Entrepôts de données XML : Développement d'un outil *Extraction Transformation Load (ETL)*

Université Libre de Bruxelles
Faculté des Sciences Appliquées

Mémoire de fin d'études présenté par
MARHOUMI Fatima Ezzahra
en vue de l'obtention du grade
d'Ingénieur Civil Informaticien
en Sciences Appliquées

Promoteur du Mémoire :
ZIMANYI Esteban

2005 - 2006

Remerciements

Je tiens à exprimer tout particulièrement ma reconnaissance envers Monsieur E. Zimanyi pour ses précieux conseils.

Merci également à ma famille, à tous ceux qui, de pré ou de loin, par un geste ou un simple sourire, ont contribué à faire de mon mémoire une expérience enrichissante.

Enfin, un remerciement spécial à Mlle Labiba MARHOUMI, pour m'avoir soutenu et encouragé tout au long de ces années.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Avant-propos

Lors des premiers projets décisionnels, la phase de la collecte des données et de leur transformation était souvent sous-estimée. C'est peut-être là une des principales explications Des échecs de réalisations et des très nombreux dépassements de budget. Pourtant cette phase d'extraction est de transformation préalable représente (selon les spécialistes du milieu) à peu près les trois quart du projet de création d'un *Data Warehouse*.

Les outils *ETL (Extract Transform Load)* prennent en charge l'une des fonctions les plus essentielle du système globale décisionnel. Il s'agit en effet de gérer toutes les étapes préalables au chargement effectif des données dans le *Data Warehouse*.

Extraire : Accéder à la majorité des systèmes de stockage de données *(SGBD, ERP*, fichiers à plat...) et récupérer les données identifiées.

Transformer : Transformer, reformater et nettoyer les données afin d'éliminer les valeurs non conforme au modèle de destination et d'éviter les doublons et autres incohérences.

Charger : Insérer les données dans le *Data Warehouse*. où elles sont mise à disposition des outils d'analyse *Data Mining*, l'analyse multimensionnelle *OLAP,* etc.

Que ce soit pour alimenter un *datamar*t ou un *Data Warehouse,* il est indispensable de traiter les processus *ETL* comme projet à part entière. Il doit donc être réalisé en se basant sur une méthodologie de développement bien définie.

- Mise en place de l'environnement de développement
- Analyse des besoins d'affaires
- La conception des mises en correspondance des données *(Logical data mapping)*
- Développement des processus *ETL*
- Tests unitaires
- Déploiement
- Maintenance

1.2 Structure du document

L'état de l'art des outils *ETL* et technologies dépendantes sera tout d'abord présenté. Viendra ensuite une introduction à l'informatique décisionnelle (*Data Warehousing*, modèle multidimensionnel, outil *ETL*). Le chapitre suivant présentera un aperçu sur la technologie *XML*, et plus précisément les bases de données *XML* et leurs performances. Les chapitres qui suivent sont quant à eux consacrés à la mise en œuvre d'un outil *ETL* (création d'un *Data Warehouse*, remplissage de ce dernier, les modules Extraction, Transformation et Chargement). En dernier lieu viendra le chapitre où sera présentée l'interface utilisateur de l'application

Chapitre 2

Etat de l'art

Integration Services de *Microsoft* [1]

Microsoft SQL Server 2005 Integration Services (SSIS) est une plate-forme qui permet de créer des solutions d'intégration de données très performantes, en particulier des packages *ETL* pour le *Data Warehouse*.

Integration Services propose les fonctionnalités suivantes :

- Des outils graphiques pour la création et la gestion d'erreurs des packages ;
- Des tâches pour la réalisation de fonctions de flux de travail telles que des opérations *FTP*, pour exécuter les instructions *SQL* ou l'envoi de messages électroniques ;
- Des sources et des destinations de données pour l'extraction des données de sources hétérogènes : fichiers plats, de feuilles de calcul *Excel*, de documents *XML* ainsi que de tables et de vues dans des bases de données relationnelles. Ainsi que pour le chargement des données ;
- Des transformations pour les opérations telles que le nettoyage, l'agrégation, la fusion et la copie de données ;
- Un service de gestion (le service *Integration Services*) pour administrer *Integration Services* et des interfaces de programmation d'application (*API*) pour programmer le modèle objet *Integration Services*.

Microsoft Integration Services est un projet très ambitieux, dont les fonctionnalités peuvent facilement s'intégrer à une application de grande envergure manipulant un grand volume de données, or le sujet de ce mémoire entre plus dans le domaine de la recherche que dans l'adaptation de modules préexistant.

R. Bourret, C. Bornhövd, A. Buchmann, Darmstadt University of Technology, Allemagne [5]

Cet article décris une utilité des *XMLDB* dont le but principal est de résoudre les problèmes classiques liés au mapping entre les bases de données relationnelles et les documents *XML*.

Cette utilité se base sur le fait de considérer les données des documents *XML* comme un arbre d'objets. Cette hypothèse permet d'utiliser les structures techniques connues des bases de données relationnelles objets et de les adapter pour gérer les problèmes propres aux données *XML*. Les principales caractéristiques de cette utilité sont les suivants :

- Elle intègre *JDBC* pour accéder aux bases de données, *SAX* et *DOM* pour accéder aux documents *XML*, et *DDML* pour les schémas *XML*. Il peut donc être utilisé pour construire des applications indépendamment de l'implémentation du *DBMS*, du *XML* parser ou encore du *DOM*.
- Offre un assistant de conversion des *DTD* en schémas *XML* ;
- Offre des composants pour générer dynamiquement des schémas relationnels à partir des schémas *XML* et des *DTD*, et vice versa ;
- Inclut un langage simple que les utilisateurs peuvent utiliser ;
- Met on disposition des utilisateurs un langage simple qui permet de programmer les différents mapping entre les bases de données relationnelles et les document *XML*.

La principale contribution apportée par ce projet, est sûrement l'intégration de concepts préexistants dans l'élaboration d'un moyen pratique est flexible de transfert de données, et la réalisation d'un mapping entre les deux technologies de gestion de données les plus importantes : Les documents *XML* et les bases de données relationnelles. Il serait de ce fait idéal pour la création d'un *ETL* qui manipule les deux sortes de base de données, le sujet de la présente étude vise uniquement des bases de données *XML*, il est toutefois intéressant de mettre l'accent sur ce genre de projets. Pour démontrer la possibilité d'une cohabitation entre le relationnel et la technologie *XML*, et donc la portabilité de l'application implémentée dans ce mémoire.

eXist [2]

eXist est une base de données *XML* native Open Source qui peut non seulement s'intégrer dans une application *Java* (grâce à la sevelet *XquerServelet*-, un composant *Cocoon*, ou encore l'utilisation de l'*API XML :DB*), mais elle peut également être utilisé dans de nombreux autre langages via ses *API REST* et *XML-RPC*.

eXist définit un modèle logique pour l'enregistrement et l'extraction des document, ce modèle considère la totalité du document *XML* comme unité de stockage, de plus *eXist* ne repose pas sur un modèle physique particulier pour le stockage. Elle peut par exemple être bâtie aussi bien sur une base relationnelle, hiérarchique, orientée-objet, ou bien utiliser des techniques de stockage propriétaires comme des fichiers indexés ou compressés.

*eXist*permet de formuler des requêtes *XPath* et *XQuery* et *XMLSchema* grâce à l'intégration des deux noyaux *XPath 2.0* et *XQuery 1.0*. Elle fournit également des extensions à *XPath,* comme des fonctions adaptées recherche textuelle. eXist offre la possibilité d'écrire des modules *XQuery* complémentaires soit directement en *XQuery* soit en *Java*. Elle supporte également quelques composants des technologie *XMLInclude*, *XPointer* et *XUpdate*. De plus *eXist* utilise un système d'indexation très performant pour améliorer la vitesse des requêtes, cette tâche est accomplie mutuellement par des indexes générés par le système ou configurés par l'utilisateur.

La flexibilité et la portabilité de ce projet font de lui l'environnement idéal pour la réalisation d'un *Data Warehouse* 100% *XML* native, de plus le fait qu'il implémente *XQuery*, et *XMLSchema,* met à notre disposition les deux outils dont en pourrait besoin, d'une part pour le questionnement de la base de données, d'autre part pour la validation des données avant leur enregistrement dans la base de données. Dernier point et non le moindre, *eXist* est Open Source, il peut donc être adapté et intégré dans une autre application.

Oracle Streamlining Oracle 9i ETL with Pipelined Table Functions [3]

Oracle propose une gamme complète de solutions modulaires et intégrées s'adressant à tous les besoins du Système d'Information Décisionnel de l'entreprise : de l'alimentation aux solutions d'analyses et de restitutions.

Oracle Warehouse Builder est l'*ETL Oracle* permettent de concevoir l'entrepôt décisionnel. Au delà des fonctionnalités classiques d'*ETL* (l'extraction, la transformation et l'alimentation des données), *Oracle Warehouse Builder* est un atelier graphique qui permet de modéliser les Datamarts[1] les plus sophistiqués de type *OLAP*. En bout de chaîne, *Oracle Warehouse Builder* est de plus, le seul *ETL* capable de générer des analyses prêtes à l'emploi pour l'utilisateur final. Il permet la création de fonctions de transformation appelées *Tables Functions*, implémentées en *Java, C* ou *PL/SQL* qui élimine le besoin d'un stockage intermédiaire des données au court de leur transformation. Effectivement ces données suivent un processus appelé pipeline, qui assure un parallélisme transparent lors de la transformation.

La figure ci-dessous illustre le parallélisme assuré par le concept des pipelines.

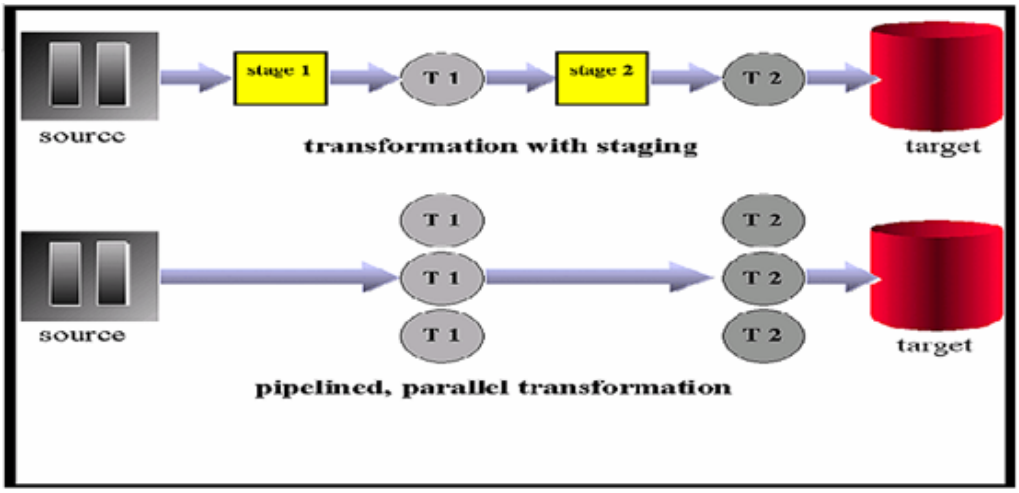

FIG. 2.1: Concept des pipelines

Comme *Microsoft Integration Services*, *Oracle* offre un large éventail de fonctionnalités toutes alléchantes mais disproportionnées par rapport aux objectifs visés par ce mémoire.

[1] ils se spécialisent dans un domaine "métier" particulier, comme la gestion de la relation client ou la gestion de la chaîne logistique par exemple.

Chapitre 3

Data Warehouse

3.1 Problématique

Les entreprises sont confrontées à une concurrence de plus en plus forte, des clients de plus en plus exigeants, dans un contexte organisationnel de plus en plus complexe et mouvant.

Pour faire face aux nouveaux enjeux économiques, l'entreprise doit anticiper. L'anticipation ne peut être efficace qu'en s'appuyant une information pertinente.

Parallèlement, les entreprises réalisent la valeur du capital d'information dont elles disposent. Au delà de ce que l'informatique leur apporte en terme fonctionnel, elles prennent conscience de ce qu'elle pourrait apporter en terme de contenu informationnel. Considérer le système d'information en tant que levier pour améliorer sa performance des entreprises et accroître leur compétitivité et leur réactivité n'est pas un concept nouveau. Par contre, étant donné l'environnement concurrentiel actuel, cela devient une question de survie.

Mais généralement, les données sont non organisées dans une perspective décisionnelle et éparpillées dans de multiples systèmes hétérogènes. Il devient fondamental de les rassembler et de les homogénéiser afin de faciliter la prise de décision.

Pour répondre à ces besoins, le nouveau rôle de l'informatique est de définir et d'intégrer une architecture qui serve de fondations au applications décisionnelles : un nouveau secteur informatique voit le jour « l'informatique décisionnelle ».

3.2 Différence entre l'informatique de production et l'informatique de décision

Les systèmes de production sont caractérisés par une activité constante composée de modifications et d'interrogations fréquentes des bases de données par de nombreux utilisateurs : ajouter une commande, modifier une adresse de livraison, rechercher les coordonnées d'un client, etc. l'intégrité des données est nécessaire pour ce genre d'applications (il faut par exemple, interdire la modification simultanée d'une même donnée par deux utilisateurs différents). La priorité est donnée en premier lieu à l'enregistrement rapide, sûr et efficace des données.

À l'inverse, nul besoin de modification ou d'enregistrement de nouvelles données dans les systèmes d'information de décision. En effet, le but principal consiste en l'interrogation du système d'information afin de permettre l'analyse des indicateurs pertinents pour faciliter les prises de décisions. Les questions posées seront de la forme : « quelles sont les ventes du produit X pendant le trimestre A de l'année B dans la région C ». Une telle interrogation peut nécessiter des temps de calcul importants. Or, l'activité d'un serveur transactionnel ne peut être interrompue. Il faut donc prévoir une nouvelle organisation qui permette de mémoriser un grand volume de jeux de données et qui soit structurée de façon à avoir un moteur de recherche de connaissances efficace et approprié.

3.2.1 L'informatique de production

L'un des formalismes les plus utilisés pour la représentation conceptuelle des systèmes d'information est le modèle Entité Association. Dans ce genre de modèle, toute redondance est considérée comme source d'erreurs, d'incohérence, et va pénaliser les temps d'exécution et complexifier les procédures d'ajout, de suppression ou de modification.

Les systèmes transactionnels temps réel, *OLTP (On-line Transaction Processing)*, garantissent l'intégrité des données. Les utilisateurs accèdent aux données de la base par de très courtes transactions atomiques et isolées. L'isolation évite la perturbation ou l'interruption de la transaction. La brièveté quant à elle garantit que les temps de réponse seront acceptables (inférieurs à la seconde) dans un environnement avec de nombreux utilisateurs.

Toutefois, le modèle Entité Association et sa réalisation dans un schéma relationnel sont pourtant des obstacles importants pour l'accès de l'utilisateur final aux données. Généralement, ces modèles contiennent plusieurs dizaines d'entités. Les bases sont alors constituées de nombreuses tables, reliées entre elles par divers liens dont le sens n'est pas toujours explicite. Il devient clair que le but était de faciliter le travail des développeurs et d'améliorer des performances de tâches répétitives, prévues et planifiées tournées vers la production de documents standards (factures, commandes...), au détriment des besoins réels de l'utilisateur final.

La dernière caractéristique de ces bases de données est qu'elles conservent l'état instantané du système. Dans la plupart des cas, l'évolution n'est pas sauvegardée. On garde simplement une trace des versions instantanées pour la reprise en cas de panne et pour des raisons légales.

3.2.2 L'informatique décisionnelle

L'informatique décisionnelle désigne les moyens, les outils et les méthodes qui permettent de collecter, consolider, modéliser et restituer les données d'une entreprise en vue d'offrir une aide à la décision et de permettre aux responsables de la stratégie d'une entreprise d'avoir une vue d'ensemble de l'activité traitée.

Ce qui caractérise d'abord les besoins, c'est la possibilité de poser une grande variété de questions au système, certaines prévisibles et planifiées comme des tableaux de bord et d'autres imprévisibles. Si des outils d'édition automatiques préprogrammés peuvent être envisagés, il est nécessaire de permettre à l'utilisateur d'effectuer les requêtes qu'il souhaite, par lui-même, sans l'intervention de programmeurs

Il sera souvent nécessaire de filtrer, d'agréger, de compter, sommer et de réaliser quelques statistiques élémentaires (moyenne, écart-type,...). La structure logique doit être prévue pour rendre aussi efficace que possible toutes ces requêtes. Pour y parvenir, il est nécessaire d'introduire de la redondance dans les informations stockées en mémorisant des calculs intermédiaires. On rompt donc avec le principe de non redondance des bases de production.

