Requêtes XPath bipolaires et évaluation

Maurice Tchoupé Tchendji, Brice Nguefack

Requêtes XPath bipolaires et évaluation

Maurice TCHOUPÉ TCHENDJI *, Brice NGUEFACK *

* Département de Maths-Informatique
Faculté des Sciences, Université de Dschang
BP 67, Dschang-Cameroun
ttchoupe@yahoo.fr
brice_nguefack@yahoo.fr

RÉSUMÉ. Le concept de requêtes bipolaires (on parle aussi de requêtes avec préférences) a émergé dans la communauté des Bases de Données Relationnelles, pour permettre aux utilisateurs d'obtenir des réponses beaucoup plus pertinentes à leurs préoccupations, exprimées via des requêtes dites avec préférences. De telles requêtes ont généralement deux parties : la première permet d'exprimer les contraintes strictes et la seconde, des préférences ou souhaits. Toute réponse à une requête avec préférences doit nécessairement satisfaire la première partie et préférentiellement la seconde. Toutefois, s'il existe au moins une réponse satisfaisant la seconde partie, toutes les réponses ne satisfaisant que la première partie seront exclues du résultat final : elles sont dominées. Dans ce papier, nous explorons une approche d'importation de ce concept dans les Bases de Données XML via le langage XPath. Pour ce faire, nous proposons le langage PrefSXPath, une extension du langage XPath permettant d'exprimer les requêtes XPath avec préférences structurelles, puis, nous présentons un algorithme d'évaluation des requêtes PrefSXPath à l'aide des automates.

ABSTRACT. The concept of bipolar queries (also call preferences queries) emerged in the Relational Databases community, allowing users to get much more relevant responses to their requests, expressed via queries say with preferences. Such requests usually have two parts: the first is used to express the strict constraints and the second, preferences or wishes. Any response to a query with preferences must necessarily satisfy the first part and preferably the latter. However, if there is at least a satisfactory answer of the second part, those satisfying only the first part will be excluded from the final result: they are dominated. In this paper, we explore an approach of importation of this concept in a XML Database via XPath language. To do this, we propose PrefSXPath language, an extension of XPath in order to express XPath queries with structural preferences, then we present a query evaluation algorithm of PrefSXPath using automata.

MOTS-CLÉS : XML, XPath, Requêtes avec Préférences, Base de Données XML, Skyline, Automates.

KEYWORDS : XML, XPath, Preferences Queries, XML Database, Skyline, Automata.

1. Introduction

Les documents semi-structurés XML ont une structure flexible [1] et ne contiennent que du texte. Ils permettent alors d'échanger et de stocker les données plus aisément qu'avec les Bases de Données Relationnelles (BDR) classiques : on parle de Bases de Données XML (BDs XML).

L'exploitation des données stockées dans une BD se fait généralement via un langage dédié d'interrogation (SQL pour les BDR, XPath [15], XQuery [16] pour les BD XML, ...) permettant à l'utilisateur de rédiger des requêtes qui peuvent dans certains cas, contenir d'une part des exigences obligatoires appelées *contraintes*, et d'autre part, des exigences optionnelles appelées *préférences* ou *souhaits* : on parle alors de *requêtes bipolaires* ou de requêtes avec préférences[1][6, 9].

Des langages spécifiques (généralement des extensions du SQL ou du XPath) ont été proposés pour l'écriture des requêtes avec préférences : *SQLf* [3], *Preference SQL* [8], *Preference Queries* [5], ... pour les BDR, *XPref* [11], *Prefrence XPATH* [14], ... pour les BD XML. Les extensions de XPath prenant en compte les préférences ([11, 14]) procèdent généralement par réécriture de requêtes vers le format XPath pure. Bien plus, ils ne s'intéressent généralement qu'aux préférences portant sur les valeurs alors qu'un document XML intègre aussi une structure qu'il convient de prendre en compte lors de son interrogation.

Ce papier a pour objectif de proposer une autre approche d'importation du concept de requête avec préférences dans les BDs XML via le langage XPath en se focalisant sur la structure : nous traitons des *préférences structurelles*. A cet effet, nous proposons le langage *PrefSXPath*, une extension du langage XPath permettant d'exprimer les requêtes XPath avec préférences structurelles. Bien plus, nous présentons aussi une approche d'évaluation des requêtes *PrefSXPath* à l'aide des automates suivant la technique utilisée par Bin Sun et al. [2] pour l'évaluation des requêtes classiques (sans préférences). Notons que notre proposition se distingue de celles de [14, 11] sur bien des points : ils se sont intéressés principalement aux préférences portant sur les valeurs quand nous le faisons sur les structures, leur extension de XPath est localisée au niveau des prédicats alors que la notre porte également sur des nœuds sans prédicats ; là où ils font une réécriture de requête nous proposons un algorithme d'évaluation.

La figure 1 donne une vue synoptique de la démarche d'évaluation proposée ; elle se décline en deux étapes. En fait, étant donné une requête *PrefSXPath* Q à appliquer à un document XML D, après construction comme dans [2] d'un automate A relativement à la requête Q et extraction d'un index T_q de D, dans la première étape, A est utilisé pour générer toutes les réponses de Q dans D sans tenir compte des préférences. Parallèlement à la production des réponses de cette étape, des structures de données contenant des informations pour la sélection des meilleures réponses sont produites et utilisées dans la seconde étape pour construire une table relationnelle *preferenceTable*. Cette table indique de façon booléenne pour chaque réponse r et pour chaque préférence p contenue dans Q si une occurrence de p a été utilisée pour produire la réponse r. L'opérateur Skyline [12] est ensuite appliqué sur la table *preferenceTable* pour choisir les meilleures réponses : ce sont celles associées à des tuples non dominés.

La suite de ce manuscrit est constituée de trois sections présentant respectivement des notions relatives aux documents XML dans la section 2, une grammaire pour le langage

1. C'est la seconde appellation que nous adopterons par la suite.

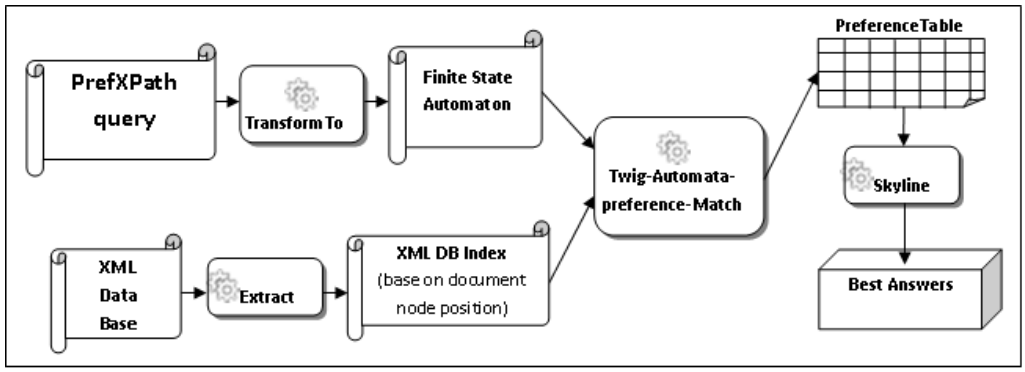


Figure 1. *une vue synoptique de notre approche d'évaluation de requêtes XPath avec préférences.*

PrefSXPath dans la section 3, l'algorithme d'évaluation des requêtes *PrefSXPath* ainsi qu'une étude de sa complexité dans la section 4, et enfin une conclusion dans laquelle nous faisons un bilan de notre travail et présentons quelques pistes pour des travaux futurs (section 5). Une annexe est également fournie (Annexe A) présentant un exemple complet d'évaluation d'une requête *PrefSXPath* contenant les différents types de motifs de requêtes adressés dans ce manuscrit.

2. Documents XML et interrogations

2.1. Documents XML : représentations et indexations

Un *document XML abstrait*[2] est représenté par un arbre étiqueté $D = (N_d, E_d)$ où N_d est un ensemble de nœuds étiquetés, E_d un ensemble d'arcs reliant chacun deux nœuds de N_d. Pour tout nœud $x \in N_d$ la fonction $label_d(x)$ retourne son étiquette.

Les documents XML sont généralement exploités à travers un index. Une présentation détaillée de ceux couramment utilisés est donnée dans [7, 10]. Dans ce papier nous utiliserons l'indexation basée sur la position des nœuds (aussi appelée la notation positionnelle) [10] dans laquelle chaque nœud du document est représenté par un triplet *(Start, End, Level)* : *Start* et *End* représentent respectivement les positions de début et de fin de l'élément dans le document ; *Level* est la profondeur du nœud dans la représentation arborescente du document (fig. 2). Avec cette convention, comme dans [17], l'index d'un document est constitué d'un ensemble de listes chainées T_a d'occurrences de nœuds de type *a* triées suivant la composante *Start* du triplet (fig. 3). L'un des avantages de cette représentation est qu'elle permet, étant donné deux nœuds *a* et *b* quelconques représentés respectivement par les triplets *(starta, enda, levela)* et *(startb, endb, levelb)*, de déterminer les relations *parent-enfant* ou *ancêtre-descendant* en temps constant. En effet, *a est un ancêtre de b* si et seulement si *starta*<*startb*<*enda*. Si de plus *levela +1=levelb*, alors, *a* est le *parent* de *b*. Par la suite, nous dirons d'un nœud *a* qu'il *recouvre* un nœud *b* si *b* est un descendant de *a*.

