Universiteit Antwerpen
Universitaire Instelling Antwerpen
Departement Wiskunde-Informatica
2002 - 2003

Mappings tussen XML documenten en relationele databases

Roel Vercammen

Proefschrift ingediend tot het behalen van
de graad van Licentiaat in de Informatica

Promotor: Prof. Dr. Jan Paredaens
Co-promotor: Dr. Ir. Geert-Jan Houben

Dankwoord

Het schrijven van een thesis is een veelzijdige taak, waarbij ik me mag gelukkig prijzen dat vele mensen me hierbij hebben willen helpen. Daarom zou ik graag deze mensen bedanken, die met hun hulp een belangrijke bijdrage hebben geleverd aan het succesvol beëindigen van deze thesis.

In de eerste plaats zou ik hier graag mijn promotor, Prof. Dr. Jan Paredaens, willen danken voor zijn steun en toeverlaat Ook mijn begeleiders Jan Hidders, Stijn Dekeyser en Philippe Michiels en de rest van ADReM, zijnde Toon Calders, An De Sitter, Nele Dexters en Hans Van Hauteghem, wil ik van harte bedanken voor hun opbouwende kritieken en de vele gedachtenwisselingen, welke een belangrijke rol hebben gespeeld bij het succesvol voltooien van dit eindwerk.

Verder dank ik Prof. Dan Suciu, van de University of Washington in Seattle, voor zijn zeer gewaardeerde en spontane hulp bij het gedeelte rond de STORED queries.

Ook Mary Fernández, Jérôme Simeon en de rest van het team van Galax worden hartelijk bedankt voor het prachtwerk dat zij hebben geleverd met hun implementatie van Galax. Deze implementatie heeft me namelijk zeer veel geholpen met het leren van XQuery en het uittesten van sommige zaken.

Tot slot zou ik ook nog graag mijn familie en vrienden willen bedanken voor de steun die ze mij hebben gegeven tijdens het maken van deze thesis.

Inhoudsopgave

Lijst van figuren

Inleiding

XML [16] is op het moment van schrijven een de facto standaard op het vlak van uitwisseling van gegevens via het Internet. De grote hoeveelheden XML-documenten vragen echter om efficiënte faciliteiten voor opslag, ondervraging en bewerking. In deze thesis zullen we de eerste twee facetten hiervan behandelen.

Indien we de XML documenten als gewone tekstdocumenten gaan bewaren en deze dan bijgevolg lineair moeten doorzoeken (er is namelijk geen index aanwezig), dan zien we direct dat deze methode niet bepaald efficiënt kan genoemd worden. Er werd dan ook reeds meermaals voorgesteld om XML gegevens in een ander formaat te bewaren en enkel bij de uitwisseling van gegevens met derden het XML formaat te gebruiken. In deze thesis zullen we XML documenten dan ook proberen op te slaan in relationele databases, zodat we een groot deel van het werk dat op het vlak van databases gedurende de laatste 20 jaar gedaan is, niet verloren laten gaan bij de introductie van deze nieuwe standaard.

We kunnen in grote lijnen twee delen onderscheiden voor het gebruik van XML documenten in relationele databases, wat zich ook zal weerspiegelen in de structuur van deze thesis.

Een eerste facet behelst de omzetting van de gegevens zelf, wat we terugvinden in deel I van dit document. De opslag van XML gegevens in een relationele database is niet vanzelfsprekend. Gegevens in een XML document zijn namelijk semi-gestructureerd, wat wil zeggen dat de gegevens in een XML document in zekere mate zelfbeschrijvend zijn (de namen van attributen zitten mee in het document) en dat het document een lagere graad van organisatie heeft dan een relationele database. Hiermee wordt bedoeld dat een schema dat de structuur van de gegevens beschrijft, afwezig kan zijn.

Bijgevolg is er geen garantie dat alle objecten (of informatie-eenheden) dezelfde attributen hebben en dat hetzelfde attribuut in verschillende objecten dezelfde betekenis heeft.

De tweede grote lijn voor het gebruik van relationele database systemen voor XML documenten is het omzetten van de queries die op de XML documenten worden uitgevoerd (bv. in XQuery) om te zetten naar SQL, zoals beschreven in deel II. XQuery is een ondervragingstaal voor XML documenten, net zoals SQL er één is voor relationele databases. XQuery is echter een standaard die nog volop in ontwikkeling is. Bijgevolg worden er ook regelmatig nieuwe working drafts uitgebracht, waarbij de XQuery syntaxis en semantiek sterk kan veranderen. In tegenstelling tot SQL heeft XQuery momenteel enkel faciliteiten om gegevens te ondervragen en dus nog niet om gegevens te updaten. Een andere veel gehoorde tekortkoming is het feit dat XQuery zelf niet kan geparsed worden als een XML document. Dit probleem is echter al grotendeels aangepakt in het project XQueryX [7], waarbij er een XML syntax wordt gedefinieerd voor XQuery. Dit werk is ondermeer nuttig om te bewerkstelligen dat queries door dezelfde parser kunnen verwerkt worden als de gegevens.

Was het niet vanzelfsprekend om semi-gestructureerde gegevens om te zetten naar het relationele model, dan ondervinden we nog een grotere hindernis bij het omzetten van XQuery naar SQL. XQuery bezit namelijk recursieve faciliteiten en ondervragingsmogelijkheden voor meta-data, die doorgaans niet in SQL aanwezig zijn. We zullen in het tweede deel van deze thesis dan ook proberen om aan te duiden wat we nu wel en niet kunnen vertalen.

De structuur van deze thesis is zeer eenvoudig. Voor elk van de twee delen zullen we eerst in een hoofdstuk bestaande technieken bespreken. In het tweede hoofdstuk beschouwen we hoe we deze technieken kunnen verrijken, zodanig dat we de doeleinden van deze thesis beter kunnen nastreven. In het derde hoofdstuk bekijken we tenslotte de kracht en de beperkingen van de voorgestelde mapping techniek. In deze thesis kan u dan ook nieuwe resultaten vinden in het tweede en derde hoofdstuk van elk deel. In de appendices vindt u nog wat uitgewerkte voorbeelden en concrete informatie rond de kleine implementatie (zoals grammatica's e.d.) die bij deze thesis hoort.

