//order[product_dimension/family/@famId eq $family]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.3 – Traduction de la requête R2 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $family := $fact/product_dimension/family/@famId
group $fact as $group by $family as $fam
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.4 – Traduction de la requête R2 pour le modèle plat avec clause de groupement.

```
for $family in distinct-values($factDoc//orders/order/product_dimension/family/@famId)
for $continent in distinct-values($factDoc//orders/order/customer_dimension/continent/
    @conId)
let $facts := $factDoc//order[product_dimension/family/@famId = $family][
    customer_dimension/continent/@conId = $continent]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.5 – Traduction de la requête R3 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $family := $fact/product_dimension/family/@famId
let $continent := $fact/customer_dimension/continent/@conId
group $fact as $group by $family as $fam, $continent as $con
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.6 – Traduction de la requête R3 pour le modèle plat avec clause de groupement.

```
for $category in distinct-values($factDoc//orders/order/product_dimension/category/@catId
    )
let $facts := $factDoc//order[product_dimension/category/@catId eq $category]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.7 – Traduction de la requête R4 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $category := $fact/product_dimension/category/@catId
group $fact as $group by $category as $cat
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.8 – Traduction de la requête R4 pour le modèle plat avec clause de groupement.

```
for $category in distinct-values($factDoc//orders/order/product_dimension/category/@catId
    )
for $continent in distinct-values($factDoc//orders/order/customer_dimension/continent/
    @conId)
let $facts := $factDoc//order[product_dimension/category/@catId = $category][
    customer_dimension/continent/@conId = $continent]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.9 – Traduction de la requête R5 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $category := $fact/product_dimension/category/@catId
let $continent := $fact/customer_dimension/continent/@conId
group $fact as $group by $category as $cat, $continent as $con
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.10 – Traduction de la requête R5 pour le modèle plat avec clause de groupement.

```
for $product in distinct-values($factDoc//orders/order/product_dimension/product/@prodId)
let $facts := $factDoc//order[product_dimension/product/@prodId eq $product]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.11 – Traduction de la requête R6 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $product := $fact/product_dimension/product/@prodId
group $fact as $group by $product as $prod
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.12 – Traduction de la requête R6 pour le modèle plat avec clause de groupement.

```
for $family in distinct-values($factDoc//orders/order/product_dimension/family/@famId)
for $country in distinct-values($factDoc//orders/order/customer_dimension/country/@couId)
let $facts := $factDoc//order[product_dimension/family/@famId = $family][
    customer_dimension/country/@couId = $country]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.13 – Traduction de la requête R7 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $family := $fact/product_dimension/family/@famId
let $country := $fact/customer_dimension/country/@couId
group $fact as $group by $family as $fam, $country as $cou
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.14 – Traduction de la requête R7 pour le modèle plat avec clause de groupement.

```
for $customer in distinct-values($factDoc//orders/order/customer_dimension/customer/
    @cusId)
let $facts := $factDoc//order[customer_dimension/customer/@cusId eq $customer]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.15 – Traduction de la requête R8 pour le modèle plat sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $customer := $fact/customer_dimension/customer/@cusId
group $fact as $group by $customer as $cus
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.16 – Traduction de la requête R8 pour le modèle plat avec clause de groupement.

A.2 Modèle hiérarchique

```
for $continent in $customerDoc//customers/continent
let $facts := $factDoc//orders/order[customer/@ref = $continent/country/customer/@cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.17 – Traduction de la requête R1 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $continent := $customerDoc//customers/continent[country/customer/@cusId eq $fact/
    customer/@ref]/@conId
group $fact as $group by $continent as $conId
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.18 – Traduction de la requête R1 pour le modèle hiérarchique avec clause de groupement.

```
for $family in $productDoc//products/family
let $facts := $factDoc//orders/order[product/@ref = $family/category/product/@prodId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.19 – Traduction de la requête R2 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $family := $productDoc//products/family[category/product/@prodId eq $fact/product/
    @ref]/@famId
group $fact as $group by $family as $famId
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.20 – Traduction de la requête R2 pour le modèle hiérarchique avec clause de groupement.

```
for $family in $productDoc//products/family
for $continent in $customerDoc//customers/continent
let $facts := $factDoc//orders/order[product/@ref = $family/category/product/@prodId][
    customer/@ref = $continent/country/customer/@cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.21 – Traduction de la requête R3 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $family := $productDoc//products/family[category/product/@prodId eq $fact/product/
    @ref]/@famId
let $continent := $customerDoc//customers/continent[country/customer/@cusId eq $fact/
    customer/@ref]/@conId
group $fact as $group by $family as $famId, $continent as $conId
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.22 – Traduction de la requête R3 pour le modèle hiérarchique avec clause de groupement.

