

CONSERVATOIRE NATIONAL DES ARTS ET METIERS

THESE

pour obtenir le grade de

DOCTEUR DU CNAM

Discipline : Informatique

présentée et soutenue publiquement

par

Laurent Mignet

le 23 Novembre 2001

Titre :

**Contrôle des changements de données
semi-structurées**

Directeurs de thèse :

Serge Abiteboul et Michel Scholl

JURY

M ^{me} .	Geneviève JOMIER	Rapporteur
Mr.	Philippe PUCHERAL	Rapporteur
Mr.	Serge ABITEBOUL	Directeur
Mr.	Jacky AKOKA	Examineur
M ^{me} .	Véronique VIGUIÉ DONZEAU-GOUGE	Examinatrice
Mr.	Alberto MENDELZON	Examineur
Mr.	Michel SCHOLL	Directeur

Remerciements

Je tiens tout particulièrement à remercier mes deux directeurs de thèse Serge Abiteboul et Michel Scholl. Serge pour m'avoir initié aux arcanes de la recherche, entre autres choses, au cours de ces trois années. Michel pour sa disponibilité à tout moment quand j'avais des problèmes métaphysique à propos de ces trois années thèse et de sa patiente lecture, relecture et re-relecture de ce manuscrit afin qu'il devienne une thèse, merci merci, merci.

Geneviève Jomier et Philippe Pucheral m'ont fait l'honneur de rapporter cette thèse, merci à eux. Pour leurs participations à mon jury je tiens à remercier Véronique Viguié Donzeau-Gouge, Jacky Akoka et Alberto Mendelzon.

Je tiens aussi à remercier Sophie Cluet pour m'avoir accueilli au sein de son équipe au cours de mon stage de DEA. Sophie, ton franc parler et ton dynamisme, non seulement au cours de mon DEA mais aussi au cours de cette thèse, ont été un des facteurs prédominant dans mon choix de continuer de travailler dans ce milieu. Avoir effectué mon stage de DEA sous ta direction ainsi que sous celle de Marie-Christine Rousset a été un des facteurs déterminant dans mon choix de faire une thèse. Merci à vous deux.

Merci aussi aux différents thésards de Verso qui m'ont intégré dans le projet à mes débuts. Je pense à Jérôme et à Sihem qui sont parti au loin et à Luc qui n'a cessé de m'encourager à écrire cette thèse, et ce dès le second mois!

Merci à Tova Milo de m'avoir accueilli au sein de son équipe pendant un mois à Tel-Aviv et à sa bonne humeur légendaire et incessante.

Je tiens aussi à remercier les différents thésards du projet Verso qui ont commencé leurs thèses en même temps que moi. Merci à Vincent Aguilera, Eirini Fountulaki et Pierangelo Veltri. Les gars: "Je vous ai battu sur le fil!" Quand aux nouveaux thésards Omar Benjelloun, Grégory Cobéna et Benjamin Nguyen je leur adresserai qu'un seul encouragement : amusez vous bien pendant cette thèse.

Merci aussi aux membres de Vertigo de m'avoir accueilli au sein de leur équipe. Merci tout spécialement à Bernd Amann avec qui j'ai eu la joie de

collaborer aux cours de mon stage de DEA et tout au long de ma thèse. Merci à Philippe pour les discussions que nous avons eu et à Dan pour sa disponibilité.

Merci au différentes autres personnes gravitant ou ayant gravité autour du projet Verso. Je pense notamment à Stéphane Grumbach parti temporairement du projet pour de plus hautes fonctions, à Victor Vianu visiteur permanent du projet, Catriel Berri, Claude Delobel, Leonid Libkin, Rodney Topor qui a partagé mon bureau au début de ma thèse et à Anne-Marie Vercoustre.

Je tiens aussi à remercier les différentes personnes ayant participé, ou continuant de participer à Xyleme. Ce fut une expérience enrichissante et sans égale. Merci à Guy Ferran pour m'avoir accueilli au sein de son entreprise entre autres choses. Enfin merci Lucie.

Ces trois années de thèse n'aurait pas été si riche en expériences sans les différentes personnes avec qui j'ai eu le privilège de travailler afin d'aboutir. Je pense aux différents stagiaires ayant eu le malheur de me côtoyer pendant leurs séjours. Merci à Amélie Marian, Sandrine Jacqmin, Sébastien Ailleret, Frédéric Hubert, Bruno Tessier d'avoir partagé de nombreuses pauses thé au cours de ma première année. Merci aussi à Sandrine Lafois, Antonella Poggi, Beiting Zhu, Benjamin Nguyen, Omar Benjelloun, Antoine Galland et Jérémy Jouglet. Certains d'entre eux, six pour l'instant, ont eu la bonne idée de continuer à faire de la recherche en entrant en thèse. Seule Amélie eu le courage de rester à Verso un an de plus en partageant mon bureau. Ce fut sans doute un an de trop car elle est parti au loin juste après. Merci Amélie pour cette année et bonne chance pour ta thèse.

Je tiens aussi à remercier les différentes assistantes de projet qui m'ont aidé, plus ou moins longtemps, au cours de ma thèse. Merci Danny, Sandra, Virginie et Karolin pour m'avoir supporté.

Merci à mes amis Frédéric, Sylvie, Pascal et Nadaige de m'avoir supporté depuis mon retour sur Paris.

Enfin merci à mon père et à ma sœur qui n'ont pas cessé de m'encourager au cours de mes longues et interminables études. Papa ça y est, j'ai enfin fini !

Table des matières

Introduction	9
I Etat de l'art	13
I.1 Construction d'un entrepôt de pages du Web	13
I.1.1 Estimation des changements d'une page	15
I.1.2 Politique d'ordonnancement simple	15
I.1.3 Politique d'ordonnancement avec personnalisation des pages	16
I.2 Base de Données Active	17
I.3 Versions	19
I.4 Souscription	20
A Données Intranet : Vues Actives	21
II Généralités sur le système ActiveView	27
II.1 Vues Actives	27
II.1.1 Modèle de vue	28
II.2 Langage de Requêtes	29
II.3 Architecture de l'application	30
III Langage de définition des vues	33
III.1 Spécification des données	33
III.1.1 Variables d'instances	33
III.1.2 Variables Locales	35
III.1.3 Simplification des définitions	36
III.2 Méthodes	36
III.3 Activités	38
III.4 Règles	38
III.5 Autres aspects du langage	40
III.5.1 Lecture et Ecriture	40

III.5.2	Matérialisation et lecture	41
III.5.3	Ecriture et Transaction	41
III.5.4	Droits d'accès	42
III.5.5	Notification et surveillance des changements	43
III.5.5.1	Notification de changements	43
III.5.5.2	Surveillance des changements	44
III.5.6	Propagation des changements	45
III.5.6.1	Algorithme de Propagation	47
III.5.7	Traces	50
IV	Personnalisation et interface	53
IV.1	Personnalisation de l'application	53
IV.2	Interface utilisateur par défaut	54
IV.3	Capture d'écran	55
IV.3.1	Vue initiale d'un client	56
IV.3.2	Vue d'un client après quelques achats	58
IV.3.3	Discussion entre un client et un vendeur	58
IV.3.4	Vue d'un vendeur	59
IV.3.5	Vue finale d'un client	61
B	Données Internet : Acquisition dans Xyleme	63
V	Présentation de Xyleme	67
V.1	Architecture fonctionnelle de Xyleme	67
V.2	Natix: L'entrepôt	69
V.3	Processeur de requêtes	70
V.4	Intégration sémantique des données	71
V.4.1	Classification sémantique des documents	71
V.4.2	Vues	72
V.5	Mécanisme de versions	73
V.5.1	Modèle Logique	73
V.5.1.1	Modèle de Données	73
V.5.1.2	Opération sur les arbres	74
V.5.2	Deltas	75
V.5.3	Le groupe des deltas complets	77
V.5.4	Gestion des identifiants	78
V.5.5	Modèle physique	79
V.6	Souscription	80
V.6.1	Architecture de Souscription	81
V.6.2	Langage de Souscription	84

V.6.2.1	Requêtes de surveillance	86
V.6.2.2	Requêtes continues	86
V.6.2.3	Rapport	87
VI	Xyleme : Acquisition et rafraîchissement des données	89
VI.1	Architecture	91
VI.2	L'Interface Web	93
VI.3	Le Gestionnaire de Métadonnées	96
VI.4	Importance des Pages	98
VI.4.1	Importance des pages XML	98
VI.4.2	Utilité des pages HTML	101
VI.4.3	Publication/Souscription	102
VI.4.4	Implantation	103
VI.4.5	Discussion	108
VI.5	Ordonnanceur de Pages	109
VI.5.1	Découverte des Pages	110
VI.5.2	Rafraîchissement des Pages	110
VI.5.3	Estimation de la fréquence de changement	114
VI.5.3.1	Existence de changement	115
VI.5.3.2	Dernière date de changement	115
	Conclusion	117
	Bibliographie Personnelle	121
	Bibliographie	123
A	Définition Langage ActiveView	139
A.1	Définition d'une Application	139
A.2	Définition des Données	140
A.3	Définition des Méthodes	141
A.4	Définition des Activités	141
A.5	Définition des Règles	141

Introduction

Depuis le début de l'informatique moderne, la distribution des données a été une source inépuisable de problèmes. Le plus célèbre exemple de distribution de données est peut être l'Internet, descendant du réseau Apanet, mis en place par l'armée américaine pour distribuer ses centres de données afin de pallier le risque d'une guerre nucléaire à la fin des années 60.

Dès lors que les données sont distribuées, un problème récurrent se pose : celui des changements de ces données. En effet, le système se doit de fournir une qualité de service de même ordre que si les données étaient hébergées dans un cadre centralisé, c'est-à-dire des données à jour.

De nos jours avec l'apparition du web [154] et son explosion au cours de ces dernières années, cette qualité de service est impossible à satisfaire du fait de la quantité gigantesque de ces données [123], de leur volatilité et des contraintes technologiques. Néanmoins, les exigences des utilisateurs n'ont pas changé. Ils veulent toujours des informations les plus récentes possibles. Des documents qui ne sont plus à jour ne les satisfont pas. De surcroît, cette mise à jour se doit d'être disponible immédiatement ou, du moins, dans un laps de temps raisonnablement court.

Afin de satisfaire leurs exigences, deux techniques ont été développées : (1) soit l'utilisateur accède à chaque fois aux données disponibles sur un site distant; (2) soit les données, "dites" intéressantes pour les utilisateurs, sont répliquées et rassemblées sur un site unique pour être plus rapidement disponibles. Chaque solution présente des avantages et des inconvénients.

La première solution est celle utilisée quand on accède au web et la seconde est utilisée dans les indexeurs du web comme Yahoo! [164]. Le moteur de recherche accède une fois à une page et stocke cette page ainsi que les informations liées à cette page dans son index. Cet index, qui par définition n'est jamais à jour, est utilisé quand l'utilisateur pose une requête au moteur de recherche.

La réplication donne de meilleurs résultats en terme de temps d'accès. Cette solution est la plus couramment utilisée même si la plupart des utilisateurs n'en ont pas conscience. Parmi les systèmes utilisant cette technologie,

citons les navigateurs modernes qui ont un système de cache permettant d'accéder une fois par session, ou par jour, à l'information. Les moteurs de recherche [152, 155, 157, 158, 164] constituent un second exemple de tels systèmes. Ces programmes accèdent aux données afin d'en extraire des informations et de les stocker, voire les documents eux-mêmes, permettant de répondre plus facilement aux requêtes des utilisateurs. Enfin, un exemple assez typique de l'utilisation de la réplication est celui des sites "miroir"[125]. Des sites entiers sont répliqués afin de diminuer le temps d'accès à l'information. Cette solution présente néanmoins un problème important : celui de l'obsolescence de l'information stockée [87]. Les mises à jour sont faites sur des répliques et leur propagation aux autres sites peut prendre du temps.

Une façon naïve de lutter contre l'obsolescence des données est de relire les données à partir de leur point de stockage le plus souvent possible. Cette solution est celle employée par les moteurs de recherche tels que [152, 157] ou les sites miroirs, même si pour ces derniers des mécanismes permettant de ne rapatrier que les données ayant changé, existent [137, 138, 144].

Pour lutter contre l'obsolescence de l'information de manière plus fine, il existe deux solutions, chacune dépendant des capacités offertes par la source d'informations servant de référence. La première solution suppose une collaboration étroite entre la source d'informations et les utilisateurs par le biais de mécanismes d'avertissement. Les problèmes liés à une collaboration étroite ont été étudiés dans le cadre des bases de données distribuées [151]. Ainsi d'après [183] les difficultés inhérentes du mécanisme d'avertissement sont liées aux changements des données. Il faut, en effet, prévenir tous les utilisateurs désirant être avertis d'un changement sur une ressource que celle-ci a effectivement changé. De plus, le coût de mise en oeuvre de tels systèmes est encore, pour l'heure, prohibitif. La seconde solution ne nécessite, quant à elle, aucun pré-requis de la part de la source d'informations. Elle consiste en la mise en oeuvre d'un système un peu plus évolué que la manière dite "naïve" de relecture complète des informations. Ce mécanisme essaie de relire les données dès que celles-ci ont changé, afin de minimiser l'obsolescence de la base. Les changements des informations sont évalués à l'aide de différentes méthodes mathématiques [37].

Cette thèse considère tour à tour ces deux visions du contrôle des changements dans un cadre distribué sur des données semi-structurées [6].

La première vision se place dans le contexte d'une application dont les données sont distribuées sur différents sites, dans le cadre d'une application de commerce électronique. Il s'agit alors de contrôler la propagation des changements entre une base de données et des vues actives [172] dans un intranet local, afin de satisfaire la contrainte de temps d'accès à l'information. Cela conduit à la notion de vues actives composantes de base du système Acti-

veView [4, 5, 7], présentée dans la première partie de la thèse. Cette partie propose un nouveau langage de spécifications de vues, doté de capacités actives. Il permet de spécifier les différents composants d’un acteur, par exemple un client ou un vendeur. Les acteurs impliqués dans une application, peuvent avoir différentes vues des données de l’entrepôt. Celles-ci sont traduites par une spécification d’ensembles d’activités, de méthodes et de règles. Des mécanismes évolués de notification, de contrôle d’accès et de trace des activités de l’utilisateur sont fournis. Ce travail a été réalisé dans le cadre d’un projet labélisé RNTL, Gaël, en collaboration avec l’équipe IASI du LRI et de feu la société MatchVision. Il fit aussi l’objet d’une coopération avec l’Université de Tel-Aviv (Israël) dans le cadre du projet “Factory of the Future” mené par l’AFIRST¹.

La seconde partie de la thèse a pour cadre général le web. La seule façon de détecter qu’une donnée a changé, est de l’obtenir une nouvelle fois afin de la comparer avec sa précédente valeur. Ce mécanisme d’acquisition et de rafraîchissement des données est présenté au cours de la seconde partie de cette thèse. Il nous a conduit à étudier et implanter la première version du mécanisme d’acquisition de données [110, 109] du système *Xyleme* [176]. La seconde partie de la thèse débute par une présentation du cadre général de ce travail, le système *Xyleme*, un entrepôt dynamique de données centré sur les pages à technologie XML [166]. Nous présentons ensuite des travaux liés à cette acquisition et auxquels nous avons participé tels que le mécanisme de versions de données [105, 106] et le mécanisme de souscription [113] afin de doter le système des mécanismes indispensables présentés lors de la première partie de la thèse. Ce travail a constitué un des apports ayant abouti à la création de la start-up *Xyleme*.

De nombreux aspects développés au cours de cette thèse ont fait partie d’un travail de groupe. En particulier j’ai participé aux contributions collectives suivantes :

- Les travaux autour du système de détection de changement d’*ActiveView* (section III.5.5.2) réalisés en collaboration avec **Amélie Marian**.
- Les travaux autour du gestionnaire de règles d’*ActiveView* (section III.4) réalisés en collaboration avec **Sébastien Ailleret**.
- Les travaux autour de l’interface graphique d’*ActiveView* (chapite IV) réalisés en collaboration avec **Bruno Tessier et Brendan Hill**.
- Les travaux autour du mécanisme de version de *Xyleme* (section V.5) dûs principalement à **Amélie Marian** et à **Grégory Cobéna**.

1. Association Franco-Israélienne pour la Recherche Scientifique et Technologique

- Les travaux autour du mécanisme de souscription de *Xyleme* (section V.6) dûs principalement à **Benjamin Nguyen** et **Grégory Cobéna**.
- Les travaux autour de l’interface web de *Xyleme* (section VI.2) réalisés en collaboration avec **Sébastien Ailleret**.
- Les travaux autour du système d’acquisition de *Xyleme* (section VI.4 et section VI.5) réalisés avec **Mihai Preda**.

Redde Caesari quae sunt Caesaris, et quae sunt Dei Deo.

Cette thèse est organisée comme suit. La première partie présente le système *ActiveView*. Cette partie commence par une présentation générale du problème de la construction d’une application locale au-dessus d’une base de données actives avant de présenter au cours du chapitre II les concepts de base de notre approche. Le chapitre III présente le langage déclaratif d’*ActiveView* ainsi que ces différentes fonctionnalités. Enfin le chapitre IV décrit les différentes manières de personnaliser une application écrite dans notre langage avant de présenter un exemple réel par le biais de capture d’écran.

La seconde partie de la thèse présente notre solution pour synchroniser un entrepôt de données extrait du web et le web. Cette partie décrit tout d’abord le système *Xyleme* dans sa globalité au cours du chapitre V avant de se concentrer sur la solution envisagée au cours du chapitre VI. Nous terminons par rappeler les contributions apportées par notre travail et les différentes perspectives envisagées.

Chapitre I

Etat de l'art

Ce chapitre présente les différents travaux existant qui traitent les problèmes abordés au cours de cette thèse.

La section I.1 fait le bilan des travaux portant sur l'acquisition et la détection des changements sur le web. La section I.2 décrit les mécanismes des systèmes de bases de données actuels. Les sections I.3 et I.4 présentent les technologies développées pour ajouter un mécanisme de version, respectivement de souscription, aux systèmes de bases de données.

I.1 Construction d'un entrepôt de pages du Web

Cette section étudie le problème de la construction d'un entrepôt de données contenant des pages extraites du web et de sa mise à jour la plus automatique possible. La gestion d'un entrepôt de données du web nécessite des mécanismes de synchronisation qui ont été étudiés dans un contexte de bases de données distribuées notamment dans le projet Esprit numero 22469 intitulé "Fundamentals of Data Warehouses" [87].

Dans [115], les auteurs montrent qu'un parcours en largeur du web donne de meilleurs résultats que toutes les autres stratégies développées ci-dessous pour peupler un entrepôt. Mais par la suite cette stratégie devient désastreuse car elle ne permet pas de modéliser correctement le rafraîchissement de l'entrepôt.

Plusieurs travaux [66, 173] se sont concentrés sur les aspects relatifs au cache des pages. D'autres se sont principalement intéressés sur la manière la plus efficace de mettre à jour un entrepôt borné de pages extraites du web, souvent grâce à des méthodes spécifiques, telles que la fouille de données, comme par exemple les travaux Cho et Garcia-Molina [54, 53, 55]. Il existe deux différents modèles afin de construire un entrepôt à partir du web :

inflationniste ou stationnaire.

Le modèle inflationniste consiste à continuer d'ajouter de nouvelles pages à l'entrepôt tout en continuant de rafraîchir les pages existantes. Ce modèle est celui présenté au cours de la seconde partie de cette thèse (Partie B, page 65).

Le modèle stationnaire est celui envisagé par les différentes études au cours de ces deux dernières années, notamment par Cho et Garcia-Molina [55]. Dans ce modèle le nombre de pages contenues dans l'entrepôt est borné, tout ajout d'un ensemble de nouvelle page se fait alors au détriment de la suppression d'un ensemble de pages de même cardinalité. C'est ce modèle qui est le plus prépondérant dans la littérature et qui fait l'objet de cette section.

Définitions et Modèle L'entrepôt de données construit peut être modélisé ainsi : Soit \mathcal{P} l'ensemble des pages du web, et \mathcal{Q} l'ensemble des pages contenues à un instant donné dans l'entrepôt. A chaque page i , $Q_i \in \mathcal{Q}$ correspond une page $\mathcal{P}_i \in \mathcal{P}$. Le fait que les pages du web apparaissent, disparaissent ou leurs contenus changent, ne permet pas de définir une application de \mathcal{Q} dans \mathcal{P} . Afin de faciliter l'exposé nous ferons l'amalgame entre une page du web et l'URL par laquelle elle est accessible. De plus, il faut comparer deux instances d'une même page au cours du temps pour savoir si ces instances sont équivalentes ou non. Nous ne nous intéressons pas, dans cette section, à la description des les changements par eux-mêmes mais juste à savoir si ce changement a eu lieu ou non. La partie du système chargée de rapatrier une page du web et de la charger dans l'entrepôt est appelée "robot".

Le modèle stationnaire peut être alors modélisé ainsi. L'ensemble des pages est fixé au départ, aucune suppression-ajout de nouvelle page n'est accepté. Soit N la cardinalité de \mathcal{Q} . Une page Q_i est considérée à jour, ou fraîche, par le système entre le moment où le robot la rapatrie du web et le moment où son instance du web \mathcal{P}_i est modifiée. A partir de ce moment elle est considérée comme obsolète. Appelons r_i cet intervalle. Le problème revient à trouver les fréquences relatives d'accès aux pages et une politique optimale d'ordonnement de l'accès aux pages qui minimise la fonction $C = \sum_{i=1}^N c_i \cdot r_i$ où c_i est un paramètre de personnalisation de la page Q_i . Cette fonction doit être minimisée sous la contrainte G symbolisant les ressources disponibles.

Le rafraîchissement d'un entrepôt basé sur ce modèle peut être décomposé en deux parties distinctes, chacune d'entre elles faisant l'objet d'une section à part entière :

1. Déterminer un ordonnancement optimal des pages basé seulement sur

leur taux de changement ($\forall i, j \in \mathcal{Q} \quad c_i = c_j$).

2. Déterminer un ordonnancement optimal des pages basé sur leur taux de changement et leurs paramètres de personnalisation.

Avant de décrire plus en détail la littérature concernant ces deux politiques d'ordonnancement des pages, nous allons présenter comment le changement d'une page au cours du temps est estimé dans les systèmes existant actuellement.

I.1.1 Estimation des changements d'une page

Une hypothèse commune dans la communauté est de supposer que les changements d'une page peuvent être modélisés par un processus de Poisson. Cette hypothèse est validée par les deux études suivantes.

La première a été réalisée par Brewington et Cybenko [35], se basant sur un ensemble de deux millions de pages. Ils ont observé les changements de cent milles pages par jour pendant une période de sept mois. Cette étude a montré que la plupart des pages de leur entrepôt ont changé pendant les heures de bureau des Etats-Unis, entre 05H00 et 17H00 (tranche horaire de la Silicon Valley) du Lundi au Vendredi.

La seconde a été réalisée par Cho et Garcia-Molina dans [53, 55]. Cette étude a porté sur la surveillance de sept cent vingt mille pages venant de 720 sites différents pendant une période de quatre mois. Ils ont trouvé que pour les sites surveillés, par exemple 40 pour cent des pages dans le domaine ".com" changent journallement alors que plus de 50 pour cent des pages des domaines ".edu" et ".gov" n'ont pas changé durant les quatre mois de leur étude. Les auteurs ont comparé leurs résultats avec les prédictions que la modélisation par un processus de Poisson aurait donné et montrent que le modèle poissonnien décrit très bien les changements des pages, à la condition que les pages aient un taux de changement λ_i compris entre un jour et quatre mois.

Dans la suite nous supposons que la page \mathcal{P}_i change au cours du temps suivant un processus de Poisson de paramètre λ_i , $\mathcal{P}_i = f(\lambda_i)$.

I.1.2 Politique d'ordonnancement simple

Coffman et al [90] modélisent la relecture des pages suivant une modélisation proche de celle des files d'attente et des systèmes décisionnels. Les auteurs supposent aussi que les durées d'accès à des pages sont des variables aléatoires indépendantes et identiquement distribuées suivant une loi Υ . Le

problème est alors proche de celui d'un système avec un seul serveur (robot) gérant plusieurs files d'attente. A chaque page Q_i correspond une file d'attente. Le changement d'une page est modélisé par l'arrivée d'un nouveau client. L'intervalle entre deux accès consécutifs à la même page correspond au temps de basculement du système. Un résultat de leur modélisation est que si Υ décroît dans l'ordre convexe-croissant, alors r_i décroît pour toutes les pages i sous n'importe quelle politique d'ordonnancement. De plus afin de maximiser l'espérance de la fraîcheur d'une page, l'accès à cette page par le robot doit être le plus équitablement espacé au cours des différents accès successifs.

Cho et Garcia-Molina ont étudié deux modélisations différentes de système dans [54]. La première modélisation suppose que les pages changent toutes suivant un même taux λ , i.e. $\forall i, j \in \mathcal{Q} \quad \lambda_i = \lambda_j$. Ils en concluent que sous cette hypothèse la meilleure stratégie pour synchroniser l'entrepôt avec le web revient à synchroniser toutes les pages de l'entrepôt dans le même ordre fixe à chaque fois. Cela revient à rafraîchir une page dans un intervalle fixe.

La seconde modélisation suppose que les pages changent suivant des fréquences indépendantes les unes des autres et que le système connaît les fréquences de changements de chaque page de l'entrepôt. Dans ce cas, les auteurs montrent que la solution optimale est plus proche d'une politique d'ordonnancement uniforme, comme pour le précédent cas. Ces résultats confirment ceux de Coffman et al décrit précédemment.

I.1.3 Politique d'ordonnancement avec personnalisation des pages

Dans [52] les auteurs montrent que la qualité d'un entrepôt peut être singulièrement améliorée en orientant le système vers les pages intéressantes pour les utilisateurs finaux de l'entrepôt. Les pages sont différenciées par une notion d'importance. Généralement plusieurs métriques sont utilisées pour définir l'*importance* d'une page. La plus couramment utilisée actuellement est celle de [37], qui la définit comme étant le résultat du calcul d'un point fixe sur la matrice des liens de la page au reste du web, matrice construite grâce à l'ensemble des pages connues du système.

Par la suite Cho et Garcia-Molina montrent dans [55] qu'une architecture basée sur un robot dit "en mode incrémental", est plus performante que celle basée sur un robot dit "en mode différé". Un robot est dit incrémental lorsque le système lui donne des pages à rafraîchir continuellement, à l'opposé d'un robot "en mode différé" où le système envoie une liste de pages à recueillir de

temps en temps par le robot, i.e. un nombre fixe de pages est relu pendant une période donnée.

En utilisant des collections différentes, Wills et Mikhailov [173] arrivent sensiblement aux mêmes conclusions.

[67] propose un modèle adaptatif suggérant qu'une stratégie de relecture efficace peut être implantée sur leur robot, ou tout autre robot incrémental sans aucune hypothèse sur le taux de changement des pages en récoltant des informations glanées au cours des précédentes relectures. Pour cela, ils ont résolu un système d'équations avec 11200 variables dont 10000 sont non linéaires et 11200 contraintes dont 3300 sont non linéaires. Cependant, la résolution d'un tel système d'équations dépend fortement des conditions initiales (même si les auteurs disent que leur modèle est robuste) et se fait de manière déconnectée du processus.

Tous les travaux présentés ont été développés suivant un modèle d'entrepôt dit stationnaire, l'acquisition de nouvelles pages se faisant au détriment d'autres pages. Ainsi un des désavantages des travaux de Cho et Garcia-Molina, est qu'ils ne portent que sur un entrepôt borné de pages du web.

Notre approche, inflationniste, basée sur les différents travaux présentés, est flexible, adaptative basée sur le web tout entier et permet aussi bien de rafraîchir l'entrepôt existant que de découvrir de nouveaux documents pertinents pour le système. La seconde partie de la thèse présente un système permettant non seulement de rafraîchir un entrepôt construit à partir du web, mais aussi de continuer d'acquérir de nouvelles pages.

I.2 Base de Données Active

Une caractéristique des systèmes de bases de données traditionnels est leur passivité : ils n'exécutent des opérations que si l'utilisateur ou un programme le leur demande explicitement. Une solution pour pallier ce problème serait d'écrire des programmes vérifiant périodiquement la base de données et, selon son état, de prendre les mesures adéquates. Malheureusement cette solution est trop dépendante de la périodicité de la vérification. Si elle est trop longue, le système risque d'être bloqué ; au contraire si elle est trop petite, le système risque de rater des états spécifiques. Une autre solution a été mise au point : les systèmes de bases de données actives. Ces systèmes peuvent exécuter automatiquement, selon l'état de la base, des règles préprogrammées et réagir à des événements [131, 172].

Les premiers systèmes de bases de données actives ont émergé à partir des années 1970 avec le langage CODASYL [77]. Les règles suivaient le paradigme ON `<event>` CALL `<procedure>` où les événements étaient limités aux

actions *INSERT*, *REMOVE*, *FIND*, Les procédures étaient des appels de programmes COBOL. Au cours du temps le langage de règles est devenu de plus en plus élaboré. Ainsi dès le système QBE [182] les règles acquièrent leur forme définitive :

```
ON <event> if <condition> do <action>
```

Les moteurs de déclenchement ou “triggers engines” en anglais, sont suggérés dès la fin des années 1970 [136] dans [68, 69]. Enfin le premier système de base de données actives intégrant des “triggers” a été proposé au début des années 80 dans [112]. Les triggers font actuellement partie intégrante de la norme SQL92.

Plusieurs systèmes implantent actuellement les fonctionnalités de bases de données actives. Les projets DIPS [133], RPL [61, 62], A-RDL [59, 60, 96, 97, 135], POSTGRES [143, 147, 148], Starburst [42, 43, 44, 45, 46, 84, 171], pour ne citer qu’eux, sont implantés à partir du modèle relationnel.

Les solutions choisies par ces divers projets ne permettent pas de spécifier dynamiquement de manière simple des événements et des règles. De plus, la plupart des fonctionnalités offertes nécessitent de redémarrer l’ensemble du système afin de prendre en compte des changements mineurs effectués sur les aspects actifs du système. Ceci entraîne un coût énorme lors du déploiement d’une application se servant des aspects actifs de ces systèmes.

Actuellement de nombreux efforts sont fournis sur les services web afin de fournir les services de notification aux changements des données. Ces travaux s’appuient sur de nouveaux standards d’échanges de données tels que SOAP [167], WSDL [168] ou encore UDDI [162]. Ces travaux sont encore immatures. Citons néanmoins [33] qui utilise le protocole d’échange SOAP afin de fournir un système réactif utilisant des règles actives.

I.3 Versions

Les utilisateurs des bases de données sont souvent intéressés non seulement par les données courantes stockées dans la base de données, mais aussi par les changements de ces données au cours du temps. Les versions dans les bases de données ont été le sujet de nombreux travaux. Dès le début des années 80 [13] discute des différentes utilisations des versions d'un même document au cours du temps.

La plupart des travaux sont centrés sur les données plutôt que sur les changements; c'est-à-dire que le modèle de données physique choisi permet de suivre les changements d'un n-uplet ou d'un objet très facilement au cours du temps au détriment de l'état de la base, ou d'un document à un moment donné. Ainsi [39, 40, 41] définit une approche pour les bases de données objets. D'un certain point de vue, les auteurs attachent à chaque objet (oid) une relation R . Le schéma de cette relation R est composé d'un attribut valeur et d'un attribut représentant l'estampille dans l'arbre de versions de la base à partir de laquelle la valeur est affectée à l'objet. Ce mécanisme permet de retrouver facilement les différentes valeurs d'un objet au cours du temps. Une approche similaire a été suivie dans [48] pour gérer l'historique dans les bases de données semi-structurées. Se basant sur le modèle OEM [120] les auteurs attachent à chaque nœud de l'arbre des labels contenant une estampille temporelle ainsi que son ancienne valeur.

Les travaux de [75], basés sur le modèle relationnel, utilisent la notion de delta afin de faire des prévisions sur l'état futur de la base en cas de changement.

D'autres travaux portant sur les différents mécanismes de version s'attachent non pas aux changements des données eux-même mais aussi aux changements de schéma de la base de données. Citons ainsi les travaux fondateurs de [29] pour le modèle relationnel et de [98, 181] pour le modèle objet.