Deel I

XML documenten in een relationele database bewaren

Hoofdstuk 1

Bestaande opslagtechnieken

Zoals in de inleiding reeds vermeld werd, zal een eerste belangrijk onderdeel van deze thesis het omzetten van de gegevens van een XML document naar een relationele database omhelzen. Studie rond deze methodes hangt echter nauw samen met de omgekeerde omzetting van gegevens van relationele databases naar XML documenten. We zullen in dit hoofdstuk dan ook beide soorten opslagmethodes bekijken.

Een grote moeilijkheid die we bij deze omzetting ondervinden is het feit dat XML documenten semi-gestructureerde gegevens bevatten, zodat er een conflict onstaat met de sterk gestructureerde relationele databases. Bijgevolg is er geen natuurlijke mapping tussen beide opslagformaten. We beschrijven in wat volgt een aantal gangbare methodes voor de mapping tussen XML documenten en relationele databases. In het volgende hoofdstuk zal dan een combinatie van opslagmethodes van de eerste soort verder worden uitgewerkt.

1.1 Relationale opslagschemas voor XML documenten

Een eerste mogelijkheid om de mapping te verzorgen tussen XML documenten en relationele databases is om bestaande XML documenten om te zetten naar een relationele database. Na deze omzetting moet de gebruiker de indruk krijgen dat hij nog steeds een XML document ondervraagt, ook al zitten de gegevens momenteel in een relationele database. We zullen in wat volgt enkele van deze methodes bespreken.

De methodes die volgen, kunnen we onderverdelen in twee categoriëen, nl. de algoritmes met een vaste vertaling en diegene met een niet-vaste vertaling. Onder vaste vertaling verstaan we dat de resultaten van de vertaling niet beïnvloed of aangepast kunnen worden door diegene die de vertaling opstart. Vertrekkende van een XML document kunnen we dus met één algoritme slechts één mogelijke output genereren. Bij een niet-vaste vertaling kan de persoon die opdracht geeft tot de omzetting wel invloed uitoefenen op het resultaat. Dit kan bijvoorbeeld gebeuren door parameters aan het algoritme mee te geven. Bij elke methode zullen we aangeven of het gaat om een vaste of een niet-vaste vertaling

1.1.1 Model-driven mapping

Vertrekkende van het XML document dat voldoet aan een bepaald model, kunnen we het XML document dat een instantie is van dit model omzetten naar een relationeel schema. Onder model verstaan we een schema (DTD of XML Schema) of een welbepaald vast vormvoorschrift waaraan het XML document moet voldoen. Afhankelijk van het gebruikte model kan de omzetting op twee manieren gebeuren. Deze zijn table-based en object-relational mapping. In beide gevallen zullen we zien dat het hier een vaste vertaling betreft. In appendix C vindt u voorbeelden die deze methode van mappen illustreren.

Table-based mapping

De table-based mapping methode [3] veronderstelt dat we niet werken op willekeurige XML documenten, maar dat de XML documenten aan een bepaalde vorm voldoen die rechtstreeks kan omgezet worden naar een relationele database. De XML documenten moeten er dan uitzien zoals het onderstaande. De mapping zal gebeuren op basis van het XML document zelf en is dus onafhankelijk van zijn XML Schema of DTD.

```
<Database>
  <Table_1>
    <Row>
      <Column_1>...</Column_1>
       ...
      <Column_n>...</Column_n>
```

```
    </Row>
     ...
  </Table_1>
   ...
  <Table_n>
    <Row>
      <Column_1>...</Column_1>
       ...
      <Column_m>...</Column_m>
    </Row>
     ...
  </Table_n>
</Database>
```

Een document dat aan deze vormvoorwaarden voldoet, wordt eenvoudig omgezet. De tabelnamen, kolomnamen en de databasenaam kunnen we bekomen door de elementnamen te gebruiken van het XML document.

Object-relational mapping

Object-relational mapping gaat in tegenstelling tot table-based mapping zich niet baseren op het eigenlijke XML document, maar op zijn schema [16] om het XML document om te zetten naar een relationele database. Dit schema kan de vorm aannemen van een DTD of een XML Schema. In wat volgt wordt het schema voor het XML document eerst omgezet naar een object schema. Vervolgens wordt dit object schema omgezet naar een relationeel schema. In de praktijk gebeurt deze vertaling echter vaak rechtstreeks. Bij de volgende paragrafen horen voorbeelden die de methode illustreren. U kan deze vinden in bijlage C.

Mapping van DTD naar object schema Het mappen van een DTD naar een object schema [3] is gebaseerd op enkele eenvoudige principes. Zo zullen we elementen van het type #PCDATA of CDATA steeds omzetten naar een String, aangezien er verschillende soorten data de inhoud kunnen zijn van elementen met dit type. Elk element dat van een ander type een instantie is, zal omgezet worden naar een klasse. De velden van deze klasse halen we dan uit de typebeschrijving, door wederom het voorgaande principe

te gebruiken. Er kunnen ook veel complexere content models gebruikt worden bij DTD's. Zo zullen we sequenties, keuzes, herhalingen van kinderen, optionele kinderen en subgroepen moeten kunnen omzetten. Herhalingen worden dan bijvoorbeeld omgezet naar arrays en keuzes naar nullable pointers (elementen die tot een keuzemogelijkheid behoren, mogen maar moeten immers geen waarde aannemen).

Sequenties vergen een speciale constructie. Bij een DTD is er immers een volgorde gedefinieerd op de verschillende onderdelen van een element. Dit is niet het geval bij de velden van een klasse in een object schema. Dit kunnen we oplossen door nieuwe klasses te maken voor elk veld van een klasse A. Die klasses zouden dan bestaan uit het originele veld, een getal dat de volgorde aangeeft van dit veld en een verwijzing naar de originele klasse A. De klasse A zou als enige veld slechts een waarde hebben die als primaire sleutel dient.