```
for $category in $productDoc//products/family/category
let $facts := $factDoc//orders/order[product/@ref = $category/product/@prodId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.23 – Traduction de la requête R4 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $category := $productDoc//products/family/category[product/@prodId eq $fact/product/
    @ref]/@catId
group $fact as $group by $category as $cat
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.24 – Traduction de la requête R4 pour le modèle hiérarchique avec clause de groupement.

```
for $category in $productDoc//products/family/category
for $continent in $customerDoc//customers/continent
let $facts := $factDoc//orders/order[product/@ref = $category/product/@prodId][customer/
    @ref = $continent/country/customer/@cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.25 – Traduction de la requête R5 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $category := $productDoc//products/family/category[product/@prodId eq $fact/product/
    @ref]/@catId
let $continent := $customerDoc//customers/continent[country/customer/@cusId eq $fact/
    customer/@ref]/@conId
group $fact as $group by $category as $catId, $continent as $conId
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.26 – Traduction de la requête R5 pour le modèle hiérarchique avec clause de groupement.

```
for $product in $productDoc//products/family/category/product
let $facts := $factDoc//orders/order[product/@ref = $product/@prodId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.27 – Traduction de la requête R6 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $product := $productDoc//products/family/category/product[@prodId eq $fact/product/
    @ref]/@prodId
group $fact as $group by $product as $prod
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.28 – Traduction de la requête R6 pour le modèle hiérarchique avec clause de groupement.

```
for $family in $productDoc//products/family
for $country in $customerDoc//customers/continent/country
let $facts := $factDoc//orders/order[product/@ref = $family/category/product/@prodId][
    customer/@ref = $country/customer/@cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.29 – Traduction de la requête R7 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $family := $productDoc//products/family[category/product/@prodId eq $fact/product/
    @ref]/@famId
let $country := $customerDoc//customers/continent/country[customer/@cusId eq $fact/
    customer/@ref]/@couId
group $fact as $group by $family as $famId, $country as $couId
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.30 – Traduction de la requête R7 pour le modèle hiérarchique avec clause de groupement.

```
for $customer in $customerDoc//customers/continent/country/customer/@cusId
let $facts := $factDoc//orders/order[customer/@ref eq $customer]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.31 – Traduction de la requête R8 pour le modèle hiérarchique sans clause de groupement.

```
for $fact in $factDoc//orders/order
let $customer := $customerDoc//customers/continent/country/customer[@cusId eq $fact/
    customer/@ref]/@cusId
group $fact as $group by $customer as $cus
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.32 – Traduction de la requête R8 pour le modèle hiérarchique avec clause de groupement.

A.3 Modèle XCube

```
for $continent in $customerDoc//dimensionData/classification/level[@sid eq "continent"]/
    node
let $couId := $customerDoc//dimensionData/classification/level[@sid eq "country"]/node[
    rollUp/@toNode eq $continent/@id]/@id
let $cusId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    rollUp/@toNode = $couId]/@id
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid="customers"]/@node = $cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.33 – Traduction de la requête R1 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $cusId := $fact//dimension[@sid eq "customers"]/@node
let $couId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    @id eq $cusId]/rollUp/@toNode
let $conId := $customerDoc//dimensionData/classification/level[@sid eq "country"]/node[
    @id eq $couId]/rollUp/@toNode
group $fact as $group by $conId as $con
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.34 – Traduction de la requête R1 pour le modèle XCube avec clause de groupement.

```
for $family in $productDoc//dimensionData/classification/level[@sid eq "family"]/node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "category"]/node[
    rollUp/@toNode eq $family/@id]/@id
let $prodId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[
    rollUp/@toNode = $catId]/@id
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid="products"]/@node = $prodId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.35 – Traduction de la requête R2 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $prodId := $fact/dimension[@sid="products"]/@node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[@id
     eq $prodId]/rollUp/@toNode
let $famId := $productDoc//dimensionData/classification/level[@sid eq "category"]/node[
    @id eq $catId]/rollUp/@toNode
group $fact as $group by $famId as $fam "
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.36 – Traduction de la requête R2 pour le modèle XCube avec clause de groupement.