2. Dans un document abstrait, on fait abstraction des nœuds de textes et des nœuds d'attributs : ceux-ci ne sont pas d'un intérêt avéré pour les traitements purement structurels qui nous intéressent dans ce papier.

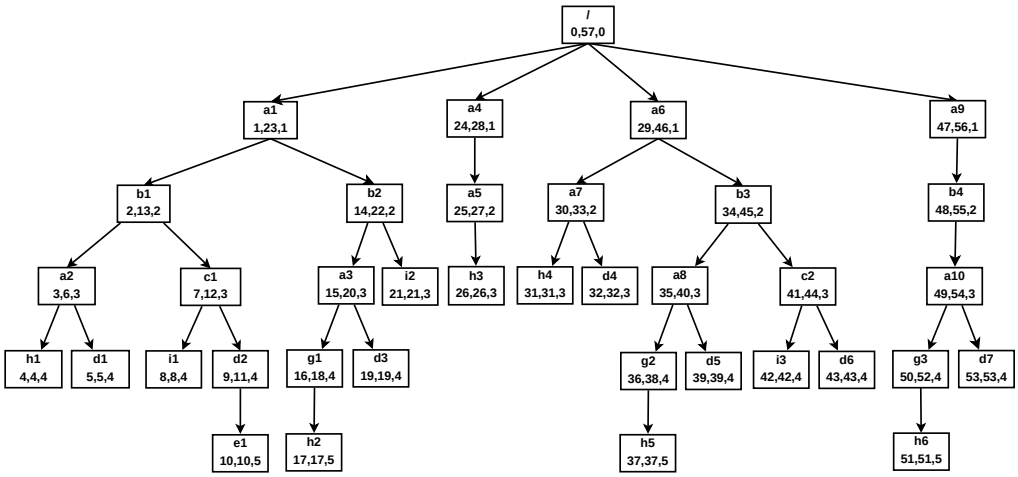


Figure 2. *Document XML décoré à l'aide des triplets (start, end, level).*

La figure 2 est une illustration de la représentation arborescente d'un document XML dans laquelle chaque nœud est décoré par sa notation positionnelle ; la figure 3 en est un index.

2.2. Documents XML : requêtes et évaluations

A l'instar d'une BD classique, un document XML contient des informations (des données). Il encapsule aussi une structure dont il faut impérativement tenir compte lors de son interrogation. Ainsi une requête XML concerne non seulement le contenu (les données) mais aussi la structure (les relations structurelles que l'utilisateur souhaite avoir entre les différentes occurrences des éléments). Comme pour la représentation des documents, une requête XML Q peut être représentée par un arbre $Q = (N_q,\ E_q)$ dans lequel N_q est un ensemble des nœuds étiquetés, E_q un ensemble d'arcs reliant chacun deux nœuds de N_q. Dans E_q on distingue deux types d'arcs : ceux reliant un nœud père à un nœud fils noté

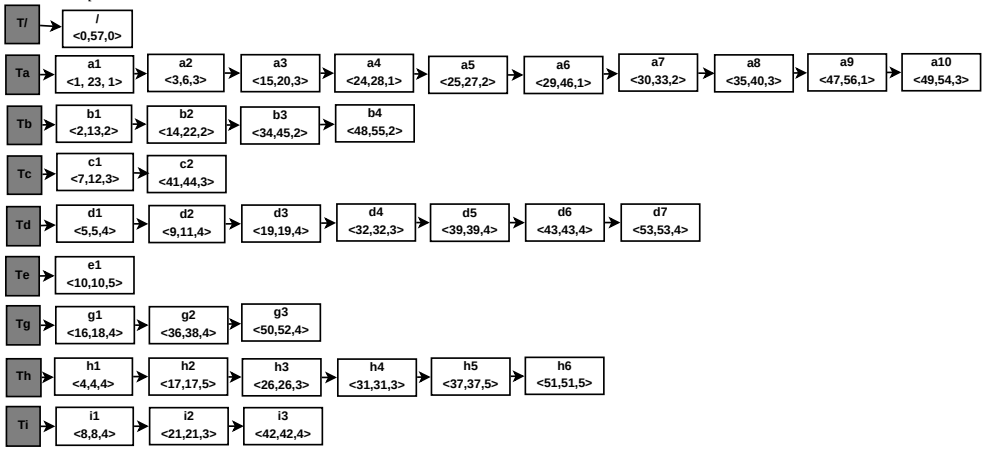


Figure 3. *Index du Document de la figure 2.*

x/y et ceux reliant un nœud à un de ses descendants noté $x//y$. Pour tout arc $x \in N_q$ la fonction $label_q(x)$ retourne son étiquette, et on note $\Sigma = \{label_q(x),\ x \in N_q\}$.

Soient, $Q = (N_q,\ E_q)$ une requête, $D = (N_d,\ E_d)$ un document XML, deux nœuds $n_d \in N_d$ et $n_q \in N_q$: On dira que n_d *est une occurrence de* n_q dans D si $label_q(n_q) = label_d(n_d)$. On dira aussi que la requête Q est *satisfaite* dans D si chacun des nœud de N_q possède au moins une occurrence dans N_d et telle que les relations de parenté existant entre les nœuds de N_q soient les mêmes qui existent entre leurs occurrences.

2.3. Requêtes XPath et évaluations

XPath (XML Path) [15] est un langage essentiel[3] de requête XML permettant de sélectionner des nœuds d'un document XML de sorte que le chemin allant de la racine du document à chacun des nœuds sélectionnés satisfait un motif (la requête) donné. En plus des relations parent-enfant et ancêtre-descendant, on trouve aussi dans les requêtes XPath des prédicats associés à des nœuds. Ce sont des expressions booléennes comprises entre les symboles '[' et ']' et accolées à un nœud requête pour signifier que les occurrences licites de ce nœud sont celles pour lesquelles le prédicat est évalué à *True*.

De nombreuses techniques d'évaluation de requêtes XPath existent [13, 4], parmi lesquelles celle développée par Bin Sun et al. [2] sur laquelle nous nous appuyons pour étayer notre approche d'extension du langage XPath pour y intégrer les préférences. Dans l'approche de [2] une requête XPath est transformée en un automate en appliquant un ensemble de schémas de construction associés aux différents types de chemin existant dans le langage XPath. Certaines transitions de l'automate produit sont décorées par des *actions* dont l'exécution dans l'algorithme de matching, lors du franchissement de telles transitions permet de construire progressivement la solution de la requête.

3. PrefSXPath : un langage pour l'expression des requêtes XPath avec préférences

Dans cette section, nous présentons la grammaire de *PrefSXPath* ainsi que des éléments du vocabulaire utilisé dans la suite de ce manuscrit.

3.1. Une notation pour les préférences dans la grammaire de XPath

Nous introduisons la notation ' !' comme opérateur unaire dans la grammaire de *PrefSXPath* (voir ligne 7 du listing ci-dessous) pour l'expression des préférences dans un (sous-)chemin XPath. Par exemple, dans la requête *PrefSXPath* $Q_1 = /a/(b[c])!/d$ les sous-chemins $/a$ et $/d$ représentent des *contraintes*, alors que $/(b[c])!$ représente une *préférence*. Q_1 est interprétée comme une requête retournant toutes les occurrences *di* du nœud résultat *d* satisfaisant : (1) Le chemin allant de la racine à *di* possède (obligatoirement) une occurrence de *a* et éventuellement une occurrence de *b* si celle-ci est parente d'une occurrence de *c*. Les occurrences *di* candidates à faire partie de la réponse sont donc celles pour lesquelles il existe dans le document un sous-arbre de l'une des formes */ai/bi[ci]/di* ou */ai/di*. (2) S'il existe même une seule réponse candidate, disons *dp*, telle que le chemin allant de la racine à *dp* contienne une occurrence de *b* (*/ap/bp/dp)*), alors, seules les réponses candidates intégrant une occurrence de *b* seront retournées comme réponses à la requête : on dit qu'elles *dominent* les autres solutions. En fait, vue que le

3. Il est utilisé par bien d'autres langages d'interrogation XML comme XQuery [16], XSLT, ...

chemin $(b[c])!$ permet d'exprimer une préférence, on ne retournera prioritairement (et de façon exclusive) que les occurrences de *d*, fils d'une occurrence de *b* elle même parente d'une occurrence de *c*. Ainsi présenté, la notion de préférence permet de ne retenir comme résultat à la requête que les *meilleures réponses* c-à-d. les plus préférées[4].

En nous basant sur la grammaire donnée dans [2] pour décrire un sous ensemble significatif du langage XPath, nous considérons la grammaire suivante pour les expressions *PrefSXPath*.