Tot slot bespreken we nog kort het omzetten van attributen. Attributen van elementen kunnen ofwel enkelwaardig zijn (CDATA, ID, IDREF, NMTOKEN, ENTITY, NOTATION en enumerated) ofwel meerwaardig (IDREFS, NMTOKENS, ENTITIES). Zoals verwacht zullen de enkelwaardige attributen afgebeeld worden op enkelwaardige velden van een klasse en meerwaardige attributen afgebeeld worden op meerwaardige velden (arrays) van een klasse.

Mapping van XML Schema naar object schema XML Schema is een veel krachtigere Data Definition Language (DDL) dan DTD's. De mapping van XML Schema naar object schema [4] biedt dan ook de mogelijkheid om meer primitieve types, zoals gehele en vlottende komma getallen te gebruiken.

De vertaling van de meeste van deze primitieve types is voor de hand liggend. Indien het 'maxOccurs' attribuut een waarde groter dan 1 heeft (of 'unbounded' is), dan zal de vertaling hiervan een array zijn van het type van het element. Een eenvoudig type dat een restrictie is van een basis, wordt afgebeeld op z'n basis en verliest dus in zekere zin een deel van haar betekenis. In de rechtstreekse vertaling naar SQL kunnen we echter wel constraints toevoegen die overeenkomen met de restrictie.

Elementen met een complex type worden vertaald naar een klasse. Deze klasse bevat het resultaat van de omzetting van de kinderen (elementen en attributen). Wanneer een aantal elementen echter tussen choice-tags staan, dan zijn deze elementen nullable (i.e. ze kunnen de waarde NULL aannemen). Als een aantal elementen tussen sequence-tags staan, dan is de

volgorde van deze elementen belangrijk en wordt een constructie analoog aan die van bij de omzetting van DTD's gebruikt.

Mapping van object schema naar relationeel schema Het bekomen object schema moet nog omgezet worden naar een relationeel schema [3, 4]. Een klasse wordt omgezet naar een tabel. De enkelwaardige velden van deze klasse met een primitief type worden omgezet in kolommen in die tabel. Indien er niet alleen enkelwaardige velden zijn met een primtief type, dan is het ook noodzakelijk om een extra kolom ID toe te voegen, die als primaire sleutel dient voor de gemaakte tabel. Enkelwaardige velden die een instantie zijn van een klasse A, zorgen ervoor dat in de tabel die gemaakt werd door omzetting van de klasse B waarvan het veld een instantie is, een foreign key wordt toegevoegd, die verwijst naar de primaire sleutel van de tabel van klasse A. Meerwaardige velden (arrays) van een klasse A worden omgezet naar een nieuwe tabel, waarin een foreign key wordt toegevoegd die verwijst naar de primaire sleutel van de tabel van klasse A.

1.1.2 Edge/Value mapping

De techniek van Edge/Value Mapping [14] is een verzameling van zes methodes, die allen zeer eenvoudig zijn van opzet en die een vaste vertaling bewerkstelligen. Het omzetten van XML gegevens naar tabelvorm gebeurt zuiver op basis van het XML document, zonder hulp van DTD of XML Schema. Het XML document wordt omgezet naar een graaf, waarbij elke knoop een XML Element voorstelt. Verder worden element-subelement relaties weergegeven door de takken tussen twee knopen. Om de volgorde uit het XML document te kunnen mappen, worden de takken die uit een bepaalde knoop vertrekken ook genummerd. De waarden zullen tot slot bladeren zijn in de graaf. De methodes van de Edge/Value Mapping techniek worden ingedeeld volgens methode van mapping van de takken en mapping van de waarden, waarbij er een willekeurige combinatie kan gemaakt worden van de mappingstrategiëen van deze twee categoriëen. Over het algemeen valt de performantie [13] van al deze zes methodes zeer goed mee op het vlak van ondervraging en grootte van de gegevensbank. Enkel de reconstructie van het XML document is opmerkelijk traag.

Mapping van de takken

Edge Approach Deze aanpak is de meest eenvoudige. Alle takken van de XML graaf worden in een enkele tabel bewaard. Het schema van de tabel '`Edge`' ziet er dan als volgt uit:

```
Edge(source, ordinal, name, flag, target)
```

De sleutel van deze methode is het koppel `(source, ordinal)`. Het attribuut '`flag`' geeft het type van de knoop waarin de tak arriveert aan (int, string, ref, ...).

Binary Approach De tweede methode zal alle takken met dezelfde label in 1 tabel groeperen. Dit komt eigenlijk neer op een horizontale partitionering van de tabel '`Edge`' uit de eerste methode. Bijgevolg is de sleutel in dit geval ook het koppel `(source, ordinal)`. Het schema ziet er als volgt uit:

```
B_name(source, ordinal, flag, target)
```

Deze methode kwam als beste van de drie methodes voor het mappen van takken uit een vergelijkend performantieonderzoek naar boven.

Universal Table De laatste methode om de takken te mappen, genereert een enkele tabel '`Universal`' die alle takken zal opslaan. Dit komt overeen met een 'full outer join' van alle binaire tabellen. Het schema ziet er als volgt uit, indien $n_1, \ldots, n_k$ de namen van de labels zijn.

```
Universal(source, ordinal_n1, flag_n1, target_n1, ...,
                  ordinal_nk, flag_nk, target_nk)
```

Een instantie van dit schema zal voor een groot deel uit NULL-waarden bestaan en zal bovendien in grote mate redundantie vertonen. Met andere woorden, het schema is gedenormaliseerd.

Mapping van de waarden

Gescheiden tabellen met de waarden Een eerste manier om waarden te bewaren is om aparte waarde-tabellen te maken voor elk mogelijk datatype. Het schema van zo'n tabel ziet er dan als volgt uit:

```
V_type(vid, value)
```

De kolom '`flag`' van de mapping van de takken zal het mogelijk maken om na te gaan in welke tabel we de gewenste waarde moeten gaan zoeken. Bijgevolg zullen we eerst een query moeten uitvoeren om te weten in welke tabel we de antwoorden moeten zoeken, om vervolgens deze tabel te joinen met de tabel waarin de takken bewaard worden en hierop de eigenlijke query uit te voeren.

