Dataontsluiting met XML, XQuery en XSLT

Door Anne Schuth en Maarten Marx,
anne.schuth@uva.nl | *maartenmarx@uva.nl*

Om de kracht van de combinatie van XML, XQuery en XSLT te demonstreren beschrijven we een webapplicatie die volledig in XML-gerelateerde W3C-standaarden geïmplementeerd is. We transformeren de oorspronkelijke, platte data met XSLT naar een gestructureerde XML-database en ontsluiten de bijbehorende video's met behulp van *full-text search*, geïmplementeerd in XQuery. We laten voorbeeldcode van XSLT en XQuery zien en geven een onverzicht van de mogelijkheden van de XML-database *eXist*.

Introductie

Dit artikel laat zien hoe met louter XML-technologie data op een volwaardige manier ontsloten kan worden. We laten zien hoe W3C-standaarden als XML, XPath 2.0, XSLT 2.0, XQuery 1.0, XQuery Full-Text 1.0 en de XQuery Update Facility 1.0 (zie ook het artikel *XQuery Novelties Revisited* van Geert Josten in dit nummer) geïntegreerd kunnen worden om een gebruikersvriendelijke webinterface, een ingang naar data, te ontwikkelen.

Om de kracht en voordelen van het gebruik van deze technieken te illustreren beschrijven we in dit artikel een klein maar volledig uitgewerkt voorbeeld. Marianne Winslett heeft tussen 2002 en 2010 32 interviews afgenomen met vooraanstaande database-onderzoekers. Deze interviews zijn gepubliceerd in *SIGMOD Record* en te lezen, beluisteren en bekijken op de ACM-website [1]. Ons doel is om de video en uitgeschreven interviews samen te brengen in een systeem dat gebruikers in staat stelt de uitgeschreven tekst te bevragen en als antwoord de relevante delen in de video te krijgen. Eerder hebben we een vergelijkbaar systeem ontwikkeld voor het zoeken in de videobeelden van debatten in de Tweede Kamer (zie http://openkamer.tv). Dit maakte echter gebruik van een relationele MySQL-database. We stellen de volgende eisen aan ons systeem:

1. alles gebeurt geautomatiseerd (schaalbaarheid);
2. met slechts XML-technologie (uniformiteit);
3. de gegevensindeling moet views en analytische vragen toestaan (herbruikbaarheid);
4. gebruikers worden naar de juiste plek in de data gebracht, afhankelijk van de zoekvraag (productiviteit).

Afgezien van het feit dat de aanpak met XML-technologie een elegante is, levert ze nog een voordeel op: we kunnen op inhoud *en* structuur zoeken, niet slechts op inhoud; we gaan van document- naar structuurgerichte ontsluiting.

In het algemeen wordt veel data geproduceerd om door mensen gelezen te worden; de meeste documenten volgen een eenvoudig chronologisch datamodel. Dit geldt ook voor onze dataset: elk document bevat een verslag van een interview dat van boven naar beneden gelezen kan worden. Als de data in een dergelijk model, waarbij een document de kleinste eenheid is, is opgeslagen kan een zoekmachine niet veel beter doen dan een lijst met documenten teruggeven als antwoord op een zoekvraag. We noemen dit documentgericht ontsluiten. De toegang tot de datacollectie wordt bepaald door de manier van opslaan, en er is in dit geval slechts één wijze van toegang en één soort antwoord mogelijk: altijd een heel document. Zoeken naar het eigenlijke antwoord op de zoekvraag is dan aan de gebruiker en diens vaardigheid met de toetsen `Ctrl+F` [2].
Het kan ook anders. Wanneer de verslagen van interviews nader bekeken worden, valt er meteen iets op: van elk woord is bekend wie het gezegd heeft, de interviewer of de geïnterviewde. Deze informatie is jammer genoeg alleen impliciet, verstopt in opmaak, aanwezig. Voordat je een computer kan vragen om "alle zinnen gesproken door David Maier" zal die impliciete structuur eerst expliciet gemaakt moeten worden. Vaak wordt dezelfde vraag aan bijna alle geïnterviewden gesteld. Zodra we de structuurinformatie expliciet gemaakt hebben zijn we in staat alle antwoorden op een enkele vraag naast elkaar als antwoord op een zoekvraag te geven.
Dit is wat we bedoelen met structuurgerichte ontsluiting. Dezelfde data kan nu opeens vanuit een heel ander perspectief bekeken worden. Het antwoord op een zoekvraag bij documentgericht ontsluiten is altijd een selectie van een of meer documenten. Het antwoord bij persoonsgericht ontsluiten is veel specifieker, het document is niet langer de kleinste eenheid, en sluit dichter aan op de zoekvraag; `Ctrl+F` wordt overbodig.