La cohérence assurée par les systèmes de production est toute relative. Elle se contrôle au niveau de la transaction élémentaire mais pas au niveau global et des activités de l'organisation. A l'inverse des systèmes d'informatique décisionnelle, la

cohérence requise doit être interprétable par l'utilisateur. Par exemple, si les livraisons n'ont pas été toutes saisies dans le système, comment garantir la cohérence de l'état du stock ?

Les systèmes d'informatique décisionnelle doivent donc assurer plutôt une cohérence globale des données. Pour ce faire, leur alimentation doit être une opération réfléchie et planifiée dans le temps. Les transferts de données du système opérationnel vers le système décisionnel seront réguliers avec une périodicité bien choisie dépendante de l'activité de l'entreprise. Chaque transfert sera contrôlé avant d'être diffusé.

Une dernière caractéristique importante de l'informatique décisionnelle, qui est aussi une différence fondamentale avec les bases de production, est qu'aucune information n'y est jamais modifiée. En effet, on mémorise toutes les données sur une période déterminée, les données ne seront jamais remises à jour car toutes les vérifications utiles à la cohérence globale sont procédées lors de l'alimentation. L'utilisation se résume donc à un chargement périodique, puis à des interrogations non régulières, non prévisibles, parfois longues à exécuter.

S'il existe effectivement des informations importantes, il n'en est pas moins nécessaire de construire une structure pour les héberger, les organiser et les restituer à des fins d'analyse. Cette structure est le *Data Warehouse* ou *"entrepôt de données"*. Ce n'est pas une usine à produire l'information, mais plutôt un moyen de la mettre à disposition des utilisateurs de manière efficace et organisée.

3.3 *Data Warehouse*

3.3.1 Définittion

Le *Data Warehouse ("entrepôt de données")* d'une entreprise regroupe tout ce qui peut concourir à donner une image de la situation de l'entreprise à un moment donné, tel que des données détaillées avec leur historique de vente et de chiffres d'affaires, mais aussi des données marketing, des traces des contacts avec les clients.

Cette collection unique de données, réactualisée régulièrement, peut atteindre des tailles considérables, de l'ordre de plusieurs dizaines de gigaoctets, notamment dans les grandes entreprises de distribution (c'est-à-dire vendant quotidiennement une quantité considérable d'articles).

3.3.2 Principe

Un *Data Warehouse* est une base de données qui se caractérise par des données :

– orientées « métier » ou business : par exemple, pour une banque, un compte débiteur sera agrégé avec les prêts accordés par la banque et non pas avec les autres comptes restés créditeurs, à la différence de ce qui se passe dans la comptabilité et le système de production d'origine ;

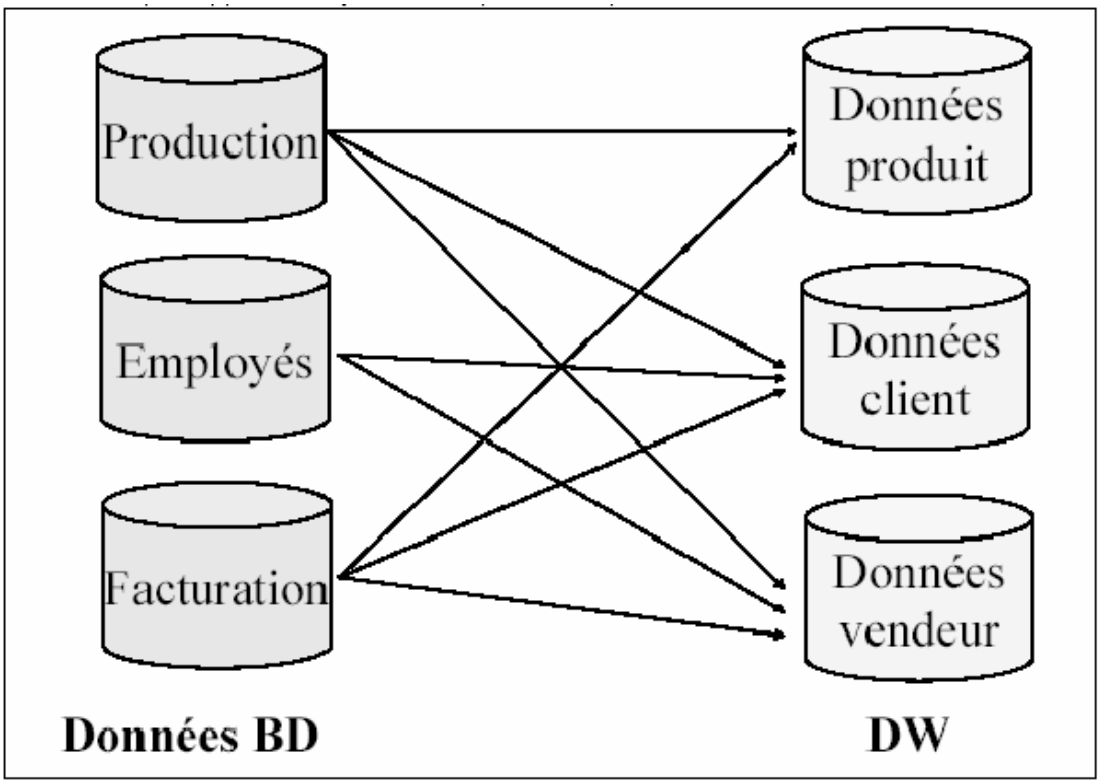

FIG. 3.1: Données Orienté sujet

– présentées selon différents axes d'analyse ou dimensions : par exemple, le temps, les types ou segments de clientèle, les différentes gammes de produits, les secteurs régionaux ou commerciaux, etc ;
– intégrées en provenance de sources hétérogènes ou d'origines diverses. Avant d'être intégrées dans le *Data Warehouse* les données doivent être mises en forme et unifiées afin de correspondre au standard défini dans le *Data Warehouse* ;

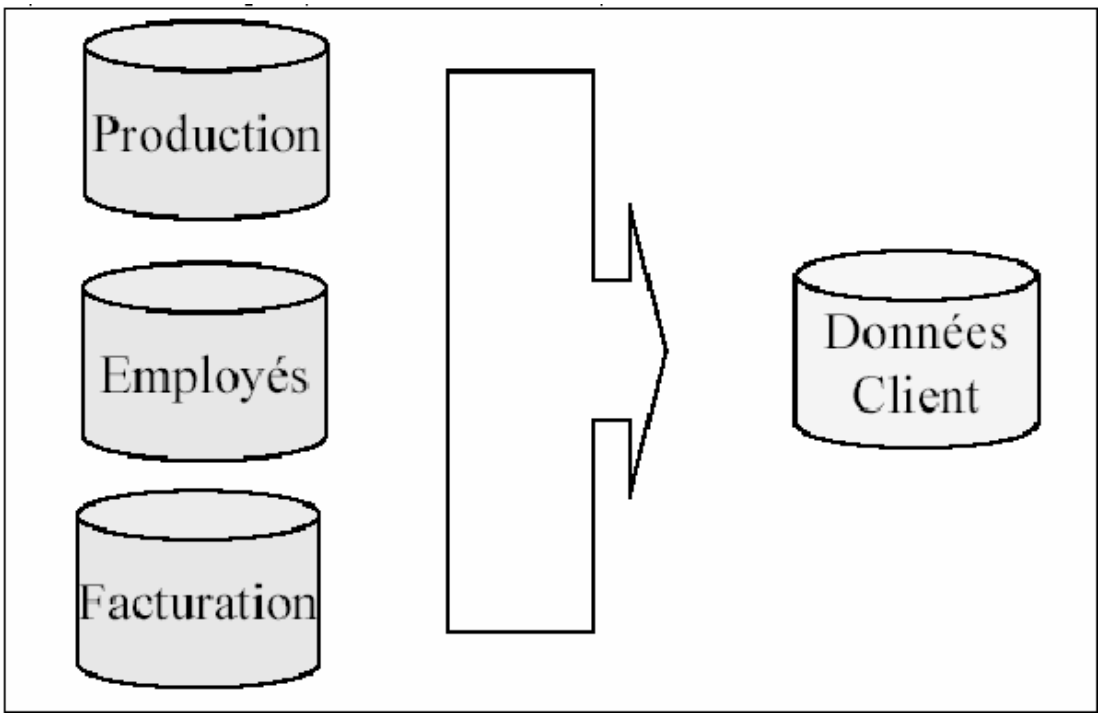

FIG. 3.2: Données intégrées

- historisées et donc datées : dans un système de production ; la donnée est mise à jour à chaque nouvelle transaction. A l'inverse dans un *Data Warehouse*, la donnée ne doit jamais être mise à jour. Un référentiel temps doit être associé à la donnée afin de pouvoir identifier une valeur particulière dans le temps pour permettre les analyses comparatives (par exemple, d'une année sur l'autre) ;
- non volatiles : stables, en lecture seule et non modifiables. La non volatilité est une conséquence directe de l'historisation. Une même requête effectuée à des mois d'intervalle en précisant la date de référence de l'information recherchée doit donner le même résultat.

Le model multidimensionnel répond à ces critères.

3.3.3 Modèle multidimentionnel

Le modèle conceptuel doit être simplifié au maximum pour permettre au plus grand nombre d'utilisateurs d'appréhender l'organisation des données et de comprendre ce que le *Data Warehouse* mémorise. On parle de modèle multidimensionnel. Ce dernier est utilisé dans les applications dont l'objectif est d'analyser les données plutôt que de procéder à des transactions on-line.

La technologie des bases de données multidimensionnelles est un facteur clé dans l'analyse de larges quantités de données dans l'informatique décisionnelle. En effet, contrairement aux technologies précédentes, les données sont vues comme des cubes particulièrement adaptés à l'analyse de données dans le modèle multidimensionnel.

3.3.3.1 Cubes

Dans le modèle multidimensionnel, les données seront toujours des faits à analyser suivant plusieurs dimensions. Par exemple, dans le cas de ventes de produits à des clients dans le temps (trois dimensions, un vrai cube), les faits sont les ventes, les dimensions sont les clients, les produits et le temps. Cela revient à dire que pour chaque combinaison des trois dimensions (clients, produis, temps), ont peut accéder à la mesure numérique associée au fait vente (cellule non vide). Les interrogations s'interprètent souvent comme l'extraction d'un plan, d'une droite de ce cube (par exemple, lister les ventes du produit A ou lister les ventes du produit A sur une période de temps D), ou l'agrégation de données le long d'un plan ou d'une droite (par exemple, obtenir le total des ventes du produit A revient à sommer les éléments du plan indiqué en figure 2.3).

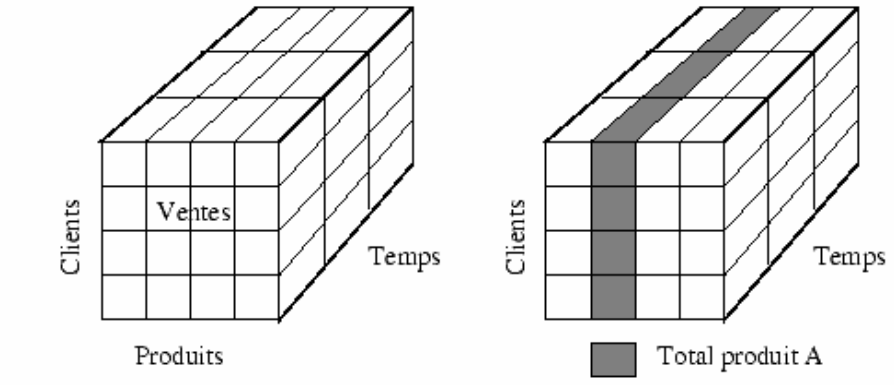

FIG. 3.3: Modèle en cube

Un *Data Warehouse* est composé d'un ensemble de cubes reliés. Théoriquement, un cube peut contenir un nombre infini de dimensions. Mais dans le monde réel, la plupart des cubes contiennent quatre à douze dimensions. Des problèmes de performance sont observés au-delà de cet intervalle.

Un avantage évident de ce modèle est la simplicité qu'il représente dès que les mots ventes, clients, produits et temps sont précisés. Il reprend les termes et la vision de l'entreprise de tout utilisateur final concerné par les processus de décision. Même si dans un entrepôt important il peut exister plusieurs cubes, parce qu'il est nécessaire de suivre plusieurs faits dans des directions parfois identiques, parfois différentes, l'utilisateur pourra accéder à des versions simplifiées, car plus ciblées, dans des *datamarts*[1].

3.3.3.2 Faits

Un fait représente un sujet d'analyse. Il est constitué de plusieurs mesures relatives au sujet traité. Ces mesures sont numériques et généralement valorisées de façon continue.

Dans la plupart des modèles multidimensionnels, les faits sont implicitement représentés par la combinaison des valeurs des dimensions. Un fait n'existe que si une combinaison particulière des dimensions découle vers une cellule non vide du cube le représentant.

3.3.3.3 Dimensions

Les dimensions sont un concept essentiel dans les bases de données multidimensionnelles. Elles sont les critères suivant lesquels on souhaite évaluer, quantifier et qualifier le fait. Elles peuvent être utilisées pour la sélection des données selon le niveau de précision désiré. Aussi, elles peuvent être affinées, décomposées en hiérarchies, afin de permettre à l'utilisateur d'examiner ses indicateurs à différents niveaux de détail, de "descendre" dans les données, en allant du niveau global au niveau le plus fin. Par exemple, une date pourra être décomposée en <*année, mois, semaine, jour*>. On aura alors une vision pyramidale des données, la base de la pyramide représentant le niveau le plus détaillé et le haut le niveau le plus global.

[1] ils se spécialisent dans un domaine "métier" particulier, comme la gestion de la relation client ou la gestion de la chaîne logistique par exemple.

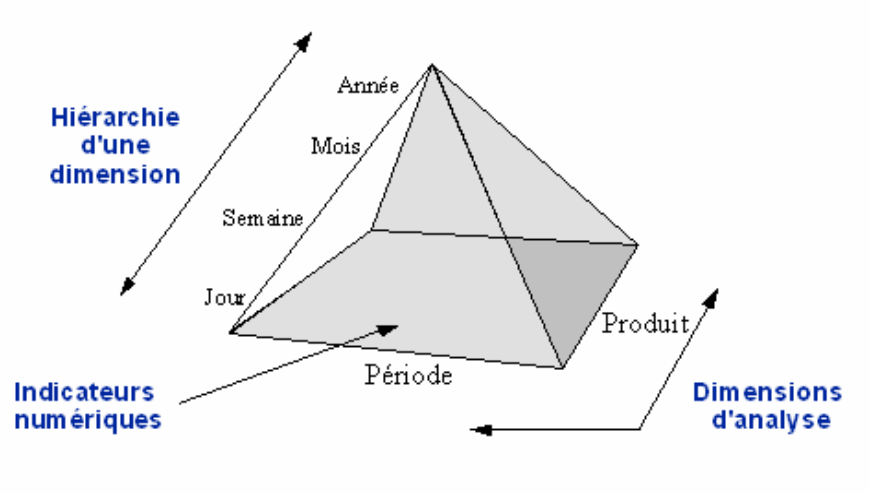

FIG. 3.4: Représentation pyramidale

L'utilisateur peut avoir besoin de personnaliser le modèle défini par l'administrateur en incorporant par exemple ses propres attributs dans les dimensions ou en modifiant certaine hiérarchies.

3.3.4 Modélisation

Quatre axes permettent de qualifier un modèle décisionnel :

- lisibilité du point de vue de l'utilisateur final ;
- performances de chargement ;
- performances d'exécution ;
- administration, c'est-à-dire faire vivre le *Data Warehouse.*

Partant du principe que les données sont des faits à analyser selon plusieurs dimensions, il est possible de réaliser une structure de données simple qui correspond à ce besoin de modélisation multidimensionnelle. Cette structure est constituée du fait central et des dimensions.

Au niveau logique, cela peut se traduire par trois modèles différents : en étoile, en flacon de neige ou en constellation.

- Modèle en étoile : le centre est la table des faits, et les branches en sont les dimensions. Pour une dimension, il existe plusieurs faits. La structure est dissymétrique. en effet, la table des faits est énorme et les tables de dimensions sont petites.

La table de faits contient une clé composée des clés des tables de dimensions. Les tables de dimensions contiennent quant à elle une clé primaire unique correspondant exactement à l'un des composants de la clé de la table des faits.
Les faits sont généralement numériques alors que les dimensions sont qualitatives, elles contiennent des informations textuelles pour décrire les faits.