Inlining Het duidelijke alternatief voor vorige methode is om de waarden en attributen in dezelfde tabel op te slaan, wat overeenkomt met een outer join van de tabel '`Edge`' en de waarde-tabellen. Bijgevolg is er een aparte kolom nodig voor elk soort data type. Het is duidelijk dat in dit geval de flag-kolom niet meer nodig zal zijn en er een groot aantal NULL-waarden zullen voorkomen. Volgens de performantietesten komt deze methode als beste naar voor, op het vlak van het mappen van de waarden, onder meer omdat er bij queries minder joins nodig zijn.

1.1.3 Virtual Generic Schema

De methode van Virtual Generic Schema [24] verzorgt een vaste vertaling van het XML document via het XML Data Model. Het XML Data Model geeft een XML document weer als een boom, waarbij de knopen elementen, attributen, text of processing instructions zijn. Voor meer informatie over het XML Data Model verwijs ik u naar [16].

Het document wordt bij deze mapping techniek dus omgezet naar de abstracte syntax boom, die overeenkomt met de boom van het XML Data Model. We gaan deze boom proberen te bewaren in verschillende relaties, waarin elk tupel (afhankelijk van de relatie) zal overeenkomen met een knoop of een tak. Deze omzetting gebeurt als volgt (primaire sleutels worden aangeduid met een sterretje):

```
Document(docID*, docURIID, rootElemID)
URI(uriID*, uriValID)
ProcInstr(piID*, piVal1ID, piVal2ID)
QName(qNameID*, qnPrefixID, qnLocalID)
Attribute(attrID*, attrElID, attrNameID, attrValID)
```

```
Element(elID*, elQNameID, elTypeID)
Namespace(nsID*, nsValID, nsURIID)
Comment(commID*, commValID)
Value(valID*, value)
Child(parentID*, childID, childValID, index*)

TransClosure(parentID, childID)
```

Dit schema is een volledig genormaliseerde relationele versie van de hiërarchie van het XML document. De laatste tabel TransClosure is eigenlijk redundant t.o.v. de rest van het schema. Het verbindt de paren van elementen die gelinkt zijn door een voorouder/nakomeling relatie. Het is eigenlijk de transitieve sluiting van de relatie Child die expliciet wordt gemaakt, aangezien anders een aantal XQuery queries in het stricte geval niet kunnen vertaald worden naar SQL zonder gebruik te maken van recursie, waarvoor we bijvoorbeeld SQL-3 nodig zouden hebben. Een voorbeeld van zo'n query is de XPath expressie waarin de self-or-descendant operator (//) gebruikt wordt.

1.1.4 Inlining technieken

De verzameling technieken die hier als inlining technieken [31] worden beschouwd zullen op basis van DTD's een XML-document overbrengen naar een relationele database, met een vaste vertaling. Het basisconcept van de inlining technieken is het op een logische manier inlinen van een aantal elementen en attributen in relaties gemaakt voor andere elementen.

In een eerste stap worden de DTD's vereenvoudigd volgens een aantal eenvoudige regels. Een voorbeeld van zo'n regel is '$e_1*? \rightarrow e_1*$', wat wil zeggen dat we een mogelijk voorkomen van 0 of meer keer het element e_1 hetzelfde is als het 0 of meer keer voorkomen van dat element. Deze regels zullen echter niet alle constraints behouden, maar hiervoor zien we later nog het Constraint Preserving Inlining (CPI) algoritme. Een voorbeeld van een omzettingsregel waarin de constraints niet bewaard worden, is '$e_1|e_2 \rightarrow e_1?, e_2?$'.

Basic Inlining Technique

Deze techniek lost het probleem van fragmentatie op door zoveel mogelijk nakomelingen van een element in één enkele relatie te inlinen. Er wordt echter een relatie gecreëerd voor elk element, omdat een XML document als wortel om het even welk element kan hebben bij een DTD. Bijgevolg zullen er vele en grote relaties of tabellen aangemaakt worden.

Er zijn echter nog twee complicaties, nl. set-valued attributes en recursie. Set-valued attributen zijn attributen die (zoals de naam het zegt) een verzameling als waarde kunnen hebben, zoals bijvoorbeeld een lijst van elementen (cf. '*' en '+' operatoren). We zetten deze om door voor elk van deze attributen een relatie te creëren en vervolgens foreign keys te gebruiken. Onder recursie verstaan we de recursie die in de element beschrijving van een DTD kan voorkomen. Zo kan bijvoorbeeld een element A als één van zijn nakomelingen opnieuw het element A hebben. Om deze recursie te vertalen, kunnen we niet inlinen, omdat dit immers zou impliceren dat er slechts een beperkt niveau van recursie kan zijn. Dit probleem wordt opgelost door eerst enkele concepten te definiëren en vervolgens deze te gebruiken om relaties te maken.

Eerst wordt het concept van de DTD graaf geïntroduceerd, waarbij de knopen overeenkomen met elementen, attributen en operatoren. Elk element komt slechts één keer voor in deze graaf, terwijl attributen en operatoren evenveel voorkomen in de graaf als in de DTD. Cycles in deze graaf geven de aanwezigheid van recursie aan. Een tweede concept dat wordt aangehaald is dat van een element graaf, die als volgt ontstaat. Eerst doen we een 'depth first traversal' van de DTD graaf, startend van de elementknoop waarvoor we relaties aan het construeren zijn. Elke knoop wordt gemarkeerd wanneer hij voor de eerste keer bezocht wordt. Deze markering wordt ongedaan gemaakt vanaf dat alle kinderen zijn bezocht. Wanneer we een niet gemarkeerde knoop tegenkomen in de DTD graaf bij depth first traversal, dan wordt er een nieuwe knoop aangemaakt (met dezelfde naam) in de elementgraaf. Bovendien wordt er ook nog reguliere tak toegevoegd van de meest recent gecreëerde knoop (in de elementgraaf) met dezelfde naam als de ouder in een diepte-eerst zoektocht (DFS) van de huidige DTD knoop, naar de nieuw aangemaakte knoop. Wanneer een poging wordt ondernomen om een gemarkeerde knoop te bezoeken, dan wordt er een terugverwijzingstak toegevoegd aan de elementgraaf van de meest recent toegevoegde knoop naar de meest recent toegevoegde knoop met dezelfde naam als de gemarkeerde DTD knoop.