Syntaxe BNF des expressions *PrefSXPath*

```
1  AbstTwigExpr ::= '/' RltvTwigExpr
2                 | '(' RltvTwigExpr ')'
3  RltvTwigExpr ::= RltvTwigExpr BinaryOp RltvTwigExpr
4                 | RltvTwigExpr UnaryOp
5                 | Step
6  BinaryOp ::= '/' | '//'
7  UnaryOp ::= '*' | '+' | '?'|!
8  Step ::= Name
9           | Name '[' Predicate ']'
10 Predicate ::= RltvTwigExpr
```

L'opérateur unaire '*!*' est introduit à la ligne 7 du listing précédent.
$/a/(b[(c[d/e!])!/i])!/a[d]/g!/h$ est un (exemple de) mot pour le langage dénoté par cette grammaire : c'est un chemin *PrefSXPath* c'est-à-dire une requête avec préférences.

3.2. Quelques définitions et notations

Considérons la requête avec préférence $Q_3 = /a/(b[(c[d/e!])!/i])!/a[d]/g!/h$ ayant *h* pour *nœud résultat* (fig. 4). En remarquant que plusieurs nœuds d'une expression *PrefSXPath* peuvent avoir le même label, pour pouvoir désigner sans ambigüité tout nœud par son label, on peut lui ajouter un indice relativement à sa position dans l'expression. Q_3 par exemple est reécrit en $Q_3 = /a_1/(b[(c[d_1/e!])!/i])!/a_2[d_2]/g!/h$.

Dans Q_3, aux nœuds *b*, *c* et a_2 sont associés des prédicats donnés respectivement par les *chemins prédicats* $[(c[d_1/e!])!/i]$, $[d_1/e!]$ et $[d_2]$; de tels nœuds sont appelés *nœuds clés*[5] . Le chemin obtenu en supprimant dans la requête tous les chemins prédicats est appelé *chemin principal* : Q3 a pour chemin principal $/a_1/b!/a_2/g!/h$. Le sous-chemin $(b[(c[d_1/e!])!/i])!$ permettant d'exprimer une préférence est appelé *chemin préférence* et le nœud clé *b* de ce chemin est appelé *nœud préférence* de ce chemin. En fait, on appelle *nœud(s) préférence(s)* d'un chemin préférence le(s) nœud(s) résultat(s) de ce chemin[6].

Quand un nœud préférence figure dans un chemin prédicat, il est lié au chemin principal par un unique nœud clé qui est son pivot. Le *pivot*[7] d'un nœud préférence est en fait,

4. Notons que le concept de *préférence* est différent de celui de *optionnelle* déjà présente dans le langage XPath et symbolisé par '*?*'. Par exemple, avec la requête $Q_2 = /a/(b[c])?/d$ dans laquelle le sous chemin $/(b[c])?$ est optionnel, les occurrences di de d qui seront retournées sont celles appartenant à un sous arbre du document de l'une des deux formes $/ai/bi[ci]/di$ ou $/ai/di$.

5. Lors de l'exécution de la requête, quand on arrive sur un nœud clé, on doit suivre deux chemins parallèlement : celui du prédicat et celui du *chemin principal*.

6. Par exemple, les requêtes suivantes *a!, (a[b/c])!, (a /b([c/d])!/e)!, (a/(b[c/d])!)!* ont pour nœuds préférences respectivement {a}, {a}, {e}, {a, b}.

7. En considérant la notation positionnelle, si $np(sn,en,ln)$ est un nœud préférence et $nc1(sc1,ec1,lc1),\ldots,nck(sck,eck,lck)$ sont k nœuds clés ancêtres de np (on a dans ce cas,

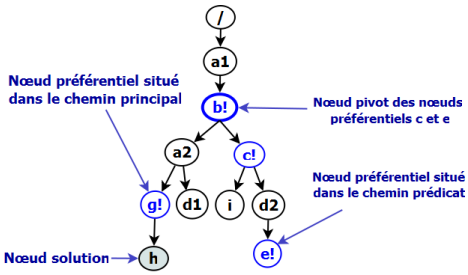


Figure 4. *Illustration d'une requêtes XPath avec préférences.*

le nœud clé situé sur le chemin principal tel que le chemin qui les relie ne passe pas par un autre nœud clé situé sur le chemin principal. Notons que si un nœud préférence est situé sur le chemin principal, il est son propre pivot. Dans Q_3, les nœuds préférences *b*, *c*, *e* et *g* ont respectivement pour pivots les nœuds *b*, *b*, *b* et *g* (voir fig. 4).

L'approche d'évaluation utilisée ici est descendante : on évalue une requête en commençant par l'évaluation du nœud le plus proche de la racine et en évoluant vers les feuilles. Nous appellerons donc par la suite *requête partielle* tout (chemin) préfixe d'une requête : $/a/(b[(c[d/e!])!/i])!$ est une requête partielle de Q_3.

4. Evaluation des requêtes PrefSXPath à l'aide des automates

Visuellement, si on exécute la requête Q_3 (fig. 4) sur le document de la figure 2, on obtiendra comme réponse l'occurrence $h1$ de *h*. Cette section est consacrée à la description de notre approche d'évaluation de telles requêtes et en l'étude de sa complexité.

4.1. De l'expression PrefSXPath à l'automate

Le tableau 1 donne à l'exemple de celui présenté dans [2] pour chaque motif de requête *PrefSXPath*, le schéma de l'automate à construire correspondant. Les transitions de ces automates sont étiquetées soit par des éléments de Σ, soit par λ quand la requête permet d'exprimer une relation *ancêtre-descendant* ($x//y$), soit par le nom d'une des actions décrites à la section 4.2.2.

Une requête *prefSXPath Q* ayant un prédicat sur le noeud "C" peut être décomposé sous la forme : *Q= ch1/C[ch2]/ch3* où les *chi*, i=1,2,3 sont des chemins *prefSXPath*. L'automate correspondant à la requête *Q* est construit en connectant suivant les motifs présentés dans le tableau 1 les trois automates A_1, A_2 et A_3 correspondants respectivement aux (sous-)requêtes *ch1*, *ch2* et *ch3* (voir figure 5). Par application de ce principe à la requête 4, on obtient l'automate de la figure 6.

$sci < sn \leq en < eci,\ \forall i, 1 \leq i \leq k$) et situés sur le chemin principal, alors le nœud pivot de np est le nœud ncl (scl, ecl, lcl), $1 \leq l \leq k$ tel que ($ecl == minimum(ec1, \ldots, eck)$) : c'est le nœud y ayant la plus petite valeur de *endPos*. On peut donc statiquement calculer le nœud pivot de tout nœud préférence. On supposera donc dans la suite qu'on dispose d'une fonction *getPivotId* qui retourne le pivot d'un nœud préférence donné. Par exemple, *getPivotId (c) = b*.

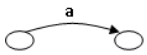
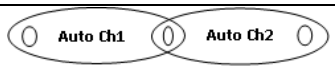
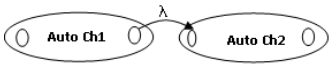
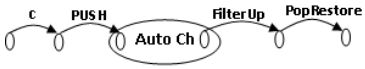
Chemin *PrefSXPath*	Automate correspondant
$a \in \Sigma$, $/a$	
$Ch1/Ch2$	
$Ch1//Ch2$	
$Ch?$	
$Ch!$, a étant un nœud préférence de Ch	
$c \in \Sigma$, $c[Ch]$	
Ch^*	

Tableau 1. *Motifs de requêtes PrefSXPath et schémas d'automates correspondant*

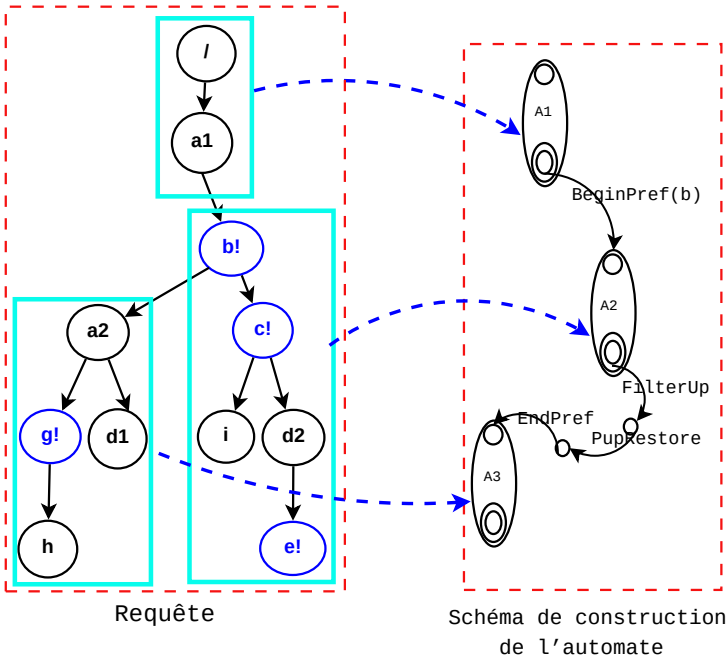


Figure 5. *Principe de construction de l'automate dérivé d'une requête.*

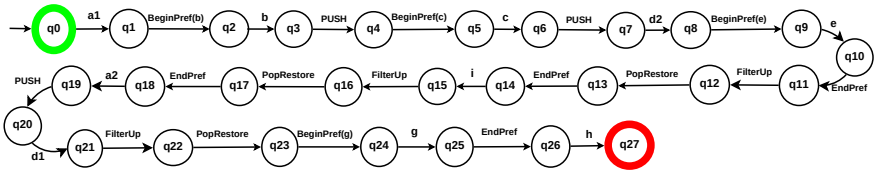


Figure 6. *Automate construit à partir de la requête 4 en utilisant les motifs du tableau 1.*