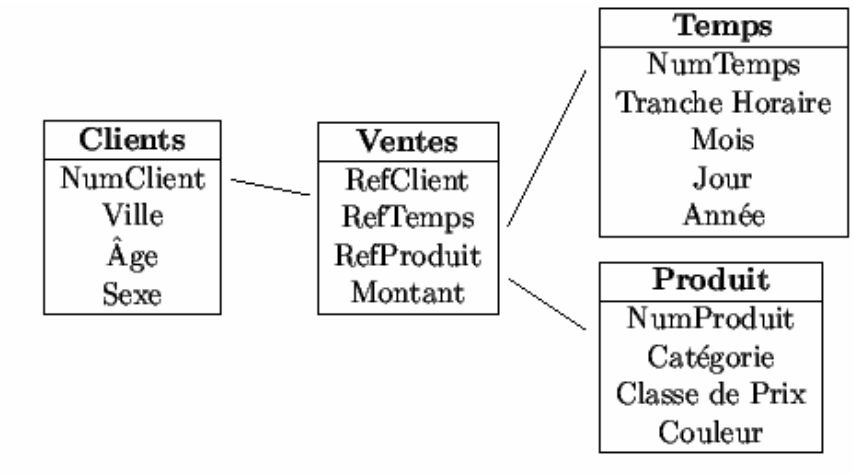

FIG. 3.5: Modèle en étoile

- Modèle en flocon : Le principe est le même que pour le modèle en étoile, mais en plus, les dimensions sont décomposées. Le but est d'économiser ainsi de la place. Cela permet également d'instaurer une hiérarchie au sein des dimensions, mais engendre par contre une complexification du modèle.
- Modèle en constellation : Il est encore basé sur le modèle en étoile, mais on rassemble plusieurs tables des faits qui utilisent les mêmes dimensions.

Pour nourrir cet entrepôt de données, il faut établir des flux de données entre l'entrepôt et les bases de données spécialisées où elles ont été créées (les bases de production). Pour ce faire, on met en place des interfaces d'extraction, de transformation et d'alimentation entre les bases de production et l'entrepôt. C'est ce qu'on appelle un outil *Extract Transform Load* (*ETL*).

Chapitre 4

Extract Transform Load

Les données du *Data Warehouse* sont, pour la plupart, issues des différentes sources de données opérationnelles de l'entreprise. Des solutions logicielles sont alors nécessaires à leur intégration et à leur homogénéisation. Celles-ci peuvent aller de l'écriture de batchs à l'utilisation de logiciels spécialisés dans l'extraction et la transformation d'informations (*ETI*, *Prism*, *Carleton*, ..., etc). Ces outils ont pour objet de s'assurer de la cohérence des données du *Data Warehouse* et d'homogénéiser les différents formats trouvés dans les bases de données opérationnelles. Les solutions de réplication sont souvent citées comme pouvant répondre à ce besoin. Très liées à un éditeur et à une technologie, elles ne sont en fait adaptées qu'à un très petit nombre de cas, où les données sont issues de sources homogènes et n'ont pas besoin d'être transformées.

Lors des premiers projets de l'informatique décisionnels, cette phase de collecte et de préparation des données était généralement sous-estimée. C'est peut-être là une des principales explications des échecs de réalisations et des très nombreux dépassements de budget. On retient que cette phase de collecte et de préparation préalable représente à peu près les $\frac{3}{4}$ du projet.

Les outils d'*ETL* (*Extract Transform Load*) ont en charge cette fonction essentielle du système globale décisionnel. Il s'agit en effet de gérer toutes les étapes de la collecte et de préparation des données. La figure 3.1 reprend une vue globale des besoins et outils d'un *Data Warehouse*.

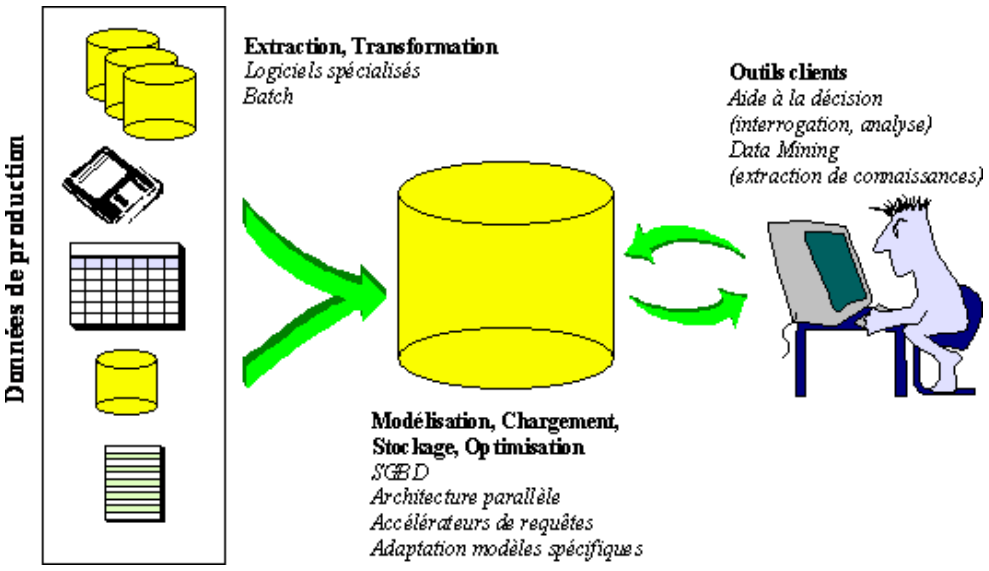

FIG. 4.1: Besoins et outils d'un Data Warehouse.

4.1 Extraction des données

L'extraction des données consiste à collecter les données utiles dans le système de production (*SGBD*, *ERP*, fichiers à plat, etc). Pour rafraîchir la base de données décisionnelle, il faut identifier les données ayant évolué afin d'en extraire le minimum, puis planifier ces extractions afin d'éviter les saturations de production.

Généralement l'extraction de données est différentielle, les données sont historisées. Cette fonctionnalité devient importante lorsque le volume des données est important. L'intégrité des données est indispensable et nécessite la synchronisation des différents processus d'extraction. Les problèmes liés à cette synchronisation peuvent être complexe, soit fonctionnellement, soit techniquement dans des environnements très hétérogènes.

Un autre problème est de traiter les données externes. Il faut maintenir une surveillance du système d'information pour pouvoir les identifier et s'assurer que ce sont les bonnes données qui sont recensées. De plus, la forme de ces données, qui est souvent totalement anarchique, accentue la difficulté. Pour être utilisables, ces données nécessitent un reformatage pour pouvoir les incorporer dans une forme exploitable pour l'entreprise.

Enfin le troisième problème vient de l'apparition imprévisible de ces données qui les rend difficiles à capter. En conséquent, l'outil d'extraction doit attaquer toutes sorte de sources de données sans être perturbé et s'adapter au future.

Pour extraire les données sources, il y a plusieurs technologies utilisables :

- des passerelles, fournies principalement par les éditeurs de bases de données. Ces passerelles sont généralement insuffisantes car elles sont mal adaptées aux processus de transformation complexes ;
- des utilitaires de réplication, utilisables si les système de production et décisionnel sont homogènes et si la transformation à appliquer aux données est légère ;
- des outils spécifiques d'extraction. Ces outils sont certainement la solution opérationnelle au problème de l'extraction, mais leur prix relativement élevé est un frein à leur utilisation dans les premières applications.

4.2 Transformation des données

Le nettoyage des données est une discipline sur laquelle de nombreux éditeurs travaillent actuellement. Outre la qualité des données qu'ils permettent d'auditer et éventuellement d'améliorer, les outils de nettoyage permettent de supprimer les doublons dans les fichiers. Il s'agit à ce stade d'appliquer des filtres prédéfinis sur les données afin d'attribuer des valeurs cohérentes aux variables mal ou non renseignées ou encore d'harmoniser les formats (date : jj/mm/aaaa). On peut également avoir à convertir les données d'un format *EBCDIC* vers *ASCII*. Aussi, des données du système opérationnel doivent être agrégées ou calculées avant leur chargement dans la basse décisionnelle.

Il faut également pouvoir associer des champs sources avec des champs cibles. Il existe plusieurs niveaux de complexité pour ces associations (cardinalités *1-1*, *1-N*, *N-1*, *N-N*), comme par exemple :

- le transfert du « nom du client » vers le champs cible
- la décomposition d'une « adresse » vers les champs « numéro », « rue », « ville » ou l'inverse.

Certains outils peuvent également réaliser des analyses lexicales des champs sources. Ils seront dons capables de comprendre que les champs suivants signifient la même chose : « Boulvd », « Bd », « Boulevard ».

En complément, on trouve des outils d'audit et d'analyse pour assurer le suivi du processus afin de pouvoir contrôler les rejets par exemple.

4.3 Le chargement des données

Le chargement est la dernière phase de l'alimentation du *Data Warehouse*. C'est une phase délicate notamment lorsque les volumes sont importants. Pour obtenir de bonnes performances en chargement, il est impératif de maîtriser les structures du *SGBD* (tables et index) associées aux données chargées afin d'optimiser au mieux ces processus. Les techniques de parallélisation optimisent les chargements lourds. Pour les mettre en œuvre, des utilitaires particuliers existent chez la majorité des éditeurs de bases de données.

Chapitre 5

Technologies *XML*

5.1 Qu'est ce que *XML* ?

XML, qui signifie *eXtensible Markup Language* (traduisible par langage de balisage extensible, ou encore langage à balises extensible), est un langage constitué de balises, ou de *tags*. Mis au point par le *XML Working Group* sous la direction de la *World Wide Web Consortium (W3C)* dès 1996, il est destiné à la structuration des documents. En 1998, les spécifications *XML* sont devenues des recommandations[1].

XML dérive du *Standard Generalized Markup Language*, ou *SGML*, défini en 1986 par le standard *ISO 8879*. La recommandation du *W3C* indique d'ailleurs que le but de *XML* est de permettre au *SGML* générique d'être transmis, reçu et traité sur le Web.

Le but du *SGML* était de dissocier complètement le contenu du document de sa présentation, tout en lui ajoutant une structure de description ne laissant aucune ambiguïté quant aux éléments de contenu. Pour exemple,

<description>un exemple</description>

ne décrit que ce que représente le texte, c'est-à-dire, qu'il s'agit d'une description sans aucun souci de présentation. C'est là qu'intervient le *XMLSchema* [4].

Pour être conforme au *SGML*, il faut respecter strictement et scrupuleusement une description type issue d'une norme complexe et complète. Prévu à l'origine pour

[1] c'est-à-dire que le *XML* a été officiellement reconnu comme standard

l'industrie, le SGML permet de gréer de grandes documentations techniques dans des domaines aussi complexes que l'aviation, l'industrie automobile, etc. Cette norme, également utilisée dans le domaine de la gestion électronique (ou *GED*), a été jugée beaucoup trop difficile à mettre en œuvre pour être accessible aux sociétés souhaitant structurer leurs informations de manière rapide et productive. De plus, lors de sa création, Internet était encore une préoccupation futuriste et *SGML* n'a pas été conçu pour une utilisation on-line. *XML* est donc venu régler ces problèmes.

La création de *XML* s'explique aussi par ses différences avec *HTML*. Bien que ces langages soient proches, *XML* est un langages beaucoup plus puissant et diversifié dans ses applications. En effet, les balises *HTML* sont définies et arrêtées par le *W3C*, alors que celles du *XML* sont limitées par le champ d'application qu'on pourrait y trouver. Il nous appartient de créer nos balises.

En fait, *XML* peut être vu comme un métalangage permettant d'en définir d'autres. Du fait de son extensibilité, *XML* possède la capacité de décrire n'importe quel domaine de données dont on saura la structure et le vocabulaire à employer.

De plus, une autre différence entre *XML* et *HTML* tient au fait que le premier permet de séparer le contenu de la présentation. En fait, *XML* a été largement promu pour cela. À l'aide de ce dernier, on pourrait transmettre un document sans nous soucier de son apparence finale ce qui n'était pas le cas avec *HTML*.

Pour résumé, *XML* permet de répondre à un tas de critères :

- *XML* peut être utilisé sans difficulté sur Internet ;
- Il soutient une grandes variétés d'applications ;
- Il facilite l'écriture de programmes traitants des documents *XML* ;
- Les documents *XML* sont lisibles par l'homme et raisonnablement claires ;
- La conception du *XML* est rapidement exécutable ;
- La facilité de créer des documents *XML*.

XML rassemble aussi certaines caractéristiques le positionnant comme une solution logique à l'ensemble des problèmes rencontrés dans l'échange et la structuration de documents :

- Le langage peut s'autodécrire ;
- Le langage peut être étendu par lui-même : Les langages dérivés du *XML* sont définis en *XML*, ce qui permet donc de gérer tous les domaines d'application ;

- Le contenu d'un document peut être lisible dans n'importe quel éditeur de texte et être compréhensible par une personne n'ayant aucune connaissance particulière ;
- Le langage permet la description arborescente des données afin d'apporter par l'intermédiaire des arbres des structures de données efficaces ;
- Il peut être facilement distribué par n'importe quel protocole à même de transporter du texte, comme le *HTTP* ;
- Toute application munie d'un parseur texte peut utiliser le document *XML*.

5.2 *XML* et les bases de données

XML est en passe de devenir le standard utilisé systématiquement par les entreprises pour échanger des données dans des documents structurés, que ce soit en interne, avec des partenaires commerciaux ou dans des applications accessibles à tous sur internet. Toutefois, ces données sont souvent stockées dans un format de fichier autre que le *XML* par exemple, dans une base de données relationnelle. Il faut donc mettre en place un processus de conversion des données du format actuel de la base en documents *XML*, puis du *XML* au format utilisé pour le traitement des données, cela se traduit par un plus long traitement des données.

Des outils sont disponibles avec les bases de données relationnelles pour traiter ce genre de tâche, comme *Oracle9i* et *IBM DB2*. Ils peuvent traduire en *XML* les données, structurées ou non. Toutefois, si ces outils sont utilisés pour traduire des données en direct lors d'un échange *XML*, cela risque d'augmenter les temps de traitement des transactions *XML* et de ralentir également les autres applications qui accèdent à la base de données relationnelles.

Il existe cependant une autre approche : les bases de données *XML* natives.

5.2.1 Bases de données *XML* natives

L'une des définitions possibles, développée par les membres de la liste de diffusion *XML :DB*, est la suivante :

Une base de données *XML* native définit un modèle (logique) de document *XML* (modèle est ici opposé aux données du document), et stocke et retrouve les documents en fonction de ce modèle.

Le document *XML* est l'unité fondamentale du stockage (logique) dans une base de données *XML* native, tout comme une ligne d'une table constitue l'unité fondamentale du stockage (logique) dans une base relationnelle.

Une base de données *XML* native ne repose pas sur un modèle physique particulier pour le stockage. Elle peut par exemple être bâtie aussi bien sur une base relationnelle, hiérarchique, orientée-objet, ou bien utiliser des techniques de stockage propriétaires comme des fichiers indexés ou compressés.

Les architectures des bases de données *XML* natives

Il existe deux grandes catégories d'architectures de bases *XML* natives : les architectures basées sur le texte et celles qui sont basées sur un modèle.

5.2.1.1 Les bases de données *XML* natives basées sur le texte

Une base de données *XML* native basée sur le texte stocke le *XML* en tant que texte, les index sont communs à toutes les bases de données *XML* natives basées sur le texte. Ils permettent au moteur de recherche de naviguer facilement en tout point d'un document *XML* quelconque. Cela procure à ce genre de base un avantage considérable en matière de vitesse quand on recherche des documents entiers ou des fragments de documents.

De ce point de vue, une base de données *XML* native basée sur le texte est similaire à une base hiérarchique en ce sens que toutes deux surpassent une base relationnelle quand on recherche et retourne des données selon une hiérarchie prédéfinie. À l'instar également des bases hiérarchiques, les bases *XML* natives basées sur le texte sont susceptibles de rencontrer des problèmes de performance quand on recherche et retourne des données sous une forme non structurée, contrairement aux bases données relationnelles dont l'utilisation des pointeurs logiques permet à toutes les questions d'une même complexité d'être exécutées avec la même vitesse.

5.2.1.2 Les bases de données *XML* natives basées sur un modèle

Plutôt que de stocker un document *XML* en tant que texte, ces bases de données construisent un modèle objet interne du document et stockent ce modèle. La manière dont le modèle est stocké dépend de la base. Certains produits stockent le modèle dans une base relationnelle ou orientée objet. D'autres bases utilisent

un format de stockage propriétaire adapté à leur modèle. Les bases *XML* natives basées sur un modèle et construites sur d'autres bases possèdent vraisemblablement des performances similaires à ces bases sous-jacentes lors de la recherche des documents, et ce, pour la raison évidente qu'elles reposent sur ces systèmes pour retrouver les données.

A l'instar des bases *XML* natives basées sur le texte, les bases natives basées sur un modèle rencontrent probablement des problèmes de performance lorsque l'on recherche et retourne des données dans un format quelconque autre que celui sous lequel ces données sont stockées, par exemple lorsque l'on inverse la hiérarchie de certaines de ses parties. La question de savoir si ces bases seront plus rapides ou non que les systèmes basés sur le texte n'est pas claire non plus.