```
for $continent in $customerDoc//dimensionData/classification/level[@sid eq "continent"]/
    node
for $family in $productDoc//dimensionData/classification/level[@sid eq "family"]/node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "category"]/node[
    rollUp/@toNode eq $family/@id]/@id
let $prodId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[
    rollUp/@toNode = $catId]/@id
let $couId := $customerDoc//dimensionData/classification/level[@sid eq "country"]/node[
    rollUp/@toNode eq $continent/@id]/@id
let $cusId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    rollUp/@toNode = $couId]/@id
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid eq "products"]/@node = $prodId
    ][dimension[@sid eq "customers"]/@node = $cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.37 – Traduction de la requête R3 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $cusId := $fact/dimension[@sid="customers"]/@node
let $couId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    @id eq $cusId]/rollUp/@toNode
let $conId := $customerDoc//dimensionData/classification/level[@sid eq "country"]/node[
    @id eq $couId]/rollUp/@toNode
let $prodId := $fact/dimension[@sid="products"]/@node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[@id
     eq $prodId]/rollUp/@toNode
let $famId := $productDoc//dimensionData/classification/level[@sid eq "category"]/node[
    @id eq $catId]/rollUp/@toNode
group $fact as $group by $conId as $con, $famId as $fam
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.38 – Traduction de la requête R3 pour le modèle XCube avec clause de groupement.

```
for $catId in $productDoc//dimensionData/classification/level[@sid eq "category"]/node
let $prodId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[
    rollUp/@toNode eq $catId/@id]/@id
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid="products"]/@node = $prodId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.39 – Traduction de la requête R4 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $prodId := $fact/dimension[@sid="products"]/@node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[@id
     eq $prodId]/rollUp/@toNode
group $fact as $group by $catId as $cat
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.40 – Traduction de la requête R4 pour le modèle XCube avec clause de groupement.

```
for $catId in $productDoc//dimensionData/classification/level[@sid eq "category"]/node
for $continent in $customerDoc//dimensionData/classification/level[@sid eq "continent"]/
    node
let $prodId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[
    rollUp/@toNode eq $catId/@id]/@id
let $couId := $customerDoc//dimensionData/classification/level[@sid eq "country"]/node[
    rollUp/@toNode eq $continent/@id]/@id
let $cusId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    rollUp/@toNode = $couId]/@id
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid eq "products"]/@node = $prodId
    ][dimension[@sid eq "customers"]/@node = $cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.41 – Traduction de la requête R5 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $cusId := $fact/dimension[@sid="customers"]/@node
let $couId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    @id eq $cusId]/rollUp/@toNode
let $conId := $customerDoc//dimensionData/classification/level[@sid eq "country"]/node[
    @id eq $couId]/rollUp/@toNode
let $prodId := $fact/dimension[@sid="products"]/@node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[@id
     eq $prodId]/rollUp/@toNode
group $fact as $group by $conId as $con, $catId as $cat
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.42 – Traduction de la requête R5 pour le modèle XCube avec clause de groupement.

```
for $prodId in $productDoc//dimensionData/classification/level[@sid eq "product"]/node
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid="products"]/@node = $prodId/
    @id]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.43 – Traduction de la requête R6 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $prodId := $fact/dimension[@sid="products"]/@node
group $fact as $group by $prodId as $prod
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.44 – Traduction de la requête R6 pour le modèle XCube avec clause de groupement.

```
for $couId in $customerDoc//dimensionData/classification/level[@sid eq "country"]/node
for $family in $productDoc//dimensionData/classification/level[@sid eq "family"]/node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "category"]/node[
    rollUp/@toNode eq $family/@id]/@id
let $prodId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[
    rollUp/@toNode = $catId]/@id
let $cusId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    rollUp/@toNode = $couId/@id]/@id
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid="products"]/@node = $prodId][
    dimension[@sid="customers"]/@node = $cusId]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.45 – Traduction de la requête R7 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $cusId := $fact/dimension[@sid="customers"]/@node
let $couId := $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node[
    @id eq $cusId]/rollUp/@toNode
let $prodId := $fact/dimension[@sid="products"]/@node
let $catId := $productDoc//dimensionData/classification/level[@sid eq "product"]/node[@id
     eq $prodId]/rollUp/@toNode
let $famId := $productDoc//dimensionData/classification/level[@sid eq "category"]/node[
    @id eq $catId]/rollUp/@toNode
group $fact as $group by $couId as $cou, $famId as $fam
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.46 – Traduction de la requête R7 pour le modèle XCube avec clause de groupement.

```
for $cusId in $customerDoc//dimensionData/classification/level[@sid eq "customer"]/node
let $facts := $factDoc//cubeFacts/cube/cell[dimension[@sid="customers"]/@node eq $cusId/
    @id]
where exists($facts)
return
<group>
  <count>{count($facts)}</count>
</group>
```