Des études récentes se sont appliquées au modèle XML. [51] stocke les différentes versions d'un même document au cours du temps. Néanmoins les auteurs utilisent un procédé permettant de diminuer l'espace de stockage nécessaire. Les parties communes entre deux versions consécutives d'un document ne sont stockées qu'une seule fois. Ils remplacent ces parties communes dans la nouvelle version du document par des pointeurs vers l'emplacement physique de cette partie dans la version précédente. Le gain en place est appréciable, néanmoins le coût de reconstruction d'un document est pénalisé par la relecture de la chaîne des pointeurs dans les différentes versions pour un sous-arbre commun.

I.4 Souscription

De nos jours, offrir l'accès aux informations au travers du web n'est plus suffisant pour démarquer une entreprise par rapport à ses concurrentes. Les utilisateurs attendent d'autres services tels que recevoir des informations ciblées. Ainsi, on peut souscrire par le biais de son opérateur de téléphonie mobile afin de recevoir des informations aussi diverses que la météo ou les cotations boursières sur son téléphone portable. De plus, il n'est pas rare de recevoir des messages électroniques offrant des promotions sur des articles auxquels nous étions intéressés lors d'une précédente visite.

De tels systèmes de souscription/notification se doivent d'être efficaces et d'offrir le maximum de fonctionnalités et flexibilités possibles afin d'être facilement paramétrables par les utilisateurs. De nombreux travaux ont porté sur ces sujets.

Plusieurs systèmes implantent déjà de tels mécanismes. La suite de ce paragraphe classe ces systèmes suivant leur architecture et la puissance de leurs langages de souscription.

Le système Field [126], le service d'événements de Corba [119] et [88] sont destinés à des architectures centralisées et offrent un langage de souscription basé sur des messages. S'appuyant sur le même type d'architecture, Tool-Talk [91] propose un langage basé sur les événements alors que Elvin [130] relaie les informations du serveur vers les clients. Les systèmes Keryx [175] et [179] offrent les mêmes fonctionnalités au niveau du langage sur une architecture dit "client/serveur", alors que Gryphon [26] porte ces fonctionnalités sur une architecture dite "pair à pair" (peer-to-peer). GEM [103] et YEAST [99] entre autres, permettent de raffiner un peu plus encore le langage de souscription en ajoutant des fonctionnalités permettant de filtrer les événements suivant des motifs. SIENA [38] étend ce type de langage sur les architectures "client/server" hiérarchiques et "pair à pair".

Avec la venue de XML, de nombreux nouveaux systèmes offrent soit des fonctionnalités du type "clé - valeur" [71, 159, 160] ou permettant de poser des filtres sur la structure du document grâce à des expressions de chemin [19, 121, 163]. De tels systèmes préfigurent certainement ce que sera le web dans les années futures mais les performances effectives ont encore besoin d'être améliorées afin de soutenir l'utilisation réelle des utilisateurs à venir.

Première partie

Données Intranet : Vues Actives

De nos jours, il est possible de construire des applications permettant aux clients de la toile d'interagir et de partager des données, au prix d'intenses développements de logiciel. Nous croyons que (i) la nécessité de faire des déploiements rapides d'applications, (ii) la généralisation de telles applications, et (iii) le besoin, souvent rencontré, de vérifier les propriétés de ces applications, requièrent une spécification déclarative de telles applications. La situation est, quelque part, similaire à ce qui a conduit dans les années 70 aux langages de requêtes déclaratifs pour interroger et manipuler de grandes bases de données. Ce chapitre propose un tel langage de spécification. Il explique comment il est compilé/implanté/supporté par le système *ActiveView*.

Pour illustrer ce travail, une application de commerce électronique basée sur un catalogue sur le web sera prise comme exemple. Les activités autour du commerce électronique sont multiples telles que, la sécurité, l'authentification, le paiement électronique, et la conception de modèles d'affaires [178]. Le commerce électronique est souvent amené à gérer de grandes collections de données (e.g. catalogue de produits, pages jaunes), et doit fournir un moteur transactionnel, des fonctions de contrôle de concurrence, de distribution et de reprise sur panne. Il implique aussi de fortes interactions entre les participants (e.g. entre des clients et des vendeurs) et le contrôle de séquences d'activité (i.e. gestion de flux). Tous ces aspects seront pris en charge par les *vues actives*, composant principal du système *ActiveView* [5, 7].

Plus généralement, nous nous intéressons aux applications qui : (i) demandent de partager les données, et (ii) ont besoin de travail coopératif entre des acteurs connectés par le réseau. Ces aspects typiques peuvent aussi se retrouver dans les bibliothèques électroniques ou les mémoires d'entreprises.

Nous croyons que la technologie des bases de données fournit la colonne vertébrale de telles applications. Le système *ActiveView* peut être vu comme un *générateur d'applications de bases de données*. Il permet une spécification *déclarative* d'un *certain type* d'application de bases de données. Par *déclaratif*, nous entendons (i) peu (ou pas) de programme à écrire et (ii) une description d'applications s'effectuant dans un langage de haut niveau (ou via une interface graphique). La spécification décrit les acteurs principaux de l'application. Une application est composée d'acteurs exécutant un certain nombre d'activités. Chaque activité comporte des données ainsi que des opérations. Chaque acteur est spécifié par :

- les données et les opérations accessibles pour cet acteur particulier (un mécanisme de *vue*) liées à un contrôle d'accès sophistiqué.
- les activités que l'acteur peut exécuter ainsi que les données et les opérations disponibles dans chaque activité.

- les règles actives qui non seulement définissent les séquences d'activités (composant de flux), mais permettent aussi de contrôler les changements, les événements pouvant être notifiés aux acteurs (composant de souscription) et ceux qui doivent être journalisés (composant de trace).

Par conséquent, le langage *Active View* permet de spécifier, de manière organisée et déclarative, bon nombre d'aspects souvent considérés séparément. Une des contributions principales de cette partie est de montrer comment ces différents aspects peuvent être combinés dans une structure simple et cohérente. Le système *Active View* rassemble fortement les quatre composants principaux suivants :

1. *XML* : Du point de vue des données, nous choisissons XML [166] comme modèle de données. Toutes les données seront stockées, échangées ou présentées aux utilisateurs en XML, grâce à l'API DOM [165].
2. *Règles actives* : Nos règles actives sont plutôt simples en comparaison de celles qu'on peut trouver dans la littérature [122, 172]. La nouveauté est qu'elles sont intégrées dans un cadre général et qu'elles sont utilisées pour différents objectifs (contrôle de flux, contrôle des changements, trace).
3. *Appels de méthodes et notifications* : Les données peuvent être manipulées par le biais de méthodes. De plus chaque événement déclenché par les règles actives est un appel de méthode. Ceci permet au système de relier les événements au mécanisme de souscription. Ainsi, les vues peuvent être notifiées par certains événements.
4. *Gestion des vues* : Nous nous appuyons ici sur l'expérience de O₂-Views [65], un système développé à l'INRIA. Les vues que nous considérons ici sont cependant plus simples. La nouveauté se trouve dans leur intégration avec des aspects actifs.

Considérons un exemple : supposons qu'un produit soit ajouté à un catalogue de vente électronique. Une notification est alors envoyée à tous les acteurs intéressés par cet événement qui se traduit par un changement dans leur vision du catalogue. Comme les vendeurs veulent visualiser la version la plus récente du catalogue, leur spécification devrait inclure une règle active actualisant leur vue du catalogue, quand tel ou tel changement du catalogue a lieu. Observons que la détection d'un événement et la maintenance doivent toutes les deux tirer partie des techniques de mise à jour incrémentale. En particulier, si le changement affecte une portion du catalogue non intéressante pour un vendeur spécifique, le système doit éviter d'actualiser sa vue.

De plus, quand la vue d'un vendeur doit être réactualisée, cette opération doit être effectuée incrémentalement afin d'éviter de renvoyer de larges portions du catalogue sur le réseau. La première contribution du système *ActiveView* est de permettre la spécification déclarative de ces composants à l'aide d'un langage de haut niveau.

Une seconde contribution est un système qui implante ces concepts en s'appuyant le plus possible sur les standards. Une spécification *ActiveView* est compilée vers une application basée sur l'environnement suivant :

- La couche XML de la base de données O₂ développée par ArdentSoftware [153] et son interface DOM [165] pour stocker et interroger les données XML et les méthodes.
- Le langage de requêtes standard pour XML [170] quand il sera disponible. En attendant (et pour illustrer les exemples de ce chapitre), nous utiliserons un langage simple inspiré de Lorel [2] appelé X-OQL [15].
- Le mécanisme de notification du système O₂. Ce mécanisme existait déjà pour le langage C++ [117], nous avons dû l'adapter pour l'utiliser avec le langage Java [24]. Chaque vue est un programme Java "multi-threadé" utilisant l'API Java-DOM.
- L'utilisation des Interfaces web pour afficher les documents XML et des navigateurs XML interagissant avec les vues, via les méthodes d'invocation à distance de JAVA (RMI [140]). En attendant que les navigateurs XML offrent les fonctionnalités dont nous avons besoin, nous utilisons des pages HTML dynamiques en conjonction avec des applets JAVA. Du point de vue de l'utilisateur, une application se présente comme une séquence de pages contenant des données (modifiables) et des boutons. Les pages peuvent évoluer dynamiquement (e.g. de nouvelles promotions peuvent apparaître).
- La génération automatique d'une application à partir d'une spécification de vue. Nous offrons des moyens flexibles de personnaliser de telles applications.

De telles spécifications sont compilées par le système *ActiveView* dans plusieurs applications qui permettent à différents utilisateurs de naviguer sur un ensemble d'activités contrôlées, de travailler interactivement sur des données précises. Une application *ActiveView* peut, bien sûr, utiliser une application déjà existante, et, d'une certaine manière, peut être aussi vue comme une manière d'exporter sur le web une application de base de données existant sur l'intranet de manière contrôlée [70].

Cette partie est organisée de la façon suivante. Le chapitre II introduit notre système de vues actives. Les besoins en fonctionnalités sont illustrés grâce à un exemple qui présente brièvement le modèle de données et le langage de requête retenus. L'architecture générale de notre système conclut ce chapitre. Le chapitre III présente le langage de définition d'une application "ActiveView". Enfin, le chapitre IV discute de l'application générée par défaut par le système et des différents moyens de la personnaliser.

Chapitre II

Généralités sur le système ActiveView

Ce chapitre introduit successivement les vues actives de *ActiveView*, le modèle de données et le langage de requêtes. Pour finir, l'architecture générale du système *ActiveView* est décrite.

II.1 Vues Actives

Une application *ActiveView* permet à plusieurs utilisateurs de travailler de manière interactive sur les mêmes données dans le but d'exécuter un ensemble particulier d'activités contrôlées.

Avant d'entrer dans de plus amples détails, prenons un exemple. Un magasin virtuel implique plusieurs types d'acteurs, tels que des clients et des vendeurs. Cette application manipule aussi de grandes quantités de données, e.g. les produits vendus et présentés dans un catalogue ou les informations sur les produits en promotion. De plus, chaque acteur peut voir différentes parties des données. Ainsi, un client peut seulement voir ses ordres et les promotions correspondant à une catégorie particulière, alors qu'un vendeur peut voir tous les ordres et toutes les promotions. Chaque acteur peut exécuter différentes *actions* sur les données. Les droits d'accès dépendent de l'acteur. Par exemple, les promotions ne peuvent être mises à jour que par certains vendeurs. Aussi, les demandes sur la "fraîcheur" des données peuvent varier d'un acteur à un autre. Quand une promotion apparaît, le client peut souhaiter que son écran soit immédiatement rafraîchi, mais quand un changement intervient sur le catalogue, il n'est pas forcément nécessaire de déranger un client en train de regarder le catalogue ou une ancienne version du catalogue. Il suffit de l'avertir qu'un changement sur un produit susceptible de l'intéres-

ser a eu lieu. Le rafraîchissement de la valeur de ce produit sera, alors, fait par une requête explicite de la part du client.

Chaque acteur exécute plusieurs activités. Par exemple, un client *recherche* un produit sur le catalogue, *commande* des produits, *modifie* une commande passée. Dans chacune de ces activités, la possibilité de montrer différentes pages aux acteurs est offerte. Ces pages contiennent seulement les données et les actions disponibles pour cet acteur particulier. De plus, des actions effectuées par un acteur dans une activité particulière peuvent déclencher d'autres actions. Par exemple, la commande d'un produit déclenche la mise à jour du stock. Finalement, il peut être utile de gérer l'historique de certaines opérations des acteurs. Ceci permet d'effectuer des analyses postérieures ainsi que de prévoir de futures contestations.

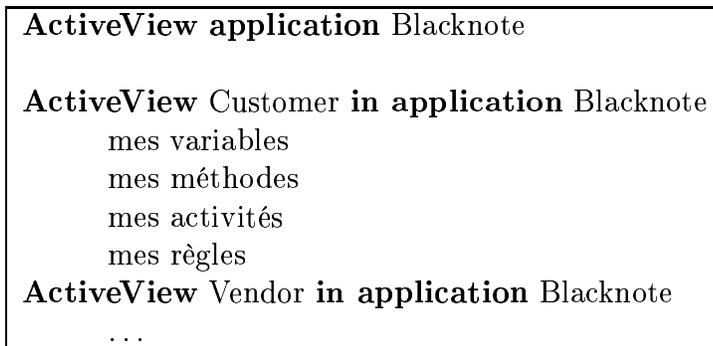
Bien que motivé par des applications de commerce électronique le système *ActiveView* s'applique de manière générale à un large panel d'applications pour le web. Au delà d'une *spécification déclarative* notre approche permet une *génération* automatique de l'application (par compilation), plutôt que de produire de grandes quantités de code pour une application spécifique. Nous allons présenter la spécification déclarative de telles vues actives, la génération et la personnalisation de ces vues sera décrite dans le chapitre IV.

II.1.1 Modèle de vue

Une spécification dans le langage déclaratif est définie pour chaque type de vue active participant à l'application :

- les données accessibles par la vue;
- les opérations (méthodes) que l'on peut faire dans la vue;
- les différentes activités ou ensemble de méthodes et de données accessibles pour effectuer une action particulière, par exemple l'activité recherche ou achat;
- les règles actives qui sont les composantes actives de base du système *ActiveView*.

La spécification générale de l'application "Blacknote" comporte deux types d'acteur: "customer" et "vendor" et suit la forme suivante :



Une description plus formelle de la définition d'une application est décrite à la section A.1 Page 139.

Le modèle de données utilisé pour les vues actives repose sur le langage XML [166] et sur l'API DOM [165] pour sa représentation arborescente des données.

Nous présentons dans la suite de cette section le langage de requêtes et nous terminons par un aperçu de l'architecture du système *ActiveView*.

II.2 Langage de Requêtes

En attendant qu'un standard de langage de requêtes pour XML soit disponible [170] le langage de requête appelé X-OQL [15] est utilisé. C'est un langage proche de Lorel [2] basé sur des expressions de chemin pour interroger les arbres DOM. Nous illustrons ici ses caractéristiques par deux exemples.

La requête suivante recherche les éléments *Item* qui ont un prix inférieur à 50 euros sur le document connu par le processeur sous le nom "Catalog".

<pre> select i from i in Catalog.*.Item where i.Price < 50 </pre>

Observons que le symbole "*" dénote un chemin quelconque de longueur arbitraire (équivalent au symbole "/" du langage XPath [169]).

La requête ci-dessus construit un nouvel arbre DOM dont les fils sont les éléments *Item* sélectionnés.

Plus précisément :

1. Le module de mise à jour reçoit le flot des changements de l'entrepôt. Il avertit les vues appropriées des changements qui les concernent. Ce mécanisme est basé sur le mécanisme de notification fourni par l'interface XML de la base de données de O₂ [104].
2. Le module de trace garde le journal des événements spécifiés. Ces événements sont générés par l'application ou par des vues.
3. Le module de règles actives gère l'ensemble des règles spécifiées par le créateur de la vue [17]. Ces règles sont lancées conformément aux événements et peuvent avoir un impact sur la base XML et sur les vues actives. Elles forment, par exemple, les composants essentiels pour spécifier un modèle d'affaire d'une application de commerce électronique.

Ces notifications sont générées en accord avec la spécification des vues. Deux types d'événements peuvent être notifiés : (i) les événements générés par le gestionnaire de données après la création/suppression/changement d'objets et (ii) les événements définis par l'utilisateur générés par les clients. Le mécanisme de notification d'O₂ sur lequel nous nous reposons a été partiellement développé par l'équipe Verso de l'INRIA [24].

Une Vue Active est l'objet fondamental de notre application. Dans la version courante du système, elle est implantée en Java. Cet objet appartient à (est sous classe d') une classe particulière appelée *ActiveView*. Cette classe contient certaines variables d'instance, et en particulier la variable *owner* qui stocke des informations sur l'utilisateur ayant initié la vue. Cette classe contient aussi des méthodes telles que *transaction*, *commit*, *abort* pour capturer les modes transactionnels, ou *init*, *sleep*, *quit*, *resume*. Une vue active est généralement reliée à une fenêtre sur le web ouverte par un utilisateur du système. Des vues n'ayant pas d'interface web peuvent être aussi introduites, comme pour la gestion de la comptabilité. Une vue active a accès aussi bien à la base XML qu'aux données locales (les variables d'instance des objets vues). Elle réagit aux demandes des utilisateurs et peut être rafraîchie par les notifications envoyées par le serveur ou le gestionnaire de vues. Les méthodes accessibles par la vue dépendent des droits d'accès des utilisateurs qui peuvent leur permettre de lire, écrire, charger une partie ou l'ensemble des données qu'ils peuvent voir.

Les Interfaces Utilisateur sont actuellement implantées comme des documents HTML dynamiques communiquant avec le système par le biais d'un applet Java. Les composantes dynamiques sont constituées par des petites fonctions Javascript encapsulées à l'intérieur des pages HTML. Notre but est de passer vers XML dès que les navigateurs XML supporteront les caractéristiques dynamiques souhaitées. Une activité est associée à un document HTML pour chaque utilisateur. L'applet Java est construit au-dessus d'une API générée par le système selon la spécification de la vue. Bien que le système génère des interfaces par défaut, le programmeur de l'application peut les redéfinir/personnaliser en utilisant l'API générée qui capture la sémantique de l'application [149]. En principe, le serveur, les clients et les interfaces s'exécutent sur des machines différentes. Les données de la vue sont obtenues par enregistrement/désenregistrement explicite, par conséquent les changements de la base XML ne sont généralement pas propagés immédiatement vers les vues, sauf si le programmeur l'a spécifié. De cette façon, les vues et la base XML peuvent voir les mêmes données.

Chapitre III

Langage de définition des vues

Le chapitre précédent a introduit une application de vue active contenant plusieurs types d'acteurs, chacun ayant une vue différente du système. La spécification de chaque type de vue consiste en quatre parties qui définissent respectivement : (i) les données, (ii) les méthodes, (iii) les activités, et (iv) les règles.

Dans notre système, les activités sont spécifiées en deux temps. Premièrement, pour chaque type d'acteurs (i.e. chaque vue), le programmeur déclare un ensemble d'activités avec les données et les méthodes qui peuvent être utilisées dans ces activités. Ensuite, un ensemble de règles spécifie la sémantique de la vue. Les règles spécifient comment réagir à certains événements.

Ce chapitre illustre chacune de ces parties dans des sections différentes, de la section III.1 à la section III.4. Ce chapitre finit par présenter des aspects transversaux du langage tels que les modes d'accès à la section III.5. Une définition plus exhaustive de la syntaxe du langage est fournie à l'annexe A.

III.1 Spécification des données

Une vue active est constituée de deux types de variables : (i) des variables d'instances qui dérivent de la base XML et (ii) des variables locales de la vue. Cette section discute la définition de telles variables. Enfin, une simplification de la définition des variables est proposée en fin de section.

III.1.1 Variables d'instances

La spécification des variables est illustrée en utilisant un exemple très simple dans lequel trois types d'utilisateurs interagissent : un ensemble de clients et de vendeurs et un seul contrôleur. Un client peut feuilleter le cata-

logue, passer et modifier des commandes. Le contrôleur associe un vendeur à des clients. Un seul contrôleur peut être actif en même temps. Si une personne essaie d'entrer dans le système comme contrôleur, il est déconnecté si un contrôleur existe déjà ou s'il n'a pas le droit d'exister en tant que tel. Un vendeur est en charge de quelques clients et peut interagir avec eux, e.g. en offrant de nouvelles promotions.

Considérons, en premier, la vue d'un client. L'exemple A illustre l'importation dans la vue des données d'un catalogue.

let	catalog: CatalogElem
be	RepCatalog
with	catalog.*
mode	read all

Exemple A: *Définition (simplifiée) de la Variable d'instance catalog de la vue Client*

Cette spécification importe l'élément racine de la base XML nommé *RepCatalog* (dans nos exemples, les entrées de la base XML seront toujours préfixées par *Rep*). Cet élément racine sera connu par la vue sous le nom, ou variable d'instance, *catalog* de type *CatalogElem*. De plus, la définition d'une variable de la vue est étendue par une clause *with* afin de fournir un moyen de spécifier la portée exacte de la variable, comme décrit dans [10]. Cette clause supplémentaire décrit, en utilisant les expressions de chemin, les sous-arbres atteignables à partir des éléments sélectionnés qui seront inclus dans la vue. Par exemple, l'ajout à la définition précédente, de la clause *with catalog.** spécifie que tout le sous-arbre DOM du point d'entrée *RepCatalog* devra être importé. En général, une clause *with* peut contenir des expressions de chemin complexes et introduire de nouvelles variables.

De plus, il peut être utile de spécifier les droits d'accès de la vue sur les variables. Ce rôle est dévolu à la clause *mode*. Dans notre exemple, cette clause spécifie que nous pouvons lire toutes les données importées. Cette dernière clause est, en fait, inutile puisque le mode par défaut des données importées est *read*. Les autres modes possibles sont :

write qui permet de spécifier que l'acteur peut changer la valeur des éléments spécifiés;

append qui autorise l'acteur à ajouter un nouvel élément du type spécifié; et enfin

remove qui autorise l'acteur de la vue à supprimer un élément.

Nous allons illustrer la puissance du langage pour spécifier une variable dérivée de la base XML avec l'exemple suivant qui spécifie la vision du catalogue dans la vue *Vendor* :

let	catalog: (CATEGORIES)*
be	select m.CATEGORIES.CATEGORY from m in RepCatalog.META_CATEGORIES.META where m.NAME == 'Musical Instrument'
with	self.* X, X.ARTICLE Y, X.NAME Z, Y.PRICE P, Y.NAME N
mode	deferred read X, immediate read Y Z N, write P

Exemple B: *Définition du Catalogue dans la vue Vendeur*

Cet exemple spécifie que la vision du catalogue par les vendeurs consiste en une collection d'éléments de type *CATEGORIES*. Dans certains cas, il est possible de dériver le type d'une requête XML [111]. Cette possibilité n'est pas considérée dans ce langage. Les éléments sélectionnés par la requête X-QL sont les éléments *META* contenant un fils *NAME* ayant pour valeur "Musical Instrument". De plus, la clause *with* permet d'associer plusieurs variables (X, Y, Z, P, N) à différents éléments du sous-arbre résultant de la définition des clauses *let* et *be*. Le mot clé *self* est un raccourci syntaxique pour représenter la variable définie par la clause *let* (dans notre cas *catalog*). La clause *mode* permet d'associer différents modes d'accès aux variables introduites par la clause *with*. Ainsi, un vendeur pourra changer le prix d'un article (*write* P). Les autres modes de lecture seront expliqués en section III.5.1.

III.1.2 Variables Locales

Pour l'instant, nous avons seulement défini les variables dérivées de la base XML. Les variables locales de la vue sont définies de la même manière.

local	caddy: (ARTICLE)*
mode	append, remove

Exemple C: *Définition de la variable locale caddy de la vue Client*

Le mot clé *local* permet de spécifier une variable locale à la vue. L'exemple C définit une variable locale *caddy* grâce à laquelle un client peut ajouter ou enlever des éléments du type *ARTICLE*.

III.1.3 Simplification des définitions

Comme illustré dans l'exemple B, la clause *with* est utilisée pour spécifier quelles données peuvent être atteintes à partir des objets liés par des variables d'instance. Néanmoins, spécifier pour chaque variable sa portée peut être fastidieux, spécialement quand le même type d'élément est atteignable par différentes variables et que nous voulons avoir la même portée et le même droit d'accès dans tous les cas. Une façon de simplifier les définitions est de permettre de spécifier les différentes portées et droits d'accès d'un élément lors de sa définition, i.e. permettre de définir pour chaque type d'élément, les données qui peuvent être lues ou modifiées quand un tel élément est accédé.

L'instruction suivante, quand elle est ajoutée à la définition de la vue d'un client, spécifie qu'à chaque fois qu'un élément "fournisseur" est inclus dans la vue, son "nom" et sa "description" complète sont aussi inclus, en mode lecture, et ses éléments fils "évaluation" sont inclus en mode ajout, c'est-à-dire qu'un client peut ajouter sa propre évaluation d'un fournisseur :

element	SupplierElem
with	self.name, self.description.*, self.evaluation E
mode	append E

Exemple D: *Définition globale de l'élément SupplierElem dans la vue Client*

III.2 Méthodes

La spécification d'une vue active contient aussi la définition de méthodes accessibles par l'utilisateur dans un contexte donné. Ainsi, un client voulant ajouter un article à son caddy activera la méthode suivante :

method	add_to_caddy(article: ARTICLE)
range	article in catalog
is	caddy.getDocumentElement().appendChild(article); recalculate_budget();

Exemple E: *Méthode add_to_caddy de la vue Client*

La clause **range** permet de spécifier que le paramètre *article* de type *ARTICLE* provient de la sélection de la variable *catalog*. Le code de la méthode est assez simple. Il est essentiellement constitué d'appels de méthodes connues

par la base XML ou de la vue. Ces méthodes sont soit écrites en C++ soit en Java ou n'importe quel langage supporté par l'API DOM et la base XML. Ainsi la vue contient très peu de code si ce n'est les requêtes XML. Elle est donc indépendante d'un langage particulier. Dans notre exemple cette méthode utilise l'API DOM pour ajouter l'article sélectionné à la fin du panier et appelle ensuite la méthode *recalculate_budget* afin de soustraire le prix de l'article du budget spécifié par l'utilisateur.

De plus, le système permet d'exécuter des appels de méthodes à distance. Prenons, comme exemple, un client souhaitant contacter un vendeur afin que celui-ci l'aide dans sa recherche d'un produit.

La vue client contiendra alors la définition de la méthode suivante :

method	contact_vendor(categorie: CATEGORIE)
range	categorie in category

Exemple F: *Méthode contact_vendor de la vue Client*

Cette méthode ne contient pas de code propre. Le système la traduira comme un simple *message* avec le nom de la méthode "contact_vendor" comme titre et les arguments comme contenu. La vue Vendeur contient une règle active ayant comme nom d'événement *contact_vendor*. Cette règle est donnée ici pour des raisons de consistance. Néanmoins, la spécification d'une telle règle est donnée dans la section III.4.

on	contact_vendor
if	(category == c and Customer == null)
do	Customer = sender.name; accept_contact(sender.name); notify_me(sender.name + " needs help");

Exemple G: *Règle contact_vendor déclenchée dans la vue Vendeur par un Client*

Cette règle s'applique lors de l'arrivée de l'événement *contact_vendor*. Des tests d'application sont évalués (clause *if*). Si les conditions sont satisfaites, alors la variable "Customer" est affectée à l'expéditeur de cet événement (le client ayant besoin d'aide), la méthode "accept_contact" est déclenchée et un message, méthode "notify_me", est envoyé à l'interface.

III.3 Activités

Une activité est un ensemble de variables d’instances ou locales et de méthodes. Les activités permettent de modéliser les différentes étapes qu’un utilisateur pourra effectuer dans l’application. Par exemple, l’activité de *shopping* définie à l’intérieur de la vue *Customer* montrera le catalogue, des promotions, une collection d’objets sélectionnés (c’est-à-dire un chariot) et des boutons permettant aux utilisateurs de spécifier un budget maximum, d’ajouter des produits dans le chariot, de passer des commandes, c’est-à-dire de changer d’activité ou de quitter l’application. Toutes ces fonctionnalités sont définies ainsi :

activity	shopping
includes	catalog, promotions, caddy, search() ... , goto_order(), add_to_caddy(), quit()

Exemple H: *Activité achat de la vue Client*

L’activité *search* comporte trois variables et quatre méthodes. Une feuille de style par défaut est attachée à chaque activité. La spécification générale d’une activité peut être trouvée en section A.4.

Du point de vue de l’utilisateur, chaque activité correspond à un document hypertexte contenant des données et des boutons.

Il est à noter que, bien que les activités ne voient qu’une partie de la vue, comme définie par la spécification, *toutes* les vues des données actives sont maintenues (au moins virtuellement) par le système. Ce mécanisme permet à différentes activités consécutives de partager des données, ce qui est particulièrement utile quand un utilisateur redémarre une activité après l’avoir suspendue.

III.4 Règles

Une règle est définie classiquement suivant trois composants :

1. L’événement qui déclenche la règle.
2. Les conditions nécessaires au déclenchement de la règle.
3. Les actions lancées par cette règle si les conditions sont satisfaites.

Nous considérons ici des règles actives très communes et relativement simples. Les règles sont spécifiées à l’intérieur de la vue. Les règles sont prises

en charge par le *gestionnaire de règles*. Plus précisément, les composants des règles actives sont définis comme suit :

- Les événements sont des appels de méthode, locale à la vue ou non (e.g. changement d’activité), des opérations sur des variables ou des objets (i.e. écriture, lecture, concaténation, suppression) ou des détections de changement. Ces événements sont globaux pour un type d’acteur. Le mot clé **local** permet de restreindre la portée de l’événement à l’instance l’ayant déclenché.
- Les conditions sont des requêtes XML retournant un booléen.
- Les actions sont des appels de méthode (distante), des opérations sur des variables ou des objets, des notifications ou des traces.

Les règles actives sont illustrées par des exemples simples. Les discussions sur les changements des variables, les notifications ou les traces sont traitées dans la section suivante.

Supposons qu’un client ajoute un nouvel article dans son caddy. Tout d’abord, il sélectionne un article du catalogue. Ensuite il enclenche la méthode “add_to_caddy” pour l’ajouter dans son panier. La définition de cette méthode est donnée par l’exemple E. Lors de l’exécution de cette méthode, un événement portant son nom est alors envoyé. La vue Client peut alors enclencher une action spécifique lors de la réception de ce message, action indépendante du code de la méthode :

```
on local add_to_caddy  
do      update("remaining_budget");
```

Exemple I: *Définition de la règle add_to_caddy dans la vue Client*

A chaque ajout de produit dans le panier du client, cette règle exécute la méthode *update* sur la variable “remaining_budget” afin d’afficher la valeur du budget restant après soustraction du prix du produit ajouté. Le mot clé **local** spécifie que la portée de la règle est “locale” à la vue, c’est-à-dire que seule la vue ayant enclenché la méthode “add_to_caddy” recevra cet événement.

La puissance des règles dans notre langage est double. (1) Elles permettent de spécifier des actions dynamiquement sans recharger toute l’application, au contraire des méthodes. Il faut, en effet, recompiler toute l’application pour prendre en compte des changements du code des méthodes. (2) De plus, comme la portée des événements est globale à l’application ce mécanisme

permet l'appel de méthode distante. L'exemple G montre un tel cas lorsqu'un client veut contacter pour la première fois un vendeur.

L'exemple J intervient tout de suite après cette négociation. Cette règle dans la vue Client est déclenchée par une méthode portant le nom *message_to_customer* dans la vue Vendeur. Elle permet au vendeur de communiquer, par le biais de message, avec son client. Si un dialogue bilatéral est souhaité, il suffit d'ajouter une méthode identique, disons *message_to_vendor* dans la vue Client et une règle dans la vue Vendeur affichant le message envoyé.

```

on message_to_customer
if sender.name == Vendor
do notify_me(Vendor + ": " + message);
```

Exemple J: Règle affichant le message du Vendeur dans la vue Client

Dans ce cas, la clause **if** sert à filtrer les messages afin de n'afficher que ceux du Vendeur que le client a contacté, et non pas ceux d'un autre vendeur. Les événements étant globaux à l'application, tous les clients reçoivent le message du vendeur.