Het schema dat voor de DTD wordt gecreëerd is de unie van de verzamelingen van relaties die voor elk element werden aangemaakt. Gegeven een elementgraaf, wordt er een relatie gemaakt voor elke wortel van de graaf. Alle nakomelingen van een element worden geinlined in die relatie. Hierop zijn echter twee uitzonderingen, nl. de kinderen van een '*'-knoop (worden gesplitst in afzonderlijke relaties) en elke knoop die een terugverwijzing op zich heeft gericht (hiervoor wordt een afzonderlijke relatie gecreëerd).

Op vlak van performantie valt op te merken dat de Basic Inlining techniek een inlining techniek is die we beter vermijden, omdat deze techniek altijd veel meer relaties zal aanmaken dan de andere twee technieken en bovendien de meeste queries op het XML document gesplitst moeten worden in meer queries dan bij de andere inlining technieken.

Shared Inlining Technique

De shared inlining techniek zal de vorige techniek proberen te verbeteren door ervoor te zorgen dat elementknopen in exact één relatie voorkomen. Relaties worden aangemaakt voor elke knoop uit de DTD graaf, waarin er twee of meer takken aankomen, of waarin geen enkele tak aankomt (omdat ze niet bereikbaar zijn via een andere knoop). Knopen waarin juist één tak aankomt, worden geinlined. Net zoals in het Basic Inlining algoritme worden ook hier voor de kinderen van een '*'-knoop aparte relaties aangemaakt. Tot slot wordt er van de mutueel recursieve elementen, waarbij in elks hiervan juist één tak aankomt, één element gekozen om hiervoor een aparte relatie te maken. Zulke mutueel recursieve elementen kunnen we vinden door te zoeken naar sterk geconnecteerde componenten in de DTD graaf.

Deze techniek zal opvallend minder relaties creëeren dan de Basic Inlining techniek. We moeten echter opmerken dat er meer joins nodig zijn voor het oplossen van een query met de Shared Inlining techniek dan met de Basic Inlining techniek.

Hybrid Inlining Technique

De laatste techniek uit de categorie van inlining technieken zal proberen het beste van beide vorige technieken te combineren. De techniek is grotendeels gebaseerd op de Shared Inlining Technique, maar elementen in wiens knopen meer dan één tak toekomt, die niet recursief zijn en ook niet bereikt worden

door een '*'-knoop worden extra geinlined. Deze techniek is gemiddeld genomen qua performantie even goed als de Shared Inlining Technique.

1.1.5 Constraint Preserving Inlining Algoritme

Alle mappingtechnieken die we tot hiertoe hebben gezien en die gebaseerd zijn op een DTD of XML Schema hebben een grote tekortkoming. Hoewel de structuur door deze algoritmes goed wordt vertaald, hebben ze toch grotendeels de semantiek van het schema (DTD of XML Schema) genegeerd. Het CPI algoritme [18, 19, 25] zal de constraints uit een DTD vertalen naar constraints in het relationele schema.

Het CPI algoritme gebruikt voor het omzetten van de structuur het Hybrid Inlining Algoritme, maar kan eveneens relatief eenvoudig worden omgezet voor een ander algoritme om de structuur te mappen. Alvorens we het Hybrid Inlining algoritme kunnen uitvoeren, moeten we echter de keuze operator ('|') elimineren, terwijl we de semantiek ervan trachten te behouden. Dit doen we door eerst de keuze-operator van binnen naar buiten te migreren, vervolgens de keuze te converteren en tot slot de bekomen DTD te 'flattenen'.

We kunnen in een DTD volgende semantische constraints onderscheiden:

- **Domeinbeperkingen**: Attributen kunnen in een DTD beperkt worden tot een bepaald domein. Indien er een opsomming is van mogelijke waarden, dan kan dit gechecked worden door een SQL statement van de vorm `CREATE DOMAIN ... CHECK (VALUE IN (...))`. In een DTD kan eveneens vermeld worden dat een bepaald attribuut steeds moet voorkomen door dit `#REQUIRED` te maken. Dit kan naar SQL vertaald worden door in het commando van `CREATE TABLE` het overeenkomstige relationele attrbituut de constraint `NOT NULL` ($X \nrightarrow \emptyset$) toe te kennen.

- **Cardinality Constraints**: In een DTD declaratie zijn er slechts vier mogelijke cardinaliteitsrelaties, nl. $[0,1], [1,1], [0,\infty[, [1,\infty[$. Een cardinality constraint kan vertaald worden naar drie constraints in een relationele database. De eerste zegt of een attribuut al dan niet NULL mag zijn. De tweede is de zogenaamde singleton constraint en zegt of al dan niet meer dan één subelement kan voorkomen. De derde constraint zal zeggen of een subelement al dan niet zal moeten voorkomen.

De tweede constraint kan thuisgebracht worden in de categorie van Equality-Generatings Dependencies (EGDs), de derde in de categorie van de Tuple-Generating Dependencies (TGDs).

- **Inclusion Dependencies (IDs)**: Deze constraint is een generalisatie van het concept van referentiële integriteit en verzekert dat de waarden in kolommen van een bepaald fragment ook moeten voorkomen in de kolommen van de andere fragmenten. IDs worden indien mogelijk omgezet door middel van foreign keys, maar dit kan enkel als het attribuut waarnaar verwezen wordt een primaire sleutel is. Indien dit niet het geval is, moeten we de `CHECK`, `ASSERTION`, of `TRIGGERS` faciliteiten van SQL gebruiken.