Structuur extraheren met XSLT

Om aan eisen 2, 3 en 4 te voldoen moeten we de interviews vanuit PDF omzetten en de aanwezige

```
start = element pdf2xml {
  element page {
    element t {
      (text
      | element i { text })+
    }+
  }+
}
```

Figuur 1: RelaxNG-schema voor syntactische representatie.

```
start = element interview {
  element title { text },
  element intro { text },
  element author { text },
  element qa {
    element q { element p { text }+ },
    element a { element p { text }+ }
  }+
}
```

Figuur 2: RelaxNG-schema voor semantische representatie

impliciete structuur expliciet maken. De interviews hebben een zeer regelmatige indeling: ze bestaan uit vraag-antwoordparen met wat inleidende informatie aan het begin. Lange vragen of antwoorden zijn in alinea's opgedeeld. We extraheren de tekst uit de PDF met behulp van het programma `pdftohtml` met de argumenten `-xml` en `-hidden`. We hebben kleine eigenaardigheden in het resultaat van dit programma gerepareerd met behulp van het Python-pakket BeautifulStoneSoup om tot welgevormde XML te komen. De resulterende XML-documenten volgen het schema in figuur 1.

Een document met een dergelijke indeling kan getransformeerd worden in een XML-document dat voldoet aan het schema in figuur 2, waarin dus alle structuur expliciet is gemaakt, met een zogenoemde *tekstbehoudende* transformatie, zie ook figuur 3. In het algemeen is dat een transformatie die toegepast kan worden op XML-bomen met tekst in de bladeren waarbij de (document-)volgorde van die bladeren van belang is. Een dergelijke transformatie kan een boom bijvoorbeeld meer of minder diepte geven zolang deze volgorde niet verandert. Dat is precies wat wij willen doen: in onze data is de tekstvolgorde van groot belang en we willen van een relatieve platte boom (figuur 1) naar een diepere boom met meer structuur (figuur 2) transformeren.

Uiteraard is XSLT de aangewezen taal voor een dergelijke transformatie van en naar XML. Zoals weergegeven in figuur 3 doen we de transformatie in twee stappen. Eerst annoteren we ieder element met een `q`, `a` dan wel `p`. Merk op dat deze transformatie *tekstbehoudend* is; de volgorde van de tekst in de bladeren is niet gewijzigd.

Hierna is het mogelijk om de vraag-antwoordparen samen te pakken door middel van een XSLT-transformatie als in figuur 4. Merk op dat ook dit algoritme in principe een *tekstbehoudende* transformatie is, waarbij we ons de vrijheid gepermitteerd hebben om aangrenzende bladeren met slechts tekst samen te voegen tot een enkel blad. Het *stylesheet* blijft klein en overzichtelijk doordat we de constructie "pak alle knopen totdat..." kunnen implementeren met de operator `except` in XPath 2.0. Bij grote input wordt dit wel erg inefficiënt en kan men hier beter een recursieve XSLT-functie voor definiëren. Jammer genoeg bevat XPath 2.0 (nog?) geen afsluitingsoperator als de Kleene-ster welke het dure gebruik van `except` overbodig zou maken.

Meteen naar de juiste plek

Nu we nette data hebben, met een expliciete structuur, wordt het mogelijk deze data zo te indexeren en vervolgens te bevragen dat we het structuurgericht kunnen ontsluiten (zie ook [2], [3] en [4]). Anders dan conventionele zoekmachines (zoals Google, Yahoo en Baidu) die altijd hele documenten teruggeven kunnen we stukjes uit documenten teruggeven. In principe

```
<pdf2xml>                <tussenstap>
  <t><i>0</i></t>          <q>0</q>
  <t><i>1</i></t>          <q>1</q>
  <t>2</t>                 <a>2</a>
  <t>3</t>                 <a>3</a>
  <t>4</t>                 <a>4</a>
  <t/>                     <p/>
  <t>5</t>                 <a>5</a>
  <t>6</t>                 <a>6</a>
  <t><i>7</i></t>          <q>7</q>
  <t>8</t>                 <a>8</a>
  <t>9</t>                 <a>9</a>
</pdf2xml>               </tussenstap>

<interview>
  <qa>
  <q><p>0 1</p></q>
  <a><p>2 3 4</p><p>5 6</p></a>
  </qa>
  <qa>
    <q><p>7</p></q>
    <a><p>8 9</p></a>
  </qa>
</interview>
```