Il est à noter que le langage ne permet pas de détecter des cycles lors des déclarations des règles. Un mécanisme de contrôle peut être ajouté dans le "Gestionnaire de Règles" comme dans [72]. Cette fonctionnalité n'a pas été implantée.

III.5 Autres aspects du langage

Cette section présente les autres aspects du langage. Les problèmes de lecture/écriture des variables, des droits d'accès et de la propagation des changements, ainsi que la notification et la surveillance des variables, seront abordés. Pour finir, une discussion sur la spécification et l'utilisation d'un journal est proposée.

III.5.1 Lecture et Ecriture

Nous considérons maintenant deux problèmes apparentés à la lecture ou à l'écriture. Le premier à résoudre est le degré de matérialisation des variables d'instance. Le second est lié aux écritures et aux transactions.

III.5.2 Matérialisation et lecture

Le système peut décider de matérialiser soit entièrement l'arbre DOM associé à la variable lors de son initialisation, soit partiellement. En cas de matérialisation partielle, le système peut, par exemple, matérialiser les deux premiers niveaux de l'arbre DOM de la variable d'instance et matérialiser les autres niveaux seulement lors de leurs accès.

Par défaut, les variables d'instance sont entièrement matérialisées lors de l'initialisation de la vue. Aussi, lors de l'accès à une variable d'instance, tous les éléments spécifiés dans la clause *with* contenus dans le même document sont lus. Les éléments accessibles par référence vers d'autres documents seront seulement chargés par une demande explicite. La même philosophie est suivie lors de la lecture d'un élément spécifié par une clause *element*. De plus, certaines applications peuvent avoir des exigences différentes :

1. Considérons, par exemple, une application qui permet à des utilisateurs de charger des rapports pour pouvoir ensuite travailler chez eux déconnectés du gestionnaire de données. Dans ce cas, le système devra charger entièrement le rapport. Supposons maintenant que ce rapport contienne des citations bibliographiques référant d'autres documents bibliographiques. Le système devra alors aussi charger ces documents immédiatement puisque la connection risque de ne plus exister quand l'utilisateur voudra accéder à l'un de ces documents.
2. Considérons maintenant une application de cotations boursières. Nous ne voulons pas charger à l'avance les cours, puisque de telles informations deviennent rapidement obsolètes. Dans ce cas, il est préférable d'attendre une requête explicite de l'utilisateur pour charger de telles données.

Pour contourner la lecture par défaut du système, nous ajoutons deux modes de lecture, à savoir *deferred read* ou *immediate read*. Dans le premier mode, le système charge les éléments seulement à la demande, alors que le second indique que les éléments devront être chargés lors de leur premier accès. Le mot clé *read* peut être remplacé à n'importe quel moment par l'un de ces deux mots clés, plus spécifiques, lors de la spécification de la vue.

III.5.3 Ecriture et Transaction

Une fois que de telles données sont matérialisées et chargées dans l'interface du client, l'utilisateur peut les voir, les modifier, en accord avec les droits d'accès spécifiés. Observons que ces changements ne sont pas propagés vers

la base de données à moins d'une demande explicite de la part de l'utilisateur, comme par un appel de la méthode *write* qui est une part de l'interface de la vue. Le problème de la propagation des changements est considéré en détail en section III.5.5. Pour l'instant, seuls les problèmes de transactions sont abordés.

Par défaut, une vue n'est pas en mode transactionnel. En utilisant les méthodes de base de la classe *ActiveView*, une vue peut démarrer une transaction et la terminer par une validation ou une annulation. Les lectures sont permises en dehors des transactions; donc, par défaut, toutes les lectures sont "sales", i.e. aucun verrou n'est posé. Pour les changements, toutes les mises à jour venant d'un appel de méthode, dans la base XML, provoquées par une vue, doivent être effectuées au travers d'une transaction. Si une mise à jour est demandée comme conséquence de l'appel d'une méthode et que la vue n'est pas dans un mode transactionnel, une erreur est provoquée. La seule exception provient de l'appel explicite de la méthode *write* de la part d'un utilisateur. Dans ce cas, si la vue n'est pas encore dans un mode transactionnel, alors le système démarre automatiquement une nouvelle transaction pour la durée de l'opération d'écriture.

III.5.4 Droits d'accès

Les droits d'accès doivent être relativement sophistiqués dans le type d'applications web envisagées. Le langage offre donc le moyen d'attacher des prédicats d'accès à n'importe quelle variable ou méthode de la vue. Ce prédicat peut utiliser, par exemple, les valeurs de données courantes de la vue ainsi que l'identification de l'utilisateur.

En général, les droits d'accès déterminent le mode des variables d'instance (e.g. lecture/écriture), et si les méthodes peuvent être activées ou non. Par exemple, un client peut ne pas être autorisé à soumettre des commandes si l'approvisionnement de sa carte de crédit est négatif ou s'il appartient à la liste noire des clients. Ces droits d'accès peuvent être implantés en ajoutant une clause pour contrôler les accès dans la spécification des méthodes comme montré ci-dessous :

```

method submit_order() is self.owner.pass_order(neworder)
if      (owner.approved_credit() >= 0 and
          !("blacklisted" in owner.group))
```

Exemple K: *Illustration des droits d'accès sur une Méthode.*

Les conditions de la clause *if* peuvent exécuter des requêtes XML retour-

nant comme résultat un booléen.

Remarque III.5.1 Les droits d'accès peuvent être assez coûteux à évaluer. Dans de nombreux cas, ils dépendront seulement des paramètres de l'initialisation des procédures de la vue. Les droits d'accès peuvent, alors, être évalués une fois pour toutes, lors de l'initialisation de la vue. Cette optimisation peut donner d'énormes gains en performances. Mais, observons qu'il peut être difficile de détecter que c'est, en fait, le cas pour une définition particulière, i.e. que ses droits dépendent seulement de valeurs immuables de la vue. Ainsi, pour indiquer au compilateur que les droits d'accès doivent être évalués lors de l'initialisation de la vue seulement, le mot clé *if* peut être remplacé par le mot clé *static if*, afin de spécifier que le système ne doit évaluer les droits d'accès qu'à l'initialisation de la vue. □

Pour conclure cette section, nous considérons la notification et la surveillance des changements ainsi que la propagation de ces changements.

III.5.5 Notification et surveillance des changements

III.5.5.1 Notification de changements

Les notifications sont basées sur des appels de méthode distante qui peuvent être envoyés par l'interface d'une vue afin de notifier que certains événements (cf. la section précédente) sont apparus. Les changements des variables d'instance constituent un type important d'événements. Dans l'interface par défaut de l'application, la détection du changement d'une variable d'instance (ou d'un objet) aura pour résultat, par exemple, le changement de couleur dans l'affichage du fond de la variable. Les notifications des autres événements peuvent avoir pour résultat l'affichage d'un message à l'utilisateur avec une icône "notification". Ces messages ressemblent aux appels de méthode distante précédemment discutés. En effet, les notifications peuvent être personnalisées de la même manière que les appels de méthode distante.

Supposons que les vendeurs veulent être notifiés de la soumission de commandes importantes par des clients qu'ils ont en charge. La règle suivante de la vue *Vendor* spécifie cette notification :

```

on submit_order(owner,neworder)
if neworder.amount > 10000 and owner in MyCustomers
do notify-me
```

Exemple L: *Notification qu'un client a passé un nouvel ordre*

Dans cette règle, le mot clé *notify-me* spécifie que cet événement particulier doit être notifié à la vue. Dans un certain sens, la vue a créé une souscription à certains événements. Observons que seules les vues qui ont explicitement souscrit seront notifiées.

III.5.5.2 Surveillance des changements

Maintenant, considérons la surveillance des variables d'instance. Le mécanisme de notification de O_2 permet assez facilement de détecter qu'un objet de la base a changé [104, 24, 117]. Si une variable d'instance est définie par une requête complexe, la situation est plus difficile. Considérons la variable d'instance *promos* dans la vue *Customer*. La règle suivante *changed_promos*, indiquant que la variable *promos* a changé, est incluse dans le but de notifier les clients d'un changement dans les promotions :

on	<i>changed_promos</i>
do	<i>notify-me</i>

Exemple M: *Règle notifiant le client d'un changement sur les promotions*

La variable *promos* peut avoir changé à cause d'une promotion ajoutée ou supprimée, s'appliquant à un client particulier. Elle peut avoir aussi changé si une promotion existante a été modifiée. Cette vue maintient aussi la liste des objets dont le changement peut affecter une donnée dérivée. Dans ce cas, le changement peut aussi bien affecter une collection d'objets que chaque élément dans cette collection. Quand un changement possible a été détecté, deux cas apparaissent :

1. La donnée dérivée ne peut être maintenue incrémentalement. Dans ce cas, une notification est créée. Clairement, cette situation peut générer de fausses alarmes.
2. La donnée dérivée peut être maintenue incrémentalement. Une évaluation incrémentale des changements est effectuée dans le style de [10] pour voir si, en effet, un changement est apparu. Aucune fausse alarme ne peut être créée.

Les notifications sont juste des avertissements. Les données peuvent être changées par les utilisateurs en cliquant sur le bouton de *lecture* ou automatiquement par une interface personnalisée (voir section IV). Même lors de l'utilisation de l'évaluation incrémentale et si la nouvelle valeur est connue par le système, elle ne sera envoyée à l'utilisateur que si elle est explicitement

demandée. Il est néanmoins possible d'inclure une règle dans la vue afin de forcer les changements à être envoyés aux clients lorsqu'ils sont détectés. La règle suivante illustre cette possibilité :

```

on  changed_promos
do  promos.read()
```

Exemple N: Règle entraînant une relecture des promotions lors d'un changement.

Dans ce cas particulier, la variable d'instance dérivée est assez simple pour être maintenue incrémentalement. Néanmoins, un tel cas peut être coûteux puisque chaque détection d'un changement possible déclenchera un re-calcule de la variable et son expédition aux clients.

Remarque III.5.2 Le besoin de “surveiller” les variables d'instance ou de les “rafraîchir” quand un changement intervient est rencontré dans bon nombre d'applications. Le langage fournit un raccourci syntaxique pour spécifier de telles caractéristiques, sans avoir à écrire les règles correspondantes. A la place d'utiliser **let** <variable> dans la spécification de la vue, nous pouvons utiliser **let monitored** <variable> ou **let fresh** <variable>. Ces raccourcis génèrent les règles appropriées pour notifier les changements et éventuellement (dans le cas de **fresh**) déclencher automatiquement une lecture quand un changement est détecté. □

III.5.6 Propagation des changements

Dans certains cas, quand une variable de la vue est modifiée et qu'une écriture est requise, nous devons propager le changement de la vue vers la base XML. Dans d'autres cas, quand la base XML change et qu'une lecture est demandée pour des variables dérivées des valeurs précédemment calculées pour la vue, le système doit propager les changements de l'entrepôt vers les vues.

Regardons d'abord les problèmes liés aux *changements des vues*, c'est-à-dire que la valeur d'une variable de vue a changé et que ce changement doit être transmis à l'entrepôt XML. Nous touchons ici un des problèmes critiques des bases de données. La plupart des travaux sur les changements des vues ont porté sur le changement des vues définies par des requêtes relationnelles complexes impliquant des jointures et des projections, e.g. [29]. C'est un problème un peu plus complexe que le nôtre. Ce problème est évité ici en utilisant de manière extensive des objets et en ne permettant pas de

propager les changements des variables définies à partir de requêtes trop complexes. De manière simple, la propagation des changements concernant des données spécifiques à un acteur (nom, adresse) ne pose aucun problème car ces données ne sont pas partagées, au contraire des changements de variables partagées. Dans ce cas, il faut retrouver dans chaque vue, quelles sont les parties visibles de ces variables, et propager le minimum de messages utiles pour changer la valeur de ces variables en minimisant la gêne de l'utilisateur.

Une correspondance entre la base XML et les portions modifiables de la vue est maintenue. Dans le meilleur des cas, une valeur atomique dans la vue (disons une chaîne de caractères) correspond à une valeur atomique dans l'entrepôt, et la modification de la valeur dans la vue est facilement propageable dans l'entrepôt. Dans d'autres cas, une valeur de la vue n'a pas de correspondance exacte. Par exemple, la valeur est définie comme une sélection sur des collections de données. Le système accepte quand même de propager des changements de telles données vers la base dans des cas simples et non ambigus. Dans plusieurs cas, le système rejette simplement les changements au travers des vues à moins que le programmeur de l'application ne fournisse une méthode pour le faire. Dans ce cas, le système n'est pas responsable de la propagation correcte des changements.

Nous mentionnons maintenant deux cas importants où les changements des vues sont propagés vers la base. De plus amples informations sur le sujet peuvent être trouvées dans [9] :

1. Stricte correspondance entre deux collections. Quand chaque élément dans la base a un correspondant dans la vue, e.g. un ensemble d'objets et le même ensemble d'objets avec une interface différente spécifiée par la vue, les changements des éléments sont propagés quand ils sont possibles, i.e. quand la propagation est définie au niveau de l'élément. Si un élément est enlevé d'une collection, nous l'enlevons de la collection de l'entrepôt. Si un élément est inséré, nous construisons l'élément correspondant (éventuellement avec une valeur par défaut) si possible.
2. Correspondance partielle entre les collections. Ce scénario est possible si la vue est obtenue en filtrant seulement quelques éléments de l'entrepôt (parfois en les restructurant). La principale difficulté apparaît lors de l'insertion d'un élément (de la vue). Il faut alors vérifier que l'élément de la vue, qui a été inséré lors de la propagation du changement de la base XML, est bien passé par le test de filtrage. Si ce n'est pas le cas, le changement est simplement rejeté.

Pour illustrer la discussion précédente, considérons une définition de *Customer* où un client peut modifier ses ordres en mettant à jour la variable

d'instance *myorders* définie dans la vue :

let	myorders: (MyOrderElem)*
be	select O
	from O in RepOrders
	where O.buyer = owner
with	O.*
mode	write all except O.buyer

Exemple O: *Définition de la variables d'instance myorders*

Un client peut maintenant changer le résultat du filtrage de l'ensemble entier des ordres, *RepOrders*. Le système maintient une correspondance entre les éléments de *myorders* et ceux de *RepOrders*, permettant la propagation des changements vers la base XML. La suppression de tels éléments ne peut résulter que de la disparition des éléments correspondant à *RepOrders*. La concaténation d'un nouvel élément résulterait de l'ajout d'un nouvel ordre dans *RepOrder*. Observons, qu'un client ne peut modifier l'identité de l'acheteur à cause de la clause *except O.buyer*. De plus, la spécification courante permet, en principe, au client d'ajouter un nouvel ordre comme s'il résultait d'un autre client; en ajoutant simplement la description d'un autre client dans le champ *buyer*. Cette possibilité peut être anticipée en utilisant les règles actives qui seront définies plus tard.

Considérons maintenant le cas de la *propagation d'un changement de la base XML*. Un des problèmes est de recalculer certaines valeurs des vues quand la base de données change. Supposons qu'un utilisateur ait chargé dans sa vue le catalogue et demande de le relire plus tard. La séquence de changements entre deux lectures n'est pas disponible. Une possibilité serait d'utiliser le mécanisme de versionnement de la base XML, s'il existe. Ce serait suffisant pour calculer la différence ou Δ entre les versions de l'utilisateur et de la base. Le gain en communication en envoyant le Δ est loin d'être négligeable. Les versions ne sont pas considérées dans le système *ActiveView*. La section précédente a montré comment, dans certains cas, la liste des changements peut être obtenue. Cette liste de changements permet la maintenance incrémentale des vues.

III.5.6.1 Algorithme de Propagation

Cette partie présente de manière simple la procédure employée par le système afin de propager les changements de la base vers les vues des utilisateurs. Pour cela, prenons l'exemple P qui spécifie la vision qu'un vendeur

a du catalogue.

let fresh	catalog: (CATEGORY)*
be	select m.CATEGORIES.CATEGORY from m in RepCatalog.META_CATEGORIES.META where m.NAME match 'Musical Instruments'
with	self.* X , X.ARTICLE Art, X.NAME Name, Art.PRICE Pri, Art.NAME Name1
mode	deferred read X, immediate read Art Name Name1 write Price

Exemple P: *Définition du catalogue dans la vue Vendeur*

Brièvement, le vendeur voit les instruments de musique du catalogue. La lecture des éléments “article” ainsi que des éléments “nom” est immédiate alors que le chargement du catalogue vers la vue se fait de façon incrémentale par le mot clé **deferred read**. Les requêtes X-OQL étant forcément matérialisées, le problème de matérialisation des données ne se pose qu’entre les vues actives et l’interface client, afin de minimiser les accès réseau lors du démarrage de la vue et d’optimiser la mémoire utilisée par l’applet Java. De plus cette variable bénéficie d’une surveillance. A chaque changement du catalogue, la vue est avertie qu’un changement a eu lieu ainsi que de la nouvelle valeur de cet élément grâce au mot clé **fresh**. Un vendeur ayant des privilèges étendus par rapport à un client, peut changer le prix d’un article.

Le reste de cette section décrit les différentes étapes de l’algorithme de propagation des changements de la vue ainsi que les actions sous jacentes du système pour maintenir à jour cette variable.

Compilation de la vue Lors de la compilation de la spécification vers une application, le compilateur attache à cette variable un automate d’états finis

$$A = \langle Q, Q_0, Q_F, \Sigma, \delta \rangle$$

Q liste des noms définis dans la clause **with** et **let**
(catalog, X, Art, Name, Name1, Pri);

Q_0 état de départ ou nom de la variable définie dans la clause **let**, ici *catalog*

Q_F état d’arrivée de l’automate, ici *Name, Name1, Pri*;

Σ ensemble des expressions de chemin capturées par le langage = $\{1, *\}$

δ clauses de with : $\{catalog.* \rightarrow X, X.ARTICLE \rightarrow Art, \dots\}$

Chaque état de l'automate est annoté avec les droits d'accès correspondants. De plus, une table des documents à surveiller DS est construite. Dans notre exemple, seul le document RepCatalog est à surveiller.

Initialisation de la vue La requête X-OQL est matérialisée au sein de la vue active. L'automate A associé à la variable est chargé en mémoire. Une table de correspondance C (table de hash) à double entrée associant à chaque élément de la requête un identifiant unique, est initialisée. Cette table permet au système de connaître quelles sont les parties de la requête que l'utilisateur a chargé dans son espace de travail. A chaque élément de l'arbre construit par la requête le système attribue un identifiant unique (son oid par exemple) et le stocke dans la table de correspondance C .

Connection de l'utilisateur à la vue Le système construit le sous-arbre visible S de la requête en partant de l'état de départ de l'automate A . Les états accessibles de l'automate (**immediate read**) sont attachés à S . L'identifiant de chaque élément de S (stocké dans C) est ajouté à sa liste d'attributs. Le sous-arbre construit S est alors envoyé à l'utilisateur.

De son côté, l'utilisateur affiche ce sous-arbre et initialise, lui aussi, une table de correspondance D ; élément - identifiant; afin de savoir en permanence quelles sont les données affichées et quelles sont celles chargées en mémoire mais non affichées.

Parcours du catalogue par l'utilisateur A chaque demande explicite d'un sous-arbre de la part de l'utilisateur, l'interface transmet au système l'identifiant id de l'élément à parcourir. Le système déroule alors l'étape précédente en prenant cette fois comme état de départ, l'élément correspondant à l'identifiant id . La correspondance est obtenue grâce à la table de correspondance C .

Détection d'un changement Le Gestionnaire de mise-à-jour notifie à la vue qu'un élément E du document D a changé et que sa nouvelle valeur est V . La vue vérifie si le document D apparaît dans la table des documents à surveiller DS . S'il n'apparaît pas, la notification est ignorée.

Sinon la vue regarde si l'élément a été chargé par le client. Cette détection est effectuée par la table C qui associe un élément chargé par le client à son oid. Si oui, il regarde le mode de surveillance de la variable. Dans notre cas,

le système envoie un message à l'utilisateur disant que la nouvelle valeur de l'élément E est V . En fait, il n'envoie pas l'élément lui-même, mais son identifiant.

L'interface utilisateur doit alors traiter ce message. Elle modifie dans l'arbre DOM associé à la variable, la valeur de l'élément. Si cet élément est actuellement visible à l'écran, elle change par le biais des mécanismes actifs de l'interface, la valeur actuellement affichée.

Si la variable était en mode **monitored**, alors seul l'identifiant de l'élément aurait été envoyé vers l'interface. Celle-ci aurait alors annoté que cet élément a vu sa valeur changée. Lors d'une relecture explicite de cet élément de la part de l'interface la nouvelle valeur est alors transmise.

Des affichages d'écrans (voir figure IV.1 à IV.5) permettent de mieux visualiser ce que les acteurs voient avant et après la modification du prix d'un article de la part d'un vendeur.

III.5.7 Traces

Les bases de données offrent classiquement un mécanisme de journalisation qui est (1) de bas niveau et (2) difficile, voire impossible d'accès. Cependant, il est essentiel de pouvoir avoir une trace de l'exécution de certaines opérations. Cela peut être requis pour des raisons légales, ou pour pouvoir anticiper sur d'éventuels conflits entre les participants, ou pour analyser les motifs d'achat. Dans le système *ActiveView*, les événements et les règles sont le noyau du module de trace. Ce module est l'objet de cette section.

Dans la spécification de la vue, de la même manière que l'on demande de notifier la vue de certains événements, on peut les *tracer*. Le traçage consiste en une notification du traceur. Par exemple, nous pouvons demander de tracer toutes les soumissions de commandes :

<pre> on submit_order(owner,neworder) do trace </pre>

Exemple Q: *Journalisation d'un événement*

Si une telle déclaration est incluse, le Gestionnaire de trace est aussitôt averti des nouvelles commandes et les enregistre dans l'entrepôt. Le traceur enregistre aussi le contexte de tels événements (e.g. la date). Le journal peut être vu comme un historique partiel d'une activité de l'application et peut être aussi interrogé. Par exemple, la requête suivante retourne les commandes d'un client particulier en 1998 :

```
select O
from O in RepTrace.submit_order
where O.neworder.buyer.name = "J. Doe" and O.date = 98
```

Exemple R: *Requêtes sur le journal*

Dans cette requête, *RepTrace* est le point d'entrée de la base XML qui permet d'accéder aux journaux.

Chapitre IV

Personnalisation et interface

En comparaison avec les applications traditionnelles des bases de données, les applications de commerce électronique et, plus généralement, les applications sur le web évoluent très rapidement suivant les nouveaux besoins commerciaux. Pour cette raison, le système *ActiveView* supporte non seulement la définition déclarative des vues et des activités du côté du serveur d'application, mais aussi une exploitation rapide du côté de l'utilisateur final en générant des *interfaces par défaut*. Dans ce chapitre, nous discutons brièvement des différents moyens pour personnaliser l'application et les interfaces ainsi que l'interface par défaut que le système propose. Ce chapitre finit par donner un sketch de l'application *Blacknote* utilisé au cours de cette partie grâce à des captures d'écran.

IV.1 Personnalisation de l'application

Comme montré au cours de cette partie, le système *ActiveView* n'inclut pratiquement pas de code à l'exception des requêtes XML. Bien que nous n'insistions pas sur ce point, il est évident qu'un appel aux méthodes de la base XML implantées dans des langages de programmation conventionnels tels que C++ ou Java, ne peut se faire que dans ces langages. De tels codes peuvent faire partie de l'application de base de données, quelque peu indépendants de la vue elle-même. Les vues actives peuvent être aussi personnalisées de nombreuses manières au prix de l'écriture du code spécifique pour la vue. La redéfinition de la procédure d'authentification ou l'agencement d'une page HTML sont des exemples. Nous présentons les différentes personnalisations dans la suite de cette section.

Personnalisation des composants de la vue. Chaque interface web client communique avec une vue active qui est une instance d'une sous-classe de la classe *ActiveView*. Par exemple, *Customer* est une sous-classe particulière d'*ActiveView*. Les instances de cette classe fournissent les concepts nécessaires pour rentrer dans le système comme (i) le type de la vue (telles *client*, *vendeur*, (ii) le propriétaire, (iii) la date de création, (iv) l'état courant (e.g. *running*, *asleep*), (v) l'activité courante, et (vi) la liste des activités accessibles.

La personnalisation est essentiellement possible en créant des sous-classes et en surchargeant le code des méthodes existantes. Par exemple, la méthode *init* est exécutée quand un nouvel utilisateur rentre dans le système (en créant une nouvelle instance de la vue). Il exécute une méthode privée d'authentification qui vérifie (avec un mot de passe ou des services d'authentification plus sophistiqués) l'identité de l'utilisateur et remplit la variable *owner*. Les deux méthodes *init* et *authentication* sont définies dans la classe *ActiveView* et peuvent être redéfinies dans les sous-classes, e.g. *Customer*, par l'administrateur pour changer, par exemple, les règles de droit d'accès (se référer à la section III pour la définition de telles règles) ou le mécanisme d'autorisation.

Personnalisation de l'Interface. La personnalisation de l'interface peut intervenir à plusieurs niveaux : présentation, redéfinition des méthodes, ou réécriture totale de l'interface.

L'utilisateur choisit entre des activités variées correspondant (par défaut) à différentes pages HTML.

IV.2 Interface utilisateur par défaut

Quand un utilisateur démarre une nouvelle vue active en suivant une URL, e.g.

```
http://www.activestore.com/customer/
```

une page HTML par défaut est affichée et demande alors des informations personnelles avant de proposer les activités qui peuvent être exécutées par l'utilisateur. Par exemple, tous les clients non identifiés peuvent naviguer dans le catalogue (activité de recherche), mais seuls les clients enregistrés peuvent acheter les produits sélectionnés (activité paiement).

Les activités forment la première représentation de l'interface perçue par les utilisateurs finaux (e.g. clients) qui interagissent avec le système *ActiveView*. Chaque activité est représentée par des pages HTML différentes qui

affichent les variables accessibles et les boutons dans la forme simple de boutons HTML. Les pages HTML sont gérées par un applet qui implante les caractéristiques actives pour les appels de méthodes, la modification et la surveillance des variables en communiquant avec le système via le protocole RMI de JAVA [140] (voir l'architecture globale du système Figure II.1, page 30). Chaque variable correspond essentiellement, à une variable de l'applet général qui interagit avec le navigateur via des instructions Java-Script [141].

L'édition de variable est un problème intéressant. Le système *Active-View* repose sur le modèle de données XML et sur le langage de requête XML X-OQL pour représenter et interroger les données. Cette modélisation a pour conséquence que les variables sont des fragments de document XML et qu'elles doivent fusionner dynamiquement dans des documents XML compréhensibles et uniformes.

Les méthodes des vues sont représentées simplement par des boutons. Par exemple, pour ajouter un produit au chariot, l'utilisateur appelle la méthode *add_to_caddy*, laquelle ajoute le produit sélectionné au chariot. Il est nécessaire, par exemple, de pouvoir sélectionner un produit, e.g. dans le catalogue, et ainsi le fournir comme argument de la méthode *add_to_caddy*.

IV.3 Capture d'écran

Les captures d'écrans suivantes proviennent d'une démonstration de l'application [4]. Elles sont incluses dans cette thèse afin de mieux visualiser les différentes possibilités du système *ActiveView* notamment en ce qui concerne la surveillance automatique de variables et la puissance des règles.

La démonstration a pour contexte un magasin de vente d'instruments de musique. Le catalogue XML a été fourni par un magasin français de vente en ligne. L'application définit deux vues, la première pour les clients, la seconde pour les vendeurs. Les captures d'écrans qui suivent illustrent une session d'un client en train de regarder le catalogue et de communiquer avec le vendeur.

Les actions qu'un client peut entreprendre sont :

1. définir un budget maximal
2. parcourir le catalogue.
3. contacter des vendeurs.
4. ajouter des articles à un panier dans la limite d'un budget spécifié
5. vérifier le budget restant

6. créer et valider un ordre

Les vendeurs quant à eux peuvent :

1. modifier le catalogue (le prix des articles)
2. convaincre un client d'acheter

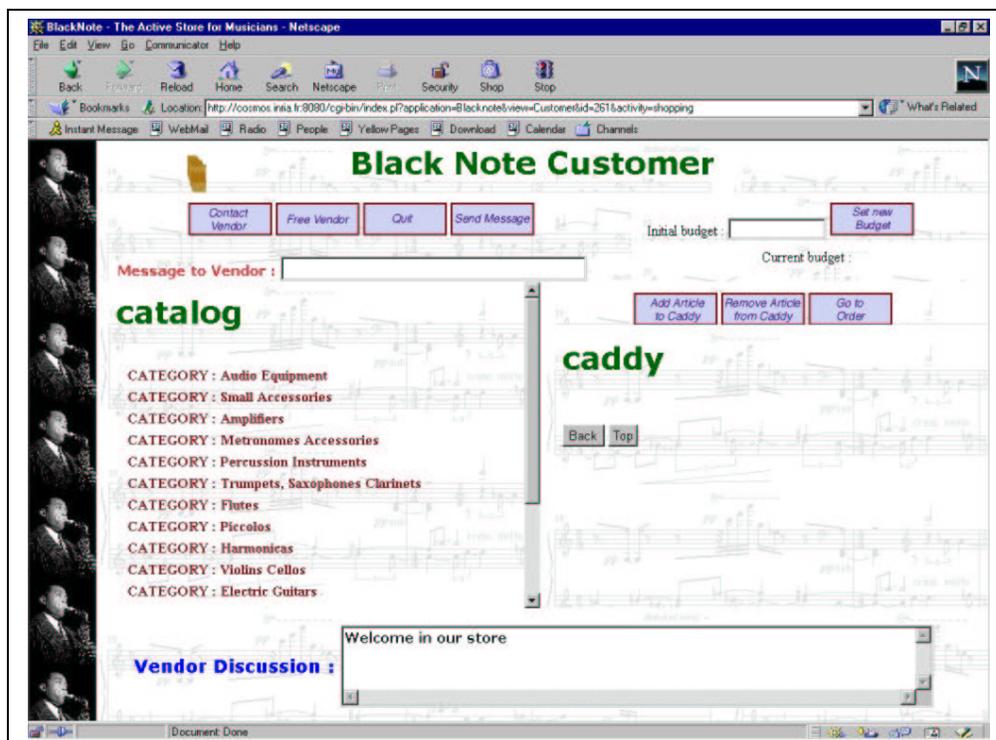
Certains composants de ces vues ont été définis au cours de la section précédente.

IV.3.1 Vue initiale d'un client

La première capture d'écran, figure IV.1, montre l'écran d'un client qui est entré dans le magasin électronique et a commencé l'activité "shopping". La page HTML affichée contient toutes les variables et les méthodes définies dans cette activité. La fenêtre de droite, ayant pour titre "catalog", correspond à la variable catalogue spécifiée dans l'exemple A. La valeur de cette variable est un document XML obtenu par la requête X-OQL incluse dans la spécification. Observons les différents modes, pour les éléments, du résultat de la requête. Le catalogue est en mode de lecture différée, i.e. les nœuds du document sont chargés dans l'interface à la demande durant le parcours du catalogue de la part du client. Les seules exceptions sont les nœuds correspondant aux éléments de type "ARTICLE" et "NOM" qui sont chargés immédiatement avec leur parent. Les boutons affichés correspondent aux méthodes de la vue. Par exemple, le bouton "Add Article to Caddy" correspond à la méthode "add_to_caddy" (exemple E page 36). Avant de cliquer sur ce bouton, le client doit parcourir le catalogue et sélectionner un article, qui devient l'argument de l'appel de méthode déclenché.

let monitored	catalog: (CATEGORY)*
be	select m.CATEGORIES.CATEGORY from m in RepCatalog.META_CATEGORIES.META where m.NAME match 'Musical Instruments'
with	self.* X , X.ARTICLE Art, X.NAME Name, Art.NAME Name1
mode	deferred read X, immediate read Art Name Name1

Exemple A: *Définition du catalogue dans la vue Client*

FIG. IV.1: *Vue initiale d'un client*

IV.3.2 Vue d'un client après quelques achats

La seconde capture d'écran, figure IV.2 montre l'état du client après avoir défini un budget initial, parcouru le catalogue et ajouté quelques articles à son panier (caddy). Observons que le budget restant est calculé automatiquement comme la différence entre le budget initial et la somme des prix des articles du caddy.