4.2. Algorithme d'évaluation

4.2.1. Les structures de données

Rappel : l'évaluation d'une requête *PrefSXPath* est descendante. Au cours de l'évaluation, le résultat de l'évaluation du nœud requête courant est stocké dans une variable appelée *currentAnswer*. Ce qui suit est une description des autres variables utilisées par l'algorithme.

partialSolutionStack : pile utilisée lors du traitement des (sous-)chemins optionnels et préférences. En effet, lors du franchissement d'un arc étiqueté par l'une des actions *BeginOP* ou *BeginPref* ou encore *Push*[8], la solution partielle courante (contenue dans *currentAnswer*) y est stockée en vue de son utilisation à la fin du traitement du sous-chemin optionnel ou préférence (c'est-à-dire, traversé d'un arc étiqueté *EndOp* ou *EndPref*) pour construire la nouvelle solution courante.

prefNodeStack : pile utilisée pour stocker et restituer les nœuds préférences contenus dans le chemin préférence courant : lors du franchissement d'un arc étiqueté *BeginPref(a)*, le nœud préférence *a* y est empilé et lors du franchissement d'un arc étiqueté *EndPref*, on y effectue un dépilement.

answerPivotTable : tableau associatif dont chaque entrée contient une paire *(clé, valeur)*. *clé* est un identifiant d'un nœud pivot, et la valeur associée est le résultat de l'évaluation de la sous-requête ayant ce nœud comme nœud résultat ; il est initialisé à *currentAnswer* au moment du franchissement d'un arc étiqueté *clé*. Lors du franchissement d'un arc étiqueté *EndPref* l'entrée associée à la clé du pivot du nœud préférence courant (c'est celui se trouvant actuellement au sommet de la pile *prefNodeStack*) est utilisée pour filtrer les occurrences du pivot à stocker dans l'entrée de la table *infoPrefNodeTable* correspondant au nœud préférence courant.

infoPrefNodeTable : tableau associatif dont chaque entrée contient une paire *(clé, valeur)*. clé est un identifiant d'un nœud préférence, et la valeur associée est constituée des occurrences du pivot de ce nœud recouvrant au moins une de ses occurrences.

preferenceTable : tableau de booléens renseignant pour chaque réponse *r* de la requête et pour chaque nœud préférence *p* si une occurrence de *p* a été utilisée pour la production de la solution *r*. Le cas échéant, l'entrée correspondante est mise à « *True* » sinon, elle est mise à « *False* ».

4.2.2. Les actions

Le tableau 2 associe à chaque action la description des traitements à effectuer lors du franchissement d'un arc étiqueté par cette action. Ces traitements sont implémentés dans la fonction *perform_action*[9] dont le code est disséminé dans les cellules adéquates du tableau 2.

8. Voir la description des actions à la section 4.2.2.

9. La fonction *perform_Action(action, q, currentAnswer,)* ; (algo. 1 ligne 12) permet d'effectuer les traitements liés à l'action *action* étiquetant l'arc associé à l'état *q*, les réponses courantes étant dans *currentAnswer*.

Tableau 2: Actions et traitements associés

Nom action	**Description des traitements associés**
Push	Empile la solution partielle courante dans la pile *partialSolutionStack* 1 `Push (partialSolutionStack, currentAnswer);`
FilterUp	Filtre dans le résultat se trouvant au sommet de la pile *partialSolutionStack* ceux qui sont ancêtres d'une occurrence se trouvant dans *currentAnswer*. 1 `filterUp ((top partialSolutionStack), currentAnswer);`
PopRestore	Dépile le résultat au sommet de la pile *partialSolutionStack* et en fait la solution courante. 1 `currentAnswer = top (partialSolutionStack);` 2 `pop (partialSolutionStack);`
BeginPref (a)	Annonce le début de traitement d'un chemin préférence ayant *a* pour nœud préférence : on doit empiler la solution (partielle) courante dans *partialSolutionStack* ainsi que le nœud préférence *a* du chemin préférence qu'on s'en va traiter dans la pile *prefNodeStack*. 1 `Push(partialSolutionStack, currentAnswer);` 2 `Push(a, prefNodeStack);` 3 `retourner (currentAnswer, δA(q, labelTrans q));`
BeginOp	Annonce le début de traitement d'un chemin optionnel : on doit empiler la solution (partielle) courante dans *partialSolutionStack* : 1 `push(partialSolutionStack, currentAnswer);` 2 `retourner (currentAnswer, δA(q, labelTrans q));`
EndOp	Annonce la fin de traitement d'un chemin optionnel. On doit mettre à jour la solution courante (*currentAnswer*) en y ajoutant les occurrences du résultat présentement au sommet de la pile *partialSolutionStack*. 1 `tmpSol = top (partialSolutionStack);` 2 `pop(partialSolutionStack);` 3 `currentAnswer = currentAnswer ∪ tmpSol;` 4 `retourner (currentAnswer, δA(q, labelTrans q));`

Suite, Page Suivante . . .

Nom action	Description des traitements associés
EndPref	Annonce la fin du traitement d'un chemin préférence. On doit tout d'abord restaurer le nœud préférence courant (c'est celui au sommet de la pile *prefNodeStack*) puis, filtrer l'entrée correspondant à son pivot dans la table *answerPivotTable* avec la solution courante *currentAnswer* pour n'y retenir que les occurrences du pivot recouvrant au moins une occurrence du nœud préférence courant contenue dans *currentAnswer*. Le résultat du filtrage est stocké dans la table *infoPrefNodeTable* à l'entrée correspondant au nœud préférence courant. Pour finir, on doit effectuer tous les traitements énumérés ci-dessus pour l'action *EndOp*.

```
1  currentPrefNode = pop(prefNodeStack);
2  pivotCurrentPrefNode = pivotId(currentPrefNode);
3  indicePrefNode = indice(currentPrefNode, infoPivotPrefTable);
4  indicePivot= indice(pivotCurrentPrefNode, answerPivotTable);
5  tmpOccPivot = answerPivotTable[indicePivot].valeur;
6  si (pivotId(currentPrefNode) )==currentPrefNode /*ie. si un noeud
7                                          est sont propre pivot*/
8   infoPivotPrefTable[indicePrefNode] = AnswerPivotTable[indicePivot];
9  sinon
10      tmpOccPivot = {x ∈ tmpOccPivot, ∃y∈ currentAnswer,
11                                y ''occurrence de'' currentPrefNode,
12                                x//y};
13      infoPivotPrefTable[indicePrefNode] = tmpOccPivot;
14 tmpSol = top (partialSolutionStack);
15 pop(partialSolutionStack);
16 currentAnswer = currentAnswer ∪ tmpSol;
17 retourner (currentAnswer, δA(q, label q));
```

BeginStar	Annonce le début de traitement d'un chemin étoilé (chemin avec star (*)) du type $ch1/(ch)*/$. On doit ajouter à la solution partielle résultant du traitement de la sous requête *ch*1, les solutions provenant du traitement de zéro ou plusieurs occurrences de *ch*. La solution partielle V de $ch1/(ch)*/$ sera la somme $V = V_0 + V_1 + V_2 \ldots + V_n$, où V_i est la solution partielle résultant de la consommation de *ch* pour la i^{ime} fois. Les V_i ne sont pas indépendants : $V_{(i+1)}$ dépend de V_i dans la mesure où, les éléments de la solution partielle $V_{(i+1)}$ sont des descendants des éléments de V_i qui correspondent à la chaine ch : $V_{(i+1)} = \{v_i | \exists v_k \in V_i, v_i Desc_{ch}(v_k)\}$. $v_i Desc_{ch}(v_k)$ signifie que v_i est un descendant de v_k suivant la chaine *ch*.

```
1  V= VStar= currentAnswer;
2  qstar=q;
3  etiq=labelTrans  qstar //étiquette de l'arc sortant de l'état qstar
4  tantque ((labelTrans  qstar)!= EndStar )faire
5       (V, Tetiq)= filter CHILD currentAnswer Tetiq;
6       qstar = δA(qstar, etiq);
7       etiq=labelTrans  qstar;
8       si (etiq== EndStar)
9        (VStar, qstar ) = perform_ActionEndStar(V, VStar, q, qstar);
10 currentAnswer = VStar;
11 retourner (currentAnswer , δA(qstar, EndStar));
```

Suite, Page Suivante . . .