Ces bases de données, relativement récentes sur le marché, ne remplacent pas les bases de données relationnelles. Elles servent plutôt de cache intermédiaire entre les applications web et les sources de données pour améliorer les performances.

De plus les bases de données *XML* natives permettent de traduire des données provenant de plusieurs sources en *XML*, et inversement. Il s'agit donc d'une amélioration nette sur l'implémentation de techniques de traduction *XML* pour chaque source de données. De cette manière on réduit les besoins de traitement que requiert la gestion des traductions*XML*.

5.2.1.3 Les inconvénients

Comme c'est le cas de toute technologie, il y a quelques désavantages avec les bases de données *XML* natives. Tout d'abord, il s'agit d'une technologie récente qui n'a pas encore été largement testée. De plus, les bases de données *XML* natives sont très chères. On voit toutefois arriver des bases de données *XML* natives à des prix plus raisonnables. La plupart d'entre elles ont été développées par la communauté open-source et arrivent désormais sous une forme commerciale. Elles restent intéressantes à considérer, car elles proposent des fonctionnalités presque identiques à celles de leurs concurrentes.

Les bases de données *XML* natives sont assez faciles à installer. Des techniciens seront néanmoins nécessaires pour intégrer la base de données *XML*, les sources de données et les applications. Un développeur expérimenté peut terminer l'intégration rapidement, mais il lui faudra être familier avec les *API* et les liaisons.

Un expert sera également requis pour configurer et administrer le tout de façon optimale.

Si toutefois une entreprise prévoit d'utiliser le *XML* à grande échelle ou à long terme, ces inconvénients ne sont pas si importants comparés aux avantages : performances améliorées, standardisation du *XML* et automatisation des processus commerciaux.

5.2.2 Performances des *Data Warehouse XML*

Les systèmes de gestion se données (*SGBD*) *XML* natifs actuels présentent globalement des performances nettement inférieures à celles des *SGBD* relationnels. Cependant, stocker des données *XML,* notamment si elles présentent des structures complexes et irrégulières, dans une base relationnelle n'est pas trivial. Leur interrogation nécessite également des opérations coûteuses pour traduire les requêtes du langage *XQuery* en *SQL*, puis les résultats obtenus sous forme de relation en documents *XML*. Pour qu'une entreprise sache si elle doit mettre en place une base de données *XML* native, elle doit réfléchir à celles, parmi ses sources de données, qui peuvent nécessiter une traduction *XML*. Si elles sont plusieurs sources de données à utiliser des applications *XML*, une base de données *XML* native est intéressante. Par ailleurs, elle permettra de réduire l'influence du traitement *XML* sur les autres applications si vous avez beaucoup de données à traiter. Les bases de données *XML* intègrent des interfaces de programmations d'application (API) que peuvent être utilisées pour intégrer les données *XML* avec les applications.

Les bases de données *XML* natives permettent d'accéder aux sources de données (base de données relationnelle ou données dans le système de fichiers par exemple) en utilisant les méthodes standardisées d'accès aux bases de données, comme Java Database Connectivity *(JDBC)* et Open Database Connectivity (*ODBC*). En clair, pratiquement toutes les bases de données, de *FileMaker*, *MySQL*, et *PostgreSQL* à *Oracle*, *DB2* et *Sybase*, sont accessibles en source ou en cible pour les traductions *XML*. Les bases de données *XML* natives disponibles aujourd'hui en sont encore à leurs débuts. La plupart des utilisateurs qui ont déjà adopté ces technologies travaillent dans les secteurs de la finance ou de la production, et utilisent principalement les bases de données *XML* natives pour accélérer les vitesses de transaction.

Chapitre 6

Présentation de l'application

6.1 L'environnement de travail

Avant toute chose il a fallu construire un *Data Warehouse* simple pour l'utiliser comme environnement de travail. Pour ce faire, le choix s'est porté sur une entreprise américaine qui détient un ensemble de centres de vente dans quatre états différents : Californie, Floride, Oregon et Louisiane. L'activité principale de cette entreprise consiste en l'achat d'articles (généralement des boissons) de ses différents fournisseurs et de les revendre sur le marché américain.

Voici un schéma représentatif du système de stockage des sources et du Data Warehouse. Les carrés sont des collections, et les cercles des documents *XML*, les collections « Ventes », « ProdFournisseur », « Fournisseur » et « Etats » contiennent eux aussi des documents *XML*, leur contenu est détaillé plus bas.

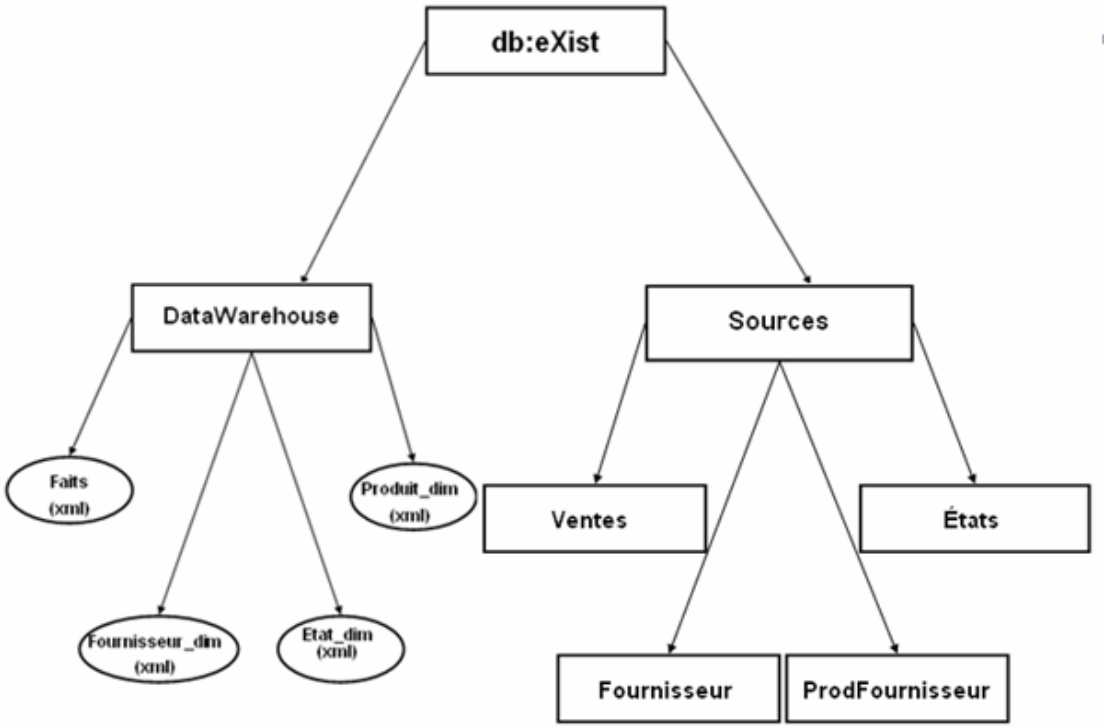

FIG. 6.1: Schéma du système de stockage : les carrés représentent des collections, les cercles des documents *XML*

6.1.1 Les sources

Les sources sont représentées par quatre collections distinctes : « Fournisseurs », « ProdFournisseurs », « Ventes » et « Etats ». Chaque collection contient un ou plusieurs documents *XML* dans lesquels on peut retrouver toutes les informations nécessaires à l'alimentation du *Data Warehouse*.

6.1.1.1 Fournisseurs

Cette collection contient des documents *XML* renfermant chacun les informations correspondant aux fournisseurs collaborant avec l'entreprise (*Cool Canadian*, *High Tech Drinks*, *Long Ridge Rd*). Dans chaque document on peut accéder au nom du fournisseur et son adresse (rue, numéro, zip, ville, état et pays).

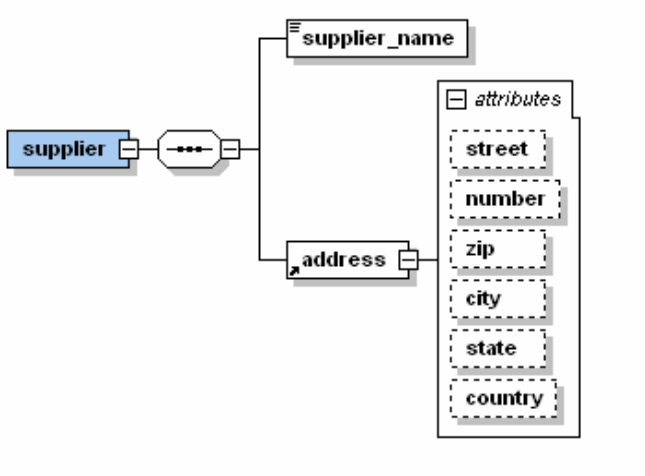

FIG. 6.2: Le schéma *Fournisseur.xsd* du document *Fournisseur.xml*

6.1.1.2 ProduitsFournisseurs

Cette collection contient le document *XML ProdFournisseur.xml* où sont listés tous les produits disponibles en vente ainsi que le nom du fournisseur qui les procure. Les produits sont identifiés grâce à l'attribut unique « id », à leur code, l'attribut booléen « caffeinated », leur type d'empaquetage et leur date d'introduction en vente.

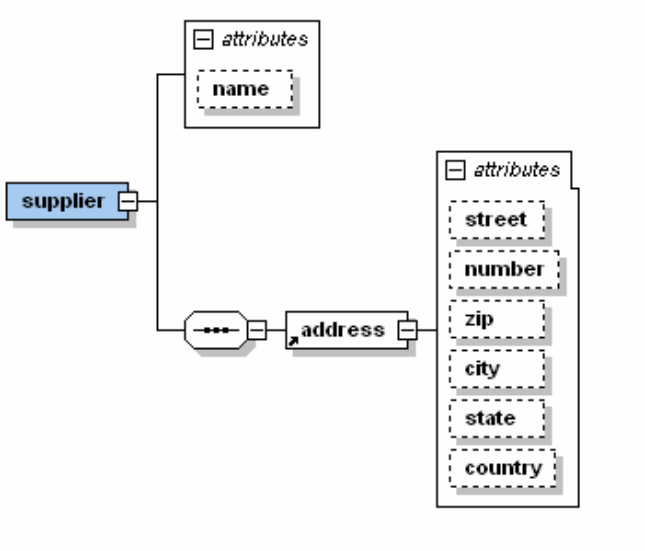

FIG. 6.3: Le schéma *ProdFournisseur.xsd* du document *ProdFournisseur.xml*

6.1.1.3 Ventes

Dans cette collection se trouvent les documents *XML* contenant une liste d'informations détaillées sur les ventes de chaque état. Dans l'élément « sale » sont répertoriés le nom qui est l'identifiant du produit vendu (l'attribut « sku_id »), l'attribut temps, et le total des ventes.

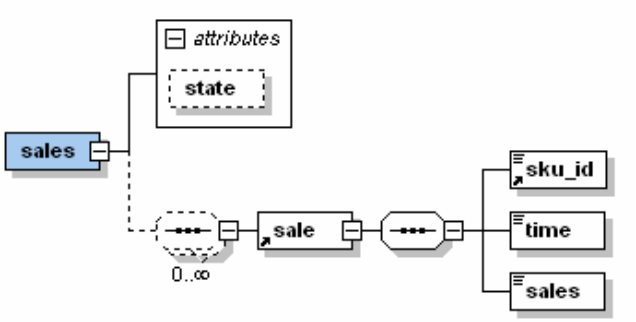

FIG. 6.4: Le schéma *VentesEtats.xsd* du document *VentesEtats.xml*

6.1.1.4 Etats

Cette collection contient des données supplémentaires sur les centres de vente, à savoir : La région, le directeur de région, la taille et le type du marché.

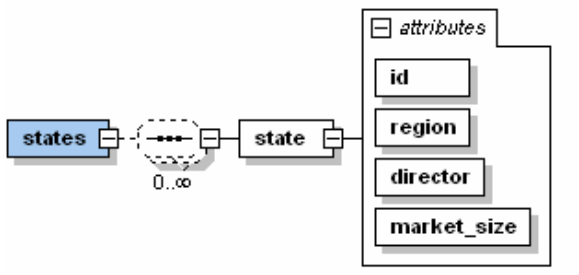

FIG. 6.5: Le schéma *Etats.xsd* du document *Etats.xml*

6.1.2 Le *Data Warehouse*

L'objectif majeur d'un système décisionnel est l'analyse de la performance. On mesure cette performance au travers des indicateurs capables d'orienter la prise de décision. Ces indicateurs seront donc la base de la modélisation dimensionnelle et vont être regroupés dans la table de faits.

Dans ce travail, au vu des sources disponibles, le seul sujet d'analyse susceptible d'être pris en compte par un système décisionnel est sans contexte les ventes de chaque état. On pourrait par exemple, mesurer le total des ventes de chaque état sur une durée de temps donnée et les comparer afin de décider de quel marché il serait bénéfique d'augmenter l'activité. Viendront s'ajouter à ce fait quatre dimensions : Etat, Fournisseurs, Produits et temps.

Il est indispensable à ce stade de définir dans le *Data Warehouse* un schéma standard pour chaque partie de ce dernier, qu'il fasse partie de la collection faits ou de la collection dimensions. Pour se faire on va inclure un schéma *XML* avec chaque document *XML*.

N'importe quelle modification, suppression ou ajout ne pourra être enregistrée dans le *Data Warehouse* que si elle n'altère pas la validité du document par rapport au schéma *XML* qu'on lui aura assigné. Ce test de validité permettra non seulement de contrôler la validité des flux entrant, mais en plus d'homogénéiser les connaissances qui se trouvent dans le *Data Warehouse* en un format standard et donc faciliter leur exploitation par un système décisionnel.

6.1.2.1 Modélisation

Pour représenter le *Data Warehouse*, on se basera sur le modèle multidimensionnel, plus précisément le modèle dit en étoile. Le cube sera composé du fait vente, et de quatre dimensions état, Fournisseurs, Produits et temps (mois). Cela revient à dire que pour chaque combinaison de ces quatre dimensions ont peut accéder à la mesure numérique (si elle n'est pas nulle) du fait.

Par exemple, la combinaison *{"California", "Cool Canadian", "Diet Cream", "Jan"}* découlera sur la valeur numérique : *11700*. La combinaison *{"California", "Cool Canadian", "Diet Cream", "Jan"}* est quant à elle vide !

Ci-dessous le schéma représentant le modèle du *Data Warehouse* :

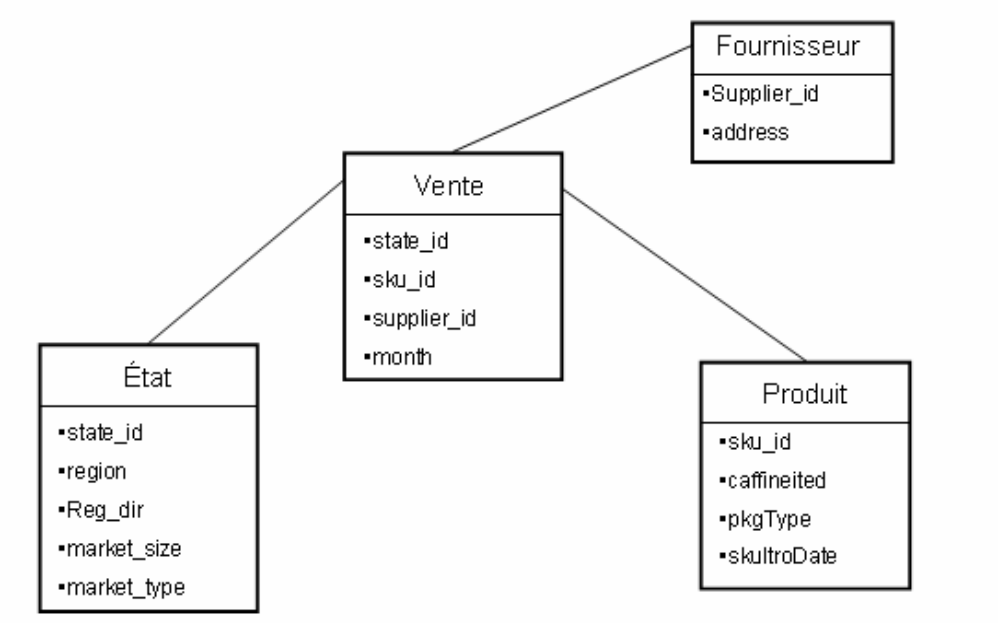

FIG. 6.6: Schéma en étoile du Data Warehouse

6.1.2.2 Le fait vente

Ce document *XML* se compose de Faits *(« facts »)*. Chaque fait est une séquence des éléments suivants :

- « state_id » : l'identifiant de la dimension état ;
- « sku_id » : l'identifiant de la dimension Produit ;
- « supplier_id » : l'identifiant de la dimension Fournisseur ;
- « month » : le mois pendant lequel sont effectuées les ventes ;
- « sales » : le total des ventes pour ce mois.