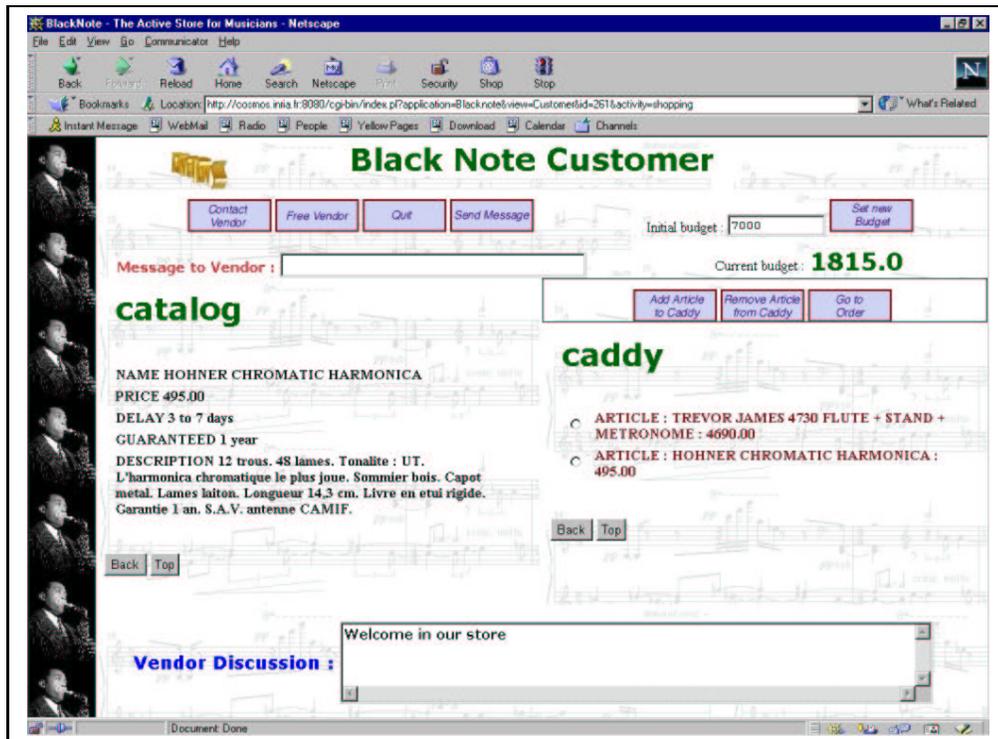


FIG. IV.2: Vue d'un client après quelques achats

IV.3.3 Discussion entre un client et un vendeur

Supposons que le client ait appuyé sur le bouton "contact_vendor". Cette action a créé une notification qui a déclenché une règle de la vue vendeur (exemple G page 37). Avant d'exécuter l'action, le gestionnaire de règles vérifie si le vendeur est bien le responsable pour la catégorie correspondante "category == c" et s'il n'est pas déjà en discussion avec un autre client "customer==null". La règle initialisera alors la variable "Customer" de la vue Vendeur et exécutera la méthode "accept_contact" pour informer la vue du client que la demande de contact a été acceptée. La capture d'écran de la

figure IV.3 montre que le client est en contact avec un vendeur et a demandé de diminuer le prix de l'article affiché.

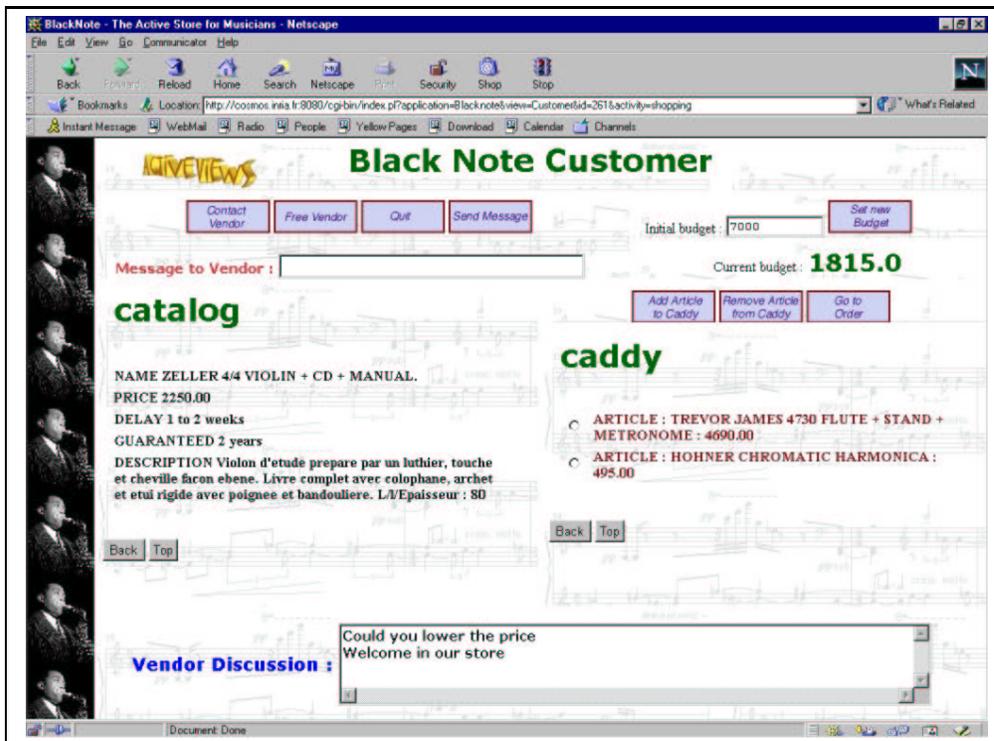
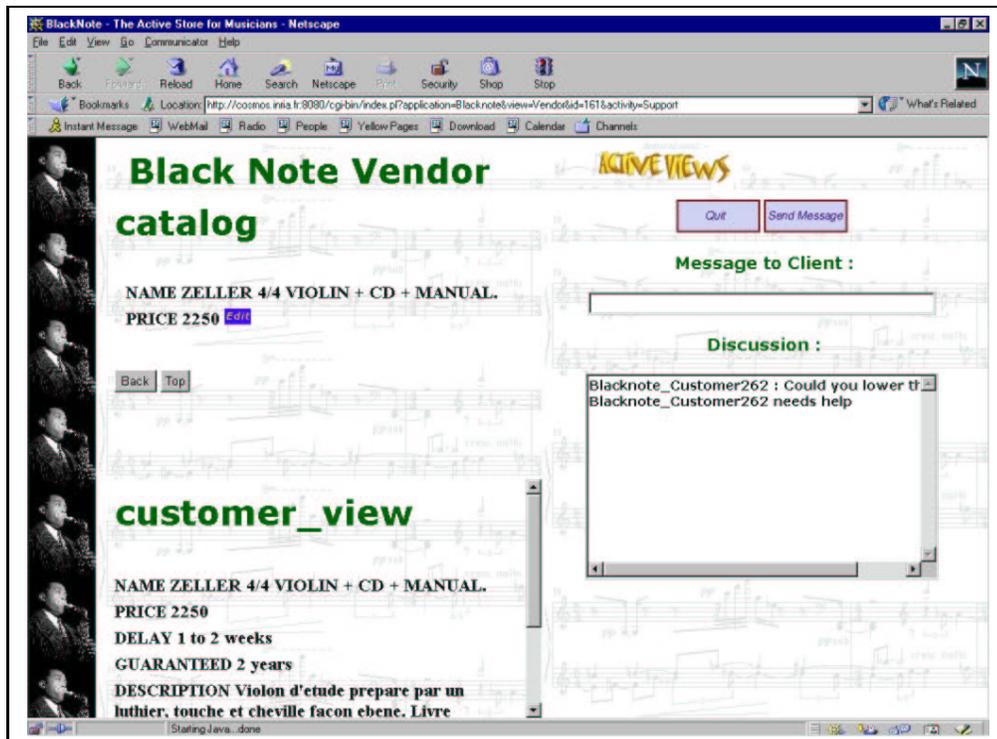


FIG. IV.3: Discussion entre un client et un vendeur

IV.3.4 Vue d'un vendeur

La capture d'écran de la figure IV.4 montre l'écran d'un vendeur après avoir reçu une question de la part du client (Discussion: "Could you lower the price"). Observons que le vendeur est automatiquement informé de la vision actuelle du catalogue de la part du client (fenêtre "customer_view"). Les informations de cette fenêtre sont automatiquement rafraîchies par une règle active déclenchée lors des différentes étapes du parcours du catalogue de la part du client qui crée alors des notifications capturées par la vue du vendeur. Puisque le vendeur a la permission de changer le prix d'un article (exemple P page 48), il va changer le prix.

FIG. IV.4: *Vue d'un vendeur*

IV.3.5 Vue finale d'un client

La capture d'écran finale, figure IV.5 montre l'écran du client après la modification du prix par le vendeur. Observons le message à côté de l'attribut "PRICE". En fait, le prix n'a pas encore été changé et le client a simplement à appuyer sur le champ correspondant pour obtenir le nouveau prix.

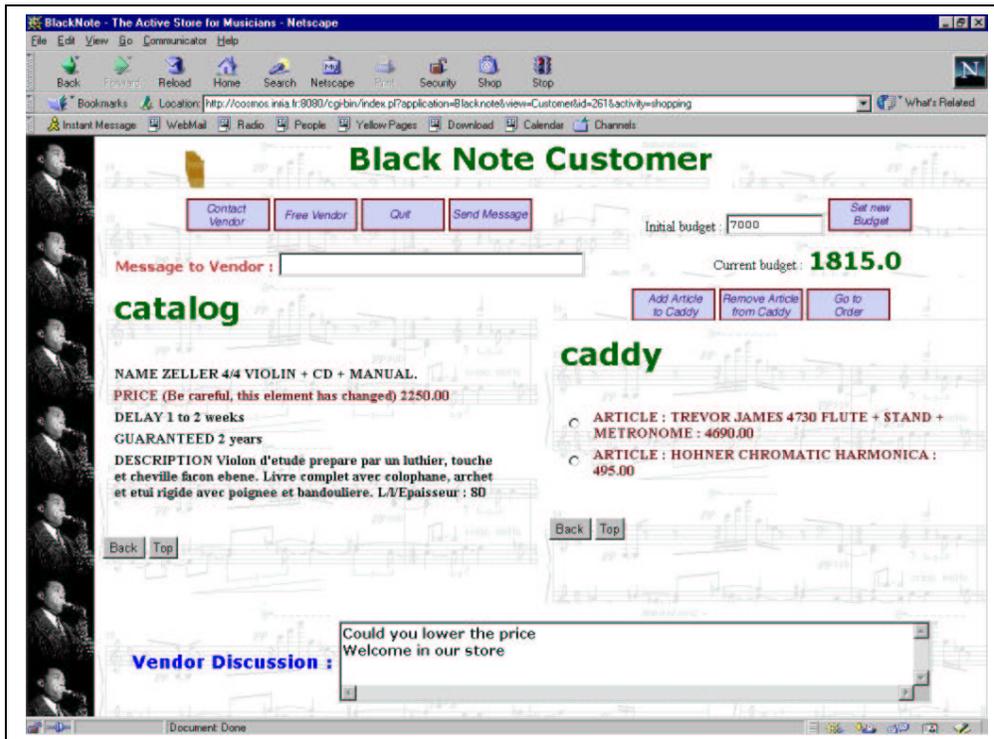


FIG. IV.5: *Vue finale d'un client*

Deuxième partie

Données Internet : Acquisition dans Xyleme

Cette partie étudie le problème de la détection des changements dans le monde Internet. Ce travail a pour cadre le projet de recherche Xyleme [176]. Nous allons, tout d'abord, présenter les axes de recherche de Xyleme dans le chapitre V pour nous intéresser ensuite dans le chapitre VI aux solutions envisagées pour contrôler les changements dans les données se trouvant sur Internet.

Chapitre V

Présentation de Xyleme

Ce chapitre présente les différents modules de Xyleme en commençant par son architecture fonctionnelle en Section V.1 puis en énumérant les différentes composantes logiques de Xyleme (Section V.2 à V.6) :

- Système de stockage.
- Processeur de requêtes.
- Classification sémantique des données.
- Système de vue.
- Système de gestion des versions.
- Système de souscription aux événements.
- Système d’acquisition et de rafraîchissement des données.

Le module d’acquisition et de rafraîchissement des données fera l’objet d’un chapitre entier. Notre contribution aux mécanismes de version et de souscription justifie le développement un peu plus élaboré des sections correspondantes.

V.1 Architecture fonctionnelle de Xyleme

Cette section présente l’architecture générale du système. Le système est organisé de manière fonctionnelle en quatre niveaux (voir Figure V.1) :

- le niveau physique (l’entrepôt Natix optimisé pour le stockage de données au format d’arbres et un gestionnaire d’index);

- le niveau logique (acquisition des données et processeur de requêtes);
- le niveau applicatif (gestionnaire de changements et intégration sémantique des données);
- le niveau permettant d'interfacer l'application avec les clients de Xyleme au travers du web.

Tous les modules sont développés en C++, pour des raisons de performances, et communiquent entre eux via une interface Corba [142, 118] à l'exception des interfaces client écrites en JAVA afin d'améliorer leur portabilité. Les clients de Xyleme fonctionnent sur des navigateurs web standard utilisant des applets Java. La Figure V.1 distingue (logiquement) plusieurs types de machines.

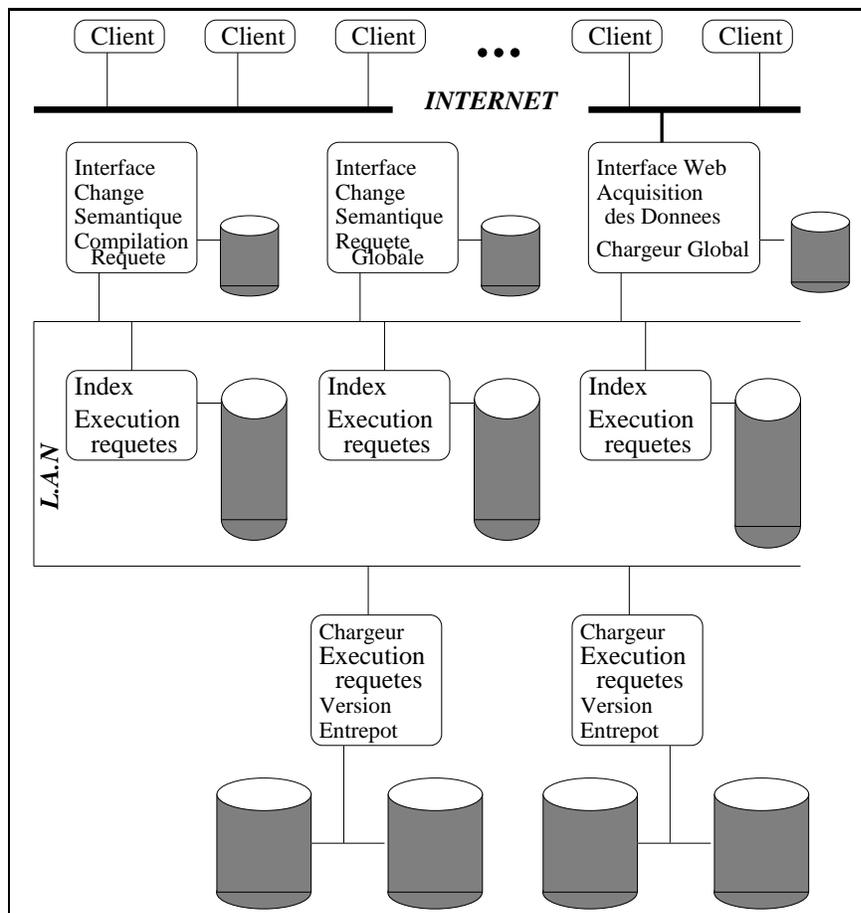


FIG. V.1: Architecture globale de Xyleme

V.2 Natix : L'entrepôt

Xyleme nécessite l'utilisation d'un stockage efficace et réutilisable de données XML. C'est pourquoi nous avons utilisé la plate-forme Natix (**NAT**ive **X**ML) [92] développée à l'Université de Mannheim car elle était la première à exister.

Typiquement, trois approches peuvent être utilisées pour stocker de telles données.

La première solution consiste à stocker les fichiers XML par une séquence d'octets. Pour des flux importants, des mécanismes sont utilisés pour distribuer le flux d'octets sur des pages disque. Ce mécanisme peut être un système de fichiers ou un gestionnaire de BLOB (Binary Large Objects) dans un SGBD. Cette méthode est très rapide dans le cas du stockage ou du chargement de documents entiers. Cependant, accéder à la structure du document n'est alors possible que par le biais d'une analyse syntaxique [8].

La seconde solution consiste à modéliser et à stocker les documents ou arbres de données en utilisant les SGBDs traditionnels et leur modèle de données [107, 134]. Dans ce cas, l'interaction avec les bases de données structurées du même SGBD est facile. Reconstruire des documents ou des parties d'un document, par exemple pour en obtenir une représentation textuelle, est plus lent que par la méthode précédente : la restructuration d'un document et de son arbre abstrait de syntaxe demandent de retrouver (peut-être) des milliers de tuples dans la base. De plus, une traduction de format à la YAT [56] peut être nécessaire. D'autres représentations requièrent des opérations de correspondance complexes pour reproduire une représentation textuelle [134]. De plus, l'élimination de doublons peut être requise [64]. En général, cette approche introduit des niveaux additionnels dans le SGBD entre les couches logique et physique, ralentissant le processeur de requêtes.

Deux tentatives ont pour souci de rassembler le meilleur des deux précédents mondes : la redondance et l'approche hybride.

La redondance prend le meilleur des deux mondes. Les données sont stockées dans deux SGBDs redondants l'un plat et l'autre structuré [177]. Les changements sont propagés soit dans un seul entrepôt, soit dans les deux. Cette technique permet de retrouver rapidement les données, mais diminue les performances des mises à jour et implique des problèmes de consistance des données.

Dans les approches hybrides, un certain niveau de détail des données est configuré comme "seuil". Les structures plus grosses que ce seuil sont visibles dans la partie structurée du SGBD et les structures plus fines sont stockées par une séquence d'octets [30, 101].

Natix utilise l'approche hybride. Les données sont stockées sous forme

d'arbres, jusqu'à une certaine profondeur, à partir de laquelle les flux d'octets sont utilisés. De plus, quelques structures minimales sont utilisées dans la partie déstructurée afin de faciliter l'accès avec des niveaux de granularité variables.

Natix est construit, de manière standard, au-dessus d'un gestionnaire d'enregistrements avec des pages de taille flexible et des segments (séquence de pages non forcément contiguës). Les fonctionnalités de base pour la gestion des enregistrements (e.g. "bufferisation") ou accès rapide (e.g. index) sont fournies. Un des aspects intéressants de Natix est la répartition des arbres XML dans des pages.

Les performances de Xyleme dépendent fortement de l'efficacité du stockage. Natix, en tirant parti des particularités arborescentes des données XML, offre des performances appropriées.

V.3 Processeur de requêtes

Cette section présente l'évaluation statique d'une requête par le processeur de requêtes de Xyleme.

L'évaluation des requêtes sur des données semi-structurées a été étudiée de manière approfondie durant ces dix dernières années [6]. Les techniques diffèrent légèrement suivant le système utilisé pour stocker les données : orienté objet, dédié ou relationnel. Toutefois, aucune des approches existantes ne permet de passer à l'échelle du web. Nous considérons ici des milliards de documents, Google [158] a récemment indexé plus d'un milliard trois cents millions de documents, et traite des millions de requêtes par jour. Par conséquent, l'évaluation de requêtes acquiert dans Xyleme une toute autre importance.

De façon classique, le processeur de requêtes transforme une requête en un arbre algébrique. La nouveauté de l'approche réside dans l'utilisation :

- d'une distribution des plans algébriques sur différentes machines afin de passer à l'échelle du web (voir Figure V.1);
- de l'implantation efficace d'un opérateur algébrique complexe, appelé *Pattern Scan*, qui permet de filtrer une collection de documents suivant un motif d'arbre et d'extraire les informations des documents sélectionnés.

De plus amples informations sur ce travail peuvent être trouvées dans [16].

V.4 Intégration sémantique des données

Les requêtes dans Xyleme tirent parti de la structure des documents. Dans quelques domaines, des types de documents standards ou DTD ont été définis (Dublin Core [156], ou [161]), mais la plupart des compagnies qui publient en XML ont leurs propres définitions. Dans ce cas, une simple requête peut avoir pour portée des collections de documents XML ayant des dizaines de types différents. L'intégration sémantique aide l'utilisateur à poser des requêtes sans avoir à connaître les très nombreuses DTDs des documents impliqués par ces requêtes. Xyleme fournit un mécanisme d'intégration sémantique, qui permet à l'utilisateur de poser une requête sur une structure simple, résumant un domaine d'intérêt, au lieu de plusieurs schémas hétérogènes. Ce mécanisme se base sur une classification sémantique des documents [127] et sur une intégration de données hétérogènes à l'aide de vues [57].

V.4.1 Classification sémantique des documents

La méthode choisie génère de manière semi-automatique la classification des documents par centre d'intérêt commun. Ce regroupement est réalisé en utilisant différentes techniques provenant de l'intelligence artificielle et de l'apprentissage du langage naturel.

La première tâche est de classifier les DTDs dans des domaines (e.g. tourisme, finance), en se basant sur des analyses statistiques de similarités entre les mots trouvés dans différentes DTDs et/ou documents. Les similarités sont définies par le biais d'ontologies ou de thésaurus tel que Worldnet [146].

Une seconde tâche est de choisir une "DTD abstraite", c'est-à-dire une structuration virtuelle de tous les documents de ce domaine. Les documents seront alors regroupés suivant cette structuration virtuelle dans un cluster concret. L'utilisateur choisira d'abord un cluster concret et posera ensuite sa requête en se servant de la DTD abstraite de ce cluster concret. Il suffit ensuite de faire correspondre les DTDs présentes dans l'entrepôt appelées "DTD concrètes" à ces DTDs abstraites. De cette manière une requête de l'utilisateur pourra être traduite, au travers de la DTD abstraite, en requêtes XML sur les clusters de documents stockés dans Natix. Chaque élément est identifié par son chemin à partir de la racine de la DTD (e.g, *hôtel/adresse/ville*). Le problème revient alors à faire correspondre à un chemin de la DTD abstraite, l'ensemble des chemins correspondants dans les DTDs concrètes. De façon évidente, tous les tags appartenant à un chemin (i) ne sont pas toujours des mots du langage usuel et (ii) n'ont pas tous la même importance. Concernant le premier point (i), des techniques de langue naturelle (e.g, pour reconnaître les abréviations ou les mots composés) ainsi qu'un thésaurus, sont utilisés.

Pour le second point (ii), une interaction humaine est maintenue lors du processus. Notamment, le concepteur de la DTD abstraite peut indiquer au générateur de correspondance que tel tag n'est pas particulièrement important dans le chemin, ou au contraire, essentiel. Considérons, par exemple, le chemin *hôtel/adresse/ville*. Nous pouvons imaginer que le tag *hôtel* est important mais que le tag *adresse* l'est moins. Dans ce cas, le système sera capable de faire correspondre *auberge/ville* avec *compagnie/adresse/ville*, mais pas *hôtel/adresse/ville* car le mot clé *hotel* n'apparaît pas dans le chemin d'origine. Le lecteur est renvoyé à [127] pour de plus amples renseignements sur ce mécanisme.

Une fois la *DTD abstraite* définie comme structure pour un domaine particulier, la tâche suivante est de faire correspondre les éléments de la DTD abstraite aux éléments des DTD concrètes correspondant aux documents XML stockés dans Natix. Les DTDs sont appelées *DTD concrètes*.

V.4.2 Vues

Comme dans un système de base de données classique, les vues dans Xyleme sont utilisées pour manipuler les données provenant de collections différentes au travers d'une structure unique et pratique. Néanmoins, les données dans Xyleme ne sont pas stockées dans des relations par des applications propriétaires mais rapatriées par les robots du web. Pour ordonner cette quantité énorme de documents, Xyleme a choisi de les relier par un mécanisme de classification qui partitionne les documents dans des clusters thématiques. Chaque cluster étant défini par une DTD abstraite. Ainsi, le cluster "art" contient tous les documents relatifs à l'art. Une analogie peut être faite avec modèle relationnel, les clusters concrets jouant le rôle des relations. Ainsi, les vues [3] dans Xyleme combinent plusieurs clusters concrets, ou DTDs abstraites, dans un cluster abstrait. De surcroît, alors que les tuples d'une relation partagent un type commun, un cluster abstrait est une collection de documents hautement hétérogènes. Ainsi, une requête est définie comme une union de plusieurs requêtes sur différents sous-clusters.

Les utilisateurs créent une vue (i) en sélectionnant le domaine de la vue, i.e. les différents clusters (ii) en définissant le schéma/DTD de la vue (DTD abstraite). Alors, l'outil de génération de vue extrait, de façon semi-automatique, les nombreuses DTDs du domaine choisi et, à l'aide d'un thésaurus [146] trouve les différents mappings entre le schéma de la vue et les schémas des clusters définis par l'outil de classification. Ces mappings utilisent la technique de correspondance "chemin-à-chemin" pour faire le lien entre le schéma de la vue et les DTDs abstraites des clusters. Cet outil de génération de vue ainsi que de réécriture de requêtes est expliqué dans [57].

V.5 Mécanisme de versions

Cette section décrit un mécanisme de versionnement développé par Xyleme [105], auquel j'ai partiellement contribué. Ce mécanisme utilise une représentation centrée sur les changements, par opposition aux versions dans les bases de données qui sont en général centrées sur les données [51].

Dans le contexte de Xyleme, le point de départ est une séquence de valeurs d'un document XML au cours du temps, obtenues du Web (ou calculées dans le cas de requêtes évaluées régulièrement au cours du temps). Chaque nouvelle version est comparée avec la version précédente par un algorithme de *diff* [58] qui fait correspondre les nœuds entre les deux versions, c'est-à-dire les "identifiants". Ces identifiants persistants sont appelés XId (pour Xyleme Ids). Ceci permet de construire la différence ou le "delta" entre les deux versions. Pour chaque document géré par le mécanisme de version, la dernière version du document est stockée ainsi que la *séquence de deltas* dits "*complet*" (voir section V.5.3 pour leur définition).

La section V.5.1 présente le modèle logique utilisé. Ensuite, des *ensembles* d'opérations fondamentales décrivant globalement les changements sont présentés en section V.5.2. La section V.5.3 présente la notion de deltas complets qui pallient certains défauts des deltas. Les deltas complets sont en un certain sens connectés aux journaux dans les bases de données [128]. La section V.5.4 introduit les identifiants logiques ou persistants, utilisés pour identifier un nœud au cours du temps, appelés XId. Finalement, une politique de stockage est décrite en section V.5.5.

V.5.1 Modèle Logique

Cette section introduit un modèle simplifié de changements des documents XML. Nous définissons ensuite des opérations fondamentales. Une séquence de telles opérations définit un *script d'édition*.

V.5.1.1 Modèle de Données

La représentation des changements utilise des arbres dont tous les nœuds ont des identifiants. Par souci de clarté, seul un modèle simplifié, suffisant pour décrire les aspects principaux des changements, sera présenté. Cependant, l'implantation dans Xyleme utilise le modèle complet de XML. Soit \mathcal{Q} l'ensemble infini des chaînes de caractères.

Définition V.5.1 Un arbre XML est une paire $\langle t, v \rangle$ où (i) t est un arbre fini ordonné, suivant l'ordre induit par le document XML, avec des nœuds

prenant leurs valeurs dans \mathbb{N} ; et (ii) v une fonction qui associe une valeur (qui peut être nulle) dans \mathcal{Q} à chaque nœud de t .

□

Le modèle complet fait la distinction entre les nœuds textes, éléments et attributs car les opérations de changements varient légèrement suivant les types des nœuds étudiés. La valeur d'un nœud élément est son label, alors que pour un nœud texte, c'est une chaîne de caractères de type PCDATA.

V.5.1.2 Opération sur les arbres

Les opérations fondamentales pour modifier un arbre XML T sont :

1. $delete(m)$ qui supprime le sous-arbre XML de T de racine m , où m n'est pas la racine de T .
2. $insert(n,k,T')$ qui insère l'arbre XML T' comme le k -ème fils du nœud n de l'arbre T .
3. $move(n,k,m)$ qui déplace le sous-arbre XML ayant pour racine m comme étant le k -ème fils du nœud n de T .
4. $update(m,v)$ qui change la valeur du nœud m de T à v .

Avant d'appliquer un script d'édition, il faut vérifier des règles de cohérence. Par exemple, pour l'insertion, T doit avoir un nœud n avec au moins $k-1$ fils, et T ne doit pas avoir de XId en commun avec T' . Un script d'édition est une séquence cohérente de telles opérations.

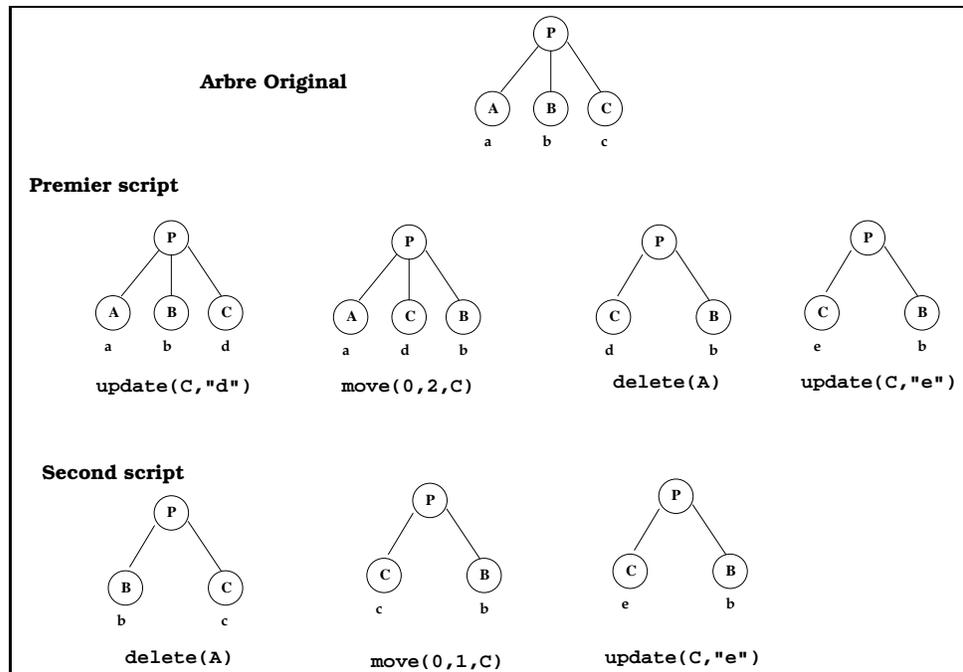
Remarque V.5.2 Considérons un arbre T de racine P, avec les fils A, B, C ayant pour valeur "a", "b", "c" et les scripts d'édition suivants :

1. $update(C,"d"); move(0,2,C); delete(A); update(C,"e")$.
2. $delete(A); move(0,1,C); update(C,"e")$.

Ces deux scripts ont le même effet sur T . La Figure V.2 présente l'arbre T original ainsi que les différents arbres résultant des deux scripts précédents.

□

En conséquence, il existe, en général, plusieurs scripts pour décrire les changements d'un arbre T à un arbre T' . La représentation avec des *deltas*, décrite ci-dessous, présente l'avantage de décrire un changement quelconque par un *ensemble unique* d'opérations.

FIG. V.2: Différents Scripts d'édition sur un arbre T

V.5.2 Deltas

Un delta est un *ensemble* d'opérations delete (D), update (U), insert (I) et move (M) définies plus bas. Etant donné deux versions d'un document avec des nœuds identifiés, il existe un unique delta minimal décrivant les opérations qui transforment une version dans l'autre [105]. Le document résultant de l'application du delta est donc indépendant de l'ordre d'exécution des opérations de l'ensemble. Les paramètres des opérations sont des positions absolues dans l'un des deux documents, et non pas relatives comme dans les opérations du script d'édition. Par exemple, l'opération de suppression d'un sous-arbre se réfère au document V_1 , alors que pour l'insertion d'un sous-arbre XML, le document de référence est V_2 . Plus de précisions sont données dans [105].