- **Equality-Generating Dependencies (EGDs)**: Wanneer het elementtype X aan de singleton constraint voldoet t.o.v. zijn subelementtype Y, dan zal als een element x dat een instantie is van type X en bovendien twee subelementen y_1, y_2 van type Y heeft, moeten gelden dat y_1 en y_2 hetzelfde element zijn. Dit wordt genoteerd als '$X \rightarrow Y$'. Voor de vertaling van dit soort EGDs, kunnen we gebruik maken van het keyword `UNIQUE` in SQL.

- **Tuple-Generating Dependencies (TGDs)**: Constraints van deze klasse zorgen ervoor dat in een relationeel model een aantal tupels aanwezig moeten zijn in de tabel en wordt weergegeven door het '$\twoheadrightarrow$'-symbool. Twee belangrijke vormen van TGDs zijn de child constraints en de parent constraints. Child constraints (Parent $\twoheadrightarrow$ Child) zeggen dat elk element van het type Parent minstens één kind van het type Child moet hebben, wat het geval is bij een $[1, 1]$ en $[1, \infty[$ cardinaliteit. Parent constraints (Child $\twoheadrightarrow$ Parent) zeggen dat elk element van het type Child een ouder moeten hebben van het type Parent.

Zoals reeds eerder vermeld werd, bouwt het CPI algoritme voort op de inlining technieken. De DTD graaf wordt echter voor dit algoritme geannoteerd, met volgende informatie: indegree (aantal inkomende takken), type (naam van het type van het element of attribuut), tag (een vlag die aangeeft of de knoop een element of een attribuut is en in het tweede geval of het keywords zoals ID of IDREF bevat, etc.) en status (de 'visited' flag voor DFS). Het CPI algoritme zal aan de hand van deze informatie de bovenstaande constraints eenvoudig kunnen herkennen en deze mee kunnen vertalen naar SQL door uitvoering van een licht gewijzigde vorm van het Hybrid Inlining

algoritme.

1.1.6 STORED

Het STORED (Semistructered TO RElational Data) project [9, 10] bestaat uit twee componenten. Eerst wordt er een speciale querytaal gedefinieerd. In deze taal is het mogelijk om de gegevens van het XML document ofwel in een relationele opslagplaats, ofwel in een overflow graaf te bewaren. De tweede component gaat deze queries automatisch genereren aan de hand van een data mining algoritme.

De overflow graaf is een soort container waarin de XML data zitten die niet in relaties worden bewaard. Deze overflow graaf kan bijvoorbeeld als gewone tagged text opgeslagen worden.

De taal waarin je queries kan formuleren om XML documenten om te zetten naar een relationele database ziet er als volgt uit. De queries om XML data van een relationele opslagplaats te voorzien zijn van de vorm: '`FROM ... STORE ...`'. De queries om gegevens in de overflow graaf te bewaren hebben de vorm: '`FROM ... OVERFLOW ...`'.

Bij het genereren van een opslagschema willen we de gegevens zo efficiënt mogelijk bewaren. Dit komt neer op een kostenoptimalisatie probleem. Naast deze soft-constraint, moet er ook voldaan worden aan hard-constraints. Deze worden gekarakteriseerd door de opslag-generatie parameters, zijnde het maximum aantal tabellen, het maximaal aantal attributen per tabel, de maximale opslagruimte op schijf, de collection size treshold en de minimum support. Het bepalen van een optimale storage mapping M is echter NP-moeilijk in de grootte van de semi-gestructureerde data (en dus niet in de grootte van de query). Daarom wordt er het data mining algoritme van Wang en Li gebruikt als heuristiek.

Het Automatic Storage Generation Algorithm dat hiervoor wordt voorgesteld in [10], gebruikt de opslag-generatie parameters. Voor meer informatie over dit algoritme verwijzen we de lezer naar de paper over STORED [10].

We kunnen ten slotte nog opmerken dat het STORED algoritme het enige van de geziene methode's is die een niet-vaste vertaling van het XML document naar de relationele database impliceert.

1.2 XML views van relationele tabellen definiëren

Het is soms gewenst om in plaats van in een relationele database een opslag proberen te voorzien voor een bestaand XML document, voor een bestaande relationele database een XML view te genereren. Deze moet dan ondervraagd kunnen worden als een XML document, zodat er voor de gebruiker geen verschil zichtbaar is met het ondervragen van een gewoon XML document. In deze sectie zullen we enkele van de methodes bespreken om een XML view te maken voor relationele tabellen.

1.2.1 Flat Translation (FT)

Deze vertaling [20, 25] genereert een view waarbij de namen voor de verschillende elementen overeen komen met die van de kolomnamen en tabelnamen uit de database, vergelijkbaar met de vorm van de XML documenten die als input dienen voor de table-based mapping methode. Deze techniek is dus zeer eenvoudig, maar heeft een grote tekortkoming. Doordat deze vertaling zo eenvoudig is, zal het resultaat van de vertaling een flat XML view zijn en zal er dus van een heel aantal non-flat mogelijkheden van XML geen gebruik gemaakt worden, zoals het weergeven van herhalingen van subelementen door reguliere expressie operatoren ('*', '+'). Bovendien zal de nestingsdiepte van de XML view nooit groter kunnen zijn dan 3 of 4.

1.2.2 Nested Translation (NeT)

Om de tekortkomingen van FT proberen te verhelpen, proberen we in onze XML view nesting te introduceren. De nesting operator die we hiervoor invoeren werkt als volgt: voor een tabel t met een verzameling kolommen C, zal de nesting operator voor een niet-lege kolom $X \in C$ alle tupels die overeenkomen voor de overige kolomen $C-X$ verzamelen in een verzameling. Als we na het nesten vaststellen dat kolom X slechts één verzameling heeft met waarde $\{v\}$, dan zeggen we dat het nesten mislukt is en stellen we dat $\{v\} = v$. Als het nesten niet mislukt is, dan zal kolom X een verzameling met meerdere waarden $\{v_1, \ldots, v_k\}$ zijn, met $k \geq 2$ voor minstens één tupel. In dit geval is het nesten gelukt.