Nom action	Description des traitements associés
EndStar	Annonce la fin de la i^{ieme} consommation d'un chemin étoilé. Si lors de ce i^{ieme} passage on a récolté une solution, c.-à-d. si (V != ϕ), on doit l'ajouter à la solution partielle courante et se reconnecter au début du chemin étoilé. Sinon, on sort de ce chemin pour poursuivre le traitement de la requête résiduelle.

```
1  si (V!= { })  //on vient  de calculer la solution partielle  Vi
2       VStar = VStar + V;
3       qstar=q  //on se replace au début du chemin
4  sinon  //on ne fait rien
5  retourner  (VStar, qstar);
```

Dans le pseudo-code disséminé dans le tableau 2, la fonction *etiquette* retourne l'étiquette de l'unique arc associé à un état donné, et la fonction *perform_ActionEndStar(V, VStar ,q, qstar)* ; est une spécialisation de la fonction *performAction*. Elle est appelée pour invoquer les traitements liés à l'action *EndStar*. Ici, *V* est la solution obtenu lors du matching du *chemin étoilé*[10] pour la $i^{\text{ième}}$ fois, *VStar* la somme des solutions obtenues lors du matching au cours des (i-1) fois précédentes, *q* l'état correspond à l'action *BeginStar* associé à l'action *EndStar* courant, *qstar* l'état courant associé à l'action *EndStar*.

4.2.3. L'algorithme

Rappelons que l'évaluation d'une requête avec préférences se fait en deux étapes : (1) recherche de toutes les solutions de la requête (sans tenir compte des préférences) puis, (2) sélection des meilleures solutions (c'est celles qui *intègrent*[11] un maximum d'occurrence de nœud préférence).

Notons que quand un nœud préférence apparait dans un chemin prédicat, il est impossible à postériori de savoir s'il a été impliqué dans la production d'une réponse donnée de la requête. Pour solutionner ce problème, pendant le déroulement de la première étape (décrite section précédente), on a construit en parallèle une table *infoPrefNodeTable*.

Etape1 : évaluation sans tenir compte des préférences

L'algorithme de la première étape d'évaluation d'une requête *PrefSXPath Q* sur un document *D* (algo. 1) prend en entrée un index Tq de *D* et l'automate *A* associé à *Q*. Il retourne le triplet *(currentAnswer, infoPrefNodeTable, answerPivotTable)* utilisé dans la seconde étape pour construire la table *preferenceTable* de laquelle sont extraites les meilleures réponses. Dans l'algorithme 1, on parcourt l'automate *A* à partir de l'état initial en exécutant suivant le type de l'étiquette de la transition qui part de l'état courant le traitement correspondant : si c'est une valeur $a \in \Sigma$, (on est sur un chemin de type $ch1/a/ch2$), alors, on filtre dans la liste Ta les occurrences de *a* qui sont fils d'un élément de la solution courante *currentAnswer* (algo. 1, ligne 15) et si *a* est un pivot, on remplit l'entrée correspondante dans la table *answerPivotTable* (algo. 1, lignes 16-18) ; sinon si c'est λ, (on est sur un chemin de type $a//b$) alors, on filtre dans la liste Tb les occurrences de *b* qui sont descendants d'un élément de la solution courante *currentAns-*

10. Nous appelons ainsi un chemin requête contenant l'étoile (*). Par exemple, dans la requête $i(j/k)*$, j/k est un chemin étoilé.

11. Soit *Q* une requête avec préférence ayant pour nœud résultat Nr et possédant un nœud préférence Np ; Np est situé soit sur le chemin principal de *Q* soit dans un chemin préférence de *Q*. Nous disons qu'*une occurrence nr de Nr intègre* une occurrence np de Np si np *recouvre* nr dans le cas où Np est situé sur le chemin principal ou alors, s'il existe une occurrence $npiv$ du pivot de np et telle que $npiv$ recouvre à la fois np et nr.

wer (algo 1, lignes 6-8). Enfin, si c'est une action *(BeginStar, BeginPref, EndPref, ...)*, les instructions prévues pour le traitement de celle-ci (tableau 2) sont exécutées.

entrée: - L'index T_q du document ;
- L'automate A associé à la requête;
sortie : Toutes les occurrences du nœud résultats de la requête satisfaisant la requête

```
1  currentState = q0 ; answerPivotTable = ϕ ; infoPrefNodeTable = ϕ;
2  currentAnswer ='/' ;                /* On initialise à la racine */;
3  tantque currentState != FA faire
4      a=labelTrans currentState        /* On récupère le label de la transition
       courante  */;
5      si a= = λ alors                                              /* ch//b */
6          currentState = δA(currentState, a);
7          b= labelTrans currentState;
8          (currentAnswer, Tb) = filter ANC-DESC currentAnswer Tb;
9          currentState = δA(currentState, b);
10     sinon
11         si a ∈ Σact alors
12             (currentAnswer, qc) = perform_Action (currentAnswer, q, a);
13             currentState = qc;
14         sinon                                                    /* a ∈ Σ */
15             (currentAnswer, Ta) = filter CHILD currentAnswer Ta;
16             si (isPivot(a)) alors
17                 indiceA= indice(a, answerPivotTable);
18                 answerPivotTable[indiceA] = currentAnswer ;
19                 currentState = δA(currentState, a);
20 return (currentAnswer, answerPivotTable, infoPrefNodeTable ) ;
```

Algorithm 1: Twig-Automata-Preference-Match pour l'étape 1

Etape 2 : extraction des meilleures réponses

La seconde étape est divisée en deux phases : dans la phase 1 on construit la table *preferenceTable* et dans la phase 2 on sélectionne les meilleures réponses en appliquant *l'opérateur skyline*[12][12] sur les tuples de la table *preferenceTable* pour ne retenir que les réponses contenues dans l'ensemble constitué des tuples non dominés.

Dans ce qui suit, pour des besoins d'illustrations, nous considérons la requête *PrefSX-Path* $Q = /a_1/\ldots/a_{(k-2)}/b![\ldots]/a_k/\ldots/a_{(k+l)}/c[d_1/\ldots/d_{k_{pred}}/g![\ldots]/\ldots]/a_{(k+l+2)}/\ldots/a_s/f$ ayant f comme nœud résultat, possédant deux nœuds préférences b et g situés respectivement sur le chemin principal et sur un chemin prédicat. Considérons aussi que :

• Q a pour chemin principal $/a_1/\ldots/a_{(k-2)}/b!/a_k/\ldots/a_{(k+l)}/c/a_{(k+l+2)}/\ldots/a_s/f$ de longueur s+1.

• $/a_k/\ldots/a_{(k+l)}/c/a_{(k+l+2)}/\ldots/a_s/f$ est le suffixe du chemin principal comprenant $K =$

12. L'opérateur skyline [12] permet de sélectionner les meilleurs n-uplets d'une table relationnelle : ce sont ceux qui ne sont pas dominés au sens de la relation de préférence. De façon sommaire, il peut être présentée comme suit : soient deux tuples $p = (p_1,\ldots,p_k,p_{k+1},\ldots p_n)$ et $q = (q_1,\ldots,q_k,q_{k+1},\ldots q_n)$ d'une table relationnelle R de schéma $R(P_1,\ldots P_K,P_{K+1},\ldots P_n)$. Pour les requêtes dans lesquelles les préférences portent sur les champs $P_{k+1},\ldots,P_n$, on dira que ***p domine q*** et on note $p > q$, si les trois conditions suivantes sont satisfaites : (1) $p_i = q_i$, pour tout $i = 1,2,\ldots k$. (2) $p_i \geq q_i$ pour tout $i = (k+1),\ldots,n$. (3) il existe i, $(k+1) \leq i \leq n$, et $p_i > q_i$.

$s+1-(k-1) = s-k+2$ nœuds, parmi lesquels on suppose avoir $p,\ 0 \leq p \leq K-1$ nœuds préférences.

• Le (sous-)chemin $d_1/\ldots/d_{(k_{pred})}$ contenu dans le prédicat associé au nœud c est le préfixe d'un chemin principal se terminant juste avant le nœud g et contenant k_{pred} nœuds[13], parmi lesquels on suppose qu'on ait $p_{pred},\ 0 \leq p_{pred} \leq k_{pred}$ autres nœuds préférences.

• $a_{(k+l+2)}/\ldots/a_s/f$ sont les $k_{princi} = s+1-(k+l+1) = s-k-l$ nœuds du chemin principal de Q compris entre c et le nœud résultat f, parmi lesquels on suppose y avoir $p_{princi},\ 0 \leq p_{princi} \leq k_{princi}$ nœuds préférences.

A l'aide d'une telle requête, présentons comment s'effectue le remplissage de la table *preferenceTable*. Elle comporte pour le cas de cet exemple au moins deux colonnes étiquetées b et g (ce sont les deux seuls nœuds préférences effectivement mis en exergue dans la requête Q). De façon générale, nous examinerons le cas où le nœud préférence est situé sur le chemin principal et celui dans lequel il ne l'est pas.