Figuur 3: Schematische weergave van de tekstbehoudende transformatie in twee stappen. Om het geheel overzichtelijk te houden zijn de elementen `title`, `intro` en `author` weggelaten

```
<!-- Loop through all text elements that start a question that are not preceeded by a question node -->
<xsl:for-each select="//tussenstap//q[not(./preceding::*[position() le 2][self::q])]">
  <!-- Collect all following question and paragraph nodes before the next answer -->
  <xsl:variable name="q" select=". union (./(following::q|p) except ./following::a/following::*)"/>
  <!-- Collect all following answer and paragraph nodes before the next question -->
  <xsl:variable name="a" select="$q[last()]/(following::a|p) except $q[last()]/following::q/following::*"/>
  <qa>
    <q>
      <!-- Loop through all question nodes that start a question paragraph -->
      <xsl:for-each select="$q[self::q and not(./preceding::*[1][self::q])]">
        <p>
          <!-- Glue all following question nodes together -->
          <xsl:value-of select="normalize-space(string-join(. union (./following::q except ./following::p/
               following::q), ' '))"/>
        </p>
      </xsl:for-each>
    </q>
    <a>
      <!-- Loop through all answer nodes that start an answer paragraph -->
      <xsl:for-each select="$a[self::a and not(./preceding::*[1][self::a])]">
        <p>
          <!-- Glue all following answer nodes together -->
          <xsl:value-of select="normalize-space(string-join(. union (./following::a except ./following::p/
               following::a), ' '))"/>
        </p>
      </xsl:for-each>
    </a>
  </qa>
</xsl:for-each>
```

Figuur 4: Tekstbehoudende XSLT 2.0-transformatie van de tussenstap naar een XML-document dat voldoet aan het schema in figuur 2.

kunnen we ieder willekeurig XML-element teruggeven, idealiter *die* elementen die het meest relevant *en* het meest specifiek zijn. Dat laatste, de specificiteit, is precies waar het bij conventionele zoekmachines aan schort. Het is dan aan de gebruiker om in het resulterende document de relevante passage te vinden.
Bij tekstuele data gaat dat nog maar video wordt dat aanzienlijk lastiger. Willen we het juiste stuk video terug kunnen geven dan zullen we zeer specifiek in de transcripten van de interviews moeten zoeken.
Dat betekent uiteraard wel dat we die transcripties moeten annoteren met tijdcodes. We hebben ervoor gekozen om slechts het begin van de vragen aan te merken als ingang in de video en niet een willekeurig XML-element omdat we deze stap niet geautomatiseerd hebben en dus handmatig door de video's heen moeten gaan. We voldoen dus niet volledig aan eis 1.
Vervolgens hebben we vier *full-text* indexen gedefinieerd: één op elk van de elementen `q`, `a`, `qa` en `interview`. Wat een full-text index doet is alle `text()`-knopen die er in de boom onder één van de genoemde elementen vallen samenpakken en hier een *omgekeerde index* op maken. Een omgekeerde index is niet veel meer dan een lijst van alle woorden met verwijzingen naar alle elementen waar ze in voorkomen. We doen dit met behulp van het *open source*-softwarepakket eXist 1.4 (een XML-database) [5] dat Lucene 2.9.2 (een zoekmachine) [6] als integraal onderdeel bevat. In zekere zin is dit een implementatie van de W3C XQuery Full-Text 1.0 Recommendation (zoals besproken in het artikel van Geert Josten), al is de syntaxis iets anders.

Het definiëren van deze vier indexen heeft twee redenen. Gebruikers kunnen specificeren of ze willen zoeken in de vragen, de antwoorden, of in allebei. Verder kunnen we de indexen combineren om de ordening van de antwoorden te verbeteren [5]; dat gaat als volgt: als een gebruiker bijvoorbeeld aangegeven heeft dat het antwoord op de zoekvraag in een vraag-antwoordpaar gezocht moet worden, dan gebruiken we informatie uit de indexen voor de elementen `qa` en `interview`. We kunnen de `qa`-elementen ordenen aan de hand van *score*(zoekvraag, **qa**). Dat is de kans dat een zoekvraag voort kan komen uit een `qa`-element;met andere woorden, de kans dat een `qa`-element relevant is met betrekking tot de *query*. Dit kan berekend worden met behulp van de vergelijking in figuur 5.