Cela se traduit par le schéma XML suivant :

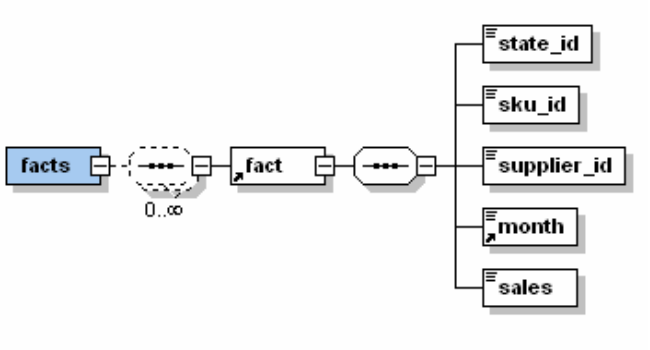

FIG. 6.7: Le schéma *Fait.xsd* du document *Fait.xml*

La dimension « month » est une énumération : *{"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}*.

6.1.2.3 Les dimensions

La dimension Etat

La dimension Etat est un document *XML* qui contient toutes les informations concernant les états où se trouvent les centres de vente :

- Le nom de l'état : « state_id », cet élément est du type *"State_id"* qui est lui-même une énumération reprenant les noms des états où se trouve les centres de vente de l'entreprise *{"California", "Florida", "Oregon", "Louisiana"}* ;
- La région où se trouve l'état : « region », il est du type *"Region"*. Ce type énuméré ne peut prendre que les valeurs suivantes : *{"Central", "East", "West", "South"}* ;
- Le directeur de la région : « reg_dir », restreint lui aussi à l'énumération suivantes : *{"Cindy Traveller", " John West ", " Eric Leaf ", " Michael Sun"}* ;
- La taille du marché : « market_size », qui peut être soit un *"Major_Market"*, soit un *"Small_Market"* ;

Ci-dessous le schéma state_dim.xsd qui reprend tous ces éléments :

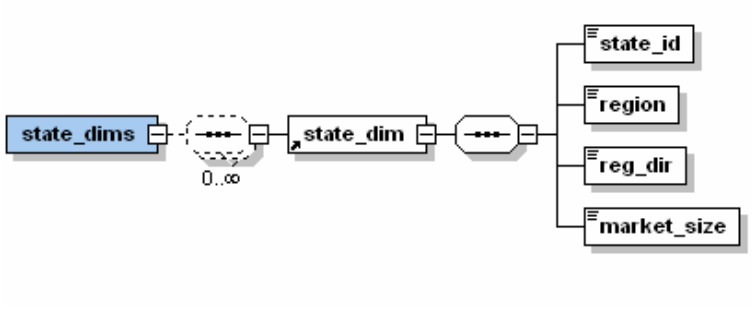

FIG. 6.8: Le schéma *Etat_ dim.xsd* du document *Etat_ dim.xml*

La dimension Fournisseur

Cette dimension reprend les données correspondant à chaque fournisseur, à savoir le nom et l'adresse de l'entreprise. L'élément adresse est un type complexe, il est représenté par la séquence suivante : *{"street", "number", "zip", "city", "state", "country"}*.

Cette façon de faire pourrait être considérée comme une manière simple de représenter le modèle multidimensionnel *snow flake* dans un *Data Warehouse XML* native. Car effectivement cela reviendrait à décomposer la dimension Fournisseur en deux dimensions : « adresse » et « Fournisseur », permettant ainsi certaine hiérarchisation au sein des dimensions.

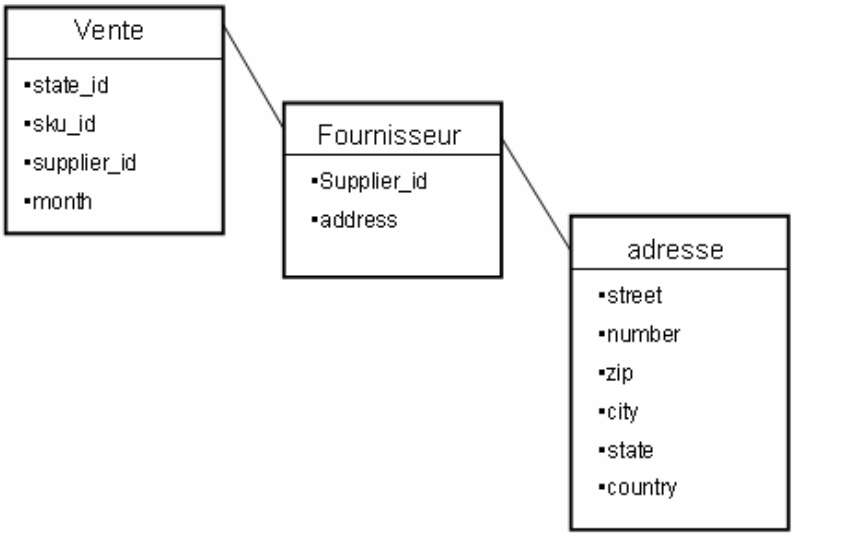

FIG. 6.9: Hiérarchisation de la dimension Fournisseur

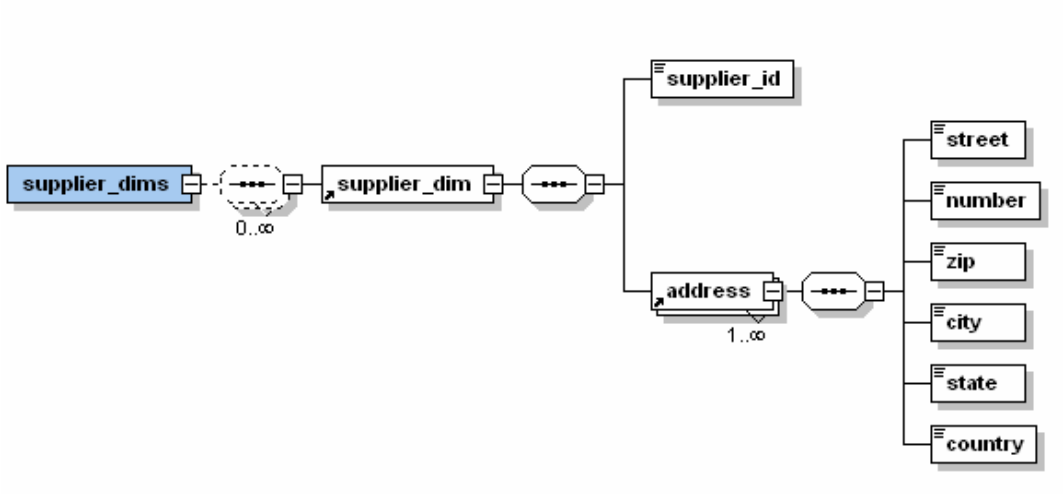

FIG. 6.10: : Le schéma *Fournisseur_ dim.xsd* du document *Fournisseur_ dim.xml*

La dimension Produit

Chaque dimension Produit : « sku_dim » est une séquence composée des éléments suivants :

- Identifiant du produit : « sku_id » de type *"Sku_ id"*, une énumération des produits disponibles dans les centres de vente :*{"Diet Cola", "Dark Cream", "Diet Cream", "Vanilla Cream", "Grape", "Orange"}* ;
- Le code du produit : « sku_code ». Ce mnémonique ne prend que des valeurs du type *"200-30"*. Il a fallu donc restreindre le type générique *"xs :string"* en utilisant la variable *"xs :pattern"* et en lui assignant la valeur : *"[1-4] + 00 + [1-4] + 0"* ;
- Le booléen « caffeinateited » ;
- Le type de l'empaquetage : « pkgType », restreint aux valeurs suivantes : *{"Can", "Bottel"}* ;
- La date de l'introduction de l'article en vente : « skuItroDate ».

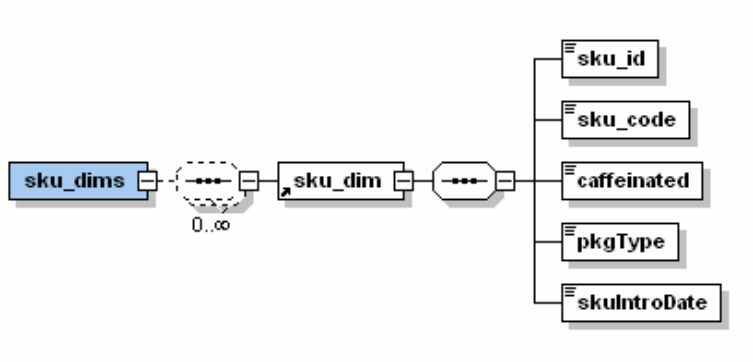

FIG. 6.11: Le schéma *Produit_ dim.xsd* du document *Produit_ dim.xml*

Chapitre 7

Implémentation de l'outil *ETL*

7.1 Introduction à *eXist*

La meilleur façon d'utiliser *eXist* dans une application java est de travailler avec l'*API* (*Application Programming Interface*) *XML :DB*. Cette application procure une interface commune pour les bases de données *XML* natives et supporte le développement d'applications portables et réutilisables.

Les composants basiques employés par *XML :DB* sont les drivers, collections, resources et services.

- Les drivers sont des implémentations de l'interface de la base de données qui encapsulent la logique des accès pour des produits *XML* spécifiques dans la base de données. Ils doivent être enregistrés avec le gestionnaire de la base de données ;
- Une collection est un contenant hiérarchique des resources et d'autres sous-collections. Deux types de ressources sont définis par cette *API* : *XMLResource* et *BinaryResource*. Une *XMLResource* représente un document *XML* ou un fragment de document, résultant de l'exécution d'une requête (*XPath, XQuery, XSLT*) ;
- Les services sont requis pour l'exécution de tâches spéciales comme le questionnement d'une collection avec l'outil *XPath* ou encore la gestion d'une collection. *eXist* offre une large palette de services, les plus importants pour ce mémoire sont le *XQueryService*, *XPathQueryService* et *CollectionManagementService*.

7.1.1 Questionnement de la base de données

Deux manières pour le traitement de cette tâche, on a le choix entre le standard *XPathQueryServise* ou la classe *XQueryService* implémentée par *eXist*. Les services de L'*API XML :DB* ne sont pas tous supportés par *eXist*. La méthode *getService ( )* de la classe *Collection* appelle l'un des services disponible. C'est dans la collection qui appelle le service où seront exécutées les différentes requêtes.

Pour exécuter une *XQuery* le service appelé est *XQueryService*. Ensuite par le biais de la méthode *query* (*xpath expression*) qui retourne un *ResourceSet*, chaque ressource est un fragment de document *XML* sélectionné par l'expression *XPath*. Vu de l'intérieur, *eXist* traite les expressions *XQuery* et *XPath* de la même manière. De ce fait *XQueryService* et *XPathQueryService* sont tous les deux implémentés dans la même classe. Néanmoins *XQueryService* est plus intéressant car, une fois compilées, les requêtes *XQuery* peuvent être représentées par des documents *XML*, qui peuvent à leur tour être stockés et réutilisés.

Le serveur *XML-RPC* enregistre automatiquement les expressions compilées.

7.1.2 Accès aux collections

Le driver utilisé dans *eXist* est *org.exist.xmldb.DatabaseImpl*, il doit être enregistré avec le *DatabaseManager*.

On extrait une collection à partir de la base de donnée en appelant la méthode statique *DatabaseManager.getCollection ( )*. Pour identifier la collection désirée, il faut lui passer comme paramètre l'*URI* complet. Le format de ce dernier doit ressembler à ceci : *xmldb : [DATABASE-ID] :// [HOST-ADDRESS] / db/ collection*. Du fait que plusieurs drivers peuvent être enregistrés avec le gestionnaire de la base de données, la première partie de l'URI (*xmldb : exist*) est requise pour déterminer quelle classe de driver sera utilisée.

Le *database-id* est utilisé par le gestionnaire de la base de donnée pour sélectionner le bon driver dans la liste des drivers disponibles. Pour utiliser *eXist*, cet *ID* doit toujours être égale à *« exist »* (sauf si l'application traite avec de multiples instances de bases de données, auquel cas *ID* pourra aussi prendre le nom des autres bases de données).

La dernière partie de l'*URI* est réservée au chemin vers la collection, éventuellement l'adresse *host* d'un serveur de base de donnée sur le réseau. Dans le cas de ce mémoire, l'application est interne (pas besoin d'extraire les données sur le web), on utilisera donc l'un des deux drivers internes d'*eXist* : Le premier s'adresse à un moteur de base de données distant utilisant des appels *XML-RPC,* le second a un accès direct aux instances locale d'*eXist*.

La collection de base est toujours identifiée par */db*. Par exemple, l'*URI xmldb : exist : // localhost : 8080/ exist/ xmlrpc/ db/ DataWarehouse* fait référence à la collection *DataWarehouse*, à l'aide d'un serveur distant utilisant l'interface *XML-RPC* comme servelet au *localhost : 8080/ exist/ xmlrpc*.

Si on omet de préciser l'adresse du host, le driver de *XML :DB* va essayer de se connecter à une instance de base de données attachée locale. Par exemple, *xmldb : exist : / / / db/ / DataWarehouse.* Pour ce cas de figure, le driver de *XML :DB* devrait créer une nouvelle instance de base de données, si aucune n'a été démarrée. Pour ce faire, on assigne la valeur *« true »* à la propriété *create_database* dans la classe *Database*, cette propriété et bien d'autres sont initiées grâce aux appels de la méthode *setProperty* qui prend en charge le bon paramétrage de la base de données.

7.1.3 Accès aux ressources

Pour récupérer les ressources (documents *XML*), on passe comme paramètre l'*Id* unique de ce document à la méthode *getRessources ( )* de l'interface *Collection*. Ensuite on peut appeler la méthode *getContent ( )*, qui retourne le contenu de la ressource, dans le cas des *XMLResources* (documents *XML*) cette méthode retourne un *String*.

7.1.4 Enregistrement des documents

Pour enregistrer un nouveau document, il faut d'abord créer une nouvelle *XML-Resource* en appelant la methode *getRessource( )*, charger le contenu du nouveau document à l'aide de la méthode *setContent ( )* et l'enregistrer en appelant la méthode *storeResource ( )* de la classe *Collection*.

7.1.5 *XQuery*

eXist permet de formuler des requêtes via *Xpath2.0/Xquery 1.0*. Il implémente toutes les spécifications du noyau *Xquery* mis à part les spécifications suivantes :

- Les éléments reliés au schéma : validation, schéma importé. Le processeur *Xquery* d'*eXist* ne supporte pas encore l'importation de schémas et leur validation. Les informations sur le type de la valeur d'un nœud ne sont pas enregistrées dans la base de données. On ne peut donc pas connaître le type de la valeur d'unnœud, et on admet que toutes les données ont le même type standard *xs :untypedAtomic*, suivant le comportement défini par les spécification *Xquery* en pareil cas. Ceci dit *eXist* supporte un typage très strict dans une expression où le type est spécifié, comme dans le cas des arguments d'une fonction ou de son retour.
- Types atomiques : *xs :gYear, xs :gMonth*, etc.
- Les axes de *Xpath* : *following*, *preceding*.

7.2 L'implémentation de l'outil *ETL*

Dans cette section, on va définir le travail effectué pour l'élaboration de chacune des trois étapes de l'outil *ETL*, d'abord en terme de requêtes *XQuery*, ensuite de leur intégration dans l'application *Java*, et pour finir, on va présenter les différentes fonctionnalités qu'offrent l'interface graphique.

En terme d'outillage, on va s'appuyer sur :

- La base de données *XML* native *eXist*
- L'environnement de développement *NetBeans*
- L'outil *XMLSpy* d'*Altova* afin de construire les documents et schémas *XML* des sources et du *Data Warehouse*, et de tester les requêtes *XQuery*.

eXist intègre directement les utilitaires permettant de d'accéder aux collections qui renferment les bases de données source et *Data Warehouse*. On n'aura donc pas besoin d'intégrer manuellement des pilotes de type *JDBC* ou *RMI* pour la connexion avec les bases de données.

7.2.1 Extraction

Dans la présente étude, les données traitées sont des données embarquées. On part du principe que toutes les données sources nécessaires à l'alimentation du *Data Warehouse* sont disponibles dans la base de données *eXist* et l'on pourra accéder à cette base de données à partir de l'application Java.

Les données sont hétérogènes dans le sens où le modèle des données sources ne correspond pas forcément au modèle du *Data Warehouse*, le format lui reste unique (document *XML*), comme l'est aussi le système de stockage : aussi bien les sources que le *Data Warehouse* sont enregistrés nativement, l'unité de stockage est un document XML ou un fragment de document XML.