• **Cas du nœud préférence « b » situé sur le chemin principal**

Les occurrences $f_j\left(f_{js}, f_{je}, f_{jl}\right)$[14] de f intégrant une occurrence $b_i(b_{is}, b_{ie}, b_{il})$ de b satisfont : $\begin{cases} b_{is} < f_{js} < f_{je} < b_{ie} & (1) \\ f_{jl} = b_{il} + m, \quad m \in \{K-p, K-p+1, \ldots, K\} & (2) \end{cases}$

L'équation (1) exprime le fait que $b_i(b_{is}, b_{ie}, b_{il})$ recouvre $f_j\left(f_{js}, f_{je}, f_{jl}\right)$. Tenant compte de ce que le (sous-)chemin allant de bi à fi peut éventuellement contenir une occurrence pour chacun des p nœuds préférences compris entre les nœuds b et f, l'équation (2) exprime le fait que f_j doit être situé à une profondeur comprise entre $(K-p)$ et (K) de bi.

• **Cas du nœud préférence « g » situé dans un chemin prédicat**

Les occurrences $f_j\left(f_{js}, f_{je}, f_{jl}\right)$ de f qui intègrent une occurrence $g_i(g_{is}, g_{ie}, g_{il})$ de g ayant pour pivot l'occurrence $c_v\,(c_{vs}, c_{ve}, c_{vl})$ de c[15] satisfont :

$$\begin{cases} c_{vs} < f_{js} < f_{je} < c_{ve} & /* \, c_v \textit{ recouvre } f_j \, */ & (3) \\ c_{vs} < g_{is} < g_{ie} < c_{ve} & /* \, c_v \textit{ recouvre } g_i \, */ & (4) \\ f_{jl} = c_{vl} + m, & m \in \{k_{princi} - p_{princi}, \ldots, k_{princi}\} & (5) \\ g_{il} = c_{vl} + n, & n \in \left\{k_{pred} - p_{pred}, \ldots, k_{pred} + 1\right\} & (6) \end{cases}$$

Les équations (3) et (4) expriment le fait que c_v doit recouvrir à la fois f_j et g_i. De même que pour le cas précédent, le (sous-)chemin allant de c_v à f_j (resp. de c_v à g_i) contient éventuellement des occurrences des k_{princi} (resp. p_{pred}) nœuds optionnels compris entre les nœuds c_v et f_j (resp. c_v et g_i). L'équation (5) (resp. 6) exprime le fait que f_j (resp. g_i) doit être situé à une profondeur comprise entre $(k_{princi} - p_{princi})$ et (k_{princi}) (resp. $(k_{pred} + 1 - p_{pred})$ et $(k_{pred} + 1)$ de c_v) de c_v.

Ainsi donc, pour tout résultat f_j de la requête, pour un nœud préférence b situé sur le chemin principal, l'entrée *preferenceTable[f_j, b] = 1* s'il existe au moins une occurrence b_{is} de b dans la liste présente dans l'entrée *infoPrefNodeTable[b]* recouvrant f_j (équation 1) et située à la "bonne profondeur"' dans le sous arbre ayant b_{is} pour racine (équations 2), sinon, elle est mise à zero. De même, pour un nœud préférence g situé dans un prédicat, l'entrée *preferenceTable[f_j, g] = 1* s'il existe au moins une occurrence c_{vs} du pivot c de g dans la liste présente dans *answerPivotTable*$[c]$ recouvrant à la fois f_j (équation 3) et au moins une occurrence g_{is} de g dans la solution partielle contenue dans *infoPrefNode-*

13. Tous appartenant au chemin principal.

14. Rappelons que f_{js}, f_{je}, f_{jl} sont respectivement les composants du triplet (start, end, level) représentant le nœud f_j dans le document.

15. Rappel : g a pour pivot c.

Table[g] (équation 4). f_j et g_{is} doivent être situées à la bonne profondeur dans le sous arbre ayant c_{vs} pour racine (équations 5 et 6) ; sinon, elle est mise à zero.

Enfin, l'*opérateur skyline* [12] est appliqué aux tuples de la table *PreferenceTable* ainsi construite pour déterminer les meilleures solutions.

4.3. Etude de la complexité de l'algorithme d'évaluation

Le temps processeur nécessaire pour l'évaluation d'une requête *PrefSXPath* est égal à la somme des temps utilisés pour l'appariement, pour la construction des tableaux *AnswerPivotTable, infoPrefNodeTable* et *preferenceTable,* et de la sélection des meilleures solutions par l'algorithme Skyline. Ces différents temps sont calculés comme suit :

• **Temps nécessaire pour l'appariement** : Il est égal au temps correspondant aux opérations de jointures (évaluation des relations père-fils ou ancêtre-descendant) et de filtre (*filterUp*). Soit n la taille moyenne des listes T_q constituant l'index. Rappelons qu'une opération de filtre ou de jointure consiste en la recherche des descendants des éléments d'une liste L_1 dans une liste L_2 toutes triées. Bien que les listes manipulées par l'opération *filterUp* soient nettement plus petites que celles constituant l'index (les listes T_q) nous considèrerons que les listes L_1 et L_2 sont aussi de taille n.

Au pire des cas, chaque parent (resp ancêtre) dans L_1 a exactement un fils (resp descendant) dans la liste L_2. Pour une opération de jointure, on a $2n-1$ comparaisons : un nœud a_i de L_1 différent du dernier est comparé aux nœuds d_i et d_{i+1} de L_2 pour successivement valider le fait que d_i est descendant de a_i et que d_{i+1} ne l'est pas.

Au meilleur des cas, la complexité de la jointure est de n ; en effet, dans ce cas, le premier élément de la liste L_1 est l'ancêtre (ou le parent) de tous les éléments de la liste des descendants L_2.

Soit n_{req} le nombre de nœuds requêtes et $n_{feuille}$ son nombre de feuilles, on a $n_{req}-1$ opérations de jointure. Le nombre d'opérations de filtre est de $n_{feuille}-1$ et correspond au nombre d'action *filterUp*. Le temps processeur de l'appariement au pire des cas est alors de $C1 = (2n-1)(n_{req}-1+n_{feuille}-1) = (2n-1)(n_{req}+n_{feuille}-2)$.

• **Temps de construction du tableau answerpivotTable** : Le temps processeur nécessaire pour conserver des occurrences d'un pivot est égal au temps de copie de ces dernières dans la case correspondante du tableau *answerPivotTable*. Soient n_{pivot} le nombre de nœuds pivots et $n_{occPivot}$ la taille moyenne des listes contenant les pivots ; elle est plus petite que celle des listes T_q, ($n_{occPivot} < n$) car elles proviennent du filtrage de la liste *CurrentAnswer*. Le temps de construction du tableau *answerPivotTable* est donc $C2 = n_{pivot} * n_{occPivot}$.

• **Temps de construction du tableau infoPrefNodeTable** :Rappelons que *infoPrefNodeTable* contient les occurrences du pivot de chaque nœud préférence. Le temps processeur nécessaire pour la construction de celui-ci est de : $C3 = n_{pref} * n_{occPref} * log(n_{0ccPivot})$. Où $n_{0ccPref}$ représente le nombre moyen d'occurrences des pivots couvrant au moins un nœud préférence, et n_{pref} est le nombre de nœuds préférences ; $log(n_{0ccPivot})$ est le temps de recherche (dichotomique) d'un élément dans une liste triée.

• **Temps de construction du tableau preferenceTable** : Il est égal à $C4 = n_{pref} * n_{sol} * log(n_{info})$ où, n_{sol} est la taille moyenne de l'ensemble solution. n_{info} est la taille moyenne des listes contenues dans les cellules du tableau *infoPrefNodeTable*.

• **Complexité de sélection des meilleures solutions :** Elle dépend de l'algorithme Skyline utilisé : lorsque l'ensemble solution est de petite taille, on peut utiliser l'algo-

rithme de boucles imbriquées (Block Nested Loop) pour sélectionner les solutions non dominées; dans ce cas, le temps processeur est de l'ordre de $C_{5BNL} = O(n_{sol}^2)$ [12]. Pour un ensemble solutions de grande taille, on peut utiliser l'algorithme basé sur l'approche de Diviser Pour Régner (DPR). Dans ce cas, la complexité est de l'ordre de : $C_{5DPR} = O(n_{sol} * (log\ n_{sol})^{n_{pref}-2} + n_{sol} * log\ n_{sol})$ [12].

En définitive, la complexité totale est égale à $C1+C2+C3+C4+C_{5BNL} = O[n(n_{req} + n_{feuille}) + n_{pivot} * n_{occPivot} + n_{pref} * n_{occPref} * log(n_{occPivot}) + n_{pref} * n_{sol} * log(n_{info}) + n_{sol}^2]$ si l'algorithme de recherche des points skyline utilisé est celui basé sur les boucles imbriquées ou alors, $C1 + C2 + C3 + C4 + C_{5DPR} = O[n(n_{req} + n_{feuille}) + n_{pivot} * n_{occPivot} + n_{pref} * n_{occPref} * log(n_{occPivot}) + n_{pref} * n_{sol} * log(n_{info}) + n_{sol} * (log\ n_{sol})^{n_{pref}-2} + n_{sol} * log\ n_{sol}]$ si l'algorithme de recherche des points skyline est basée sur l'approche DPR.

Remarquons que si la requête a peu de prédicat (le nombre de feuilles est négligeable devant le nombre de nœuds requêtes, $n_{feuille} \ll n_{req}$), on aura $C1 \approx n(n_{req})$; de même, le nombre de nœud pivot conservé sera aussi très petit, il tendra vers 1 et donc $C2$ tendra vers $n_{occPivot}$.