De scores $\mathit{score}_{\texttt{qa}}$ (zoekvraag, **qa**) en $\mathit{score}_{\texttt{interview}}$ (zoekvraag, **interview**) hoeven we niet zelf te berekenen; die krijgen we van Lucene, via de XQuery-functie `ft:score(.)`.

De λ bepaalt dus in hoeverre we de ordening op de `qa`-index dan wel op de `interview`-index moeten baseren. Sigurbjörnsson et al. [4] betoogt dat een λ van 0,6 een stabiele, optimale waarde is. Dat is dan ook de waarde die wij in onze implementatie voor `$lambda` gebruiken.

Figuur 7 toont de webapplicatie, direct nadat op 'xml' is gezocht in de vraag-antwoordparen [7]. Hetgeen daar weergeven wordt is praktisch het resultaat van de XQuery-code in figuur 6, met wat HTML-code (als in het eerste voorbeeld in het artikel van Geert Josten) eromheen, CSS om het geheel op te maken en JavaScript voor de *autocompletion*.

eXist is meer dan een XML-database

Zoals eerder genoemd gebruiken we eXist 1.4 als ons database-managementsysteem. eXist is een XML-database, wat inhoudt dat het een systeem is waar XML in opgeslagen en bevraagd kan worden. Dat bevragen ge-

$$\mathit{score}(\text{zoekvraag}, \texttt{qa}) \equiv \lambda \cdot \mathit{score}_{\texttt{qa}}(\text{zoekvraag}, \texttt{qa}) + (1 - \lambda) \cdot \mathit{score}_{\texttt{interview}}(\text{zoekvraag}, \texttt{interview})$$

Figuur 5: Vergelijking

```
let $lambda := 0.6
let $query := request:get-parameter('q', '')
let $hits :=
  for $interview in collection('/db/sigmod/')//interview[ft:query(., $query)]
    let $qas :=
      for $qa in $interview//qa[ft:query(., $query)]
        let $score := $lambda * ft:score($qa) + (1 - $lambda) * ft:score($interview)
        order by $score descending
        return
          <hit score='{$score}'>
            {$qa}
          </hit>
    order by number($qas[1]/@score) descending
    return
      <hits>
        {subsequence($qas, 1, 3)}
      </hits>
return subsequence($hits, 1, 10)
```

Figuur 6: Enigszins vereenvoudigde XQuery 1.0-code die vergelijking uit figuur 5 implementeert en maximaal drie `qa`*-elementen per* `interview` *terug geeft. De vereenvoudiging behelst het enkel opleveren van* `qa`*-elementen.*