Soit deux arbres T, T' , un *delta* Δ de T vers T' est l'*ensemble d'opérations* définies comme suit :

deletes D: pour chaque sous-arbre de racine n qui existe dans T et pas dans T' et tel que son parent est dans T' , alors $D(n)$ est une opération dans Δ . Δ ne contient pas d'autre opération de suppression.

inserts I: pour chaque nœud n qui est dans T' et non dans T et tel que

son parent est dans T , alors $I(m, k, T')$ est dans Δ , où m est le parent de n , k son rang (position dans le parent) de n et T' est le nouveau sous-arbre ayant pour racine n . Δ ne contient pas d'autre opération d'insertion.

updates U: pour chaque nœud n ayant pour valeur v' dans T et v dans T' avec v différent de v' , alors $U(n, v)$ est dans Δ . Δ ne contient pas d'autre opération de mise à jour.

moves M: $M(n, k, m)$ est dans Δ pour tout m qui se trouve dans T et T' à différentes positions. Sa nouvelle position dans T' est connue comme le k -ème fils de n .

absent nodes pour chaque nœud n appartenant à T et à T' , l'ensemble des nœuds de n n'ayant ni été supprimés, ni insérés, ni déplacés et qui sont restés les mêmes dans T et T' .

Le delta correspondant aux scripts d'édition donnés à la figure V.2 est le suivant :

- delete(A), move(0, 1, C), update(C, "e")

La Figure V.3 ainsi que la Table V.1 présente les deltas entre deux versions d'un même document.

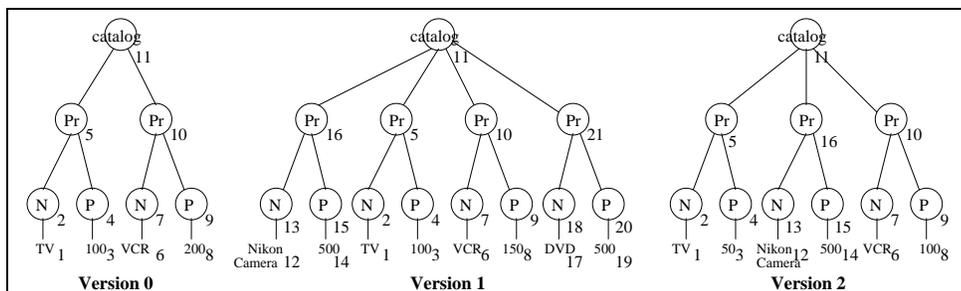


FIG. V.3: Séquences de versions d'un même document XML avec les nœuds identifiés

Il n'est pas possible de reconstruire à partir d'un delta le script d'édition, sans utiliser la version originale du document. De même qu'il est impossible d'inverser des deltas, i.e, à partir d'un delta Δ qui transforme un arbre T en un arbre T' , il est impossible de calculer le delta Δ' qui transforme T' en T sans utiliser T .

Ces limites justifient l'introduction de la notion de *delta complet*.

$\Delta_{1,2}$ (Forward Delta)	$\Delta_{2,1}$ (Backward Delta)	$\overline{\Delta}_{1,2}$ (Completed Delta)
delete (21)	insert (11, 4, B)	\overline{delete} (11, 4, B)
move (11, 2, 16)	move (11, 1, 16)	\overline{move} (11, 2, 16, 11, 1)
update (3, 50)	update (3, 100)	\overline{update} (3, 50, 100)
update(8, 100)	update (8, 150)	\overline{update} (8, 100, 150)

TAB. V.1: *Deltas pour la séquence de versions de la figure V.3 où B représente le sous arbre de racine 21*

V.5.3 Le groupe des deltas complets

Pour pouvoir composer des deltas ou les inverser, nous introduisons les deltas “complets”. Ainsi, les deltas complets gardent, par exemple, le sous-arbre supprimé dans le cas d’une suppression. Un delta complet contient, non seulement les informations nécessaires pour passer d’un arbre T vers un arbre T' , mais aussi comment revenir en arrière de T' vers T . Les opérations d’un delta complet sont les suivantes :

1. $\overline{delete}(n, k, T_1)$ qui supprime l’arbre XML T_1 dont la racine est le k -ème fils du nœud n .
2. $\overline{update}(n, v, ov)$ où ov est l’ancienne valeur du nœud n .
3. $\overline{insert}(n, k, T_1)$ qui insère l’arbre XML T_1 comme le k -ème fils du nœud n .
4. $\overline{move}(n, k, m, p, q)$ qui déplace le sous arbre XML ayant pour racine m du q -ème fils de p pour être le k -ème fils du nœud n .

Définition V.5.3 Un ensemble $\overline{\Delta}$ de ces opérations est un *delta complet* s’il existe T, T' tels que $\overline{\Delta}$ est l’ensemble des opérations qui transforment T en T' .

□

Composition. La composition peut être définie pour les deltas complets [105]. Il est facile d’obtenir la liste des nœuds insérés, supprimés, déplacés ou changés. Les valeurs des nœuds ayant subi une mise à jour, ainsi que les parents des nœuds insérés ou supprimés sont faciles à maintenir.

Inversion. Soit un delta complet $\overline{\Delta}$. Soit $\overline{\Delta}^{-1}$ le delta complet obtenu en échangeant les opérations d’insertion et de suppression, les anciennes et les nouvelles valeurs des opérations de mise à jour et en permutant les arguments des opérations de déplacement. Observons que pour chaque T :

$$\overline{\Delta}^{-1}(\overline{\Delta}(T)) = T.$$

Identité. Soit $\Delta_0 = \emptyset$ (l’ensemble vide des opérations complètes). Alors pour chaque $\overline{\Delta}$, nous avons :

$$[\overline{\Delta}; \Delta_0] = [\Delta_0; \overline{\Delta}] = \overline{\Delta}.$$

Nous montrons facilement [105] que :

Theorème V.5.4 Les deltas complets avec la loi de composition forment un groupe algébrique.

V.5.4 Gestion des identifiants

Cette section présente une partie critique de la gestion des versions, qui concerne les identifiants persistants associés aux nœuds d’un document, les XId. Une *liste de XId* fournit une correspondance entre les nœuds d’un arbre et les entiers qui identifient ces nœuds. La *liste des XId* spécifie aussi le prochain entier disponible pour empêcher la réaffectation d’un identifiant d’un nœud qui a été supprimé à un nouveau nœud. La *XId map* est la liste des XId des nœuds obtenue par une traversée post-ordonnée de l’arbre. Un exemple d’une *XId map* et de l’arbre induit est montré à la figure V.4. Dans notre exemple, le prochain XId disponible est le 29.

La méthode pour créer et gérer la liste des XId est décrite par les deux étapes suivantes :

Initialisation. A l’initialisation, la liste des XId est initialisée à $1 - n | n + 1$ où n est le nombre de nœuds de l’arbre. Cette liste signifie que les nœuds de l’arbre sont numérotés de 1 à n suivant une traversée post-ordonnée. Le prochain identifiant libre est $n + 1$.

Evolution. Rappelons que les XId sont persistants. En particulier, un nœud déplacé garde toujours le même identifiant. Pour les insertions, nous assignons de nouveaux identifiants à tous les nœuds du sous-arbre inséré en utilisant encore le parcours postordonné dans le sous-arbre inséré.

Prenons un exemple. Considérons la liste des XId $(1 - 534 | 535)$. Supposons maintenant que le sous-arbre de racine 156 et de nœud le plus à gauche 112

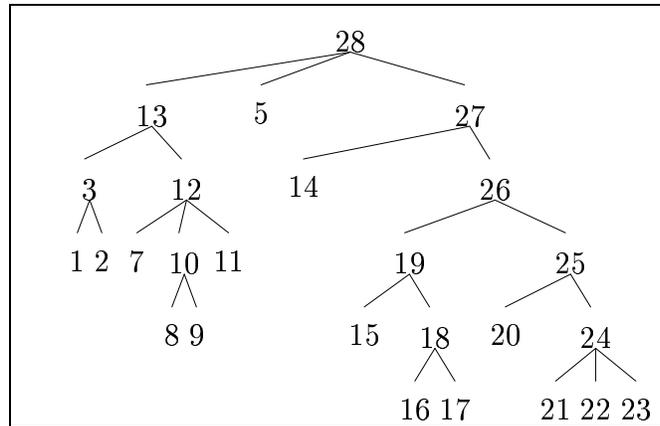


FIG. V.4: Arbre (1-3; 7-13; 5; 14-28 / 29)

soit supprimé. Ce sous-arbre n'est constitué que d'identifiants entre 112 et 156. La liste des XId après suppression sera alors (1 – 111; 157 – 534|535). Le signe “;” servant à séparer des séquence de XId.

Supposons maintenant qu'un sous-arbre contenant 23 nœuds soit inséré à droite du nœud 431. la liste de XId obtenue après cette insertion sera alors (1 – 111; 157 – 431; 535 – 557; 432 – 535|558).

V.5.5 Modèle physique

Cette section décrit l'implantation du mécanisme de versions.

Chaque delta complet est stocké comme un document XML. Quand le système décide de stocker une nouvelle version d'un document, les étapes suivantes sont exécutées :

1. La nouvelle version est obtenue du web et l'ancienne version ainsi que sa liste de XId sont chargées de la base XML.
2. L'algorithme *diff* [58] est exécuté entre les deux versions et fournit une correspondance entre les nœuds des deux versions.
3. Les XId sont attachés aux nœuds de l'ancienne version en utilisant la liste des XId de l'ancienne version. Les nœuds de la nouvelle version en correspondance avec les nœuds de l'ancienne version acquièrent leurs XId. Les nœuds insérés obtiennent de nouveaux XId. La liste des XId de la nouvelle version est alors construite.
4. Le delta complet est calculé.

5. La nouvelle version du document et la liste des XId sont stockées. Le delta complet est attaché au document XML contenant les deltas complets précédents. L'ancienne version est supprimée.

La Figure V.5 montre le contenu de la base XML pour un document profitant du mécanisme de versions à un moment donné. La dernière version du document, numéro 235, est stockée ainsi que sa liste de XId. Les deltas complets sont stockés comme un document XML. Chaque delta complet a pour racine l'élément `<unit_delta>`

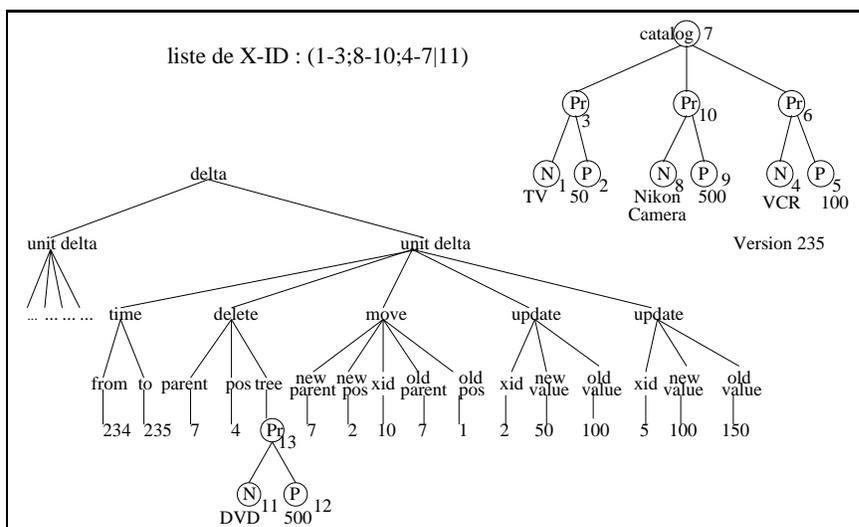


FIG. V.5: Stockage d'un document versionné

V.6 Souscription

Cette section décrit le mécanisme de souscription de Xyleme [113, 114], c'est-à-dire les mécanismes permettant de surveiller un grand flot de documents XML (stockés dans la base) et HTML. Une souscription consiste en un certain nombre de requêtes de surveillance et de requêtes continues. Ces notions de requêtes seront définies au cours de cette section. Considérons les requêtes de surveillance. Xyleme lit un flux de pages web. Ce flux peut être vu abstraitement comme une liste infinie de documents : $\mathcal{D} = \{d_i \mid i\}$, i.e. la liste des pages récupérées par Xyleme dans l'ordre où elles sont lues du web. La tâche principale est, avant de déclencher le système de souscription, de trouver pour chaque document du flux s'il existe une requête de surveillance

intéressée par ces documents. Si c'est le cas, le système de surveillance produit une *notification* qui contient le code de la requête de surveillance et quelques informations pertinentes concernant ce document. Ainsi, chaque requête de surveillance peut être vue comme un filtre sur la liste des documents \mathcal{D} produisant un flot de notifications.

Cette section décrit tout d'abord l'architecture du module de souscription (Section V.6.1) en présentant les composants un à un. Le langage de souscription est présenté en Section V.6.2.

V.6.1 Architecture de Souscription

L'architecture générale du système de souscription est présentée dans la Figure V.6. Son implantation dans Xyleme permet de surveiller un flot de millions de pages par jour tout en gérant des millions de souscriptions sur un seul PC, et peut facilement être distribuée sur plusieurs machines. Cette architecture peut être divisée en deux groupes de modules.

- les modules génériques peuvent être utilisés dans tout autre système de gestion de changements. Ces modules sont le Surveillant du Processeur de Requêtes, le Gestionnaire de Souscription, le Moteur de Déclenchement et le Rapporteur.
- les modules spécifiques à l'application dédiés aux contrôles de changements dans Xyleme. Ces modules sont les Alerteurs spécifiques que nous utilisons, le Processeur de Requête, et des modules tels que le Gestionnaire de Souscriptions de Xyleme et le module envoyant les rapports.

Dans la figure V.6, les lignes en pointillés dénotent le flot de commandes, alors que les lignes pleines représentent le flots de données. La partie générique du système est à l'intérieur du rectangle en gras. Nous allons maintenant présenter les modules un à un en commençant par donner un aperçu du fonctionnement général.

Systeme Global. Pour chaque document, les Alerteurs détectent l'ensemble des événements atomiques intéressants. Si l'ensemble est non vide, une alerte est envoyée au *Surveillant du Processeur de Requêtes*. Cette alerte consiste en l'ensemble des événements atomiques détectés lors de l'analyse du document avec en plus des informations sur le document détecté. Le *Surveillant du Processeur de Requêtes* détermine si des souscriptions sont concernées par les alertes reçues, et/ou s'il doit déclencher un traitement particulier.

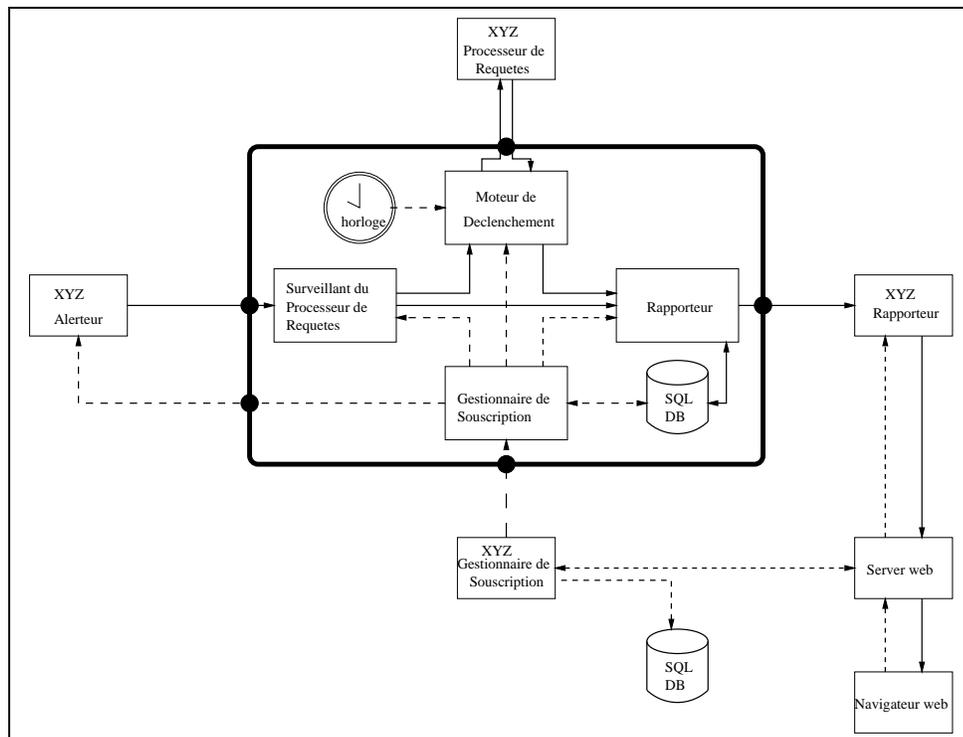


FIG. V.6: Architecture du Système de Souscription

Par exemple le *Moteur de Déclenchement* peut démarrer l'évaluation d'une requête. Les notifications venant du *Surveillant du Processeur de Requêtes* (pour les requêtes de surveillance) ou du *Moteur de Déclenchement* (pour les requêtes continues section V.6.2.2) sont envoyées au *Rapporteur*. Quand les conditions de création de rapport sont réalisées, le *Rapporteur* envoie l'ensemble des notifications reçues précédemment sous la forme d'un document XML au *Rapporteur de Xyleme* qui applique une requête XML sur ce document. Cette exécution crée un rapport envoyé par le biais du mail, ou consultable sur le web.

Les Alertes. Le système de souscription est entièrement basé sur la détection d'*événements atomiques*. Il permet la surveillance tant au niveau des pages (découverte de nouvelles pages) qu'au niveau de l'élément (insertion d'un nouveau produit dans un catalogue)¹. Son rôle est de détecter ces événements pour chaque document entré dans le système, et si c'est le cas d'envoyer une alerte pour le document concerné. Ainsi, ce module est fortement dépendant de l'application.

Surveillant du Processeur de Requêtes. Le système doit détecter des conjonctions d'événements atomiques qui correspondent aux souscriptions. Une telle conjonction "d'événements atomiques" est appelée "événement complexe". Le rôle du *Surveillant du Processeur de Requêtes* est, basé sur les alertes levées par un document (un ensemble d'événements atomiques), de détecter les événements complexes auxquels ce document correspond. Quand un tel événement complexe est détecté, le *Surveillant du Processeur de Requêtes* envoie une "notification" qui consiste dans le code de l'événement complexe avec des informations additionnelles vers le *Rapporteur* et/ou le *Moteur de Déclenchement*.

Moteur de Déclenchement. Le *Moteur de Déclenchement* peut déclencher une action externe soit à la réception d'une notification, soit à une certaine date.

Le Rapporteur. Le *Rapporteur* stocke les notifications qu'il reçoit. Quand une condition de rapport est satisfaite, il envoie ces notifications comme un document XML. Le rapporteur traite ce rapport, fondamentalement en lui appliquant une requête XML. Les rapports sont finalement, soit envoyés par courrier électronique, soit accessibles par le biais d'une adresse sur le web.

1. Cette dernière partie n'a pas fait l'objet d'une implantation.

Le Système de Souscription. Le *Gestionnaire de Souscription* reçoit les demandes des utilisateurs et gère les autres modules du système de souscription. Pour construire une souscription, un utilisateur la poste sur le serveur web. La demande est analysée par ce module.

Son premier rôle est de servir d'interface pour l'insertion d'une nouvelle souscription et la suppression ou la modification de souscriptions existantes. Pour être plus précis, le Gestionnaire de Souscription est divisé en un module générique et un module spécifique pour Xyleme.

Son second rôle est de contrôler les composants variés du système de souscription. Par exemple, il choisit les codes internes des événements atomiques et avertit (dynamiquement) les *Alerteurs* de la création de nouveaux événements.

V.6.2 Langage de Souscription

Cette section décrit brièvement le langage de souscription. Une souscription consiste en quatre parties: (i) des requêtes de surveillance, (ii) des requêtes continues, (iii) un rapport de spécification, et (iv) des clauses de rafraîchissement. Elle a la forme suivante :

```
subscription name
  monitoring... % (i)
  continuous... % (ii)
  report when... % (iii)
  refresh... % (iv)
```

Un exemple de souscription est donné dans la Figure V.7.

Cette souscription surveille un ensemble d'URLs ayant pour préfixe commun `http://www.xyleme.com/` et qui ont été modifiées depuis leur dernière lecture et les nouveaux éléments ayant pour nom de tag "Member" dans le document XML d'URL `http://www.xyleme.com/members.xml`. Elle demande aussi d'évaluer une requête Q toutes les deux semaines. Toutes les notifications résultantes sont (logiquement) stockées jusqu'à ce que la condition d'élaboration du rapport devienne "vrai", c'est-à-dire, jusqu'à ce que plus de 100 notifications soient rassemblées. Une fois cette condition atteinte, la requête de la clause rapport est exécutée sur les données stockées. Le rapport peut, par exemple, enlever les doublons dans la collection des URLs qui ont été mises à jour.

```
subscription MyXyleme

monitoring
  select <UpdatedPage url=URL/>
  where URL extends ‘‘http://www.xyleme.com/’’ and modified self

monitoring
  select X
  from self//Member X
  where URL= ‘‘http://www.xyleme.com/members.xml’’ and new X

continuous ReferenceXyleme
  % une requête XML Q qui calcule la liste de sites qui
  % référencent Xyleme
  try biweekly

report
  % une requête XML Q’ sur le flux de sortie
  ...
  when notifications.count > 100
```

FIG. V.7: *Exemple de Souscription*

V.6.2.1 Requêtes de surveillance

Une requête de surveillance a la forme suivante :

```
select result
(from from-clause)
where condition
```

From clause. Cette clause est optionnelle. Par défaut la requête porte sur les données en cours de filtrage. Le document courant est connu par la variable *self* dans la requête. La clause *from* peut être utilisée pour attacher des variables aux éléments du document courant.

Select clause. Cette clause décrit les données que le résultat des notifications devrait contenir, basées sur des constantes, le mot clé *self* ou les variables introduites par la clause *from*.

Where clause. Cette clause est une disjonction de conditions atomiques. Une condition atomique peut être une des conditions suivantes :

```
URL extends string   URL = string
DTDID = integer      DTD = string
DOCID = integer      domain = string
filename = string
```

où “domain” est l’un des domaines sémantiques que Xyleme utilise pour sa classification (voir Section V.4), “DOCID, DTDID” sont des identifiants internes, et “filename” est la fin d’une URL (e.g. `index.html`). Les autres conditions atomiques portent sur des informations internes des documents.

V.6.2.2 Requêtes continues

Dans cette section la notion de requêtes continues [49] est introduite. Une requête continue consiste en une requête Xyleme standard [16] plus une condition qui spécifie quand appliquer la requête. Typiquement, cette condition implique une fréquence (e.g. toutes les semaines). La requête continue peut être aussi déclenchée par une notification envoyée par le processeur de souscription.

Considérons la requête continue suivante :

```
continuous delta AmsterdamPaintings
select p/title
from culture/museum m, m/painting p
```

```
where m/address contains ‘‘Amsterdam’’
when biweekly
```

qui demande les noms de tous les peintres trouvés dans les musées d’Amsterdam. Ici *culture* est un des domaines *abstracts* qui fournissent une vue intégrée des ressources des musées. L’utilisation du mot clé *delta* spécifie que l’utilisateur est intéressé par les changements du résultat et non par le résultat lui-même et que nous voulons stocker le delta de ce document (voir Section V.5). Par conséquent, lors de la première évaluation de la requête l’utilisateur recevra la réponse, mais lors des évaluations suivantes, il ne recevra que les modifications. La clause *when* indique que l’évaluation de cette requête doit se faire une fois toutes les deux semaines.

V.6.2.3 Rapport

La partie rapport d’une souscription a la forme suivante :

```
select ...      % report query
when ...       % reporting condition
(atmost) ...   % limiting conditions
(archive) ...  % archiving conditions
```

Le rapport est une requête standard de Xyleme qui prend en entrée l’ensemble courant des notifications (un document XML) et produit en sortie un autre document XML. La clause *when* indique quand remplir un nouveau rapport. Cette clause est obligatoire, alors que les clauses suivantes sont optionnelles. La clause *atmost* limite la portée de la requête de rapport. Par exemple, *atmost 500* indique qu’après 500 notifications, le module arrête d’enregistrer de nouvelles notifications jusqu’à la prochaine construction d’un rapport. Finalement, la clause *archive* demande que le résultat de cette souscription soit archivée pendant une période de temps. Ainsi, *archive monthly* demande au système de stocker les rapports pendant une période d’un mois avant de les effacer. Plus de renseignements sur la partie rapport du module de souscription peuvent être trouvés dans [89].

Un exemple de rapport construit pour la souscription de la Figure V.7 est :

```
<Report>
  <UpdatePage url=‘‘http://www.xyleme.com/index.html’’/>
  <UpdatePage url=‘‘http://www.xyleme.com/members.html’’/>
  <Member><name>Jouglet</name><fn>Jerémie</fn></Member>
  <Member><name>Nguyen</name><fn>Benjamin</fn></Member>
```

```
<Member><name>Le Niniven</name><fn>David</fn></Member>
...
<ReferenceXyleme>
  <site url='‘http://www.inria.fr/~verso’’/>
  <site url='‘http://www.google.com’’/>
</ReferenceXyleme>
</Report>
```

Cette partie du langage impliquant la clause de rapport n'a pas été implantée.

Chapitre VI

Xyleme : Acquisition et rafraîchissement des données

Le chapitre précédent a introduit brièvement le système Xyleme dans sa globalité. Nous allons maintenant nous intéresser à la manière dont l'entrepôt de données est peuplé en pages XML et comment ces pages sont rafraîchies (relues) de manière automatique en gérant au mieux les ressources du système. Notre solution est basée sur un système inflationniste contrairement aux systèmes actuels qui se contentent de rafraîchir l'entrepôt au détriment de l'acquisition de nouvelles pages.

Lors de la création d'un entrepôt de données du web, les données peuvent être obtenues par deux stratégies complémentaires. Dans la stratégie dite "Pull" le système parcourt le web pour *découvrir* des données intéressantes. Dans la seconde stratégie dite "Push" (voir par exemple [18, 20, 21, 22]), les serveurs de données sur le web participent activement à la constitution de l'entrepôt en exportant vers l'entrepôt, les données qu'ils souhaitent rendre visibles.

Le système décrit dans ce chapitre appartient à la première catégorie. Le système d'acquisition comporte deux aspects essentiels : (i) la découverte de nouvelles pages et (ii) le rafraîchissement des pages connues. Il est à noter que le ratio entre les deux varie au cours du temps : la découverte des pages est typiquement plus importante dans une phase préliminaire. De plus, le nombre de nouvelles pages à obtenir est limité par l'espace de stockage disponible. Le rafraîchissement et la découverte sont guidés par le concept d'*importance d'une page* et de son estimation. Le rafraîchissement est aussi guidé par une estimation de la *fréquence de changement d'une page*.

Le choix de la prochaine page à lire est complexe. Le système peut lire un nombre limité de pages par unité de temps, disons par jour. Le nombre de pages du web étant très important, la stratégie naïve de *lire toutes les*

pages et/ou de *rafraîchir toutes les pages périodiquement* peut avoir pour conséquence de ne jamais obtenir certaines pages et des cycles de rafraîchissement de plusieurs semaines. Les observations suivantes peuvent mener à des améliorations importantes de la qualité d'un entrepôt de données sous des ressources fixées [87].

- Privilégier les pages qui sont demandées par les utilisateurs et/ou appartiennent à un domaine d'intérêt spécifique [47];
- Ne pas gaspiller les ressources pour lire une page qui change rarement ou jamais (une archive ou une page oubliée); et
- Ne pas gaspiller les ressources pour essayer de garder à jour une page qui a un taux de changement trop important (comme pour les cotations boursières).

Basé sur ces observations, le rafraîchissement devient un problème d'optimisation, à savoir minimiser la différence entre le web et l'entrepôt étant donné des ressources limitées, par des fonctions de coût prenant en compte l'intérêt spécifique des utilisateurs, l'importance des pages et le taux de changement des pages.

Nous faisons la distinction entre deux types de pages, HTML et XML. Ces pages (HTML + XML) et leur structure de liens forment ensemble la colonne vertébrale du web. Essentiellement, nous découvrons le web en suivant leurs liens. XML joue aussi pour nous le rôle de *centre d'intérêt*, puisque nous voulons construire une base XML. Différentes applications peuvent avoir différents centres d'intérêt. Toutes les techniques présentées dans ce chapitre pour les pages XML peuvent être appliquées à de tels contextes.

Stratégie employée. Nous utilisons une *stratégie guidée* pour parcourir le web. Les moteurs de recherche classiques comme Altavista [152] utilisent une *stratégie standard* qui consiste, à partir d'un point de départ, à suivre les liens pour découvrir de plus en plus de pages. Une telle stratégie est dite de recherche transversale. La stratégie standard présente un haut niveau de hasard car les pages arrivent dans un ordre dépendant fortement du temps de réponse des serveurs web.

A cause de son caractère aléatoire et de son efficacité, la *stratégie standard* présente certains avantages. En particulier elle est conceptuellement simple et peut être efficacement implantée. En fait, une stratégie mixte a été implantée au départ mélangeant cette stratégie à une stratégie dite guidée.

Dans la stratégie guidée [52], le système parcourt le web sur les "conseils" d'un module chargé de décider quelles sont les prochaines pages qui doivent

être chargées. Cette décision prend en compte des paramètres extérieurs afin d'optimiser la qualité de l'entrepôt. Après les premières expérimentations, nous avons décidé d'utiliser seulement la *stratégie guidée* pour les raisons suivantes :

Caractère stochastique. Un risque de la *stratégie guidée* est de se focaliser trop sur des portions limitées du web. Plus précisément, le calcul de l'importance de la page est biaisé (au départ du processus) car le système ne connaît seulement qu'une partie du web. Cela tend à limiter la découverte aux parties déjà connues du web, au risque d'ignorer d'importantes autres portions. Nos premières expérimentations ainsi que [100] montrent que le web est assez connecté pour que ce cas ne se présente pas.

Efficacité. Nos expérimentations montrent que l'ordonnancement des pages suivant la *stratégie guidée* peut être aussi exécuté de manière performante avec des ressources limitées.

Qualité. Avec la *stratégie guidée*, le système a les moyens de mieux sélectionner le contenu de la base XML (par la découverte) et sa fraîcheur (par le rafraîchissement).

Ce chapitre présente le système d'acquisition et de rafraîchissement des données. Il se décompose ainsi :

- L'architecture du système est présentée en section VI.1. Les sections suivantes décrivent les différents modules du système.
- L'interface web.
- Le gestionnaire des méta-données.
- Le calcul de l'importance des pages.
- Finalement, l'ordonnanceur des pages en section VI.5 sera présenté ainsi que le procédé employé pour estimer le taux de changement d'une page.

VI.1 Architecture

L'architecture fonctionnelle est présentée dans la Figure VI.1. Les modules spécifiques du système d'acquisition sont à l'intérieur du rectangle en gras. Les autres modules sont présents afin de mettre en évidence les interactions du système avec le reste de Xyleme.

Les fonctionnalités essentielles des modules sont les suivantes :

- L'*Interface Web* est le seul module qui accède au web. Afin de garantir des temps de réponse rapides, ce module effectue très peu de calculs. Son rôle est d'aller chercher des pages sur le web.
- le *Gestionnaire des Métadonnées* stocke les informations liées aux pages. Ces informations sont aussi diverses que la liste des fils pour chaque page ou la date de dernière lecture des pages.
- l'*Estimateur* lit la matrice des liens, maintenue par le *Gestionnaire des Métadonnées* afin de calculer l'importance des pages. Cette importance devient elle aussi une information associée aux pages.
- l'*Ordonnanceur des Pages* décide quelles sont les pages qui doivent, soit être lues pour la première fois, soit être rafraîchies.