On n'aura donc pas à créer des modules spécifiques pour la gestion des connexions à des serveurs distribués et de technologies différentes. Ni à implémenter des méthodes de conversion de format des différentes données extraites.

L'étape « extraction » de l'outil *ETL* en construction se résumera donc à l'accès local aux documents enregistrés sous le sigle « Sources », les disposer de manière à favoriser leur exploitation pour les étapes suivantes : transformation et chargement.

Toutefois cette application pourra être étendue afin de traiter un système de données distribuées. Pour ce faire, il suffira par exemple, d'ajouter les deux modules suivants :

1. Modules de gestion des accès à distance : Pour la connexion avec les bases de données relationnelles, on pourrait utiliser les drivers *JDBC* (*Java DataBase Connectivity*), cette technologie permet aux applications *Java,* et grâce à une interface commune, d'accéder aux sources de données où sont définis des pilotes *JDBC*. Ces derniers sont disponibles pour la majorité des systèmes connus de bases de données relationnelles. Pour les bases de données *XML* natives, le protocole *SOAP* et les *WebServices* permettent l'envoie des requêtes *XQuery*. Le résultat de ces requêtes ayant la forme de fragments de documents *XML,* il sera parfaitement exploitable par la phase suivante de l'*ETL*.
2. Modules de conversion du format des données : Ce module prendrait en charge la mise sous format *XML* des flux émanant des bases de données relationnelles sources, avant de les introduire dans la phase de transformation. Ceci est possible avec l'utilisation de *SQLXML*, c'est une extension de *SQL*

Server qui permet d'insérer ou d'extraire au format *XML* des données d'une base de données relationnelle *SQL Server*.

L'ajout de ces deux modules aura pour effet d'augmenter la portabilité du projet, et d'étendre son champ d'application au-delà des bases de données *XML* natives.

7.2.2 Transformation

Avant d'insérer les données dans le *Data Warehouse*, il faudra s'assurer qu'ils correspondent bien au modèle interne afin d'en sauvegarder la cohérence, et d'optimiser l'utilisation du *Data Warehouse* par le système décisionnel. Dans ce but, une série de transformations sera mise au point pour chacun des composants du *Data Warehouse*.

La majorité des fonctionnalités à implémenter dans cette partie du mémoire impliqueront un questionnement répétitif de la bases de données et la modification de la structure interne des documents, il était donc indispensable de bien choisir la technologie *XML* à utiliser pour la gestion des requêtes au risque d'une baisse de la performance finale. On avait le choix entre deux options : *XQuery* et *XSLT*.

7.2.2.1 Choix entre *XQuery* et *XSLT* ?

XQuery et *XSLT* ont plus de similitudes que de différences (c'est justement ce qui rend difficile le choix entre les deux). Les deux prennent comme entrée des documents *XML* et en produisent en sortie. Ils traitent tous les deux aussi bien les documents typés (validés avec un schéma XML) ou non typé et manipulent globalement la validation des documents par rapport à un schéma de la même façon.

Leurs concepts d'utilisation (essentiellement *XPath*) très similaires et le fait qu'ils partagent pratiquement la même bibliothèque de fonctions font en sorte qu'une fois qu'on a maîtrisé l'un, l'initiation à l'autre langage s'en trouve grandement facilitée.

Néanmoins ils subsistent quelques différences. En effet, contrairement à *XQuery* qu'on peut considérer comme un nouveau venu, *XSLT* est sur le marché depuis plusieurs années, il a acquis plus de maturité, et offre plus de fonctionnalités ;

la plus évidente est certainement les « *Template rules* » dont on ne trouve pas d'équivalent dans XQuery. L'un des principaux avantages à utiliser cette fonctionnalité dans un code est de le rendre beaucoup moins sensible aux éventuelles modifications de structure du document traité. Un « *Template rules* » dicte les transformations auxquelles il faut procéder sur un élément <*item*>, par exemple, indépendamment de sa position dans le document. L'inclusion de cette fonctionnalité et bien d'autres facilités dans *XSLT* reflète le but premier de sa création, à savoir la production de document *XML* à usage humain.

XQuery a quant à lui introduit l'expression *FLWOR*[1], qui même si on peut la traduire en *XSLT* (sauf quelques exceptions) facilite énormément l'écriture de requêtes complexes impliquant plusieurs joins sur différents ensembles de nœuds.

En conclusion, *XSLT* est conçu pour le traitement de volumes modestes de *XML* orienté document (semi-structuré), tandis que *XQuery* est désigné pour la gestion de grands volumes de *XML* orienté données (structuré à l'avance).

Si le but d'une application est de retourner des données pour l'usage humain, il serait plus judicieux de choisir *XSLT*. Si par contre il s'agit de gérer des requêtes sur une base de données *XML*, le choix se portera sans contexte sur *XQuery*.

Et c'est bien ce dernier point qui justifie le choix de *XQuery* plutôt que *XSLT* dans l'élaboration de ce mémoire.

7.2.2.2 Transformations au niveaux de *XQuery*

Transformations liées aux dimensions

La démarche est pratiquement la même pour les trois dimensions. En effet, Pour chaque dimension, On a besoin d'accéder à deux ressources, d'une part le document *XML* correspondant à la dimension en question, et d'autre part la collection source où on puisera les données à transformer.

L'objectif de cette étape étant de faire correspondre les données de la collection sources au schéma *XML* du document où est enregistrée la dimension, On va créer une requête *XQuery* qui va se charger d'aller extraire dans chaque ressource de la collection sources, les valeur des tags qui entrent dans la construction de la

[1] *for-let-where-order by-return*

dimension, ces valeurs seront ensuite enregistrées dans la variable *$new* dans le format de destination.

A chaque fois qu'une dimension est remplie, on vérifie si elle existe déjà dans le *Data Warehouse* avant de l'insérer au document.

Pour illustrer cette transformation, on reprendra en détails les modifications apportées aux données de la collection « Fournisseur » qui intègre le schéma *XML Fournisseur.xsd* (figure 6.2), dans le but de créer de nouvelles dimensions enregistrables (valables par rapport au schéma *Fournisseur_ dim.xsd* de la figure 6.10) dans le document *XML Fournisseur_ dim.xml* (figure 6.10) .

insertSupplier.xq (figure 7.1) est une requête qui utilise l'expression *FLWOR* (*for-let-where-order by-return*) pour sélectionner tous les sous éléments « supplier » de la collection source, et en extraire les valeurs des tags <*supplier_ id*> et <*address*> de la collection « *Fournisseur* » grâce à la fonction *data ( )* qui retourne les données textuelles enfermées dans le nœud qu'on lui passe en paramètre. Les données récupérées seront ensuite utilisées pour remplir les éléments de la variable de stockage intermédiaire *$new*, et générer une structure *XML* respectant le schéma *XML* imposé dans la ressource de destination.

A chaque itération, la variable *$new* complètement reconstruite représente une nouvelle rangée de valeurs parfaitement intégrable dans le document *XML* correspondant à la dimension en question (l'équivalent d'une ligne de la table de dimension Fournisseur dans un *Data Warehouse* relationnel). Avant d'insérer *$new* dans le document *Fournisseur_ dim.xml,* on vérifie que la combinaison des valeurs qu'elle représente n'est pas enregistrée précédemment dans le *Data Warehouse*, auquel cas cette insertion serait inutile, est équivaudrait à une perte de temps. Ce test est effectué grâce à l'instruction conditionnelle *« if »* qui se charge de vérifier si aucune des instances présentes dans le document *Fournisseur_ dim.xml* n'a un *id* égal à celui de *$new*. Si c'est le cas, *$new* sera insérée dans le document.

Pour insérer *$new* dans le document, on a recours à l'instruction « *insert* » qui fait partie des instructions *XUpdate*[2]. Celle-ci insère le contenu de la séquence *$new* à la fin du document *Fournisseur_ dim.xml*.

Le schéma ci-dessus montre le fragment résultant de la transformation du document source concernant le fournisseur *" Cool Canadian "*. Par exemple l'élément

[2]une extension de *XQuery* procurée par eXist

<*address*> n'a plus d'attribut *"street"*,*"number"*, ... Ils sont devenus maintenant une suite d'éléments imbriqués.

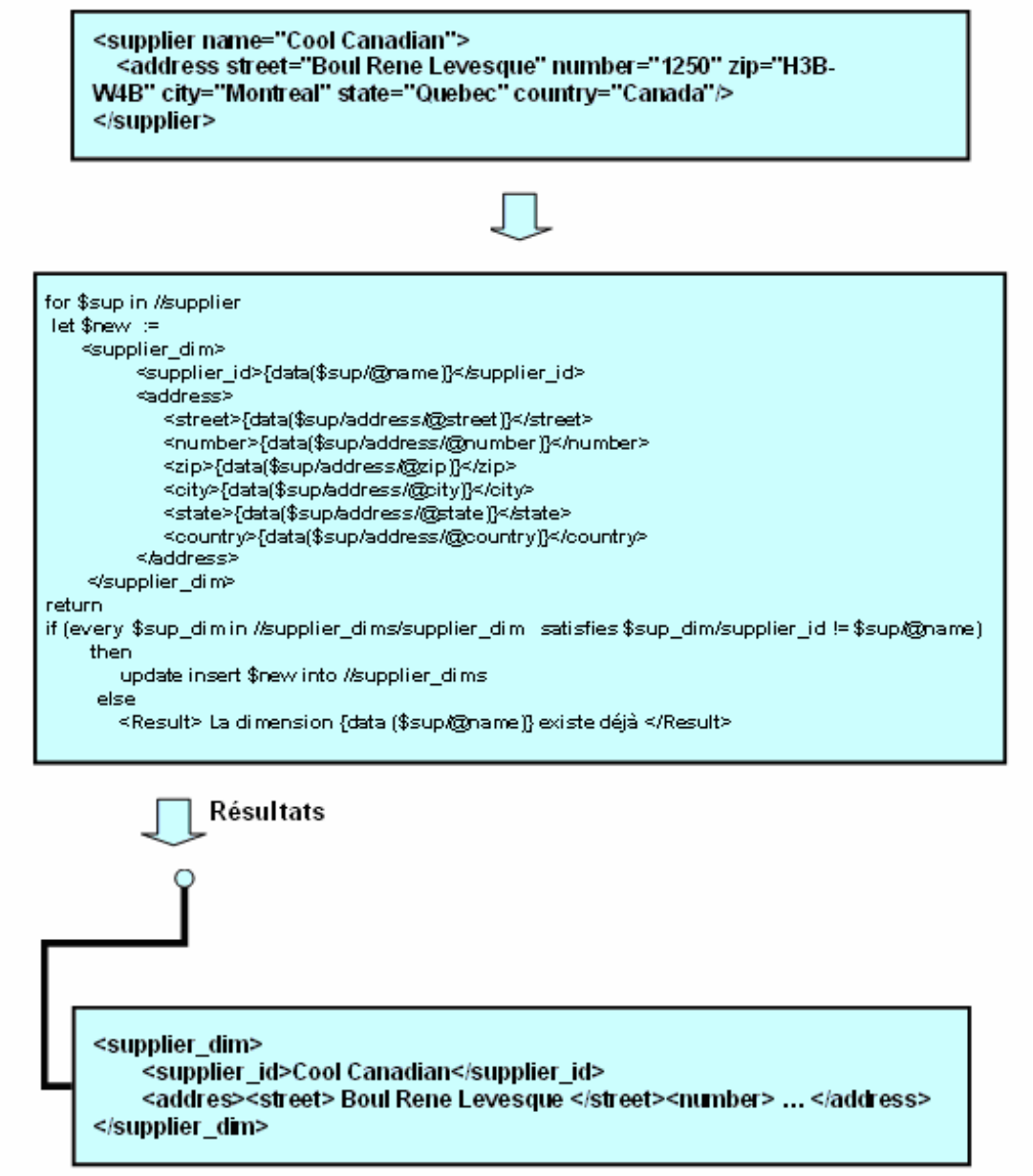

FIG. 7.1: La requête XQuery "insertSupplier.xq"

Transformations liées aux Faits

Cette transformation est plus délicate que les transformations liées aux dimensions, pour deux raisons :

- Elle utilise des données provenant de deux sources différentes « ProdFournisseur » et « Ventes ». La requête *« insertFact.xq »* (figure 7.2) qui gère cette transformation doit établir un joint entre les deux ressources.
- Dans la collection « Ventes » est enregistré le total des ventes par mois de chaque état. Le mois en question est représenté par une date, par exemple, la date *(2000-01-31)* équivaut au mois de Janvier. Il faudrait pouvoir extraire le mois correspondant à chacune de ces dates pour pouvoir le mettre dans l'élément *<month>* du document *Fait.xml* (figure 6.7). Il existe bien dans le noyau *XQuery 1.0* des fonction *getMonth ( )*, *getday ( )*, ..., qui traitent ce genre de requêtes. Malheureusement, elles ne sont pas supportées par le *XQueryService* d'*eXist*, il faut donc définir une fonction *month ( )* locale à la requête *« insertFact.xq »* qui se chargera de récupérer le mois à partir de la date.

month (string) : Cette méthode reçoit comme paramètre une date et retourne un nœud *<month>*. Elle utilise pour cela la fonction *tokenize (string, string)* qui retourne une séquence de string dont les valeurs sont les parties du premier argument séparées par les parties qui correspondent au second string. *tokenize ("2000-01-31", "-")* par exemple, retournera la séquence suivante *{2000, 01, 31}*. Récupérer le mois reviendra à récupérer le second élément de cette liste et l'insérer dans le nœud *<month>*. Ce dernier sera utilisé par la fonction locale

insert (string) : Pour chaque vente dont l'état correspond au string passé en paramètre, cette fonction établie un joint avec la collection « ProductFamily » basé sur l'identifient *Product_ Id*. L'étape suivante consistera à extraire les valeurs des éléments qui entrent dans la construction d'un Fait *vente (state_ id, sku_ id, supplier_ id, time, sales)*. ces valeurs seront ensuite stockées dans la variable *$new* conformément au schéma *faits.xsd* (figure 6.7). L'élément *<time>* sera passé en paramètre à la fonction local *month ( )*, afin d'en extraire le mois correspondant. A chaque fois que la variable *$new* est construite, elle ne sera insérée dans le document *faits.xml* (figure 6.7) qu'après avoir vérifier que cette combinaison de dimension n'existe pas dans le document.

Enfin une expression *FLOWR (for-let-where-order by-return)* fera appel à la fonction locale *insert ( )*, en lui passant à chaque fois le nom d'un état en paramètre, cela servira à grouper les ventes de chaque mois par état. Cet ordonnancement aura comme conséquence d'augmenter la vitesse des éventuelles requêtes, telles que le total des ventes pour le produit X dans l'état Y.

Dans ce schéma on trouve deux fragments de documents *XML*, extraits de la collection *« Sources »*, et la façon dont ils ont été transformés pour construire le fait vente.