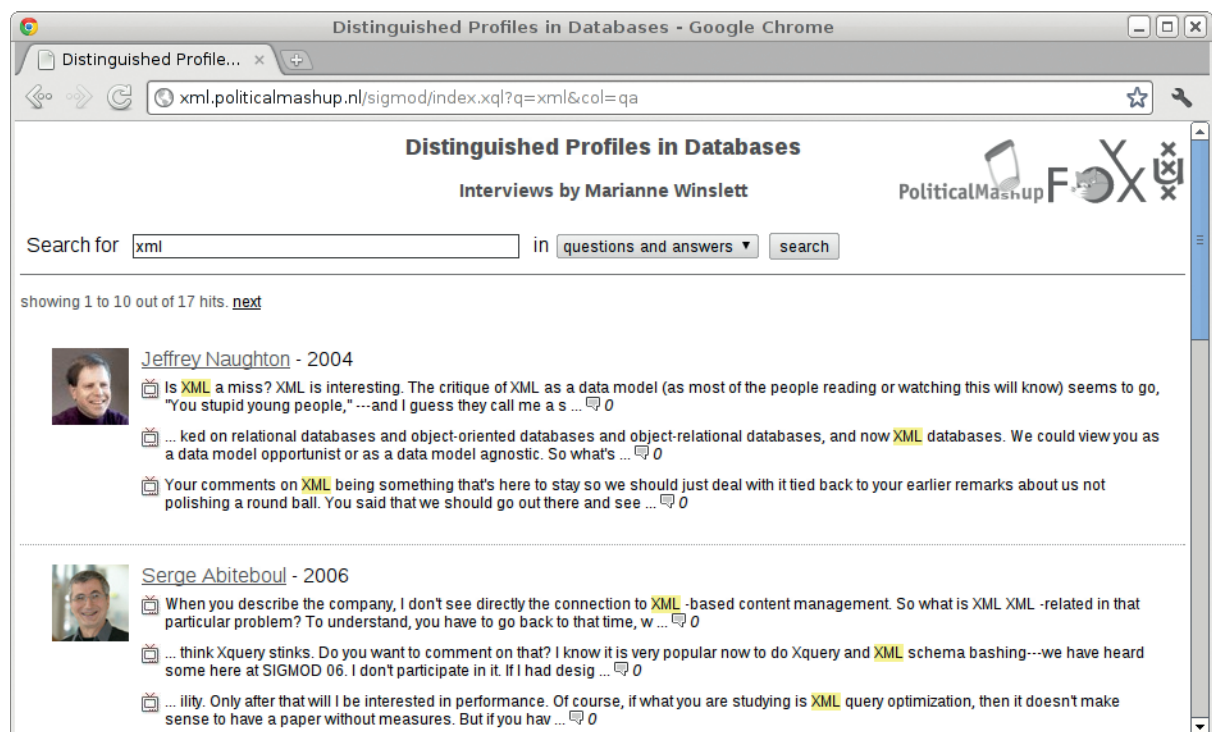
beurt uiteraard met XQuery, de taal die hier speciaal voor ontwikkeld is. Het grote voordeel van het gebruik van een database is dat het systeem in staat is indexen te genereren voor alle data die erin is opgeslagen. Zo bouwt eXist standaard *structurele indexen* op alle opgeslagen XML. Hierdoor wordt het bevragen van de database vele malen sneller dan wanneer het systeem voor iedere bevraging alle data door zou moeten. eXist kent de volgende indexen [8], allemaal bedoeld om de XQuery-rekentijd naar beneden te krijgen:

- **structurele indexen**, om (XPath-)bevragingen zoals `//interview//qa//a` sneller te maken;
- ***range*-indexen**, om `//hits//hit[@score gt 0.5]` sneller te maken;
- ***full text*-indexen**, om `ft:query(., $query)` en `ft:score($hit)` mogelijk te maken;
- ***n*-gram-indexen**, voor het indexeren van *n*-grammen;
- **ruimtelijke indexen**, om geometrische data te indexeren.

eXist als webserver

Naast het efficiënt opslaan van XML kan eXist veel meer. Het heeft een webserver ingebouwd en is zodoende in staat volledige websites te huisvesten. Data (XML uiteraard), de logica (XQuery), views (XSLT) maar ook platte XHTML, afbeeldingen, JavaScript en CSS kunnen direct in de database opgeslagen worden. Dat heeft als groot voordeel dat de hele website op één plek te vinden is. Via het HTTP-protocol ondersteunt eXist onder meer REST, XMLRPC, WebDAV en SOAP en heeft het een uitgebreid mechanisme om url's te herschrijven.

Daarnaast heeft eXist een grote hoeveelheid aan XQuery-extensies die het ontwikkelen in de taal een stuk eenvoudiger maken. Zo is er een module om met HTTP-*requests* te werken en kunnen bijvoorbeeld *query strings*, zoals geïllustreerd in figuur 5, eenvoudig uitgelezen worden met `request:get-parameter('q', '')`.



Figuur 7: Schermafbeelding van de webinterface direct nadat de zoekvraag 'xml' is ingevoerd.

```
declare function local:comment($query, $col, $doc, $start){
  let $commenttext := request:get-parameter('comment', '')
  let $name := request:get-parameter('name', '')
  let $email := request:get-parameter('email', '')
  let $null := if ( $commenttext ne '' and $name ne '' ) then
                 update insert
                   <comment doc='{$doc}'>
                     <start>{$start}</start>
                     <name>{$name}</name>
                     <email>{$email}</email>
                     <time>{util:system-dateTime()}</time>
                     <session>{session:get-id()}</session>
                     <content>{$commenttext}</content>
                     <query>{$query}</query>
                     <col>{$col}</col>
                   </comment>
                 into doc('/db/comments.xml')//comments
               else ()
  return
    <form method='post' >
      <label for='comment'>Comment*</label>
      <textarea name='comment'>{$commenttext}</textarea><br />
      <label for='name'>Name*</label>
      <input type='text' name='name' class='text' value='{$name}' /><br />
      <label for='email'>Email</label>
      <input type='text' name='email' class='text' value='{$email}' /><br />
      <input type='submit' value='send' />
    </form>
};
```