FIG. A.47 – Traduction de la requête R8 pour le modèle XCube sans clause de groupement.

```
for $fact in $factDoc//cubeFacts/cube/cell
let $cusId := $fact/dimension[@sid="customers"]/@node
group $fact as $group by $cusId as $con
return
<group>
  <count>{count($group)}</count>
</group>
```

FIG. A.48 – Traduction de la requête R8 pour le modèle XCube avec clause de groupement.

Annexe B

Résultats complets de l'évaluation de performance

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R1	hiérarchique	X	1K	5	1	452.67
R1	hiérarchique	G	1K	5	1	636.00
R1	hiérarchique	G	10K	5	1	5369.33
R1	hiérarchique	X	10K	5	1	3721.67
R1	hiérarchique	G	100K	5	1	54184.67
R1	hiérarchique	X	100K	5	1	59376.00
R1	plat	G	1K	5	1	243.33
R1	plat	X	1K	5	1	65.67
R1	plat	G	10K	5	1	2606.00
R1	plat	X	10K	5	1	644.33
R1	plat	G	100K	5	1	26980.33
R1	plat	X	100K	5	1	8532.00
R1	XCube	G	1K	5	1	3729.00
R1	XCube	X	1K	5	1	394.33
R1	XCube	X	10K	5	1	3846.33
R1	XCube	G	10K	5	1	244341.67

Tab. B.1 – Résultats de l'évaluation de la requête R1

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R2	hiérarchique	G	1K	2	1	534.00
R2	hiérarchique	X	1K	2	1	512.33
R2	hiérarchique	G	10K	2	1	5376.67
R2	hiérarchique	X	10K	2	1	6906.00
R2	hiérarchique	G	100K	2	1	54519.00
R2	hiérarchique	X	100K	2	1	117018.00
R2	plat	G	1K	2	1	247.00
R2	plat	X	1K	2	1	44.67
R2	plat	X	10K	2	1	673.00
R2	plat	G	10K	2	1	2680.33
R2	plat	G	100K	2	1	27583.00
R2	plat	X	100K	2	1	6916.67
R2	XCube	G	1K	2	1	3808.33
R2	XCube	X	1K	2	1	587.33
R2	XCube	X	10K	2	1	7862.33
R2	XCube	G	10K	2	1	244748.00

Tab. B.2 – Résultats de l'évaluation de la requête R2

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R3	hiérarchique	G	1K	10	2	993.00
R3	hiérarchique	X	1K	10	2	2934.00
R3	hiérarchique	X	10K	10	2	37631.67
R3	hiérarchique	G	10K	10	2	10031.67
R3	hiérarchique	X	100K	10	2	652519.00
R3	hiérarchique	G	100K	10	2	101281.67
R3	plat	G	1K	10	2	416.33
R3	plat	X	1K	10	2	185.33
R3	plat	G	10K	10	2	4369.33
R3	plat	X	10K	10	2	2524.00
R3	plat	X	100K	10	2	28697.67
R3	plat	G	100K	10	2	45179.00
R3	XCube	G	1K	10	2	7601.67
R3	XCube	X	1K	10	2	3431.33
R3	XCube	G	10K	10	2	492009.00
R3	XCube	X	10K	10	2	41618.33

Tab. B.3 – Résultats de l'évaluation de la requête R3

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R4	hiérarchique	G	1K	20	1	541.00
R4	hiérarchique	X	1K	20	1	248.00
R4	hiérarchique	X	10K	20	1	1402.67
R4	hiérarchique	G	10K	20	1	5429.33
R4	hiérarchique	G	100K	20	1	53970.67
R4	hiérarchique	X	100K	20	1	18141.00
R4	plat	G	1K	20	1	258.33
R4	plat	X	1K	20	1	76.00
R4	plat	X	10K	20	1	791.00
R4	plat	G	10K	20	1	2573.67
R4	plat	X	100K	20	1	9633.00
R4	plat	G	100K	20	1	27634.00
R4	XCube	G	1K	20	1	3210.00
R4	XCube	X	1K	20	1	288.67
R4	XCube	G	10K	20	1	239951.67
R4	XCube	X	10K	20	1	1839.67