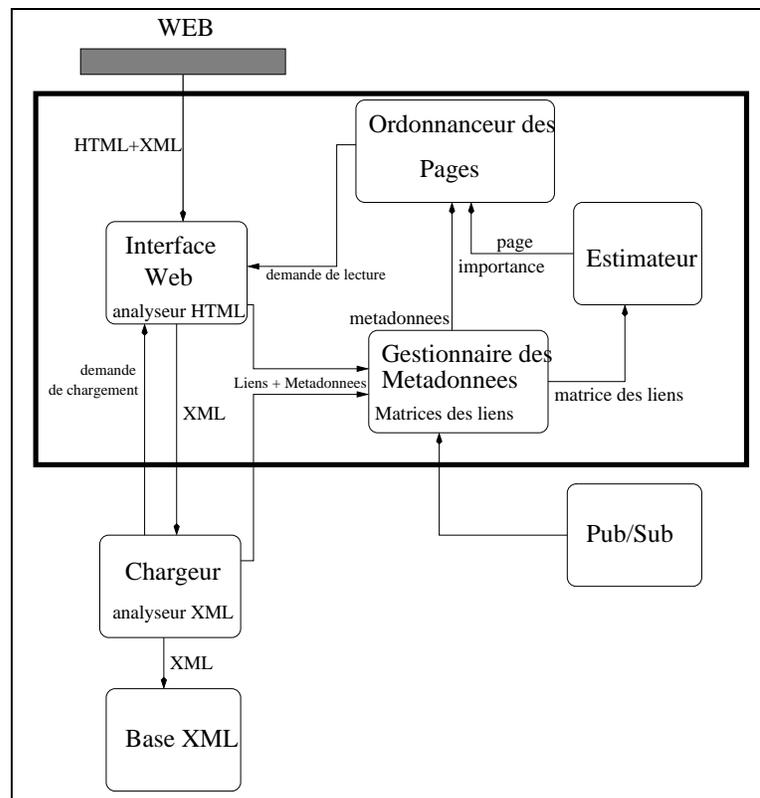


FIG. VI.1: Architecture Fonctionnelle simplifiée de l'Acquisition

Les sections suivantes de ce chapitre présentent de manière plus détaillée ces différents composants de l'architecture. Mais auparavant, le cycle de vie des pages dans notre système est défini.

Cycle de Vie d'une Page.

Le cycle de vie d'une page XML peut être décrit ainsi. L'*Interface Web* charge une page du web. Une fois la page lue, il l'envoie au *Chargeur* afin de la traiter. Ce module analyse lexicalement la page afin (i) d'en extraire les liens, (ii) de construire l'arbre DOM de la page pour la stocker dans la base XML (voir Section V.2 pour plus de précision). Si la page XML est rattachée à une DTD inconnue du système, le *Chargeur* demande alors à l'*Interface Web* d'aller chercher cette DTD.

Une fois la page analysée et stockée dans la base XML, le *Chargeur* envoie l'URL de la page avec la liste de ses fils (URL contenues à l'intérieur de la page) et des métadonnées associées à la page, au *Gestionnaire des Métadonnées* qui stocke les différentes informations de la page et construit la matrice des liens du système.

Le cycle de vie d'une page HTML est identique sauf que la page n'est pas envoyée au *Chargeur*. Cette page est analysée lexicalement par l'*Interface Web* afin d'en extraire les liens internes (ancres HTML). A la fin de l'analyse, la page, les liens contenus à l'intérieur de celle-ci ainsi que des métadonnées sont envoyés au *Gestionnaire de Métadonnées*.

La suite de la description est identique pour les pages XML ou HTML. Périodiquement, l'*Estimateur* parcourt la matrice des liens afin de calculer l'importance des pages. Cette importance devient une information supplémentaire associée à chaque page. Basé sur des critères tels que l'importance d'une page, l'*Ordonnanceur des Pages* décide de découvrir une nouvelle page ou de rafraîchir une page. Pour cela, il envoie l'URL de la page à l'*Interface Web*.

Finalement, le module de publication et/ou de souscription (Section V.6) permet de faire connaître une page à Xyleme ou de lui transmettre des indications quant à la fréquence de rafraîchissement de la page (par exemple un catalogue change ses prix tous les trois mois).

VI.2 L'Interface Web

L'interface Web implante plusieurs des caractéristiques des moteurs de recherche standard. Seules quelques particularités seront présentées.

C'est le seul module qui va chercher des pages sur le Web. Cette restriction est importante afin de garantir le respect des *lois sur les robots* [81]. Par

exemple, une des ces règles est qu'un client ne doit pas trop rapidement poser plusieurs requêtes au même site web¹. Quand l'Interface Web récupère une page HTML, il l'analyse (à la volée) pour y découvrir de nouvelles URL. L'URL de la page courante ainsi que tous les liens internes sont envoyés vers le Gestionnaire de Métadonnées afin de construire la matrice des liens (voir Section VI.3). De la même façon, lors du chargement de pages XML, l'Interface Web envoie la page au Chargeur afin qu'il en extraie, entre autres, les liens internes pour mettre à jour la matrice des liens.

Ainsi que le montre la Figure VI.1, l'Interface Web reçoit des requêtes de chargement non seulement de l'Ordonnanceur de Pages mais aussi du Chargeur. Pour être plus précis, quand l'Ordonnanceur de pages décide de la lecture d'une page, il envoie une requête vers l'Interface Web. De même, le Chargeur utilise l'Interface Web afin de récupérer différentes pages telles que les DTDs ou les entités externes d'un document XML. De façon évidente, une requête du Chargeur a une priorité plus importante puisqu'il est en attente d'une ressource particulière. Le protocole de soumission de pages entre l'Interface Web et le reste de Xyleme est donc soumis à différents niveaux de priorités.

Les métadonnées fournies par le protocole HTTP [79] telles que le type MIME de la page [78] avec des mécanismes de filtres basés sur des suffixes tels que "*.jpg" ou "*.gif" sont utilisés pour éviter de rapatrier des pages qui ne sont ni XML ni HTML ou qui ne changent jamais.

Une particularité de notre système est que la décision de rapatrier du web une page particulière est dictée par un autre module, l'Ordonnanceur de Pages. L'Interface Web "bufferise" les pages en attente de chargement (des requêtes précédentes). Ce tampon est limité, de façon expérimentale, à 100 000 URL. Ce tampon est potentiellement un goulot d'étranglement. En effet, les URL venant d'un site particulier, disons *www.popular.com* peuvent être bloquées à l'intérieur à cause de la règle contre l'accès rapide à un site. Ce tampon peut finir par être rempli de telles pages. Pour prévenir cette possibilité "d'asphyxie", i.e. un trop grand nombre de pages d'un site sont dans le tampon, elles sont rejetées. Ce rejet intervient seulement si le système est certain que les pages ne pourront pas être toutes lues au cours d'un intervalle temporel fixé par le système. L'*ordonnanceur de Page* réessaiera ces requêtes plus tard.

L'Interface Web a été implantée sur un PC standard, mono-processeur de

1. Une distinction est faite entre le comportement d'un client contrôlé par un utilisateur et celui contrôlé par un programme. Un utilisateur n'est pas capable de suivre des milliers de liens en quelques secondes alors qu'un programme le peut. Des sites célèbres sur le web ont connu de tels problèmes en Février 2000, à cause de requêtes simultanées venues de robots distribués sur plusieurs machines attaquant les mêmes sites.

600 MHz et utilise 128Mo de mémoire centrale. De plus amples informations sur son implantation peuvent être trouvées dans [109]. Elle est composée de plusieurs modules logiques, chacun d'eux étant exécuté dans un processus léger (thread) spécifique. Elle charge en moyenne 4 millions de pages par jour et peut poser 300 requêtes HTTP simultanées. Si un débit plus important est nécessaire, plusieurs Interfaces Web peuvent être utilisées en distribuant les URL entre ces différents modules.

Les principaux problèmes rencontrés lors de l'implantation sont des problèmes habituels des robots de recherche tels que :

- un thread peut être bloqué pour une durée indéterminée lors d'un appel aux serveurs de résolution d'un site web en son adresse numérique sur le réseau (adresse IP);
- des erreurs de syntaxe dans les pages HTML ou dans les fichiers "robots.txt" du web sont courantes (les fichiers spécifiant quelles portions du site sont interdites aux robots), posant ainsi de sérieux problèmes lors de l'analyse de tels fichiers;
- des fichiers de taille importante sont parfois rencontrés. Leur taille peut être plus grande que l'espace mémoire disponible.

Avant de commencer l'implantation de ce module, nous avons évalué les robots disponibles sur le web. Ces logiciels sont à la base de tous les moteurs de recherche du web, et par conséquent, seul un petit nombre est disponible sous des licences dites "libres". L'évaluation concernait deux robots: (i) le programme Unix "wget" [145] et (ii) le logiciel "ht://Dig" [139]. Ces deux programmes permettent d'aller chercher des pages sur le web et de suivre leurs liens.

La différence entre ces deux programmes est que "wget" s'apparente plus à un programme externe qui nécessite un appel externe au programme alors que "ht://Dig" fournit une API afin d'être intégré à un programme. L'un de leurs principaux défauts est qu'aucun de ces robots ne respectait le laps de temps entre deux demandes consécutives de pages sur un même site. Ainsi, lors de nos expérimentations avec "ht://Dig", le site de l'INRIA Rocquencourt s'est vu interdire l'accès à un site important pour ses pages XML en raison du non respect de cette règle².

En résumé, wget est un excellent moyen de rapatrier un document ou de dupliquer un site web, alors que ht://Dig semble être parfait pour construire une

2. Ce qui était un peu gênant car ce site était l'un des plus gros points d'entrée sur le web XML de Xyleme.

petite application sur un intranet local. Néanmoins, ces deux robots ne permettent pas une intégration fine à notre système. En particulier notre module devait (i) être pleinement intégré au reste du système et en particulier avec les modules externes tel que le chargeur de données XML; et (ii) permettre de contrôler finement la politique de rapatriement des pages. Ces raisons ont été suffisantes pour décider d'implanter un robot.

VI.3 Le Gestionnaire de Métadonnées

La gestion des métadonnées est distribuée entre plusieurs Gestionnaires de Métadonnées. La tâche d'un seul de ces processus est d'abord présentée. La distribution entre plusieurs Gestionnaires est ensuite discutée.

Tâche d'un Gestionnaire de Métadonnées. Le Gestionnaire de Métadonnées stocke et met à jour les méta-informations pour chaque URL. Ces informations sont distribuées entre plusieurs structures de données :

La Table de correspondance : Cette table maintient une bijection entre les URL et des identifiants persistants, appelés URL-Id. Un URL-id est une paire $(local, ident)$ où *local* est un identifiant sur 1 octet correspondant à la machine à laquelle est affectée cette URL, et *ident* est un entier sur 4 octets identifiant cette URL sur cette machine. Tous les autres modules du système identifient un document seulement par son URL-Id, seul le Gestionnaire de métadonnées connaît l'URL attachée à un document.

La Table de Rafraîchissement : Cette structure contient les données dont l'Ordonnanceur de pages a besoin (voir Section VI.5).

- les données temporelles : la date de dernier changement que le système a détecté pour cette page, la première et dernière date de chargement de ce document par le système, la date du dernier changement du document spécifié par l'en-tête HTTP du document. Le système garde de telles informations seulement pour les pages qu'il stocke, e.g. les pages XML.
- La signature du Document : le résultat d'une fonction de hash CRC sur 32 ou 64 bits donnée par l'algorithme MD5 [80]. Cette signature est calculée par l'Interface Web lors du dernier chargement de cette page. Pour les pages HTML, c'est l'unique moyen de détecter des changements puisque de telles pages ne sont pas

stockées par le système car ces pages doivent quand même être rafraîchies.

- Le nombre de changements et d'accès sans changement depuis une date particulière.
- Le type du document. Pour le moment, seuls les types HTML, XML, DTD, ERROR, DEAD, où DEAD permet au système de savoir si ce document a disparu du web, sont utilisés.
- L'importance de la page calculée par notre Estimateur (voir Section VI.4).
- D'autres statistiques utiles.

La Matrice des Liens : Cette relation contient, pour chaque page, la liste des fils³ de la page, plus précisément la liste des URL-id des URL référencées dans la page. Cette information est utilisée par l'Estimateur.

Le statut des données : Cette relation contient des informations au sujet du statut temporaire lors du chargement du document. Ces informations sont nécessaires lors du chargement de document contenant des entités externes.

Il est important de noter que le travail du Gestionnaire de Métadonnées est très intensif. Pour chaque page récupérée par l'Interface Web, le système met à jour les métadonnées de cette page, qui sont persistantes donc stockées sur disque. Cela signifie que le système a besoin d'effectuer plusieurs écritures sur disque par page. La principale difficulté est de mettre à jour la matrice des liens. La matrice des liens utilise les URL-id. Les liens obtenus lors de l'analyse d'une page sont des URL. Par conséquent, le système doit effectuer une transformation des URL vers les URL-id, i.e. en moyenne, N accès aléatoire à la Table de Correspondance sont nécessaires, où N est le nombre moyen de liens contenus dans une page (N est de l'ordre de 10). Pour pouvoir effectuer efficacement cette transformation, la Table de Correspondance utilise une structure en mémoire centrale.

Distribution. Le Gestionnaire de Métadonnées effectue d'intenses communications avec l'Interface Web. Les deux s'exécutent sur la même machine physique, afin de limiter le coût des accès réseaux et pour ne pas le surcharger. Cependant, ils sont tous les deux conçus pour être distribués. Pour utiliser plusieurs *Interfaces Web*, il suffit de diviser le domaine des URL (chaîne de

3. Pour être précis, nous stockons au plus les 65534 premiers fils de chaque page.

caractères) par m , où $m > 1$ est le nombre d'Interface Web, e.g. en utilisant une fonction d'indexation aléatoire des chaînes de caractères sur $[1..m]$. Un Gestionnaire de Métadonnées peut supporter la charge d'une Interface Web, donc m Gestionnaires de Métadonnées seront aussi nécessaires.

VI.4 Importance des Pages

Une notion classique d'importance de pages [37] est utilisée de manière intensive par l'Ordonnanceur de Pages pour (i) guider la découverte de nouvelles pages XML et HTML; et (ii) pour guider le rafraîchissement des pages XML déjà lues.

Plus précisément l'importance d'une page est basée sur deux critères :

- l'importance d'une page dépend du nombre de pages qui pointent vers elle et de leurs importances,
- l'intérêt spécifique de quelques pages expressément spécifiées par les utilisateurs par le biais du mécanisme de publication et/ou de souscription.

Cette section montre comment cette importance est calculée pour rafraîchir les pages XML (section VI.4.1). La même notation est utilisée pour acquérir de nouvelles pages. Le rafraîchissement des pages HTML est considéré (section VI.4.2). Nous voyons aussi comment la publication et la souscription peuvent être introduites dans cette évaluation (section VI.4.3). Finalement, quelques aspects de l'implantation et des développements futurs sont décrits en section VI.4.4.

VI.4.1 Importance des pages XML

Cette section rappelle la technique de [37] pour définir et calculer l'importance des pages. Cette technique sera utilisée pour les pages XML.

Des documents de grande qualité, au contenu clair, précis et contenant des informations utiles, sont souvent interconnectés entre eux, alors que des documents de faible qualité ont peu ou pas d'interconnexions. Aussi, les liens venant de pages importantes (comme la page principale de Yahoo! [164], par exemple) pèsent plus que des liens venant de pages moins importantes.

L'importance des pages est estimée par le *vecteur d'importance* \mathcal{I} . Les identifiants des pages sont pris dans les N premiers entiers naturels. Pour une page i , soit $out(i)$ l'ensemble des pages vers lesquelles i pointe, et $outdeg(i)$

le degré sortant de i , i.e. $outdeg(i) = Card(out(i))$. Le vecteur \mathcal{I} est donné par :

$$\mathcal{I}(i) = \sum_{j \in out(i)} \frac{\mathcal{I}(j)}{outdeg(j)} \quad (\text{VI.1})$$

Ce vecteur distribue équitablement l'importance d'une page entre ses successeurs. Pour toutes les pages, le système d'équations linéaires suivant est obtenu :

$$\mathcal{I}_{N \times 1} = M_{N \times N} \times \mathcal{I}_{N \times 1} \quad (\text{VI.2})$$

$$\text{où } M_{ij} = \begin{cases} \frac{1}{outdeg(i)} & : j \in out(i) \\ 0 & : \text{autrement} \end{cases}$$

Calculer l'importance d'une page revient à résoudre le système d'équations linéaires $M \times \mathcal{I} - \mathcal{I} = 0$, ou calculer le vecteur propre de M , ou enfin, le point fixe de φ , où $\varphi(X) = M * X$. La solution peut aussi être interprétée à partir de la notion de marche aléatoire sur une chaîne de Markov [174]. Supposons qu'un utilisateur voit une page du web i , il est susceptible de suivre n'importe quel lien de cette page avec la même probabilité. Soit $\mathcal{I}(j)$ la probabilité qu'un utilisateur aille à la page j . Supposons que \mathcal{I} soit normalisé, c'est-à-dire :

$$\|\mathcal{I}\|_1 = 1 \quad \text{où} \\ \|(v_1, \dots, v_n)\|_1 \text{ est définie par } v_1 + \dots + v_n$$

Avec cette interprétation, l'importance d'une page donne la probabilité qu'une marche aléatoire sur le graphe atteigne cette page.

Soit $\|(V_1, \dots, V_n)\|_2$ définie comme $\sqrt{V_1^2 + \dots + V_n^2}$. Le vecteur \mathcal{I} peut être calculé en utilisant une technique de point fixe itérative, en appliquant de façon répétitive n'importe quel vecteur non dégénéré de départ. Soit \mathcal{I}^0 un vecteur initial uniforme (toutes les pages ont la même importance).

$$\mathcal{I}^{i+1} = M \times \mathcal{I}^i \text{ le vecteur de la } (i+1)\text{-ème itération} \\ \|\mathcal{I}^n - \mathcal{I}^{n-1}\|_2 < \epsilon \text{ la condition d'arrêt.}$$

La convergence de cet algorithme est garantie si M est irréductible, i.e. si la matrice de transition sous-jacente correspond à une chaîne de Markov

irréductible et apériodique [94, 116]. La dernière condition (périodicité) est vraie en pratique pour le web, alors que la première condition (irréductibilité) est vraie si nous ajoutons un facteur de salissure (“dampening”) c à la propagation de l’importance [31].

$$\begin{aligned} \mathcal{I}^{i+1} &= (cM \times \mathcal{I}) + (1 - c) \times E \\ \text{où } c &\in]0, 1[\end{aligned} \tag{VI.3}$$

Le facteur de salissure c diminue la propagation de \mathcal{I} le long des longues chaînes, par conséquent diminue l’importance des pages lointaines. En fait, l’importance d’une page est propagée le long de n nœuds qui seront multipliés par un facteur c^n . Aussi, une valeur petite de c accélère la convergence de l’algorithme. Typiquement, les valeurs de c sont prises dans l’intervalle $[0.7, 0.95]$. Par exemple, l’algorithme dit de “Page-Rank” de [37] utilise $c = 0.85$. En fait, le choix de ce facteur a peu d’influence sur l’importance des pages.

Le vecteur E de l’équation précédente est un *vecteur de personnalisation* [36]. Nous fixons $E(i) \in]0, 1[$, $\|E\|_1 = 1$. Supposons $E = [\frac{1}{N}]_{N \times 1}$. La Section montrera VI.4.3 comment cette valeur est réellement fixée.

Il est facilement vérifiable que quand M est stochastique, i.e. $\forall i, out(i) \neq \emptyset$, le vecteur \mathcal{I} est lui aussi stochastique : $\|\mathcal{I}^0\|_1 = 1 \Rightarrow \forall i \|\mathcal{I}^i\|_1 = 1$. Cette propriété de normalisation de \mathcal{I} est perdue quand il existe des nœuds sans successeur. L’intuition est que la valeur de \mathcal{I} de tels nœuds est perdue par le système à chaque itération. Pour retrouver cette propriété, un facteur de normalisation r est introduit tel que :

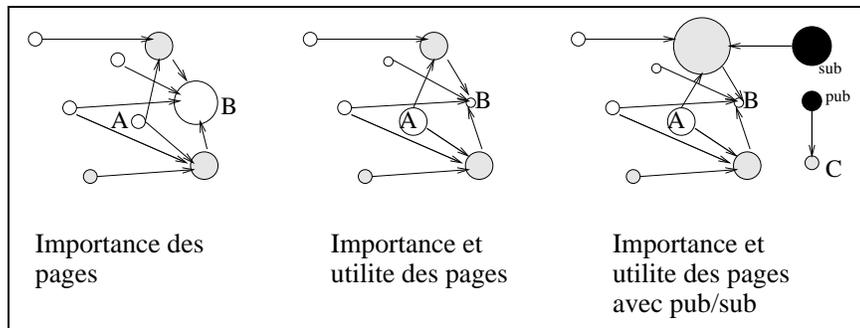
$$\mathcal{I}^{i+1} = r \times (cM \times \mathcal{I}^i + (1 - c) \times E) \tag{VI.4}$$

$$\text{où } r = \frac{\|\mathcal{I}^i\|_1}{\|cM \times \mathcal{I}^i + (1 - c) \times E\|_1} \tag{VI.5}$$

Par conséquent nous avons :

$$\|\mathcal{I}^{i+1}\|_1 = \|\mathcal{I}^i\|_1 \tag{VI.6}$$

\mathcal{I} est calculé pour l’ensemble des pages XML et HTML. Sur les pages XML déjà lues, cette importance guide leur rafraîchissement. Pour les pages XML ou HTML non encore lues, cette valeur aide à décider quelles sont les pages que le système devrait prochainement lire (voir section VI.5.1). Une mesure différente est utilisée pour rafraîchir les pages HTML.

FIG. VI.2: *Importance vs Utilité*

VI.4.2 Utilité des pages HTML

Une page HTML peut évoluer au cours du temps et pointer sur de nouvelles pages XML lors de sa prochaine relecture. Alors, pour être sûr de ne pas ignorer de nouvelles portions du web XML, les pages HTML doivent être lues plus d’une fois, i.e. il faut les rafraîchir. Toutefois, l’importance classique basée sur la structure du web n’est pas appropriée pour de telles pages. Une page HTML peut être très importante dans le web mais ne pointer sur aucune page XML, et donc être inutile du point de vue de Xyleme. Pour les pages HTML, une notion plus appropriée est utilisée. Cette notion est basée sur les pages XML sur lesquelles elle pointe et leurs importances. Pour la distinguer de la notion précédente d’importance, elle est appelée *utilité*. Comme nous pourrons le voir, l’utilité mène à un calcul du point fixe similaire mais “inverse” dans le sens que l’importance des pages est propagée vers l’arrière au lieu de l’avant.

Pour illustrer les notions d’importance et d’utilité, nous considérons le minuscule web de la Figure VI.2. Dans le graphe, les pages XML sont en gris et les pages HTML en blanc. Le noir est utilisé pour les pages résultant du processus de souscription/publication. L’importance des pages est proportionnelle au rayon du cercle. Le graphe de gauche montre l’importance des pages calculée en se basant sur la structure des liens du web. Notez que la page B est très importante car de nombreuses pages pointent vers elle, y compris deux pages “importantes”. Le graphe du milieu utilise la même notion d’importance pour les pages XML, mais utilise la notion d’utilité pour les pages HTML. Notons que, maintenant la page B est moins utile. Elle ne permet pas de découvrir des pages XML. Réciproquement, la page A est plus utile. Elle pointe sur des pages XML importantes. Finalement, observons l’impact de la souscription/publication (graphe de droite). En particulier, aucune page du graphe ne pointait sur la page C. La page C est seulement connue grâce à la

publication et tient son importance de cette publication seulement.

Une variante de l'algorithme précédemment expliqué est alors utilisée pour les pages HTML. L'heuristique suivante est utilisée : une page HTML qui ne pointe pas sur au moins une page XML (ou connue comme telle) est moins susceptible de mener (dans le futur) sur des nouvelles pages XML. Clairement, nous sommes intéressés aussi par des pages qui pointent sur des pages pointant sur des pages XML, et ainsi de suite. Intuitivement une page HTML est "utile" si elle pointe *directement* ou *indirectement* sur beaucoup de pages XML. Ainsi, le problème n'est pas totalement différent du précédent. Cependant, alors que l'importance des pages XML suit les liens, l'*utilité* pour les pages HTML est répercutée en remontant le graphe des liens.

Ainsi, pour calculer l'*utilité*, le même algorithme est utilisé mais avec une *condition de démarrage* différente et sur le graphe inverse des liens. La matrice correspondant à ce graphe est la matrice transposée du graphe initial :

$$M_{ij}^t = \begin{cases} \frac{1}{outdeg(i)} & : j \in out(i) \\ 0 & : autrement \end{cases}$$

Le calcul du point fixe de la matrice M s'effectue de la manière suivante :

$$\mathcal{J}^0 \text{ le vecteur initial } \mathcal{J}^{i+1} = cM^t \times \mathcal{J}^i + (1 - c)E'$$

$$E'_i = \begin{cases} \mathcal{I}_i & : i \text{ correspond à une page XML} \\ 0 & : autrement \end{cases}$$

$$\|\mathcal{J}^n - \mathcal{J}^{n-1}\|_2 < \epsilon \text{ la condition d'arrêt.}$$

Le point fixe est ainsi calculé sur le graphe inverse en utilisant un vecteur de personnalisation E' , lequel biaise l'algorithme vers les pages XML (en prenant en compte leur importance), et avec une valeur initiale de \mathcal{J} de 0 pour les pages HTML (aussi donnée par E').

VI.4.3 Publication/Souscription

Le même raisonnement peut être utilisé pour prendre en compte les publications et/ou les souscriptions des utilisateurs. Afin de capturer ces processus, des pages virtuelles sont introduites et modifient en conséquence le vecteur de personnalisation E' . Considérons un ensemble de pages auquel un utilisateur particulier s'intéresse. L'image précédente du web est étendue, par une page (virtuelle) qui contient la liste des liens vers ces pages. Cette page ne peut pas

être trouvée sur le web (elle n'existe pas). Cependant, nous l'utilisons pour capturer l'intérêt des utilisateurs grâce au vecteur de personnalisation. Ainsi, l'importance des pages pointées par cette page virtuelle peut être biaisée.

Le choix de l'importance de telle page de publication/souscription dépend de l'application, i.e. de l'importance que l'on donne à la publication/souscription en général et de cette publication/souscription en particulier. Il est assez difficile de donner une valeur absolue sans avoir des connaissances globales sur l'entrepôt, telles que l'importance moyenne des pages XML et la dérivation standard de celle-ci. Nous avons choisi d'utiliser une importance relative, e.g. une publication/souscription peut donner assez d'importance aux pages concernées, pour les promouvoir dans le groupe des pages les 20% plus importantes du système.

VI.4.4 Implantation

Cette section décrit quelques aspects d'une implantation efficace et permettant le passage à l'échelle. Cette implantation peut gérer des milliards de pages. Le cœur consiste en une implantation efficace du vecteur de multiplication de la matrice creuse, voir [150].

Algorithme Naïf. Premièrement, on observe que la matrice des liens est creuse. Pour chaque nœud i , la liste de ses fils ($out(i)$) est utilisée. C'est aussi la représentation originale du graphe des liens obtenue lors du parcours du web: chaque page contient la liste de ses successeurs. La métaphore de la matrice est utile pour la compréhension de l'algorithme. Nous continuerons donc à nous référer à cette représentation des liens comme étant la *matrice des liens*. La structure de données est identique à celle utilisée par [86]. La figure VI.3 illustre ce calcul.

Les lignes directrices du calcul d'une itération de l'importance sont :

```

begin
  for i=1 ... N,
    dest $\mathcal{I}_i = 0$ 
  endfor

  for i=1 ... N
    for j  $\in$  out(i),
      dest $\mathcal{I}_j = dest\mathcal{I}_j + \frac{source\mathcal{I}_i}{outdeg(i)}$ 
      source $\mathcal{I} = dest\mathcal{I}$ 
    endfor
  endfor
end

```

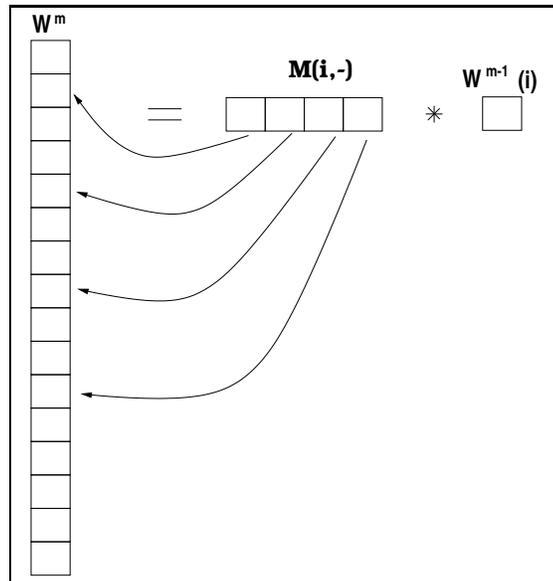


FIG. VI.3: Calcul de l'importance des pages

Pour calculer le graphe inverse (pour calculer l'*utilité* des pages HTML) la même représentation du graphe des liens peut être utilisée avec algorithme différent :

```

begin
  for i=1 ... N,
     $dest\mathcal{I}'_i = 0$ 
  endfor

  for i=1 ... N
    for  $j \in out(i)$ ,
       $dest\mathcal{I}'_i = dest\mathcal{I}'_i + \frac{source\mathcal{I}'_j}{outdeg(j)}$ 
       $source\mathcal{I}' = dest\mathcal{I}'$ 
    endfor
  endfor
end

```

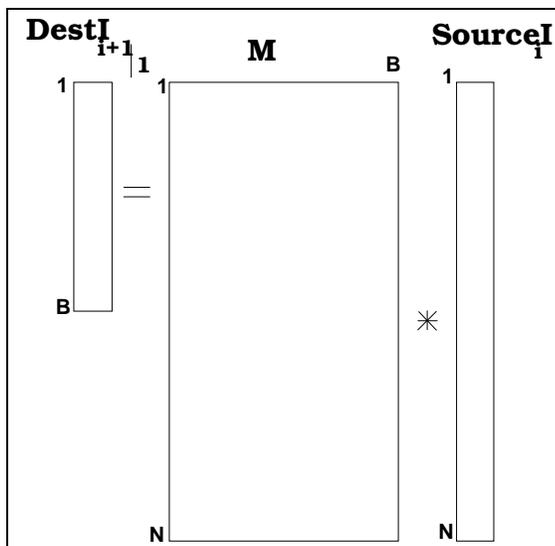
La technique de *Découpage de Matrice* [86] est implantée, technique que nous rappelons brièvement. Pour N grand, la matrice des liens ne tient pas en mémoire centrale (d'un PC). Elle est donc stockée sur disque. Par conséquent, l'astuce est de limiter les accès aléatoires à ces données.

L'algorithme d'*importance* effectue, un accès séquentiel sur $source\mathcal{I}$ et sur la matrice des liens, et un accès aléatoire sur $dest\mathcal{I}$. Si le vecteur $dest\mathcal{I}$ ne tient pas en mémoire, il est coupé en S morceaux, chacun de ces morceaux étant assez petit pour tenir en mémoire. Ainsi, chaque morceau a β éléments ($\beta = \frac{N}{S}$). Le s -ème morceau $dest\mathcal{I}$ dénoté $dest\mathcal{I}_{|s}$, contiendra les valeurs de \mathcal{I} pour les nœuds entre $s\beta$ et $(s+1)\beta - 1$. La matrice des liens est aussi coupée "verticalement" en morceaux, tels que chaque morceau de la matrice référence seulement les éléments correspondant dans le morceau $dest\mathcal{I}$. Les morceaux de la matrice sont lus un à un. De cette façon, lors du traitement d'un morceau donné, l'algorithme effectue seulement des accès aléatoires au morceau $dest\mathcal{I}$ correspondant qui est en mémoire. Quand le calcul d'un morceau de $dest\mathcal{I}$ est fini, il est écrit sur disque, et le prochain morceau est chargé en mémoire.

La figure VI.4 illustre ce découpage. Nous rappelons que le calcul matriciel est celui expliqué à la figure VI.3

Pour l'algorithme d'*utilité*, on a des accès séquentiels sur le vecteur $dest\mathcal{I}'$, et des accès aléatoires sur $source\mathcal{I}'$. Par conséquent, pour appliquer la technique de découpage en morceaux sur \mathcal{I} pour calculer l'utilité des pages, nous devons découper $source\mathcal{I}'$. Ainsi, les mêmes morceaux de la matrice pour les deux algorithmes peuvent être utilisés.