Figuur 8: XQuery 1.0 met eXist-updates om commentaar bij video fragmenten te plaatsen. In de versie die in onze applicatie staat controleren we de waarden van de velden iets uitgebreider voor we het in de database opslaan.

Verder is er een *session*-module om met gebruikersessies te werken, een *json*-module om XML te transformeren naar JSON voor eenvoudige communicatie met AJAX-technologie, een *http client* om het ophalen van bijvoorbeeld webpagina's mogelijk te maken en daarnaast nog ruim dertig modules. [9]

Full Text

Aangezien eXist *full-text*-zoekfunctionaliteit mogelijk maakt met behulp van Lucene, hét open source pakket voor het maken van zoekmachines, is het een zeer geschikt platform geworden om grote hoeveelheden data te ontsluiten. Het is mogelijk om per datacollectie (een bestandsmap in de database) in eXist aan te geven hoe en wat voor indexen er precies op gezet moeten worden. Zo is het mogelijk indexen te specificeren voor elementen met een bepaalde naam of door middel van (XPath-)paden. Wat er dan gebeurt is dat alle `text()`-knopen onder die paden samengebald worden in één document, zodat Lucene ze aankan. Lucene, op zijn beurt, is dan verantwoordelijk voor het opschonen, indexeren en bevragen van die data. En daar is het heel goed in; Lucene is terecht een van de meest gebruikte pakketten om dergelijke taken uit te voeren.

Update

Voor de meeste webapplicaties is het niet voldoende om slechts data te ontsluiten. Een mogelijkheid om data toe te voegen of te *updaten* is een onmisbare functionaliteit van eXist. De huidige implementatie wijkt enigszins af van de W3C XQuery Update Facility 1.0 Recommendation maar zal in een volgende versie gelijk zijn. In onze voorbeeldapplicatie hebben we het mogelijk gemaakt om bij videofragmenten commentaar achter te laten, een minimaal voorbeeld van de code die daar voor nodig is is weergegeven in figuur 8.

Naast `insert` kent eXist `replace`, `value`, `delete` en `rename` acties die gebruikt kunnen worden om de inhoud van de database te wijzigen. Merk op dat injecties (zoals die bij SQL een potentieel gevaar zijn) hier geen rol spelen zolang `util:eval()` niet gebruikt wordt, wat, zoals geïllustreerd, ook niet nodig is.

Referenties

[1] ACM-website: http://www.sigmod/org/publications/interview

[2] Marx, M. (2009). Meteen naar de juiste plek. In *<!ELEMENT, Jaargang 15, Nr 1*, pages 4-7. XML Holland.

[3] Fuhr, N., Kamps, J., Lalmas, M., and Trotman, A. (2007). Focused access to xml documents. In *6th International Workshop of the Initiative for the Evaluation of XML Retrieval, INEX*, volume 4862.

[4] Sigurbjörnsson, B., Kamps, J., and De Rijke, M. (2004). An element-based approach to xml retrieval. In *INEX 2003 Workshop Proceedings*, pages 19-26. Springer.

[5] eXist 1.4: http://exist.sourceforge.net/

[6] Lucene 2.9.2: http://lucene.apache.org/

[7] Vraag-antwoordparen: http://xml.politicalmashup.nl/sigmod/index.xql?q=xml&col=qa

[8] eXist indexen: http://exist.sourceforge.net/indexing.html

[9] eXist modules: http://demo.exist-db.org/exist/functions/

Maarten Marx, oorspronkelijk politiecoloog, is assistent-professor aan de Universiteit van Amsterdam. Daar doet hij onderzoek naar XML en XPath en past dit toe in de context van politieke data. Anne Schuth studeerde af in de Kunstmatige Intelligentie en werkt nu aan de Universiteit van Amsterdam binnen het PoliticalMashup-project met grote hoeveelheden XML en maakt zodoende veel gebruik van XQuery, XPath, XSLT en gerelateerde technieken.