5. Conclusion

Nous avons exploré dans ce papier une approche d'expression et d'évaluation de requêtes *XPath* avec préférences. Pour ce faire, le langage *PrefSXPath* (une extension du langage XPath) a été proposé pour l'expression des requêtes avec préférences. Nous avons aussi mis en oeuvre un algorithme d'évaluation des mots du langage *PrefSXPath* (des requêtes) sur un document XML. La technique d'évaluation fait usage des automates pour déterminer les réponses candidates sans prise en compte des préférences et ensuite, application de l'opérateur Skyline sur ces réponses intermédiaires afin de déterminer les meilleures solutions. Une étude de la complexité de l'algorithme proposé a aussi été ménée.

Bien que l'algorithme proposé dans ce manuscrit ait été déroulé sur bien des exemples (parmi lesquels celui présenté dans annexe A) avec des résultats très satisfaisants, une étude analytique complète de ses performances est en cours de réalisation pour le parfaire. Bien plus, on envisage aussi d'utiliser les techniques de déforestations développées en programmation fonctionnelle pour proposer un algorithme d'évaluation qui procède en une étape en évitant de construire explicitement les réponses candidates intermédiares.

Le travail présenté dans ce papier est le point de départ d'un travail plus ambitieux ayant pour but la généralisation de l'approche présentée dans ce manuscrit. En fait, on projette de considérer les préférences comme des *aspects* exprimables via un DSL (Domain Specific Language) qu'on construira, et de les injecter dans les algorithmes d'évaluations d'expressions *XPath* déjà existants par un *tisseur de préférences* qu'on définira.

6. Bibliographie

[1] Abiteboul, S. *« Querying Semi-Structured Data »*, In Proceedings of the International Conference on Database Theory (ICDT), Delphi, Greece, pp. 1-18, 1997.

[2] Bing Sun, Bo Zhou, Nan Tang, Guoren Wang, Ge Yu, and Fulin Jia. *« Answering XML Twig Queries with Automata »*, In Jeffrey Xu Yu, Xuemin Lin, Hongjun Lu, and Yanchun Zhang, editors, Advanced Web Technologies and Applications, 6th Asia-Pacific Web Conference, APWeb 2004, Hangzhou, China, April 14-17, 2004, Proceedings, volume 3007 of Lecture Notes

in Computer Science, pp. 170-79. Springer, 2004.

[3] Bosc P., Pivert O. *« SQLf : a relational database language for fuzzy querying »*, IEEE Trans. On Fuzzy Systems, vol(3) pp.1-17, 1995.

[4] C.Y. Chan, P. Felber, M.N. Garofalakis, R. Rastogi, *« Efficient filtering of XML documents with XPath expressions »*, in : Proceedings of the 18th International Conference on Data Engineering (ICDE), IEEE Comput. Soc., pp. 235-244, 2002

[5] Chomicki J. *« Preference Formulas in Relational Queries »*, In ACM Trans. on Database Systems (TODS), vol. 28(4), 2003.

[6] Dubois D., Prade H. *Bipolarity in flexible quering.* Proc. of the 5th Int. Conf. on Flexible Query Answering Systems (FQAS), Copenhagen, Denmark, 2002.

[7] Gang Gou and Rada Chirkova. *« Efficiently Querying Large XML Data »*, Repositories : A Survey IEEE Transactions On Knowledge And Data Engineering, VOL. 19, NO. 10, pp. 1381-1402 OCTOBER 2007.

[8] Kiebling W. *« Foundations of Preferences in Database Systems »*, In Proc. of the 28th Int. Conf. on Very Large Databases (VLDB), pp. 311-322., Hong Kong, China, 2002.

[9] Lietard L., Rocacher D. and Tbahriti S.-E. *« Preferences and Bipolarity in Query Language »*, International Conference of the North American Fuzzy Information Processing Society (NAFIPS 2008), New-York, USA. pp. 1-6, 2008.

[10] Q. Li and B. Moon. *« Indexing and querying XML data for regular path expressions »*, Proceedings of the 27th VLDB Conference, pp. 361-370, 2001.

[11] Rakesh Agrawal , Jerry Kiernan , Ramakrishnan Srikant , Yirong Xu, *« An XPath-based preference language for P3P »*, Proceedings of the 12th international conference on World Wide Web, May 20-24, Budapest, Hungary, 2003

[12] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. *« The skyline operator »*, In Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Hiedelberg, Germany, pp. 421-430, 2001.

[13] T. Green, G. Miklau, M. Onizuka, D. Suciu, *« Processing XML streams with deterministic automata »*, in : Proceedings of the 9th International Conference on Database Theory (ICDT), Springer, pp. 173-189, 2003.

[14] W. Kießling, B. Hafenrichter, S. Fischer, S. Holland : *« Preference XPATH : A Query Language for E-Commerce »*, Proc. 5th Intern. Konf. für Wirtschaftsinformatik, Augsburg, pp. 425-440, Sept. 2001.

[15] W3C Consortium. *« XML Path Language (XPath) 2.0 »*, http ://www.w3.org/TR/XPath20/, 2006.

[16] W3C Consortium. *« XQuery 1.0 : An XML Query Language »*, http ://www.w3.org/TR/xquery/. 2006.

[17] Yao, J. T. and Zhang, M. *«A Fast Tree Pattern Matching Algorithm for XML Query », Proceedings of the 2004 IEEE/WIC/ACM International Conference on Web Intelligence*, (WI 2004), 20-24 September 2004, Beijing, China, pp. 235-241, 2004.

A. Annexe : Exemple complet d'évaluation d'une requête PrefSXPath

Cette section est consacrée à la présentation des différentes étapes du processus d'évaluation d'une requête PrefSXPath contenant la plus part des motifs adressés dans ce manuscrit. La requête considérée est la suivant : $Q' = /a//b([c([d/e!])!/i(j/k)^*)]!/a[d?]/g!/h$ (fig. 8). Elle est évaluée sur la source de données (fig.2) dont l'index est représenté sur la figure 7.

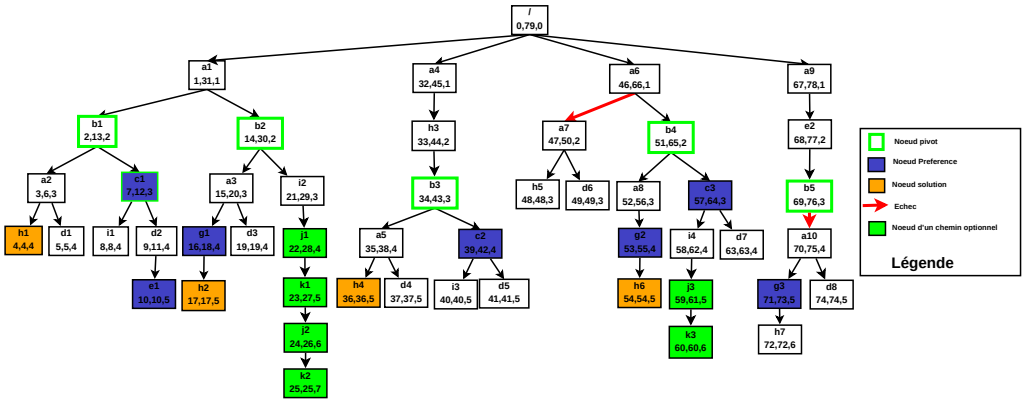


Figure 7. *La source de donnée décorée.*

Afin de pouvoir désigner de façon unique les étiquettes les nœuds requêtes de Q' à partir de leur labels, Q' est réécrit comme suit : $Q' = /a1//b([c([d1/e!])!/i(j/k)*)]!/a2[d2?]/g!/h$.

Statiquement, on déduit le nœud pivot associé à chaque nœud préférence (voir tableau 3).

A.1. Transformation de la requête en automate et extraction de l'index de la source de données

Par application des schémas de construction de l'automate à partir des motifs de requêtes *PrefSXPath* (voir Tab. 1) on obtient pour la requête Q' l'automate de la figure 9. L'index correspondant à la source de données de la figure 7 est représenté sur la figure 10

A.2. Evaluation de la requête Q' de la figure 8 sur la source de données de la figure 7

Le tableau 4 présente une trace d'exécution de l'algorithme *Twig-Automata-Preference-Match* pour l'étape 1. Les paramètres effectifs de l'appel sont l'index de la figure 10 et l'automate de la figure 9. Le résultat produit est *currentAnswer = {h1, h2, h4, h6}* (tableau 4 ligne 28). Les autres données générées au cours de l'exécution sont les tableaux *answerPivotTable* (tableau 6) et *infoNodeTable* (tableau 7).

La seconde étape de l'algorithme utilise les données contenues dans les tableaux précédents (tableaux 6 et 7) ainsi que la valeur de *currentAnswer* pour construire la table *preferenceTable* (tableau 8) sur laquelle on applique l'opérateur skyline pour obtenir comme meilleures réponses à la requête Q' les occurrences *h1* et *h6*. En effet,

• *h1* et *h2* sont incomparables car, *h1* recouvre une occurrence de *c* et de *e* ce qui n'est pas le cas pour h2 ; de même, *h2* recouvre une occurrence de *g* ce qui n'est pas le cas pour *h1*.