Het NeT algoritme [20, 25] werkt nu ongeveer als volgt. Eerst wordt voor elke tabel t_i de nesting operator herhaaldelijk uitgevoerd, totdat er geen nestings

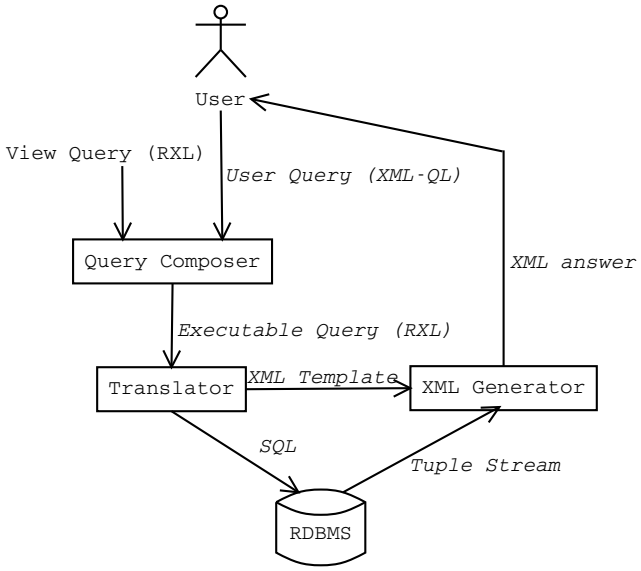
meer lukken. Vervolgens wordt de best geneste tabel gebruikt, op basis van een aantal criteria. Deze tabel zal van de vorm $t'_i(c_1, \ldots, c_{k-1}, c_k, \ldots, c_n)$ zijn, waarbij de nesting is gelukt op de kolommen $\{c_1, \ldots, c_{k-1}\}$. Indien $k = 1$ zal er FT gebruikt worden, anders gaan we het schema als volgt vertalen. Elke kolom c_i waarvoor de nesting gelukt is (dus $1 \leq i \leq k-1$) zullen we vertalen naar het element content model als c_i* indien c_i de waarde NULL mocht aannemen in het relationele schema, anders c_i+. Analoog zullen we voor de kolommen c_i waarvoor de nesting mislukt is (dus $k \leq i \leq n$), vertalen naar het element content model als $c_i?$ wanneer de kolom c_i de waarde NULL mocht aannemen in het relationele schema, anders c_i.

Het NeT algoritme zal data redundantie verminderen en een meer intuïtief schema opleveren, maar NeT beschouwt slechts tabel per tabel en zal daarom problemen kennen bij het vormen van een globaal beeld van het relationeel schema, waarbij verschillende tabellen met elkaar verbonden zijn via allerlei andere afhankelijkheden.

1.2.3 Constraint-Preserving Translation (CoT)

Het CoT algoritme [20, 25] zal het probleem van NeT om de afhankelijkheden tussen de verschillende tabellen te herkennen, oplossen door gebruik te maken van Inclusion Dependencies (IDs) van het relationele schema. Het vinden van een een ID van algemene vorm is moeilijk te realiseren en daarom wordt er enkel maar de meest voorkomende vorm van een ID beschouwd, zijnde een foreign key. We kunnen dus opmerken dat CoT ondanks zijn naam wel degelijk niet alle constraints zal bewaren.

De gedachte achter CoT is om voor twee verschillende tabellen s en t met hun respectievelijke kolomlijsten X en Y het volgende te doen. Veronderstel dat we een foreign key constraint $s[\alpha] \subseteq t[\beta]$ hebben, waarbij $\alpha \subseteq X$ en $\beta \subseteq Y$ en bovendien dat $K_s \subseteq X$ de sleutel is voor s. Als α niet NULL mag zijn, dan zal indien α uniek is er een (1,1) relatie zijn tussen s en t, wat kan weergegeven worden als `<!ELEMENT t (Y, s)>` en indien α niet uniek is, dan zal er een (1,n) relatie zijn tussen s en t (dus `<!ELEMENT t (Y, s+)>`. Als s weergegeven wordt als een subelement van t, dan zal de sleutel van s veranderen van K_s naar $(K_s - \alpha)$, maar de sleutel van t zal onveranderd blijven.



Figuur 1.1: Architectuur van de oringinele SilkRoute methode

1.2.4 SilkRoute

De originele SilkRoute methode [11] gebruikte een intermediaire query taal RXL (Relational to XML Transformation Language) die de XML view definieert. Een algemeen overzicht van de architectuur vindt u in figuur 1.1. Een RXL view query wordt samen met een XML-QL query gecombineerd tot een uitvoerbare RXL query. Deze kan dan vrij eenvoudig vertaald worden naar een SQL query.

Bij het samenstellen van de uitvoerbare RXL query zal de where-clause van de XML-QL query opgesplitst worden in patronen en filters. De construct-clause van de RXL view query wordt omgezet naar een view tree, bestaande uit een globale template en een verzameling datalog regels. De globale template bekomen we door alle templates van alle construct-clauses van de RXL view query samen te voegen. Zo worden knopen van twee verschillende templates samengevoegd als en slechts als ze dezelfde Skolem functie hebben. De datalog regels zijn niet recursief. Hun linkerkant zijn de Skolem functies en hun rechterkant bestaat uit namen van relaties en filters.

De ‘Query Composer’ zal de construct-clause van de RXL view query en de patronen uit de where-clause van de XML-QL query van de gebruiker via

pattern matching omzetten tot een oplossingsrelatie. Deze oplossingsrelatie zal samen met de filters uit de where-clause van de XML-QL query en de construct-clause van die query via het herschrijven van de query worden omgezet tot een uitvoerbare RXL query.

De uitvoerbare RXL query zal door de 'Translator' worden omgezet in enerzijds een SQL query en anderzijds een XML template. Deze template zal dienen om de resultaattupels die door de RDBMS worden teruggegeven om te zetten in de gewenste XML-vorm van het antwoord.

In een recente paper [12] wordt de aangepaste versie van SilkRoute naar XQuery besproken. Aangezien deze paper zeer belangrijk is voor mijn thesis, bespreek ik deze aangepaste methode in hoofdstuk 4.