Tab. B.4 – Résultats de l'évaluation de la requête R4

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R5	hiérarchique	G	1K	100	2	1095.00
R5	hiérarchique	X	1K	100	2	4459.67
R5	hiérarchique	G	10K	100	2	10079.67
R5	hiérarchique	X	10K	100	2	14224.67
R5	hiérarchique	G	100K	100	2	101997.00
R5	hiérarchique	X	100K	100	2	176283.67
R5	plat	G	1K	100	2	429.67
R5	plat	X	1K	100	2	973.67
R5	plat	G	10K	100	2	4440.33
R5	plat	X	10K	100	2	13327.67
R5	plat	G	100K	100	2	45169.00
R5	plat	X	100K	100	2	152089.67
R5	XCube	G	1K	100	2	6957.67
R5	XCube	X	1K	100	2	8557.33
R5	XCube	G	10K	100	2	489204.00
R5	XCube	X	10K	100	2	86299.00

Tab. B.5 – Résultats de l'évaluation de la requête R5

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R6	hiérarchique	G	1K	50	1	564.33
R6	hiérarchique	X	1K	50	1	231.67
R6	hiérarchique	G	10K	50	1	5482.33
R6	hiérarchique	X	10K	50	1	885.67
R6	hiérarchique	X	100K	50	1	10395.00
R6	hiérarchique	G	100K	50	1	54116.33
R6	plat	X	1K	50	1	95.00
R6	plat	G	1K	50	1	255.33
R6	plat	G	10K	50	1	2728.00
R6	plat	X	10K	50	1	1028.67
R6	plat	G	100K	50	1	27091.33
R6	plat	X	100K	50	1	11319.67
R6	XCube	G	1K	50	1	3227.33
R6	XCube	X	1K	50	1	288.33
R6	XCube	X	10K	50	1	1348.67
R6	XCube	G	10K	50	1	240928.00

Tab. B.6 – Résultats de l'évaluation de la requête R6

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R7	hiérarchique	X	1K	641	2	14250.00
R7	hiérarchique	G	1K	641	2	1091.33
R7	hiérarchique	X	10K	1000	2	69654.67
R7	hiérarchique	G	10K	1000	2	10181.33
R7	hiérarchique	X	100K	1000	2	955696.67
R7	hiérarchique	G	100K	1000	2	103099.00
R7	plat	G	1K	641	2	501.00
R7	plat	X	1K	641	2	2625.33
R7	plat	G	10K	1000	2	4458.67
R7	plat	X	10K	1000	2	25686.33
R7	plat	G	100K	1000	2	45684.33
R7	plat	X	100K	1000	2	332328.00
R7	XCube	G	1K	641	2	6462.00
R7	XCube	X	1K	641	2	16912.33
R7	XCube	X	10K	1000	2	94466.33
R7	XCube	G	10K	1000	2	478129.00

Tab. B.7 – Résultats de l'évaluation de la requête R7

Requête	Modèle	Méthode	Faits	Résultats	Clés	Temps moyen (ms)
R8	hiérarchique	G	1K	639	1	596.00
R8	hiérarchique	X	1K	639	1	445.00
R8	hiérarchique	X	10K	1000	1	1008.67
R8	hiérarchique	G	10K	1000	1	5512.67
R8	hiérarchique	X	100K	1000	1	6620.67
R8	hiérarchique	G	100K	1000	1	54988.67
R8	plat	X	1K	639	1	386.67
R8	plat	G	1K	639	1	294.33
R8	plat	X	10K	1000	1	1993.33
R8	plat	G	10K	1000	1	2818.33
R8	plat	X	100K	1000	1	16050.67
R8	plat	G	100K	1000	1	27264.67
R8	XCube	G	1K	639	1	2877.67
R8	XCube	X	1K	639	1	538.67
R8	XCube	G	10K	1000	1	234577.00
R8	XCube	X	10K	1000	1	1620.67

Tab. B.8 – Résultats de l'évaluation de la requête R8