Itérations de Gauss-Seidel. Pour calculer le point fixe, un algorithme similaire à *l'algorithme de Gauss-Seidel* [76] est utilisé. Il permet une convergence plus rapide en terme de nombre d'itérations, et aussi une diminution de l'espace disque nécessaire au calcul.

FIG. VI.4: *Découpage de la Matrice*

Au fur et à mesure de la convergence vers un point fixe, on espère que $dest\mathcal{I}$ soit plus proche du point fixe que $source\mathcal{I}$. Nous espérons, donc, que le nouveau morceau $dest\mathcal{I}$ calculé est plus proche du point fixe que le morceau $source\mathcal{I}$ correspondant. Lors du calcul d'un morceau s , nous avons à notre disposition tous les morceaux déjà calculés $dest\mathcal{I}_t, t < s$. Lors du calcul suivant, ces morceaux sont utilisés au lieu des morceaux de $source\mathcal{I}$ correspondants. Pour revenir à l'implantation, nous n'utilisons plus les deux vecteurs $source\mathcal{I}$ et $dest\mathcal{I}$, mais seulement un seul, appelé \mathcal{I} . Le morceau en cours de calcul est chargé en mémoire centrale. Quand le calcul est fini, il est stocké dans le vecteur \mathcal{I} , effaçant l'ancien morceau devenu inutile. Lors du calcul suivant, les valeurs nouvellement calculées sont utilisées quand elles sont disponibles.

Ce mécanisme garantit une convergence plus rapide, puisque chaque morceau est calculé à partir d'une meilleure approximation du point fixe. De plus, nous n'avons plus besoin que d'un seul vecteur \mathcal{I} au lieu des vecteurs $source\mathcal{I}$ et $dest\mathcal{I}$.

Calcul Combiné. Le calcul peut encore être accéléré, certes de façon limitée, en calculant l'*importance* et l'*utilité* des pages en même temps. Le point de blocage de notre algorithme est clairement les accès disque, puisque chaque itération effectuée une lecture complète de la matrice des liens. En calculant l'*importance* et l'*utilité* indépendamment, un parcours complet de la matrice

est effectué pour chaque itération de chacun de ces deux algorithmes. Nous pouvons mélanger et exécuter les deux algorithmes en une seule itération lors du parcours de la matrice des liens. Bien sûr, nous avons besoin de stocker en mémoire centrale les morceaux de \mathcal{T} des deux algorithmes. Par conséquent, la quantité de mémoire disponible est divisée par deux, et nous avons besoin de deux fois plus de morceaux pour exécuter le calcul mélangé. Néanmoins, malgré ce désagrément, il est (légèrement) plus efficace de gagner des lectures de la matrice que de doubler le nombre de morceaux en terme de coût.

Une difficulté lors du calcul de l'*utilité* est d'utiliser le résultat de l'*importance*. Une technique bien connue est de partir de la valeur précédemment calculée par l'algorithme, et d'utiliser ensuite les valeurs d'importance de plus en plus précises dès qu'elles sont disponibles.

Cette technique peut être généralisée pour tous les calculs multiples d'importance sur des domaines spécifiques basés sur la même matrice de liens.

Performances. Nous allons présenter, maintenant, quelques résultats d'évaluation des algorithmes développés. Des tests ont été réalisés sur des matrices de liens générées aléatoirement, paramétrées par le nombre moyen de liens par page.

Ces tests ont été effectués sur PC Intel Pentium II, doté de 128 Mo de mémoire centrale, un disque standard IDE (10 Go) sous le système d'exploitation Linux 2.0.36. Dans les mesures présentées ci-dessous, nous avons utilisé un graphe avec une moyenne de 10 liens sortant par page, ce qui semble être la moyenne des pages sur le web. Nous avons exécuté l'algorithme de calcul d'importance sur les pages XML (direct) et HTML (indirect), aussi bien que l'algorithme qui combine les deux. La Table VI.1 montre les temps d'exécution moyens pour chaque itération.

Pages\RAM		80Mo	40Mo
20 Millions Pages	Direct	178s	233s
	Inverse	149s	242s
40 Millions Pages	Direct	515s	670s
	Inverse	551s	772s

TAB. VI.1: *Performance du calcul d'importance des pages*

Nos expérimentations arrivent à un point fixe raisonnable après dix itérations. Pour une matrice aléatoire, ce nombre est en logarithme du nombre de pages. Donc pour 10^9 pages, 12 itérations sembleraient suffire. Nous espérons que la convergence soit aussi rapide pour les données réelles. Le principal

résultat est que le temps d'exécution est linéaire dans le nombre de pages grâce à la technique du découpage de la matrice. L'utilisation du découpage explique aussi pourquoi la pénalité d'utiliser peu de mémoire n'est pas si importante. La méthode combinée se révèle performante seulement quand la quantité de mémoire disponible est plus importante.

VI.4.5 Discussion

L'algorithme implanté s'applique à une machine unique. Comme le nombre de pages sur le web augmente très vite, cette solution n'est pas viable à long terme (sauf à imaginer que la taille des mémoires standard augmente aussi vite que le web). Deux alternatives sont brièvement discutées : distribuer l'algorithme ou calculer l'importance de façon incrémentale.

Calcul Distribué. Le découpage de la matrice suggère l'algorithme suivant. Un nombre de morceaux égal au nombre de machines participant au processus est alors utilisé. Lors de la phase d'initialisation de notre algorithme, chaque machine est affectée à un morceau, et récupère le morceau correspondant de la matrice. Le calcul de chaque morceau de \mathcal{I}^{i+1} s'effectue en parallèle, puisque les calculs sont indépendants. Plus précisément, lors de la i -ème itération, chaque machine C_k calcule \mathcal{I}_k^{i+1} (le k -ème morceau de \mathcal{I}^{i+1}) en utilisant \mathcal{I}^i et M_k (le k -ème morceau de M). Clairement, l'utilisation de cette distribution engendre des communications intensives sur le réseau puisque \mathcal{I}^i est distribué sur toutes les machines. Cependant, \mathcal{I}^i est petit par rapport à M , donc les accès disque continuent de prédominer.

De façon évidente toutes les machines exécutent l'algorithme de manière synchronisée, i.e. toutes les machines commencent l'itération $i + 1$ au même moment, une fois que tous les morceaux \mathcal{I}^i ont été calculés. Il est plus efficace de laisser chaque machine calculer à sa propre vitesse son propre morceau \mathcal{I}_k , obtenant des autres stations l la dernière version de \mathcal{I}_l qu'elles ont déjà calculée. Cette solution permet d'accélérer la convergence et permet de gagner sur le coût de la synchronisation.

Calcul Incrémental. Une alternative au précédent algorithme est le calcul incrémental. Un tel algorithme calcule graduellement l'importance/l'utilité, lors de l'exploration et de la mise à jour du graphe (par l'Interface Web par exemple).

VI.5 Ordonnanceur de Pages

La qualité de l'entrepôt de données dépend de façon critique des pages stockées et de leur fraîcheur. Nous voulons donc minimiser le délai entre l'arrivée d'une page XML sur le web et sa découverte par Xyleme. Nous voulons aussi minimiser le délai entre le temps du changement d'une page XML sur le web et le temps où ce changement est détecté dans Xyleme. Par conséquent, le problème principal est de rafraîchir les pages XML et aussi les pages HTML qui nous permettent de découvrir de nouvelles pages XML.

L'un des problèmes les plus critiques de Xyleme est de décider quand lire (encore) une page. Le but de l'Ordonnanceur de Pages est de minimiser l'obsolescence de la base XML de Xyleme sous certaines contraintes. Pour cela, ce module se base sur :

1. L'importance de la page (voir Section VI.4).
2. La fréquence de changement de la page. Nous verrons en Section VI.5.3 comment elle est définie et calculée.
3. La bande passante de l'Interface Web, i.e. le nombre de pages en moyenne que l'Interface Web peut rafraîchir dans une unité de temps.

L'ordonnanceur de Pages se doit de répartir la bande passante de l'Interface Web (facteur limitant essentiel du système) entre la découverte des pages et leur relecture. Le plan de cette section est le suivant. Tout d'abord, la notion de cycle de rafraîchissement est définie. La décision de lire pour la première fois une page est expliquée en section VI.5.1. La section VI.5.2 présentera la solution retenue pour rafraîchir les documents de la base XML. Finalement, le calcul de l'estimation de la fréquence de changement d'une page est présenté en section VI.5.3.

Cycle de Rafraîchissement. Le **cycle de rafraîchissement** est défini comme le plus court intervalle entre deux lectures successives de la même page. Cela signifie qu'une page ne sera jamais automatiquement lue plus d'une fois au cours d'un cycle. Notre système ne fixe pas de limite au nombre de lectures issues de demandes explicites de la part des utilisateurs. Dans nos expérimentations ce cycle a été fixé à 6 heures. La valeur du cycle de rafraîchissement ne peut pas être arbitrairement petite (de l'ordre de quelques minutes). En effet, le système doit parcourir toutes les informations du Gestionnaire de Metadonnées pour décider des pages à (re)lire. Par conséquent, la durée d'un cycle de rafraîchissement doit être égale ou légèrement supérieure à ce parcours.

Au cours du premier cycle, le système n'a aucune information sur les documents stockés dans la base XML (il n'y en a aucun). Durant ce cycle de démarrage, toutes les ressources du système seront allouées à la découverte des pages.

VI.5.1 Découverte des Pages

Nous avons besoin de parcourir de nouvelles portions du web pour découvrir de nouvelles pages. Pour allouer des ressources à la découverte, nous utilisons trois paramètres.

Notation VI.5.1 Soit σ_X (respectivement σ_H), le pourcentage de ressources allouées pour la découverte des pages XML (resp. HTML) .

□

Il est important d'observer qu'en général nous ne savons pas a priori si une page est du type XML ou HTML. Typiquement, le type de la page est connu grâce à son type MIME [78, 82] retourné par le serveur web ou en analysant la page. Lors de la découverte d'une nouvelle URL, ce genre d'information n'est pas disponible. Toutefois, le système peut *suspecter* qu'une page est XML si, par exemple, son URL finit par le suffixe “.xml”, “.wml”, “.rss”, etc. ou si c'est un lien avec un suffixe inconnu que le système identifie comme dénotant des pages XML. Le ratio σ_X est pour de telles pages.

Le choix de la page à découvrir se fait de manière assez brutale. L'Ordonnanceur des Pages maintient un seuil basé sur l'importance des pages jamais lues, calculée au cours du parcours. Les pages ayant une importance supérieure à ce seuil sont envoyées à l'Interface Web pour être lues.

Le terme $\sigma_X + \sigma_H$ représente la proportion des pages qui sont des pages lues pour la première fois (proportion affectée à la découverte). Le reste des ressources, i.e $(100 - \sigma_X - \sigma_H)$, est utilisé pour rafraîchir les pages. Ces paramètres sont fixés par l'administrateur du système, e.g $\sigma_X = \sigma_H = 20\%$, et peuvent être changés dynamiquement.

VI.5.2 Rafraîchissement des Pages

Cette section présente le rafraîchissement des pages XML. Le rafraîchissement des pages HTML utilise des techniques similaires. La seule différence est dans la définition de l'importance des pages.

Le processus de rafraîchissement doit répartir les ressources que le système lui alloue entre deux types de pages, les pages XML et HTML. Pour répartir ces ressources, un *facteur de rafraîchissement* χ est utilisé.

Définition VI.5.2 Soit χ le pourcentage de ressources allouées pour le rafraîchissement des pages XML.

□

Le pourcentage $100 - \sigma_X - \sigma_H - \chi$ représente le pourcentage de ressources allouées pour rafraîchir les pages HTML.

La stratégie de rafraîchissement est articulée comme un problème d'optimisation. Pour ce faire, une fonction de coût est utilisée (présentée ici de façon simplifiée). De plus amples informations sur cette fonction peuvent être trouvées dans [124].

Obsolescence. Un document dans Xyleme qui n'est plus identique à sa source sur le web est obsolète. Néanmoins, il existe différents degrés d'obsolescence : au cours du temps, le document web peut subir des modifications, sa copie dans Xyleme, devient pendant ce temps de plus en plus "obsolète". Une version d'un document qui est vieille d'un jour peut être satisfaisante pour un utilisateur, même si elle ne correspond plus à sa source sur le web, alors qu'une page vieille d'un an est tout bonnement inacceptable. Nous distinguons alors deux comportements dans l'obsolescence : au moins un changement a probablement été manqué, le document est dit "obsolète"; plusieurs changements ont été manqués, le document est alors dit "âgé".

Comme mentionné auparavant, Xyleme n'a pas une connaissance complète des modifications des documents du web. De même que [55, 90], l'hypothèse que les changements d'une page web suivent un processus de Poisson avec une fréquence λ est acceptée. Certaines pages suivent un motif de changement totalement différent, e.g. des pages ne sont modifiées qu'à des dates fixes. Il est facile de les capturer si des informations sur la date de dernière modification ont été fournies à Xyleme par les administrateurs de ces pages. Ce type de modification est ignoré ici. La section VI.5.3 présente l'estimation de λ_i pour chaque page i .

Pour capturer le coût de l'obsolescence, nous devons choisir pour coût, une fonction de t et de λ . En accord avec la discussion précédente, cette fonction devrait prendre en compte le fait que la page n'est plus à jour, ainsi que son âge. De façon grossière, le premier composant augmente jusqu'à ce que la page devienne obsolète, alors que le second continue d'augmenter au cours du temps. Différentes applications peuvent assigner différents poids à

chacun de ces deux composants. La fonction $(\lambda t)^\alpha$ est sélectionnée pour coût. Les avantages d'une telle fonction sont de deux ordres:

1. Elle est une approximation raisonnable d'une fonction de coût "normale" qui prendrait en compte les deux facteurs.
2. Elle facilite la recherche d'un programme optimal.

Bien qu'il soit concevable que α dépende du document, nous manquons de connaissances sur la sémantique du document pour l'estimer dynamiquement. Donc, une valeur fixe pour α est prise. Le paramètre α est utilisé pour ajuster les deux facteurs d'obsolescence mentionnés auparavant. En choisissant de façon appropriée la valeur de α , l'administrateur du système a les moyens d'influencer la stratégie du système, e.g. plus la valeur de α est petite, plus l'accent est mis sur l'obsolescence par rapport à l'âge.

Nous considérons aussi que l'obsolescence de documents importants a un impact (négatif) plus important que l'obsolescence d'un document moins important. De façon plus spécifique, le coût d'un document en train de devenir obsolète est aussi proportionnel à son importance. Par conséquent, la fonction suivante peut être utilisée pour évaluer le *coût d'une page i qui n'a pas été rafraîchie pendant un temps t* :

$$d_i(t) = w_i(\lambda_i t)^\alpha \quad (\text{VI.7})$$

où w_i est l'importance estimée de la page i , λ_i sa fréquence de changement estimée, et α est un paramètre.

Remarque VI.5.3 Une première observation sur la régularité d'un cycle de rafraîchissement peut être très utile. Supposons que la page i a une fréquence de rafraîchissement λ_i . Supposons que nous décidions d'allouer une certaine bande passante à cette page, soit m_i accès par mois, i.e. f_i accès par seconde. Alors il est optimal de rafraîchir cette page uniformément au cours de cet intervalle, i.e. la rafraîchir tout les $1/f_i$ secondes. \square

Supposons que nous utilisons une fréquence d'accès f_i pour la page i pour chaque i . Le coût moyen de la page i est :

$$c_i(f_i) = \frac{\int_0^{1/f_i} d_i(t) dt}{1/f_i} \quad (\text{VI.8})$$

Maintenant, soit N le nombre total de pages dans l'entrepôt. Nous définissons le *coût de l'obsolescence de l'entrepôt de Xyleme* comme la somme des coûts des pages individuelles, i.e. :

$$COST = \sum_{i=1}^N c_i(f_i) \quad (VI.9)$$

Cette fonction de coût représente l'obsolescence moyenne de l'entrepôt de Xyleme. Nous voulons minimiser ce coût sous une contrainte, la *bande passante* de l'Interface Web. Soit G le nombre de pages que l'Interface Web peut rafraîchir au cours d'un intervalle temporel donné (cycle de rafraîchissement). Cela introduit une contrainte sur la fréquence d'accès des pages :

$$(c) \quad \sum_{i=1}^N f_i - G = 0$$

Le problème revient donc à calculer $f_i, i = 1 \dots N$, qui minimise $COST$ sous la contrainte (c). En utilisant les facteurs de Lagrange, nous obtenons :

$$f_i = \frac{G}{S} \alpha^{+1} \sqrt{\alpha} w_i \lambda^{\alpha_i} \quad (VI.10)$$

où

$$S = \sum_{i=1}^N \alpha^{+1} \sqrt{\alpha} w_i \lambda^{\alpha_i} \quad (VI.11)$$

La Formule VI.10 est la formule de base de l'Ordonnanceur de Pages de Xyleme.

La façon dont f_i est calculée garantit que, après une période de démarrage, l'Ordonnanceur de Pages atteindra un point d'équilibre, tel que le nombre de pages en attente d'une relecture dans une même unité de temps est très proche de G .

Implantation L'Ordonnanceur de Pages maintient une variable globale S (équation VI.11). Pour une page qui vient juste d'être rafraîchie, la vieille valeur λ_i (l'estimation de la fréquence de rafraîchissement que le système avait avant la relecture), ainsi que le nouveau λ_i (l'estimation calculée après la relecture) sont connus. La valeur de S est mise à jour suivant l'équation :

$$S_{new} = S_{old} - \sqrt[\alpha+1]{w\lambda_{old}^\alpha} + \sqrt[\alpha+1]{w\lambda_{new}^\alpha} \quad (\text{VI.12})$$

De cette manière, la valeur courante de S est toujours connue.

Soit T la durée du cycle de rafraîchissement. Cette valeur est fixée par l'administrateur du système. L'Ordonnanceur de Pages effectue un parcours complet de l'ensemble des pages une fois par *cycle de rafraîchissement*. Ceci explique pourquoi nous interdisons un cycle de rafraîchissement trop court. Quand l'Ordonnanceur de Pages visite une page i (les métadonnées de la page i), il peut facilement détecter si cette page doit ou non être rafraîchie.

En effet, f_i est fonction de la valeur de S , de l'importance (w_i), de la fréquence de changement (λ_i), comme montré dans l'équation VI.10. En utilisant alors la date de dernier accès (t_i), la page doit être rafraîchie si $(t_i - t_{now}) > 1/f_i$, i.e., si le temps écoulé depuis la dernière lecture est plus grand que la période de rafraîchissement de la page. Dans ce cas, l'Ordonnanceur de Pages envoie une requête à l'Interface Web pour lire la page. La façon dont la fréquence d'accès à la page est calculée garantit que l'Ordonnanceur de Pages accédera exactement⁴ à G pages par cycle de rafraîchissement. Comme le cycle de rafraîchissement utilisé est relativement grand (six heures dans Xyleme), un parcours complet des pages par unité de temps est parfaitement acceptable, c'est-à-dire pas trop coûteux.

VI.5.3 Estimation de la fréquence de changement

Le calcul de la fréquence de changements d'une page se base fortement sur les résultats de [53] et [90].

Les informations disponibles afin d'estimer λ peuvent provenir de plusieurs observations. Ainsi quand le système relit une page il peut savoir si cette page a changé ou non depuis le dernier accès. Quand une telle information est disponible, elle est appelée *existence d'un changement* car le système sait seulement si une page a changé ou pas entre deux accès, mais ne sait pas quand ce changement a eu lieu et combien de fois cette page a changé.

Les serveurs web peuvent aussi fournir, avec la page, la date du dernier changement de la page, appelée *dernière date de changement*. De manière évidente, la *dernière date de changement* fournit plus d'informations que l'*existence d'un changement*. Ainsi à partir de séquence de telles observations, le système peut estimer le taux de changement (λ) d'un page.

4. En vérité, comme l'entrepôt change pendant que nous effectuons un cycle de rafraîchissement, nous ne rafraîchissons pas "exactement" G pages, mais un nombre de pages très proche

Le reste de la section est divisé en deux parties. La première partie (section VI.5.3.1) décrit l'estimation de λ dans le cas où seule l'*existence d'un changement* est connue du système. La seconde section (section VI.5.3.2) discute du cas où la *dernière date de changement* est connue.

VI.5.3.1 Existence de changement

Ce cas est formalisé de la façon suivante. La page est accédée $N + 1$ fois, entre les dates $t_i, i = 0 \dots N$. Le i -ème intervalle d'accès est donnée par $[t_i, T_{i+1}], i = 1 \dots N$. Si le système a observé un changement au cours de cette intervalle, il est appelé *intervalle de changement*, autrement il est appelé *intervalle de non changement*.

Définition VI.5.4 Soit C (respectivement V) le nombre d'intervalles de changement (resp. de non-changement). Nous définissons aussi $c_i, i = 1 \dots C$ (resp. $v_i, i = 1 \dots V$) la longueur de l'intervalle de changement (resp. non-changement).

□

Quand $C > 0$ et $V > 0$, nous pouvons estimer la fréquence de changement λ , comme celle qui maximise la fonction :

$$f(x) = \left(e^{-x \sum_{i=1}^V v_i} \right) \prod_{j=1}^C (1 - e^{-c_j x}) \quad (\text{VI.13})$$

Dans le cas général, la valeur qui maximise la fonction VI.13 ne peut être obtenue de manière analytique. Dans ce cas, des approximations ou des algorithmes numériques peuvent être utilisés.

Le cas particulier de l'*existence d'un changement* avec des accès réguliers ($\forall i, t_{i+1} - t_i = \text{constante}$) est présenté dans [53]. De toute façon, la contrainte d'un accès régulier n'est pas acceptable dans notre système.

VI.5.3.2 Dernière date de changement

L'estimation de la fréquence de changement λ basée sur cette information est plus précise que l'estimation présentée lors de la section précédente. Cette méthode sera donc préférée quand elle est disponible.

En ajout aux notations introduites précédemment, nous utilisons $d_i, i = 0 \dots N$ pour désigner la dernière date de changement lors du i -ème accès. Un *intervalle de changement* existe si $t_{i-1} < d_i < t_i, i = 1 \dots N$. Le premier accès

représentera toujours un *intervalle de changement* ($d_0 < t_0$). Pour tous les *intervalles de changement* i , la notation suivante $l_i = t_i - d_i$ est prise.

Les notations précédentes permettent d'estimer la valeur λ ainsi :

$$\lambda = \frac{C}{\sum_{i=1}^C l_i} \quad (\text{VI.14})$$

Un résultat similaire est présenté dans les travaux de Cho [53].

Conclusion

Cette thèse a présenté deux contributions correspondant à deux visions complémentaires pour la construction de systèmes contrôlant les changements de données semi-structurées au format XML. Ce chapitre rappelle les points forts des deux visions développées avant de terminer sur les perspectives ouvertes par cette thèse.

Intranet

La première contribution se place dans le cadre d'applications construites par une entreprise sur le web à partir d'une base de données accessible sur un intranet. La seule fonctionnalité indispensable à notre approche est que cette base de données soit dotée de mécanismes actifs permettant la notification d'un changement dans la base. Cette vision propose un langage déclaratif dans le but de déployer rapidement des applications sur le web basées sur des vues actives des données de la base. Ce langage apporte des particularités permettant de spécifier la propagation des changements entre la base de données et les interfaces web finales afin de satisfaire la contrainte de temps d'accès à l'information. Ce contrôle des changements a donné lieu à un prototype.

Deux compilateurs ainsi qu'un système complet ont été réalisés dans le cadre d'un projet appelé *ActiveView* décrit lors de la première partie de la thèse. Le premier compilateur transforme une spécification d'application écrite dans notre langage en un fichier décrivant de manière abstraite les interfaces utilisateurs et en un ensemble de fichiers Java implantant les fonctionnalités spécifiées ainsi que les couches de communication avec notre système. Les fichiers Java générés peuvent être personnalisés en changeant le code des méthodes propres à l'application. Le fichier décrivant l'interface de manière abstraite peut être lui aussi personnalisé avant d'être donné en entrée à notre second compilateur. Ce compilateur est quant à lui chargé de générer les interfaces utilisateur finales composées de fichiers HTML encapsulant des scripts écrits en Javascript et un applet Java chargé de communiquer avec le

système au travers du web par le biais de la couche RMI de Java. Les fichiers HTML générés peuvent être aussi, à leur tour, personnalisés.

Le système *ActiveView* est composé d'un moteur de requête XML, d'un gestionnaire de règles actives, d'un gestionnaire de trace ainsi que d'un gestionnaire de mise à jour. De plus à chaque utilisateur est associé un processus indépendant, qui implante les fonctionnalités spécifiées par l'application. Ce processus est chargé de faire le lien entre l'interface utilisateur et les différents composants du système cité ci-dessus.

Malgré les deux années qui se sont écoulées l'approche déclarative *Activeview* reste novatrice. De plus le choix des technologies, Java, XML sont encore au goût du jour.

Internet

La seconde vision contribution se situe dans le contexte du projet Xyleme. Le but de ce projet est de construire un entrepôt de données de documents XML accessible à partir du web et de produire des services de haut niveaux sur ces documents. Au nombre de ces services nous pouvons énumérer, entre autres, un langage de requêtes, un mécanisme de souscription aux changements et un mécanisme de versions des documents.

La deuxième partie de cette thèse présente la solution envisagée dans ce cadre afin d'acquérir et de rafraîchir automatiquement les documents XML se trouvant sur le web. Pour cela nous affectons à chacun un coût, le système n'ayant plus qu'à minimiser le coût de l'ensemble des pages sous la contrainte de la bande passante du web disponible. Le coût de la page est basé sur son importance relative par rapport aux autres pages ainsi que sur son taux de changements estimé. L'importance des pages est définie de manière classique comme le calcul du point fixe de la matrice des liens générée par les documents connus du système. Le taux de changement d'une page est lui aussi modélisé classiquement par un processus de Poisson.

La nouveauté de notre approche est double. Premièrement nous avons non seulement défini une notion d'importance pour les pages XML mais aussi nous avons introduit la notion d'utilité des pages pour les pages HTML référençant les pages XML. Cette notion d'utilité se calcule de manière analogue en injectant l'importance des pages XML aux pages HTML en remontant les liens, c'est-à-dire en effectuant un point fixe sur la matrice transposée des liens. La seconde innovation de notre approche par rapport à celles décrites dans la littérature est due à la modélisation inflationniste de notre système. En effet, il ne se contente pas de rafraîchir des pages mais il en acquiert aussi de nouvelles. L'approche a été validé par un prototype utilisé par la société

Xyleme créée à partir du projet de recherche.

La seconde partie de la thèse a aussi présenté des travaux annexes tels qu'un mécanisme de version et de souscription des documents, mécanismes ayant eux aussi fait l'objet de prototypes.

Perspectives

Les perspectives ouvertes par cette thèse sont nombreuses.

Le modèle client-serveur *ActiveView*, a perdu un peu de son attrait au profit des services web. Une activité, tels que nous l'avions défini, se re-définit alors comme une interface regroupant un ensemble de services web sur l'entrepôt de données. Néanmoins, l'approche déclarative de la gestion des vues actives afin de produire des services est maintenant universellement reconnue. Ainsi le langage WSDL permet de spécifier les services Web accessible par le biais du protocole SOAP. Tous les composants du système se renomment facilement d'interface RMI en service Web. Ils seraient intéressant de transposer de la connaissance acquise lors de cette thèse afin aux technologie émergeant actuellement.

La deuxième partie de la thèse est quant à elle en constante mouvance du fait de son intégration dans la start-up Xyleme. Ainsi de nombreux travaux ont été réalisés afin de distribuer l'algorithme de distribution. Une des avancées les plus significatives est l'implantation d'un algorithme calculant incrémentalement l'importance des pages, permettant ainsi de se passer du coûts, en temps et en espace, de stockage de la matrice des liens. Les perspectives les plus intéressantes pour la suite sont de comprendre et d'analyser les différents résultats acquis. Ainsi une analyse des fichiers XML ainsi que des différentes statistiques sous-jacentes comme leurs fréquences et leurs proportions de changements ouvriraient certainement de nouveaux horizons pour l'optimisation du système.

Bibliographie Personnelle

- [i] Serge Abiteboul, Vincent Aguiléra, Sébastien Ailleret, Bernd Amann, Sophie Cluet, Brendan Hills, Frédéric Hubert, Jean-Claude Mamou, Amélie Marian, Laurent Mignet, Tova Milo, Cassio Souza dos Santos, Bruno Tessier, and Anne-Marie Vercoustre. XML Repository and Active Views Demonstration. In Atkinson et al. [25], pages 742–745. ISBN 1-55860-615-7.
- [ii] Serge Abiteboul, Bernd Amann, Sophie Cluet, Anat Eyal, Laurent Mignet, and Tova Milo. Active Views for Electronic Commerce. In Atkinson et al. [25], pages 138–149. ISBN 1-55860-615-7.
- [iii] Serge Abiteboul, Sophie Cluet, Laurent Mignet, and Tova Milo. Declarative Specification of Electronic Commerce Applications. *IEEE Data Engineering Bulletin*, 23(1):37–42, March 2000.
- [iv] Amélie Marian, Serge Abiteboul, Grégory Cobéna, and Laurent Mignet. Change-Centric Management of Versions In an XML Warehouse. In Appear [23], pages 581–590. ISBN 1-55860-804-4.
- [v] Laurent Mignet, Vincent Aguiléra, Sébastien Ailleret, and Pierangelo Veltri. XyRo: The Xyleme Robot Architecture. First workshop Data Integration over the Web, June 2001.
- [vi] Amélie Marian, Serge Abiteboul, and Laurent Mignet. Change-centric management of versions in an XML Warehouse. In *Base de Données Avancées*, pages 281 – 303, 2000.
- [vii] Laurent Mignet, Mihai Preda, Serge Abiteboul, Sébastien Ailleret, Bernd Amann, and Amélie Marian. Acquiring XML pages for a WebHouse. In *Base de Données Avancées*, pages 241 – 263, 2000.
- [viii] Benjamin Nguyen, Serge Abiteboul, Grégory Cobéna, and Laurent Mignet. Query Subscription in an XML Webhouse. In *Proceedings of the*

First DELOS Network Excellence Workshop, pages 109–114, Zurich, December 2000. ERCIM.

Bibliographie

- [1] Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors. *Proceedings of 26th International Conference on Very Large Data Bases*, Cairo - Egypt, September 10-14 2000. Morgan Kaufmann. ISBN 1-55860-715-3.
- [2] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.
- [3] Serge Abiteboul. On Views and XML. In ACM [11], pages 1–9. ISBN 1-58113-062-7.
- [4] Serge Abiteboul, Vincent Aguiléra, Sébastien Ailleret, Bernd Amann, Sophie Cluet, Brendan Hills, Frédéric Hubert, Jean-Claude Mamou, Amélie Marian, Laurent Mignet, Tova Milo, Cassio Souza dos Santos, Bruno Tessier, and Anne-Marie Vercoustre. XML Repository and Active Views Demonstration. In Atkinson et al. [25], pages 742–745. ISBN 1-55860-615-7.
- [5] Serge Abiteboul, Bernd Amann, Sophie Cluet, Anat Eyal, Laurent Mignet, and Tova Milo. Active Views for Electronic Commerce. In Atkinson et al. [25], pages 138–149. ISBN 1-55860-615-7.
- [6] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web - From Relations to Semistructured Data and XML*. Morgan Kaufmann, San Francisco California, 2000.
- [7] Serge Abiteboul, Sophie Cluet, Laurent Mignet, and Tova Milo. Declarative Specification of Electronic Commerce Applications. *IEEE Data Engineering Bulletin*, 23(1):37–42, March 2000.
- [8] Serge Abiteboul, Sophie Cluet, and Tova Milo. Querying and Updating the File. In Agrawal et al. [14], pages 73–84. ISBN 1-55860-152-X.