• *h1* domine *h4* car, bien que *h1* et *h4* recouvrent des occurrernces de *c*, *h1* recouvre en plus une occurrence de *e* et une occurrence de *g*, ce qui n'est pas le cas de *h4* : on a

Nœud préférence	c	e	g
Nœud pivot	b	b	g

Tableau 3. *Nœud préférence et pivot associé*

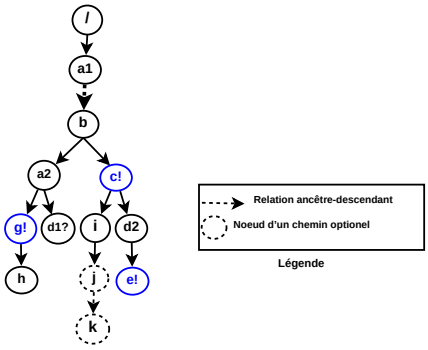


Figure 8. *La requête sous forme arborescente.*

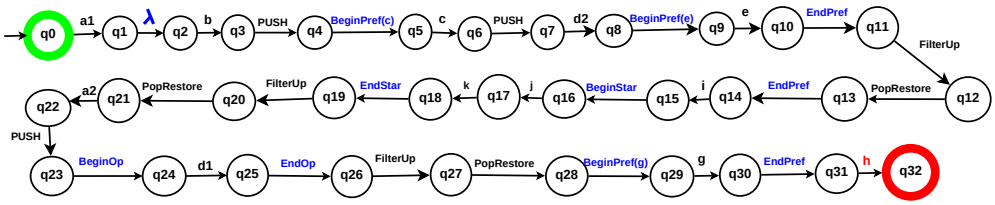


Figure 9. *Automate correspondant à la requête de la figure 8.*

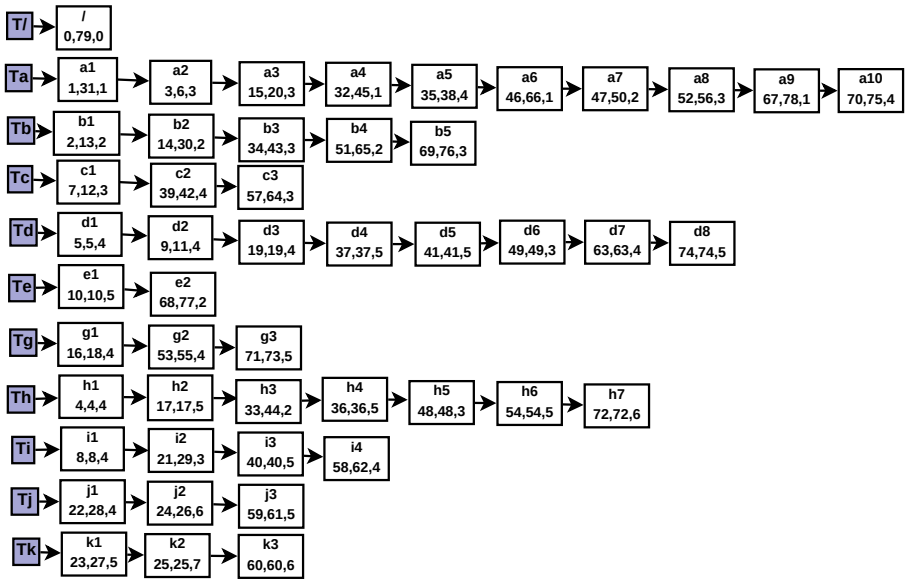


Figure 10. *Index extrait de la source de données de la figure 7.*

donc $h4 < h1$.

• *h6* domine *h2* car, bien que *h6* et *h2* recouvrent des occurrernces de *g*, *h6* recouvre en plus une occurrence de *c*, ce qui n'est pas le cas pour *h1* : on a donc $h2 < h6$.

Les occurrences *h2* et *h4* sont donc supprimées du résultat final et l'ensemble solution retourné n'est constitué que des occurrences non dominées c'est-à-dire {*h1*, *h6*}.

Le tableau 5 fait un zoom sur les étapes d'exécution correspondant au (sous-)chemin étoilé (les états q15 à q19 présentés dans le tableau 4).

Etapes	current State	Etiquette Trans (a)	next State	CurrentAnswer	Partial SolutionStack	Preference Node Stack
init			q0	{/}		
1	q0	a1	q1	{a1, a4, a6, a9}		
2	q1	λ	q3	{b1, b2, b3, b4, b5}		
3	q3	Push	q4		[{b1, b2, b3, b4, b5}]	
4	q4	BeginPref(c)	q5		[{b1, b2, b3, b4, b5}, {b1, b2, b3, b4, b5}]	[c]
5	q5	C	q6	{c1, c2, c3}		
6	q6	Push	q7		[{b1, b2, b3, b4, b5}, {b1, b2, b3, b4, b5},{c1, c2,c3}]	
7	q7	d_2	q8	{d2, d5, d7}		
8	q8	BeginPref(e)	q9	{d2, d5, d7}	[{b1, b2, b3, b4, b5}, {b1, b2, b3, b4, b5},{c1, c2,c3}, {d2, d5, d7}]	[c, e]
9	q9	e	q10	{e1}		[c, e]
10	q10	EndPref	q11	{d2, d5, d7, e1}	[{b1, b2, b3, b4, b5}, {b1, b2, b3, b4, b5},{c1, c2,c3}]	[c]
11	q11	FilterUp	q12		[{b1, b2, b3, b4, b5}, {b1, b2, b3, b4, b5}, {c1, c2,c3}]	
12	q12	PopRestore	q13	{c1, c2,c3}	[{b1, b2, b3, b4, b5}, {b1, b2, b3, b4, b5}]	
13	q13	EndPref	q14	{c1, c2,c3, b1, b2, b3, b4, b5}	[{b1, b2, b3, b4, b5}]	[b]
14	q14	I	q15	{i1, i2, i3, i4}		
15	q15	BeginStar		{i1, i2, i3, i4, k1, k2, k3}		
16	q19	FilterUp	q20		[{b1, b2, b3, b4}]	
17	q20	PopRestore	q21	[{b1, b2, b3, b4}]	[]	
18	q21	a2	q22	{a2, a3, a5, a8}		
19	q22	Push	q23		[{a2, a3, a5, a8}]	
20	q23	beginOp	q24	{a2, a3, a5, a8}	[{a2, a3, a5, a8, a10}, {a2, a3, a5, a8}]	
21	q24	d1	q25	{d1, d3, d4}		
22	q25	EndOp	q26	{a2, a3, a5, a8, d1, d3, d4}	[{a2, a3, a5, a8}]	
23	q26	FilterUp	q27		[{a2, a3, a5, a8}]	
24	q27	PopRestore	q28	{a2, a3, a5, a8}	[]	
25	q28	BeginPref(g)	q29		[{a2, a3, a5, a8}]	[g]
26	q29	g	q30	{g1, g2}		[g]
27	q30	EndPref	q31	{a2, a3, a5, a8, g1, g2}	[]	[]
28	q31	h	q32	{h1, h2, h4, h6}		
29	q32	Etat finale donc la solution à cette étape est {h1, h2, h4, h6}				

Tableau 4. *Une trace d'exécution de l'étape 1 du processus d'évaluation de Q'.*

	Etape	qstar	Etiqu.(Etstar)	V	VStar	Nextq	Commentaire
init		q16		{i1, i2, i3, i4}	{i1, i2, i3, i4}		
1	1	q16	j	{j1, j3 }			
	2	q17	k	{k1, k3 }			
	3	q18	EndStar	{k1, k3 }= v≠Ø	{i1, i2, i3, i4, k1, k3 }	q16	VStar=VStar+V
2	1	q16	j	{j2}			
	2	q17	k	{k2}			
	3	q18	EndStar	{k2} ≠Ø	{i1, i2, i3, i4, k1, k3, k2}	q16	
3	1	q16	j	{}			
	2	q17	k	{}			
	3	q18	EndStar	{}	Puisque Etiquette =EndStar et V= {} on sort de la procédure et CurrentAnswer= VStar et nextState= δ_A(qstar, Etstar) ;// q=q19,		

Tableau 5. *Zoom sur la trace d'exécution du (sous-)chemin étoilé (chemin (i/j)**

Nœuds pivots	b	g
Occurrences du nœud pivot après la traversée de l'étiquette correspondante	{b1, b2, b3, b4, b5}	{g1, g2}

Tableau 6. *AnswerPivotTable*

Nœuds préférences	c	e	g
Occurrences du pivot correspondant	{b1, b3, b4}	{b1, b3}	{g1, g2}

Tableau 7. *InfoNodeTable*

	c	e	g
h1	1	1	0
h2	0	0	1
h4	1	0	0
h6	1	0	1

Tableau 8. *PreferenceTable mettant en exergue h1 et h6 comme meilleures réponses : elles sont non dominées*