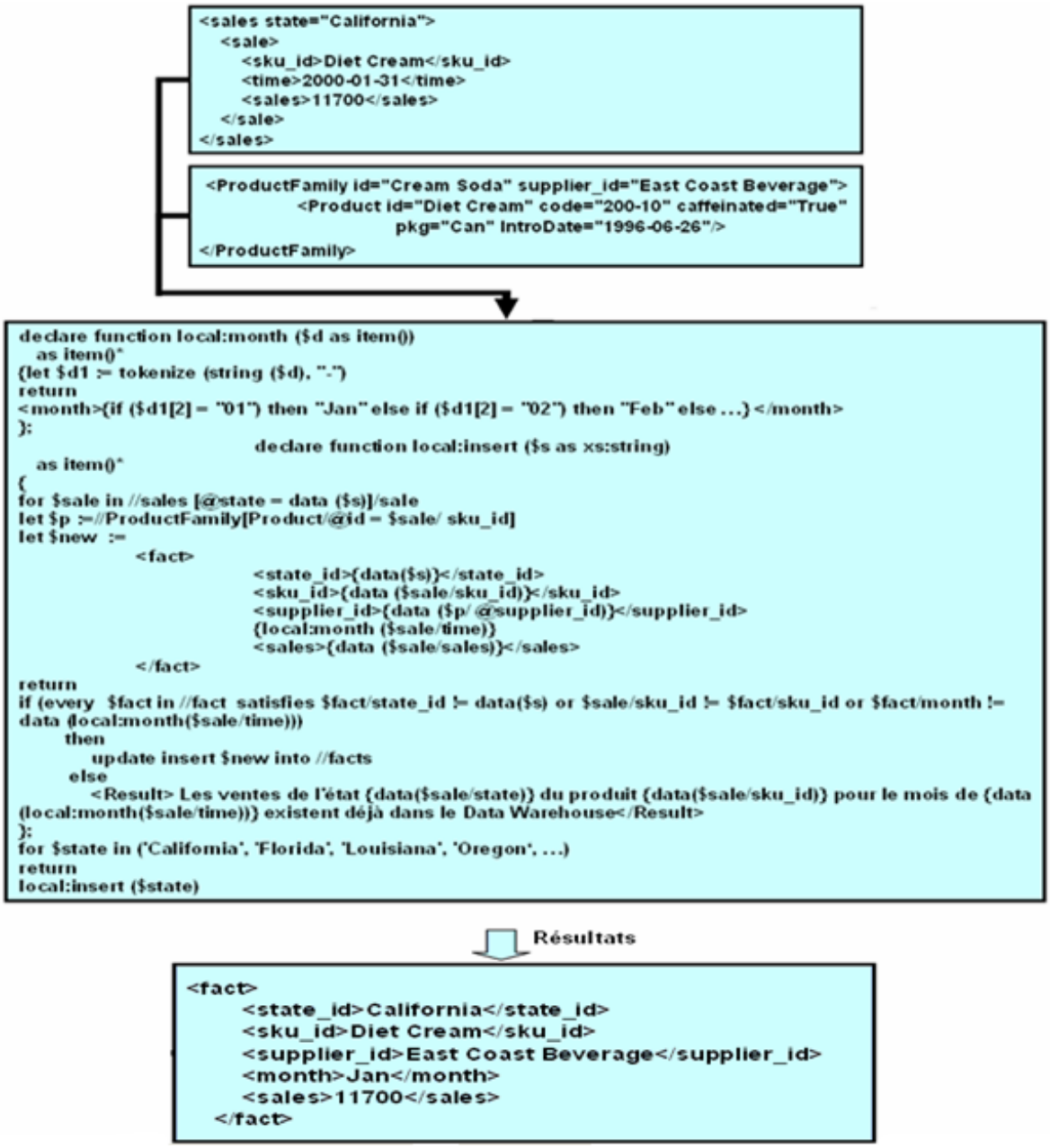

FIG. 7.2: Transformations liées au remplissage du document *Fait.xml* (*insertFact.xq*)

Mise à jour des dimensions

Une autre des missions de loutil *ETL* est de mettre à jour des dimensions enregistrées dans le *Data Warehouse*. On va essayer dans le présent paragraphe de créer des requêtes *XQuery*, qui répondent à ce besoin. Là aussi la démarche est similaire

pour toutes les dimensions, nous allons donc revoir en détail la mise à jour d'une seule des dimensions en sachant que le même processus peut être étendu aux autres dimensions.

La requête *updateAllSku.xq* (figure 7.3) prend en charge la tâche de mise à jour de la dimension Produit, autrement dit elle transpose les changements survenus dans la collection source *« Produit »* au document *Produit_ dim.xml* (figure 6.11). Cette *XQuery* va extraire toutes les valeurs texte des nœuds *<Product>* et les enregistrer dans une variable *$new* suivant la structure imposée par le schéma *XML* du document contenant la destination. A chaque itération, on procède à la mise à jour effective de la dimension en remplaçant la partie du document concernée (qui a le même *sku_ id* que *$new*) par *$new*, pour ce faire on utilise l'instruction *« insert »* qui fait partie des instructions *XUpdate*[3].

Cette requête est très élémentaire, elle remplace les valeurs texte de tous les nœuds *<sku_ dim>* du document *produit_ dim.xml* par les nouvelles données récupérées dans *$new* à partir de la source, sans se soucier de vérifier s'il y a vraiment lieu de procéder à une mise à jour.

Une version améliorée de cette méthode serait qu'à chaque itération, une suite de tests puisse filtrer les dimensions qui ont réellement besoin d'être mises à jour. Dans cette optique, on a créé la requête *updateRestrictSku.xq* qui ne remplace que les parties du document *produit_ dim.xml* qui sont en désaccord avec les données sources.

La moyenne de temps de réponse des deux requêtes est pratiquement la même, une explication plausible serait, que le temps que prend la première requête à remplacer toutes les dimensions serait compensé par le temps que prend la seconde dans l'exécution de la série de tests à chaque itération. Une autre explication serait due à la petite taille des données traitées. Il faudrait sans doute tester ces deux requêtes sur un plus grand volume de données pour avoir une comparaison plus précise des performances des deux requêtes.

Ci-dessous un exemple des résultats de la requête *updateAllSku.xq*. les nœuds *sku_ code*, *caffeinated*, *pkgType* et *skuIntroDate* selon les données du nœud *Product* correspondant dans la collection source.

[3] une extension de XQuery procurée par eXist

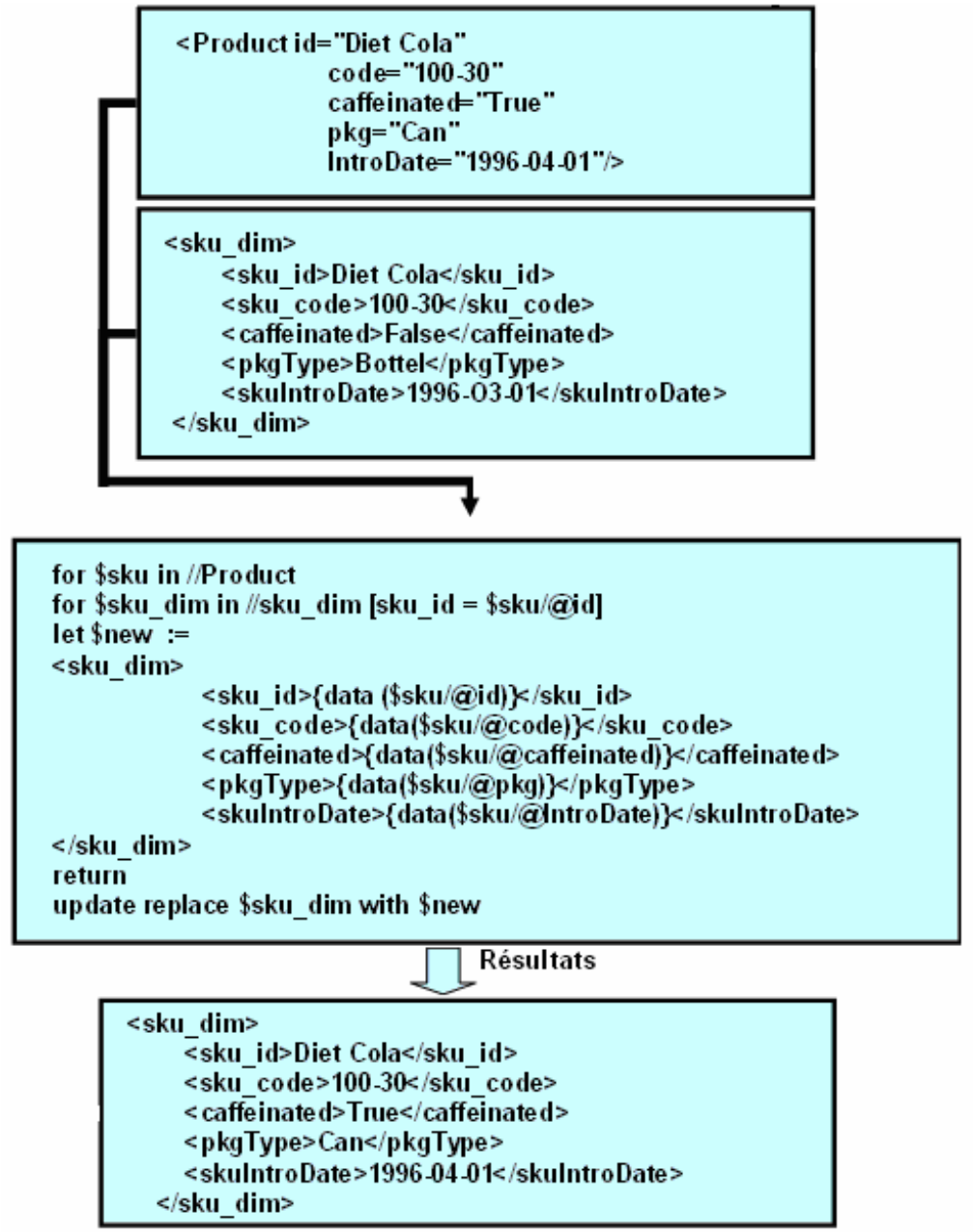

FIG. 7.3: La mise à jour du document *Produit_ dim.xml (updateAllSku.xq)*

7.2.2.3 Au niveau de l'application Java

Les différentes requêtes *XQuery* qu'on a définis précédemment sont enregistrées dans un répertoire commun.

La méthode *query ( )*

Pour exécuter les requêtes à partir de l'application *Java*, on a créé la méthode *query (String query)*. Cette dernière reçoit comme paramètre le chemin vers le fichier où se trouve la requête. Elle suit les étapes suivantes :

- Grâce à la méthode *getCollection ("xmldb :exist :///db", "admin", "")* de l'interface *DatabaseManager*, elle accède à la collection *« db »* où sont enregistrée les deux collections *« DataWarehouse »* et *« Sources »*. Ces deux collections renferment l'ensemble des ressources dont on aura besoin pour mener à bien l'exécution de toutes les requêtes.
- A partir de cette collection, elle appelle la méthode *getService ( )*. cette méthode permet d'accéder au différents services intégrés par *eXist*. Dans le cas du traitement des requêtes *XQuery,* on aura besoin du service *XQueryService*, grâce auquel on peut accéder à toutes les fonctionnalités offertes par le noyau *XQuery "1.0"* et qui sont implémentées par *eXist*. C'est dans la collection qui appelle le service où seront exécutées les différentes requêtes.
- L'étape suivante consiste à extraire la requête à partir du fichier où elle se trouve. Pour ce faire, on a créé la fonction *readFile (String path)* qui se charge d'aller récupérer le contenu du fichier dont le chemin lui est passé en paramètre et le retourne sous la forme d'un *String*.
- La version *String* de la requête, retournée par *readFile ( )*, sera passée en paramètre à la méthode *compile (String query)* du service *XQueryService* appelé précédemment. Cette méthode va compiler la requête et retourner la version compilée en cas de succès. une fois compilée, la requête est exécutée grâce à la méthode *execute (compiled),* elle aussi disponible à partir du service *XQueryService*.

Cette façon de faire rend l'application *Java* complètement indépendante de l'implémentation interne des différentes requêtes. De ce fait l'implémentation *Java* des trois types de transformations vus plus haut a découlé sur trois méthodes très similaires : *insert_ dimAction ( )*, *insert_ Fact ( )* et *update ( )*. La seule différence réside dans le paramètre *path* (chemins vers les fichiers où sont enregistrées requêtes) passé à la méthode *query ( )*, le code principale ainsi que la gestion des exceptions restent quant à eux inchangés.

7.2.3 Chargement

7.2.3.1 Enregistrement

Les deux instructions *« insert »* et *« replace »* qu'on a utilisées pour l'étape transformation, jouent un grand rôle dans la phase chargement, car même si elles ne font que modifier les documents *XML* sans pour autant les enregistrer dans le Data Warehouse, elles mettent à disposition un document *XML* modifié prêt à être charger dans le *Data Warehouse*. Ce document sera utilisé comme entrée par la méthode *save ( )* qui se chargera de son enregistrement effectif dans le *Data Warehouse*.

La méthode *save ( )* est une méthode privée qui appartient à la clase *ViewDocument*.Dans cette méthode est défini un *thread* qui accède à la collection courante considérée par défaut comme contexte de chargement. A partir de cette dernière est appelée la méthode *storeRessource (Resource res)* définie par *eXist* dans la classe *Collection* en lui passant comme paramètre le document transformé qu'on désire enregistrer. storeRessource se charge ensuite de l'enregistrement de la ressource dans la dite collection. Si la ressource n'existe pas déjà dans la collection, une nouvelle ressource sera créée, sinon la ressource est mise à jour.

Même si dans la phase de transformations on a fait en sorte de générer des fragments de documents *XML* compatibles avec le format des données du *Data Warehouse*, on n'est pas à l'abri d'enregistrer des documents qui ne sont pas conforme au modèle instauré. Pour minimiser au plus la probabilité d'enregistrement de documents incohérents dans le *Data Warehouse*, on va veiller à tester la validité des documents par rapport à une grammaire prédéfinie avant de les charger dans le *Data Warehouse*.

7.2.3.2 Validation des documents

Le mécanisme de validation de documents *XML* par rapport à un document grammatical spécifique *(XML Schema, DTD)* n'a été ajouter que récemment dans *eXist*. Ce module est encore à ces jours en cour de développement, mais il contient d'ors et déjà toutes les bases dont on aura besoin pour pouvoir valider les ressources avant de les charger dans le *Data Warehouse*.

Dans un premier temps, il a fallu choisir une base de validation. Deux option se présentaient : *DTD* ou *XML Schema*, puisque les deux grammaires sont implémentées par *eXist*. Le choix s'est porté sur *XML Schema* parce que : plus complet, typage plus strict donc plus fiable.

On a donc inclus dans chaque document *XML* du *Data Warehouse* l'attribut namespase et l'attribut *xsi :schemaLocation* du schéma *XML* correspondant. La classe *Validator* définie dans *eXist* fait correspondre l'attribut *xsi :schemaLocation* à l'argument *grammarPath* de la méthode public *validate (InputStream is, String grammarPath)*. Le premier argument étant le flux de données du document *XML* à valider. *validate ( )* utilise *SAX (Simple API of XML)* pour parser le document, et profite du fait qu'il soit événementielle pour comparer le document aux normes imposées par le schema *XML* à chaque fois qu'un nouvel événement survient (qu'une nouvelle balise est lu). A la fin, cette méthode retourne un rapport où sont saisie la totalité des exceptions rencontrées lors de la validation du document par rapport au schéma XML.

validate ( ) sera appelée à partir de *store ( )*. De cette façon on pourra sauvegarder au plus la cohérence des données en contrôlant la validité des documents avant de les enregistrer dans le *Data Warehouse*. Quant au rapport d'erreurs retourné par *validate ( )*, il servira plus avant de base dans la gestion des erreurs de chargement dans le *Data Warehouse*.

Chapitre 8

L'interface utilisateur

Cette partie du mémoire, consiste en la création d'une interface graphique qui servira non seulement à utiliser, visualiser et tester les différents modules de l'ETL, mais aussi à interagir avec le *Data Warehouse*. Effectivement on a fait en sorte que cette application soit la plus dynamique possible et qu'elle ne soit pas figée à l'exemple considéré tout au long des étapes précédentes. L'utilisateur pourra entre autre procéder à plusieurs opérations comme l'ajout, la suppression, la modification des ressources, etc.

Cinq classes ont été nécessaires pour la création de ce *GUI* :

1. *Main.java* : Dans cette classe se trouve la méthode principale *main ( )*, c'est là où sont implémentées les principales fonctionnalités du *GUI*, comme la définition de la collection courante, suppression des ressources et la recharge des collections à partir de la base de données *eXist*.
2. *Gui.java* : qui hérite de la classe *JFrame*. C'est là où sont implémentés les différents éléments qui constituent la fenêtre et les événements qui en découlent.
3. *DocumentView.java* : cette classe hérite elle aussi de la classe *JFrame*. Cette fenêtre sera utile pour l'affichage des documents *XML* et leur enregistrement.
4. *ResourceDescriptor.java* : comme son nom l'indique, elle sert à décrire les ressources (collections et documents *XML*). A partir de cette classe, on peut avoir accès au nom, propriétaire et la date de la dernière modification d'une ressource. Ces informations serviront notamment dans le remplissage de la table des ressources qui s'affichera dans la fenêtre principale

5. TextArea.java : héritée de la classe JTextArea, cette dernière à une contribution purement « esthétique », puisque c'est grâce à elle que le contenu des documents est affiché en couleurs dans la fenêtre *DocumentView.java*.

Les pages qui suivent regroupent des captures d'écran, qui permettent de mieux comprendre les différentes fonctionnalités de cette interface utilisateur.