- [9] Serge Abiteboul, Sophie Cluet, and Tova Milo. A Logical View of Structured Files. *VLDB Journal*, 7(2):96–114, 1998.
- [10] Serge Abiteboul, Jason McHugh, Michael Rys, Vasilis Vassalos, and Janet L. Wiener. Incremental Maintenance for Materialized Views over Semistructured Data. In Gupta et al. [83], pages 38–49. ISBN 1-55860-566-5.
- [11] ACM, editor. *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Philadelphia - Pennsylvania - USA, May 31 - June 2 1999. ACM Press. ISBN 1-58113-062-7.
- [12] ACM, editor. *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Dallas - Texas - USA, May 15-17 2000. ACM Press. ISBN 1-58113-214-X.
- [13] Michel E. Adiba and Bruce G. Lindsay. Database Snapshots. In Lochovsky and Taylor [102], pages 86–91.
- [14] Rakesh Agrawal, Seán Baker, and David A. Bell, editors. *Proceedings of 19th International Conference on Very Large Data Bases*, Dublin - Ireland, August 24-27 1993. Morgan Kaufmann. ISBN 1-55860-152-X.
- [15] Vincent Aguiléra. X-OQL.
<http://www-rocq.inria.fr/~aguilera/xoql/index.html>.
- [16] Vincent Aguiléra, Sophie Cluet, and Fanny Watez. Pattern Tree Queries in Xyleme. Technical Report 200, INRIA - Verso Project, 2001.
- [17] Sébastien Ailleret. Gestionnaire de Règles pour Vues Actives. Rapport de Stage Polytechnique - 1999.
- [18] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis, K. Tolle, B. Amann, I. Fundulaki, M. Scholl, and A-M. Vercoustre. Managing RDF Metadata for Community Webs. In *Proceedings of the 2nd International Workshop on The World Wide Web and Conceptual Modelling*, pages 140–151, 2000.
- [19] Mehmet Altinel, Demet Aksoy, Thomas Baby, Michael J. Franklin, William Shapiro, and Stanley B. Zdonik. DBIS-Toolkit: Adaptable Middleware for Large Scale Data Delivery. In Delis et al. [63], pages 544–546. ISBN 1-58113-084-8.

- [20] Bernd Amann and Irimi Fundulaki. Integrating Ontologies and Thesauri to Build RDF Schemas. In Serge Abiteboul and Anne-Marie Vercoustre, editors, *Proc. of the 3rd European Conference on Research and Advanced Technology for Digital Libraries (ECDL)*, volume 1696 of *Lecture Notes in Computer Science*, pages 234–253, Paris, France, September 1999. Springer Verlag.
- [21] Bernd Amann, Irimi Fundulaki, and Michel Scholl. Integrating ontologies and thesauri for RDF schema creation and metadata querying. *International Journal on Digital Libraries*, 3(3):221–236, October 2000.
- [22] Bernd Amann, Irimi Fundulaki, Michel Scholl, Catriel Beeri, and Anne-Marie Vercoustre. Mapping XML Fragments to Community Web Ontologies. In Informal Proceedings, editor, *Fourth International Workshop on the Web and Databases*, pages 97–102, 2001. in conjunction with SIGMOD 2001.
- [23] Peter M.G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors. *Proceedings of 27th International Conference on Very Large Data Bases*, Roma - Italia, September 11-14 2001. Morgan Kaufmann. ISBN 1-55860-804-4.
- [24] Sacha Arnaud. Vues actives pour le commerce électronique. Rapport de Stage Polytechnique - 1998.
- [25] Malcolm P. Atkinson, Maria E. Orłowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors. *Proceedings of 25th International Conference on Very Large Data Bases*, Edinburgh - Scotland, September 7-10 1999. Morgan Kaufmann. ISBN 1-55860-615-7.
- [26] Guruduth Banavar, Tushar Deepak Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *Proceedings of the 19th International Conference on Distributed Computing Systems*, pages 262–272, Austin, TX, USA, 31 May - 4 June 1999. IEEE Computer Society.
- [27] F. Bancilhon, C. Delobel, and P. Kanellakis. *The O2 book*. Morgan Kaufman, 1992.
- [28] François Bancilhon and David J. DeWitt, editors. *Fourteenth International Conference on Very Large Data Bases*, Los Angeles, California, USA, August 29 - September 1 1988. Morgan Kaufmann. ISBN 0-934613-75-3.

- [29] François Bancilhon and Nicolas Spyratos. Update Semantics of Relational Views. *ACM Transactions on Database Systems (TODS)*, 6(4):557–575, 1981.
- [30] Klemens Böhm, Karl Aberer, Erich J. Neuhold, and Xiaoya Yang. Structured Document Storage and Refined Declarative and Navigational Access Mechanisms in HyperStorM. *VLDB Journal*, 6(4):296–311, 1997.
- [31] Stephen L. Bloom and Zoltán Esik. *Iteration theories: the equational logic of iterative processes*. EATCS monographs on theoretical computer science. Springer, 1993. ISBN : 3-540-56378-4.
- [32] Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors. *Proceedings of 20th International Conference on Very Large Data Bases*, Santiago de Chile - Chile, September 12-15 1994. Morgan Kaufmann. ISBN 1-55860-153-8.
- [33] Angela Bonifati, Stefano Ceri, and Stefano Paraboschi. Pushing Reactive Services to XML Repositories using Active Rules. In *Tenth International World Wide Web Conference, WWW 10*, pages 633–641, Hong Kong, China, May 1-5 2001. ACM. ISBN 1-58113-348-0.
- [34] Haran Boral and Per-Åke Larson, editors. *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois - USA, June 1-3 1988. ACM Press.
- [35] Brian E. Brewington and George Cybenko. How dynamic is the Web? *Computer Networks and ISDN Systems*, 33(1-6):257–276, June 2000.
- [36] Sergey Brin, Rajeev Motwani, Lawrence Page, and Terry Winograd. What can you do with a Web in your Pocket? *IEEE Data Engineering Bulletin*, 21(2):37–47, 1998.
- [37] Sergey Brin and Lawrence Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, April 1998.
- [38] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 219–227, Portland, Oregon, USA, July 16-19 2000. ACM.

- [39] Wojciech Cellary and Geneviève Jomier. Consistency of Versions in Object-Oriented Databases. In McLeod et al. [108], pages 432–441.
- [40] Wojciech Cellary and Geneviève Jomier. Consistency of Versions in Object-Oriented Databases. In F. Bancilhon, C. Delobel, and P. Kannelakis, editors, *The O2 book*, chapter 8, pages 447–462. Morgan Kaufman, 1992.
- [41] Wojciech Cellary, Geneviève Jomier, and G. Vossen. Multiversion Object Constellations: A New Approach to Support a Designer’s Database Work. *Engineering with Computers*, 10:230–244, 1994.
- [42] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic Generation of Production Rules for Integrity Maintenance. *ACM Transactions on Database Systems (TODS)*, 19(3):367–422, 1994.
- [43] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Constraint Maintenance. In McLeod et al. [108], pages 566–577.
- [44] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In McLeod et al. [108], pages 577–589.
- [45] Stefano Ceri and Jennifer Widom. Production Rules in Parallel and Distributed Database Environments. In Yuan [180], pages 339–351. ISBN 1-55860-151-1.
- [46] Stefano Ceri and Jennifer Widom. Managing Semantic Heterogeneity with Production Rules and Persistent Queues. In Agrawal et al. [14], pages 108–119. ISBN 1-55860-152-X.
- [47] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery. *Computer Networks and ISDN Systems*, 31(11-16):1623–1640, May 1999.
- [48] Sudarshan S. Chawathe, Serge Abiteboul, and Jennifer Widom. Managing Historical Semistructured Data. *Theory and Practice of Object Systems (TAPOS)*, 5(3):143–162, 1999.
- [49] Jianjun Chen, David DeWitt, Fend Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for the internet databases. In Delis et al. [63], pages 379–390. ISBN 1-58113-084-8.

- [50] Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas - Texas - USA, May 16-18 2000. ACM. ISBN 1-58113-218-2.
- [51] Shu-Yao Chien, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient management of Multiversion Documents by Object Referencing. In Apers et al. [23], pages 291–300. ISBN 1-55860-804-4.
- [52] Junghoo Cho, Hector Garci-Molina, and Lawrence Page. Efficient crawling through URL ordering. *Computer Networks and ISDN Systems*, 30(1-7):161–172, April 1998.
- [53] Junghoo Cho and Hector Garcia-Molina. Estimating Frequency of Change. Technical report, Stanford University, 2000. <http://dbpubs.stanford.edu:8090/pub/1999-22/>.
- [54] Junghoo Cho and Hector Garcia-Molina. Synchronizing a database to improve freshness. In Chen et al. [50], pages 117–128. ISBN 1-58113-218-2.
- [55] Junghoo Cho and Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In Abbadi et al. [1], pages 200–209. ISBN 1-55860-715-3.
- [56] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your Mediators Need Data Conversion! In Haas and Tiwary [85], pages 177–188. ISBN 0-89791-995-5.
- [57] Sophie Cluet, Pierangelo Veltri, and Dan Vodislav. Views in a Large Scale XML Repository. In Apers et al. [23], pages 271–280. ISBN 1-55860-804-4.
- [58] Grégory Cobéna, Serge Abiteboul, and Amélie Marian. Detecting Changes in XML Documents. In *Base de Données Avancées*, 2001.
- [59] Christophe de Maindreville and Eric Simon. A Production Rule-Based Approach to Deductive Databases. In *Proceedings of the Fourth International Conference on Data Engineering, February 1-5, 1988, Los Angeles, California, USA*, pages 234–241. IEEE Computer Society, 1988. ISBN 0-8186-0827-7.
- [60] Christophe de Maindreville and Eric Simon. Modelling Non Deterministic Queries and Updates in Deductive Databases. In Bancilhon and DeWitt [28], pages 395–406. ISBN 0-934613-75-3.

- [61] Lois M. L. Delcambre and James N. Etheredge. A Self-Controlling Interpreter for the Relational Production Language. In Boral and Larson [34], pages 396–403.
- [62] Lois M. L. Delcambre and James N. Etheredge. The Relational Production Language: A Production Language for Relational Databases. In Larry Kerschberg, editor, *Expert Database Systems, Proceedings From the Second International Conference*, pages 333–351, Vienna, Virginia, USA, April 25-27 1988. Benjamin Cummings 1989. ISBN 0-8053-0311-1.
- [63] Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors. *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data*, Philadelphia - Pennsylvania - USA, June 1-3 1999. ACM Press. ISBN 1-58113-084-8.
- [64] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In Delis et al. [63], pages 431–442. ISBN 1-58113-084-8.
- [65] Cassio Souza dos Santos, Serge Abiteboul, and Claude Delobel. Virtual schemas and bases. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 1994.
- [66] Fred Douglass, Anja Feldmann, Balachander Krishnamurthy, and Jeffrey C. Mogul. Rate of Change and other Metrics: a Live Study of the World Wide Web. In *1st USENIX Symposium on Internet Technologies and Systems*, Monterey, California, USA, December 8-11 1997.
- [67] Jenny Edwards, Kevin S. McCurley, and John A. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *Tenth International World Wide Web Conference, WWW 10*, pages 106–113, Hong Kong, China, May 1-5 2001. ACM. ISBN 1-58113-348-0.
- [68] Kapali P. Eswaran and Donald D. Chamberlin. Functional Specifications of Subsystem for Database Integrity. In Kerr [95], pages 48–68.
- [69] K.P. Eswaran. Specifications, implementations and interactions of a trigger subsystem in an integrated database system. Technical Report 1820, IBM San Jose Research Laboratory, August 1976.

- [70] Anat Eyal and Tova Milo. Integrating and customizing heterogeneous e-commerce applications. *VLDB Journal*, 2002. To appear.
- [71] Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In Sellis and Mehrotra [132], pages 115–126. ISBN 1-58113-332-4.
- [72] Xiang Fu, Tevfik Bultan, Richard Hull, and Jianwen Su. Verification of Vortex Workflows. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy*, volume 2031 of *Lecture Notes in Computer Science*, pages 143–157. Springer, April 2-6 2001. ISBN 3-540-41865-2.
- [73] A.L. Furtado and M.A. Casanova. Updating Relational Views. In W. Kim, D.S. Reiner, and D.S. Batory, editors, *Query Processing in Database Systems*, chapter 7, pages 127–142. Springer-Verlag, 1985.
- [74] Hector Garcia-Molina and H. V. Jagadish, editors. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, USA, May 23-25 1990. ACM Press.
- [75] Shahram Ghandeharizadeh, Richard Hull, and Dean Jacobs. Heraclitus: Elevating Deltas to be First-Class Citizens in a Database Programming Language. *ACM Transactions on Database Systems (TODS)*, 21(3):370–426, 1996.
- [76] M. Gondran and M. Minoux. *Graphes et Algorithmes*. Eyrolles, 1995.
- [77] Data Base Task Group. Report to the CODASYL Programming Language Committee. *Data Base*, 2(2):11–18, 1970.
- [78] Network Working Group. MIME (Multipurpose Internet Mail Extensions part one : Mechanism for specifying and describing the format of internet message bodies. RFC 1521.
- [79] Network Working Group. The HyperText Transfert Protocol – HTTP/1.1. RFC 2616.
- [80] Network Working Group. The MD5 Message-Digest Algorithm. RFC 1321.

- [81] Network Working Group. The Robots Exclusion Protocol. Internet Draft.
- [82] Network Working Group. XML Media Types. RFC 2376.
- [83] Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors. *Proceedings of 24rd International Conference on Very Large Data Bases*, New York City - USA, August 24-27 1998. Morgan Kaufmann. ISBN 1-55860-566-5.
- [84] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene J. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(1):143–160, 1990.
- [85] Laura M. Haas and Ashutosh Tiwary, editors. *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, USA, June 2-4 1998. ACM Press. ISBN 0-89791-995-5.
- [86] T.H. Haveliwala. Efficient computation of pagerank. Technical report, Stanford Database Group, 1999.
- [87] Matthias Jarke, Maurizio Lenzerini, Yannis Vassiliou, and Panos Vassiliadis, editors. *Fundamentals of Data Warehouses*. Springer-Verlag, 2000. Rapport of Esprit Project Number 22469.
- [88] Distributed Events Specification. <http://java.sun.com/products/javaspaces/specs/>.
- [89] Jérémy Jouglet. Souscription de requêtes dans un entrepôt de données XML. Rapport de Stage Polytechnique - 2000.
- [90] E.G. Coffman Jr., Khen Liu, and Richard R. Weber. Optimal Robot Scheduling for Web Search Engines. Technical report, INRIA Number 3317, 1997.
- [91] Astrid M. Julienne and Brian Holtz. *ToolTalk and Open Protocols: Inter-Application Communication*. Prentice-Hall ECS Professional, 1994. ISBN 0-13-031055-7.
- [92] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML data. Technical Report 8/99, University of Mannheim, 1999. available

at <http://pi3.informatik.uni-mannheim.de/publications/techrep899.ps>.

- [93] Arthur M. Keller. Updates to Relational Databases Through Views Involving Joins. In Peter Scheuermann, editor, *International Conference on Data and Knowledge Base (JCDKB)*, pages 363–384. Academic Press, 1982.
- [94] John G. Kemeny and James Laurie Snell. *Finite Markov chains*. Springer, 1983. ISBN: 0-387-90192-2.
- [95] Douglas S. Kerr, editor. *Proceedings of the International Conference on Very Large Data Bases*, Framingham, Massachusetts, USA, September 22-24 1975. ACM.
- [96] Gerald Kiernan, Christophe de Maindreville, and Eric Simon. Making Deductive Databases a Practical Technology: A Step Forward. In Garcia-Molina and Jagadish [74], pages 237–246.
- [97] Jerry Kiernan, Christophe de Maindreville, and Eric Simon. The Design and Implementation of an Extendible Deductive Database System. *SIGMOD Record*, 18(3):68–77, 1989.
- [98] Won Kim and Hong-Tai Chou. Versions of schema for object-oriented databases. In Bancilhon and DeWitt [28], pages 148–159. ISBN 0-934613-75-3.
- [99] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A General Purpose Event-Action System. *IEEE Transactions on Software Engineering (TSE)*, 21(10):845–857, October 1990.
- [100] S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tomkins, and Eli Upfal. The Web as a Graph. In ACM [12], pages 1–10. ISBN 1-58113-214-X.
- [101] Tirthankar Lahiri, Serge Abiteboul, and Jennifer Widom. Ozone: Integrating Structured and Semistructured Data. In Richard C. H. Connor and Alberto O. Mendelzon, editors, *7th International Workshop on Database Programming Languages, DBPL'99, Kinloch Rannoch, Scotland, UK, September 1-3, 1999, Revised Papers*, volume 1949, pages 297–323. Springer, 2000.
- [102] Frederick H. Lochovsky and Robert W. Taylor, editors. *Proceedings of 6th International Conference on Very Large Data Bases*, Montreal - Quebec - Canada, October 1-3 1980. IEEE Computer Society Press.

- [103] M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems, 1997. In ICODP ICDP.
- [104] Amélie Marian. Détection de changements et mises à jour incrémentals dans un système de vues actives pour XML. Rapport de Stage de DEA - 1999.
- [105] Amélie Marian, Serge Abiteboul, Grégory Cobéna, and Laurent Mignet. Change-Centric Management of Versions in an XML Warehouse. In Apers et al. [23], pages 581–590. ISBN 1-55860-804-4.
- [106] Amélie Marian, Serge Abiteboul, and Laurent Mignet. Change-centric Management of Versions in an XML Warehouse. In *Base de Données Avancées*, pages 281 – 303, 2000.
- [107] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A Database Management System for Semi-structured Data. *SIGMOD Record*, 26(3):54–66, 1997.
- [108] Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors. *Proceedings of 16th International Conference on Very Large Data Bases*, Brisbane, Queensland, Australia, August 13-16 1990. Morgan Kaufmann.
- [109] Laurent Mignet, Vincent Aguiléra, Sébastien Ailleret, and Pierangelo Veltri. XyRo: The Xyleme Robot Architecture. First workshop Data Integration over the Web, June 2001.
- [110] Laurent Mignet, Mihai Preda, Serge Abiteboul, Sébastien Ailleret, Bernd Amann, and Amélie Marian. Acquiring XML pages for a Web-House. In *Base de Données Avancées*, pages 241 – 263, 2000.
- [111] Tova Milo and Dan Suciu. Type Inference for Queries on Semistructured Data. In ACM [11], pages 215–226. ISBN 1-58113-062-7.
- [112] Matthew Morgenstern. Active Databases as a Paradigm for Enhanced Computing Environments. In Schkolnick and Thanos [129], pages 34–42. ISBN 0-934613-15-X.
- [113] Benjamin Nguyen, Serge Abiteboul, Grégory Cobéna, and Laurent Mignet. Query Subscription in an XML Webhouse. In *Proceedings of the First DELOS Network Excellence Workshop*, pages 109–114, Zurich, December 2000. ERCIM.

- [114] Benjamin Nguyen, Serge Abiteboul, Grégory Cobéna, and Mihai Preda. Monitoring XML Data on the Web. In Sellis and Mehrotra [132], pages 437–448. ISBN 1-58113-332-4.
- [115] Marc Nojork and Janet L. Wiener. Breadth-First Search Crawling Yields High-Quality Pages. In *Tenth International World Wide Web Conference, WWW 10*, pages 114–118, Hong Kong, China, May 1-5 2001. ACM. ISBN 1-58113-348-0.
- [116] James R. Norris. *Markov chains*. Cambridge series on statistical and probabilistic mathematics. Cambridge University Press, 1999. ISBN : 0-521-63396-6.
- [117] *O₂ Notification User Manual*.
- [118] The Object Management Group. <http://www.omg.com/>.
- [119] The Object Management Group. CORBA services. <http://www.omg.com/>.
- [120] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object Exchange Across Heterogeneous Information Sources. In Philip S. Yu and Arbee L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 251–260. IEEE Computer Society, 1995. ISBN 0-8186-6910-1.
- [121] João Pereira, Françoise Fabret, H. Arno Jacobsen, François Llibat, and Dennis Shasha. WebFilter: A High-throughput XML-based Publish and Subscribe System. In Apers et al. [23], pages 271–280. Demonstration Paper.
- [122] Philippe Picouet and Victor Vianu. Semantics and Expressiveness Issues in Active Databases. *Journal of Computer and System Sciences*, 57(3):325–355, 1998.
- [123] Bright Planet. The Deep Web: Surfacing Hidden Value. Technical report, Bright Planet, July 2000.
- [124] Mihai Preda. Data Acquisition for an XML Warehouse. Rapport de Stage DEA - 2000.
- [125] Mirror Image Delivers Enhanced Service To Monitor Web Site Effectiveness. http://www.mirror-image.com/news/pressrelease.cfm?news_item_id=333.

- [126] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, 7(4):57–66, July 1990.
- [127] Chantal Reynaud, Jean-Pierre Sirot, and Dan Vodislav. Semantic Integration of XML Heterogeneous Data Sources. In *IDEAS*, July 2001.
- [128] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.
- [129] Mario Schkolnick and Costantino Thanos, editors. *9th International Conference on Very Large Data Bases*, Florence, Italy, October 31 - November 2 1983. Morgan Kaufmann. ISBN 0-934613-15-X.
- [130] Bill Segall and David Arnold Elvin. A publish/subscribe notification service with quenching. In *Proceedings of Australian Unix Users Group Annual Conference*, pages 243–255, 1997.
- [131] Luc Segoufin. Bases de Données Actives et Evolving Algebras. Rapport de Stage DEA - 1994.
- [132] Timos Sellis and Sharad Mehrotra, editors. *Proceedings ACM SIGMOD International Conference on Management of Data*, Santa Barbara - California - USA, May 21-24 2001. ACM Press. ISBN 1-58113-332-4.
- [133] Timos K. Sellis, Chih-Chen Lin, and Louiqa Raschid. Data Intensive Production Systems: The DIPS Approach. *SIGMOD Record*, 18(3):52–57, 1989.
- [134] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In Atkinson et al. [25], pages 302–314. ISBN 1-55860-615-7.
- [135] Eric Simon and Christophe de Maingreville. Deciding Whether a Production Rule is Relational Computable. In Marc Gyssens, Jan Paredaens, and Dirk Van Gucht, editors, *ICDT'88, 2nd International Conference on Database Theory, Bruges, Belgium, August 31 - September 2, 1988, Proceedings*, volume 326 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 1988. ISBN 3-540-50171-1.
- [136] A. Prasad Sistla and Ouri Wolfson. Triggers on Database Histories. *IEEE Data Engineering Bulletin*, 15(4):48–51, 1992.

- [137] The Apache Software Foundation. <http://www.apache.org>.
- [138] The CVS-Optimized General-Purpose Network File Distribution System. <http://www.polstra.com/projects/freeware/CVSup/>.
- [139] ht://Dig WWW Search Engine Software. <http://www.htdig.org/>.
- [140] Java Remote Method Invocation (RMI). <http://java.sun.com/j2se/1.4/docs/guide/rmi/>.
- [141] JavaScript. <http://www.javascript.com/>.
- [142] ORBacus Object Oriented Concepts Inc. <http://www.orbacus.com/>.
- [143] PostgreSQL. www.postgres.org/.
- [144] rsync. <http://rsync.samba.org/>.
- [145] Wget. <http://www.gnu.org/software/wget/wget.html>.
- [146] WordNet - a Lexical Database for English. <http://www.cogsci.princeton.edu/~wn/>.
- [147] Michael Stonebraker, Eric N. Hanson, and Spyros Potamianos. The POSTGRES Rule Manager. *IEEE Transactions on Software Engineering (TSE)*, 14(7):897–907, July 1988.
- [148] Michael Stonebraker, Marti A. Hearst, and Spyros Potamianos. A Commentary on the POSTGRES Rule System. *SIGMOD Record*, 18(3):5–11, 1989.
- [149] Bruno Tessier. Interface pour Active Views. Rapport de Stage Polytechnique - 1999.
- [150] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–725, November 1997.
- [151] Jeffrey D. Ullman. *Principles of Database and Knowledge - Base Systems*, volume 1. Computer Science Press, first edition, 1988. ISBN - 0-7167-8158-1.
- [152] AltaVista Company. <http://www.altavista.com/>.
- [153] Ardent software. <http://www.ardentsoftware.fr/>.

- [154] Tim Berners-Lee. <http://www.w3.org/People/Berners-Lee/>.
- [155] CenterFind. <http://www.centerfind.com/>.
- [156] DTD for Dublin Core Metadata Element.
<http://dublincore.org/documents/2001/04/11/-dcmes-xml/dcmes-xml-dtd.dtd>.
- [157] Excite. www.excite.com/.
- [158] Google Incorporation. <http://www.google.com/>.
- [159] IONA Technologies. <http://www.iona.com>.
- [160] New Era of Networks. <http://www.neonsoft.com>.
- [161] ACM SIGMOD Record: XML Version.
<http://www.acm.org/sigmod/record/xml/Record/DTD/index.html>.
- [162] Universal Description, Discovery, and Integration (UDDI).
<http://www.uddi.org/>.
- [163] <http://www.xmlblaster.org/>.
- [164] Yahoo! <http://www.yahoo.com/>.
- [165] World Wide Web Consortium. Document Object Model (DOM) 2.0.
<http://www.w3.org/DOM/DOMTR>.
- [166] World Wide Web Consortium. eXtensible Markup Language (XML) 1.0. <http://www.w3.org/XML/>.
- [167] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.1 . <http://www.w3.org/TR/SOAP/>.
- [168] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>.
- [169] World Wide Web Consortium. XML Path Language (XPath).
<http://www.w3.org/TR/xpath>.
- [170] World Wide Web Consortium. XML Query.
<http://www.w3.org/XML/Query>.
- [171] Jennifer Widom. The Starburst Rule System: Language Design, Implementation, and Applications. *IEEE Data Engineering Bulletin*, 15(1):15–18, 1992.

- [172] Jennifer Widom and Stefano Ceri. *Active Database Systems - Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1995. ISBN 1-55860-304-2.
- [173] Craig E. Wills and Mikhail Mikhailov. Towards a Better Understanding of Web Resources and Server Responses for Improved Caching. *Computer Networks and ISDN Systems*, 31(11-16):1231–1243, May 1999.
- [174] Wolfgang Woess. *Random Walks on Infinite Graphs and Groups*. Cambridge tracts in mathematics. Cambridge University Press, 2000. ISBN : 0-521-55292-3.
- [175] Mike Wray and Rycharde Hawkes. Distributed Virtual Environments and VRML: An Event-Based Architecture. *Computer Networks and ISDN Systems*, 30(1-7):43–51, April 1998.
- [176] Lucie Xyleme. A Dynamic Warehouse for XML Data of the Web. *IEEE - Data Engineering Bulletin*, 24(2):40–47, June 2001.
- [177] Tak W. Yan and Jurgen Annevelink. Integrating a Structured-Text Retrieval System with an Object-Oriented Database System. In Bocca et al. [32], pages 740–749. ISBN 1-55860-153-8.
- [178] Yelena Yesha and Nabil Adam. *Electronic Commerce: an Overview*. Lecture Notes in computer Science. Springer-Verlag, 1996.
- [179] H. Yu, D. Estrin, and R. Govindan. A ierarchical Proxy Architecture for Internet-scale Event Services. In *WETICE*, Stanford, CA, USA, June 1999.
- [180] Li-Yan Yuan, editor. *18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada, August 23-27 1992. Morgan Kaufmann. ISBN 1-55860-151-1.
- [181] Roberto Zicari. A Framework for Schema Updates In An Object-Oriented Database System. In F. Bancilhon, C. Delobel, and P. Kanelakis, editors, *The O2 book*, chapter 8, pages 146–182. Morgan Kaufman, 1992. reprinted from ICDE 1991.
- [182] Moshé M. Zloof. Query-by-Example: A Data Base Language. *IBM Journal of Research and Development*, 16(4):324–343, November 1977.
- [183] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall Inc., Upper Saddle River, New Jersey, 1991. ISBN 0-13-691643-0.

Annexe A

Définition Langage ActiveView

Cette Annexe décrit le langage *ActiveView* de manière plus précise. La syntaxe utilisée est proche de celle de BNF étendu. Les mots clés du langage seront écrit en gras alors que les clauses terminales seront en **type**.

A.1 Définition d'une Application

beginning	:= (application)*
application	:= application_name (application_declaration)*
application_name	:= ActiveView application application_name
application_declaration	:= application_header declaration
application_header	:= ActiveView actor-kind in application application_name
declaration	:= data_declaration method_declaration activity_declaration rule_declaration

A.2 Définition des Données

<code>data_declaration</code>	<code>:= (variable_declaration local_declaration element_declaration)*</code>
<code>variable_declaration</code>	<code>:= let_clause be_clause with_clause? mode_clause?</code>
<code>let_clause</code>	<code>:= let variable_name : variable_type</code>
<code>be_clause</code>	<code>:= be query</code>
<code>local_declaration</code>	<code>:= local_clause (mode_clause)?</code>
<code>local_clause</code>	<code>:= local variable_name : variable_type</code>
<code>element_declaration</code>	<code>:= element_head (with_clause)? (mode_clause)?</code>
<code>element_head</code>	<code>:= element variable_type</code>
<code>with_clause</code>	<code>:= with with_term (, with_term)*</code>
<code>with_term</code>	<code>:= path_expression (variable_name)?</code>
<code>mode_clause</code>	<code>:= mode (access_mode access_variable)+</code>
<code>access_mode</code>	<code>:= deferred read immediate read read write append remove</code>
<code>access_variable</code>	<code>:= variable_name all (except path_expression</code>

Les expressions de chemin de la clause `with` prennent leur point de départ à partir des noms de variables spécifiés dans les clauses `let_clause` et `local_clause`. Le langage introduit un raccourci syntaxique par le biais du mot clé **self** afin de ne pas répéter les noms des variables.

De la même manière, les noms des variables dans la clause `access_clause` sont les variables introduites par les clauses `with_term`, `let_clause`, `local_clause`.

A.3 Définition des Méthodes

method_declaration	:= method_clause range_clause code_clause (if_clause)?
method_clause	:= method signature
signature	:= signature_variable_name : signature_variable_type (, signature)?
range_clause	:= range range_predicate?
range_predicate	:= signature_variable_name in variable_name (, range_predicate)?
is_clause	:= is code
of_clause	:= if predicate

où `variable_name` renvoie au nom de la variable définie à la section précédente.

A.4 Définition des Activités

activity_declaration	:= activity_clause include_clause
activity_clause	:= activity activity_name
include_clause	:= includes variables_clause
variable_clause	:= (variable_name)* (method_name)* all

où `<nom-variable>` (resp. `<nom-méthode>`) indique quelles sont les variables (resp. les méthodes) qui sont définies dans la vue et qui sont accessibles au travers de l'activité. Le mot clé **all** peut être utilisé pour spécifier que toutes les variables et les méthodes de la vue sont visibles.

A.5 Définition des Règles

rule_declaration	:= on_clause (if_clause)? do_clause
on_clause	:= on event
if_clause	:= if condition
do_clause	:= do action

RESUME en français

La contribution de cette thèse consiste en deux approches complémentaires pour permettre à un système de mieux contrôler les changements de données semi-structurées. La première approche permet de contrôler la propagation des changements entre une base de données et des vues actives dans un intranet local. Elle est basée sur un langage déclaratif de spécification de vues, doté de capacités actives. La seconde partie de la thèse a pour cadre général la construction d'un entrepôt extrait du web centré sur des données au format XML. Cette seconde partie présente un mécanisme inflationniste d'acquisition et de rafraîchissement de pages peuplant de manière automatique cet entrepôt.

Les deux approches ont été validées par des prototypes. De plus, les technologies présentées ont été transférées dans l'industrie.

TITRE en anglais

Change Controls on Semi-structured Data.

RESUME en anglais

This thesis presents two complementary approaches to control changes in semi-structured data. The first approach makes it possible to monitor changes in a local intranet database using active views on data. Views are defined using a declarative language with active capacities. The second approach deals with data change control in a data warehouse of XML documents obtained from the web. An inflationary mechanism to acquire and refresh automatically data from the web is presented.

These two approaches have been validated by prototypes and, in addition, the techniques presented here have been transferred to industry.

DISCIPLINE - SPECIALITE DOCTORALE

Informatique

MOTS CLES

Contrôles des Changements, Base de Données, XML, Web, SGBD active, entrepôt de Données sur le web.

INTITULE ET ADRESSE DU LABORATOIRE

INRIA Rocquencourt, projet Verso	Equipe Vertigo, Acces 37-1-42
Domaine de Voluceau - BP 105	CNAM Paris 292 Rue St Martin
78153 Le Chesnay Cedex France	75141 Paris Cedex 03 France