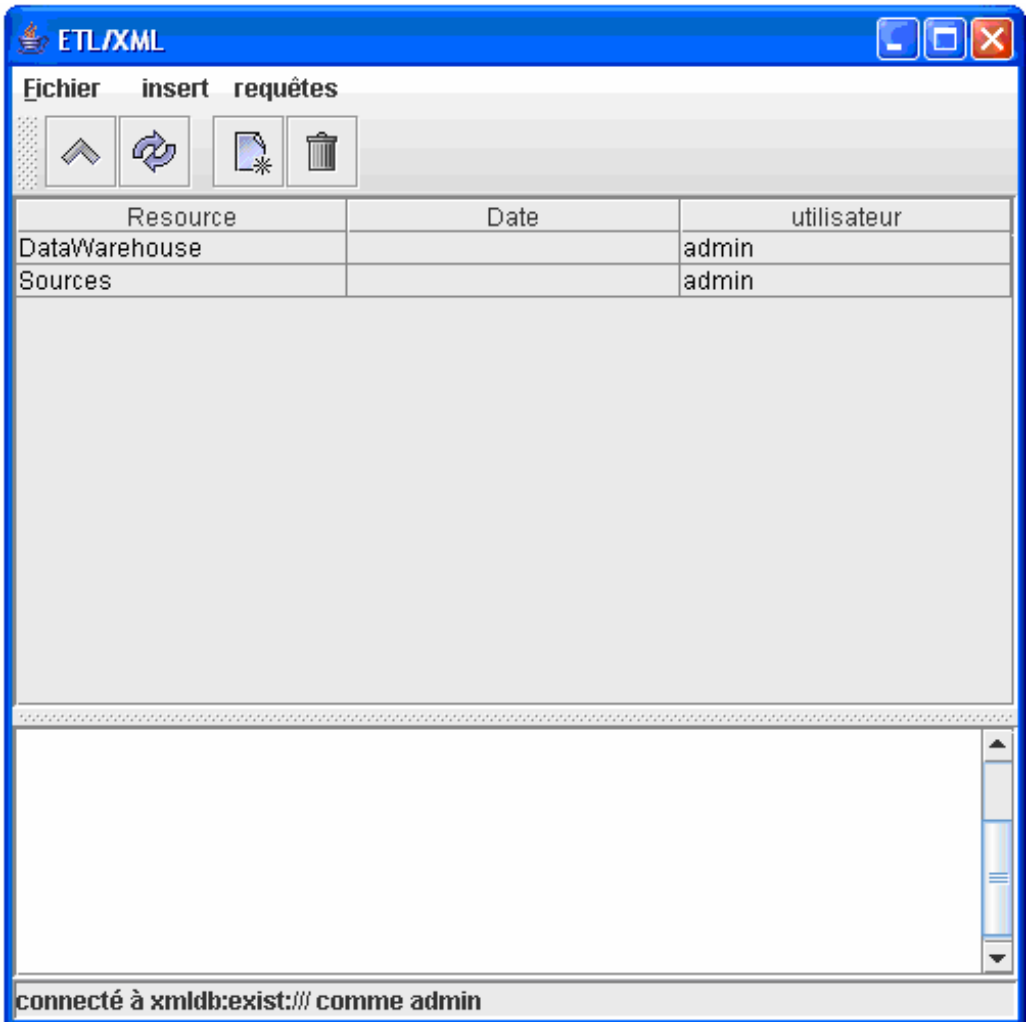

FIG. 8.1: L'interface graphique du projet *ETL/XML* (fenêtre principale)

Affichage du contenu des collections

Pour afficher le contenu d'une collection, il suffit de double cliquer sur la ligne la représentant. La table sera instantanément remplie par les sous-collections ou les

documents *XML* que contient cette dernière. Dans les lignes représentant un document *XML*, on verra s'ajouter la date de la dernière modification de ce document.

Voici par exemple, la fenêtre résultant d'un double click sur la collection *« Sources»* :

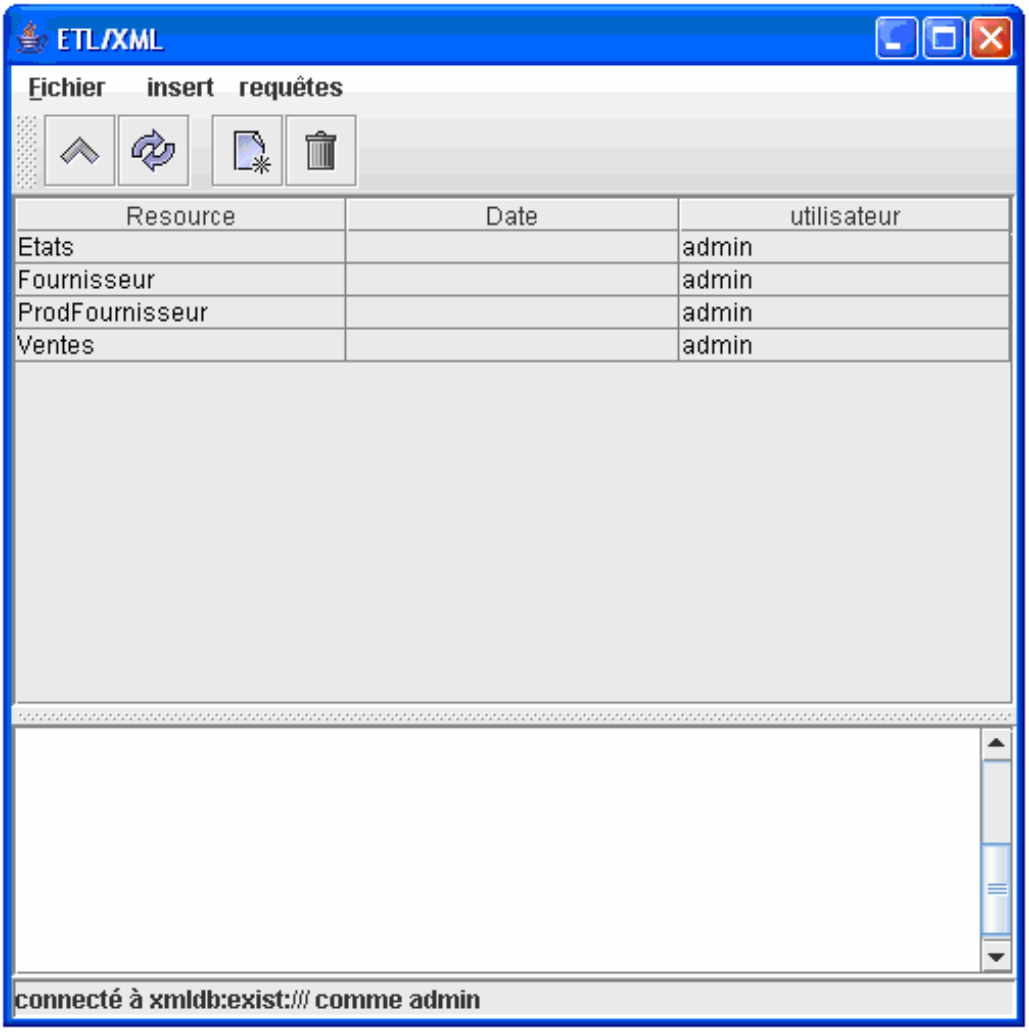

FIG. 8.2: Le contenu de la collection *« Sources »*

Affichage d'une ressource *XML*

Double cliquer sur une ligne représentant un document *XML* déclanche l'affichage du contenu de ce dernier dans une fenêtre séparée implémentée par la collection *DocumentView.java*.

Ci-dessous est la capture d'écran correspondant au document *Fait.xml* (figure 6.7) :

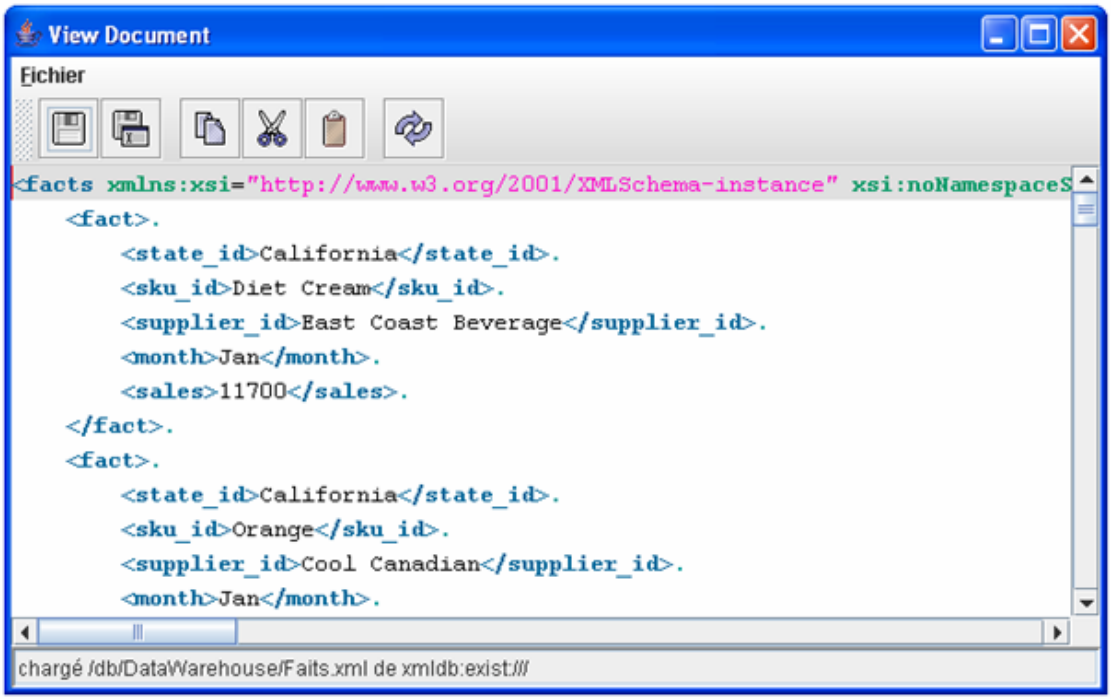

FIG. 8.3: L'affichage du document *Fait.xml*

- Le bouton représenté par une disquette permet l'enregistrement effectif de ce document une fois modifié dans la base de données. Effectivement, cliquer sur ce bouton revient à appelé la méthode *save ( )* mentionnée plus haut dans la phase chargement de l'*ETL*.
- Le bouton représenté par les deux flèches permet quant à lui de recharger le document à partir de la base de données.
- Les autres boutons exécutent respectivement les trois opérations suivantes : copier, couper et coller, sur les parties sélectionnées dans le *JTextArea*.

Suppression d'un ensemble de ressources

Pour supprimer un ensemble de ressources il faut tout d'abord les sélectionner par un simple click et puis sélectionner l'*item* Supprimer qui se trouve dans le menu *« Fichier »*. Il en découle l'appel de la méthode *removeAction ( )* qui selon que la ressource soit une collection où un document *XML* appelle respectivement les méthodes *removeCollection ( )*, *removeRessource ( )*.

L'ajout d'une collection

On peut ajouter une nouvelle collection à la collection courante en cliquant sur l'*item* correspondant à cette tâche, et qui se trouve dans le menu *« Fichier »*.

Sortir d'une collection

Le bouton représenté par la flèche dirigée vers le haut permet de sortir de la collection courante, vers la collection qui la contient.

La mise à jour des dimensions

Cliquer sur le bouton représenté par les deux flèches dans la fenêtre principale, génère l'appel de la méthode *update ( )*. Cette méthode appelle à son tour la méthode *query ( )* on lui passant comme paramètre l'adresse du fichier où se trouve la requête *XQuery* correspondant aux mise à jours. *query ( )* se charge de la compilation et de l'exécution de la requête, il en résulte la mise à jour des document XML représentant les différentes dimensions.

L'insertion des nouvelles données sources dans le *Data Warehouse*

Pour retranscrire les ajouts de nouvelles données de la collection *« Source »* vers la collection *« Data Warehouse »*, il faut cliquer sur le bouton *insert*, et choisir parmi les deux options suivantes : *« Dimensions »*, *« Faits »*. La première option permet d'insérer les nouvelles dimensions dans le *Data Warehouse* en faisant appel à la méthode *insert_dimAction ( )*, de la même manière que pour la méthode *update ( )*. Cette méthode va faire appel à la méthode *query ( )* et lui passer comme paramètre l'adresse du fichier contenant la requête *XQuery insert_dim.xq* vu plus haut dans la phase Transformation. La seconde option sert à insérer de nouveaux faits dans le *Data Warehouse*, elle appelle la méthode *insert_factAction ( )* qui suit les même étapes que *insert_dimAction ( )* en passant comme paramètre à la méthode *query ( )* le chemin du fichier où se trouve la requête *insert_fact.xq*.

Chapitre 9

Conclusion

L'objectif de ce travail est de développer un outil *ETL (Extraction Transformation Load)* pour un entrepôt de données *XML*.

La base de données *XML* native *eXist* a été choisit comme base de départ à ce projet.

Une collection *Data Warehouse*, ainsi qu'une autre collection Source ont été créées pour servir d'environnement de travail.

Des méthodes ont été mises en œuvre afin d'extraire, modifier et insérer les données provenant des sources vers le Data Warehouse. Ces méthodes ont été appuyées par un ensemble de requêtes *XQuery*.

Une interface utilisateur à été mise au point afin d'intégrer les différents modules de l'outil *ETL* dans une même application.

Cette interface a été pensée de manière à ce qu'elle soit la plus dynamique possible, elle permet en effet la création et la suppression de nouvelles ressources, et ces modifications sont visibles au niveau de l'interface graphique.

Notons que les phases qui consistaient à apprendre le fonctionnement des bibliothèques de programmation d'*eXist*, ont pris énormément de temps.

La base de données *eXist* étant en court de développement, plusieurs complications ont été rencontrées lors de l'élaboration de ce mémoire, effectivement il a fallut faire face à toutes les imperfections, principalement dues au Service *XQuery*, et garder un contact permanent avec la mailing liste d'*eXist* pour les résoudre.

L'implémentation des modules *ETL* s'est déroulée en parallèle avec l'étude des bibliothèques de programmation d'*eXist*. Des choix d'implémentation ont donc été faits, et certains se sont révélés décevants une fois l'implémentation terminée. Ce mémoire a été une expérience enrichissante, notamment en nouvelles connaissances. En effet, le développement d'un outil informatique qui entre dans le domaine de la recherche laisse la voie libre à la découverte de nouvelles technologies, tout en permettant de mieux mettre en pratique les connaissances acquises, et de comprendre les exigences qu'implique le fait de travailler sur des plates-formes déjà développées.

Chapitre 10

Bibliographie

[1] Corporation Hyperion Solutions Corporation, XML for Analysis Specification Microsoft

[2] eXist home page, http ://exist.sourceforge.net/

[3] Manipuler des données XML sous Oracle, [http ://www.oracle.com/global/fr/index.html]

[4] XMLSchema [http :xmlfr.org/w3c/XML schema-0/

[5] R. Bourret, C. Bornhovd, A. Buchmann, Darmstadt University of Technology, Allemagne, A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases (2000)

[6] Robin Grosset, The case for XML for Analysis

[7] Igor Dayen, Storing XML in Relational Databases

[8] Mykolz Dudar : XML Schema and web Service for ETL in thr Staging Area of a Scientific Data Warehouse

[9] Maryvonne Miquel, Yvan Bédar, Alexandre Brisebois, Jacynth pouliot, Pierre Marchand, Jean Brodeur, Modeling multidimentional spatio_temporal data warehouse in A Context of evolving specifications

[10] Surajit Chaudhuri, An overview of dta warehousing and OLAP technology

[11] Equipe de recherche en ingenierie des connaissances, PlateFormd ?entreposage XML de données complexes

[12] Torben Bach Pederson, Christian S.Jensen, Multidimensional Database Technology

[13] Juan Trujillo Manuel Palomar, Jaime Gomez, Designing DataWarehouses with OO Conceptual Models

[14] Richard Ewards, Sian Hope, Persistent architecture for XML repositories in relational databases

[15] Angi Voss, Vera Hernandez, Hans Voss, Simon Scheider Fraunhofer AIS, Spatial Decision Support, Sankt Augustin, German, Interactive Visual Exploration of Multidimensional Data :Requirements for CommonGIS with OLAP

[16] Michael Blaha, Data Warehouses and Decision Support Systems

[17] Vera HERNANDEZ, Angi VOSS, Wolf GÖHRING, Germany and Cornelio HOPMANN, Nicaragua, Sustainable Decision Support by the Use of Multi-Level and Multi-Criteria Spatial Analysis on the Nicaragua Development Gateway

[18] Create XML Views of Relational Data Using SQL/XML [http ://www.stylusstudio.com/xml_database.html]

[19] P.Gaspart, Websémantique

[20] P.Gaspart, Technologies XML

[21] Simba Technologies Inc., XML for Analysis (XMLA) InterOperate Workshop

[22] LA PROBLEMATIQUE DE L'ENTREPRISE (page 10)

[23] Conservatoire des arts et métiers de Lille, LE DATA WAREHOUSE, OLAP ET ANALYSE MULTIDIMENSIONNELLE, LE DATA MINING, DIVERS , ETHIQUE, WEB

[24] XPath, XQuery, and XSLT Functions, [http ://www.w3schools.com/xpath/xpath_functions.asp]

[25] Oracle Support, [http ://otn.oracle.com/tech/xml/xmldb/]

[26] Darshan Singh, Essential XQuery - The XML Query Language

[27] Stockage de documents XML dans des bases de données relationnelles, [http ://www.ianywhere.com/developer/ product_manuals/sqlanywhere/0901/fr/html/dbugfr9/dbugfr9.htm]