1.2.5 XPERANTO

XPERANTO [5, 6] (Xml Publishing of Entities, Relationships ANd Typed Objects) maakt het mogelijk om gegevens uit een (object-)relationele database beschikbaar te stellen via XML views. Dit beschikbaar stellen gebeurt door een default XML view, die overeenkomt met het resultaat van de FT techniek. Als je echter niet deze default view wilt gebruiken, maar een andere, kan je een XML query schrijven die deze view definieert in functie van de default view. In tegenstelling tot SilkRoute is hierbij dus geen intermediate query taal (RXL) nodig, die de maker van de views extra zal moeten leren.

Het ondervragen gebeurt door de XML query om te zetten naar het XML Query Graph Model (XQGM). Dit moet het onder andere mogelijk maken om een verandering in de XML query taal eenvoudig aan te kunnen passen, zonder de vertaling naar SQL te moeten aanpassen. Zo kan er in XQGM bijvoorbeeld ook gewerkt worden met driewaardige logica en kan de database ook geüpdatet worden. Het hoofddoel van het gebruik van XQGM (en het herschrijven van queries hierin) is echter de elimininatie van het overbodige construeren van XML elementen en attributen, die enkel in intermediate views voorkomen, maar niet in het uiteindelijke query resultaat.

De bekomen resultaten van de door XPERANTO gegenereerde SQL query moeten nog in het gepaste XML formaat teruggegeven worden. Dit gebeurt in twee fases. In de eerste fase wordt de data gegenereerd die nodig zijn om het resultaat document te construeren. Tijdens de tweede fase zal de data getagged worden, om zo als resultaat het gewenste XML document te bekomen. Voor de eerste fase wordt de query engine van de RDBMS

gebruikt, voor de tweede fase zullen we een 'tagger' gebruiken. Beide fases vormen samen de methode die 'sorted outer union' wordt genoemd.

Tot slot nog even vermelden dat XPERANTO claimt krachtiger te zijn dan SQL, omdat het mogelijk is om meta-data (zoals de namen van tabellen e.d.) te ondervragen, dankzij de "pure XML" filosofie, wat niet mogelijk is met SQL. In XPERANTO is dit echter mogelijk doordat deze data uit de database catalog worden gehaald tijdens het herschrijven van de XQGM representatie en in de nieuwe XQGM representatie wordt opgenomen. Deze resulterende XQGM voorstelling kan dan rechtstreeks vertaald worden naar SQL, aangezien de resultaten van de queries over meta-data reeds zijn ingevuld.

Hoofdstuk 2

Constraint-bewarende mapping

In dit hoofdstuk bouwen we verder op de methodes die we in het vorige hoofdstuk hebben besproken. Door een combinatie van enkele technieken zullen we hier namelijk een vertalingsmethode introduceren die een XML document omzet in een relationele database en daarbij een aantal constraints zal bewaren. De vertaling zelf gebeurt aan de hand van STORED queries. Een overzicht van de verschillende operaties die in dit hoofdstuk worden gedefinieerd, vindt u terug in bijlage F. Er is in het kader van deze thesis ook een prototype gemaakt dat een gedeelte van de hier beschreven mapping uitvoert. Meer info hierover vindt u in bijlage E.

2.1 Inleidende begrippen

In de volgende sectie zullen we enkele nieuwe begrippen introduceren, die we in het vervolg van het hoofdstuk nodig zullen hebben.

2.1.1 STORED queries

In hoofdstuk 1 hebben we STORED reeds kort behandeld. We zullen nu dieper ingaan op de queries die kunnen geformuleerd worden in de query taal en zullen vervolgens ook kort bespreken welke subset van deze taal we zullen gebruiken. De bedoeling van een STORED query is om een gedeelte van een XML document in een relatie te bewaren. De verzameling van STORED

queries moeten dan samen het hele XML document bewaren. Het XML document moet dus met andere woorden door de verzameling STORED queries helemaal omgezet worden in relaties, die we dan kunnen bewaren in een relationele database.

Natuurlijk moet de structuur die de STORED query van het XML document wenst te bewaren overeenkomen met de documentstructuur van het XML document en, indien dit document een DTD bevat, ook compatibel zijn met de DTD. Indien er geen DTD voor handen is kan de verzameling STORED queries beschouwd worden als een soort documentbeschrijving, waaruit we dan een DTD zouden kunnen genereren voor het te bewaren XML document. In wat volgt zullen we ter vereenvoudiging veronderstellen dat we steeds over een DTD beschikken.

In het STORED project is er ook sprake van een overflow graaf, een gegevensstructuur waarin we gegevens van het XML document bewaren die we omwille van performantieredenen niet kunnen of wensen te bewaren in de gewone relaties. Het kan hier gaan om gegevens die zeer weinig voorkomen in het XML document en waarvoor er veel NULL-values zouden nodig zijn in relaties om deze gegevens toch te bewaren. De opslag en ondervraging van de overflow graaf is op dit ogenblik nog steeds een onderzoeksonderwerp en het is zeer moeilijk om hier een zekere performantie te kunnen verkrijgen. Een mogelijke opslagmethode zou zijn om de overflow graph te bewaren volgens de methode van het "Virtual Generic Schema" [24], maar dit lost niets op aan de complexiteit van het ondervragen van gegevens uit de overflow graaf. In [24] is er immers wel een methode besproken om XQuery queries voor het virtual generic schema te vertalen naar SQL, maar de moeilijkheid zit hem vooral in de combinatie van de twee opslagmethodes. Omwille van deze moeilijkheden die gepaard gaan met het bewaren van een overflow graaf, hebben we besloten om de subset van STORED queries te beperken tot die queries zonder overflow graaf.

2.1.2 DTD graaf

Om tijdens de omzetting van het XML document naar een relationele database de documentstructuur van het XML document te bewaren, zullen we een DTD graaf gebruiken.

Definitie 1 *Een DTD graaf is een koppel (V,E), waarbij* $V = V_A \cup V_E \cup V_O$*, de verzameling van knopen en* $E \subseteq V \times V$ *de verzameling van takken. We*

definieren V_E als de verzameling van elementknopen, V_A is de verzameling van attribuutknopen en V_O de verzameling van operatorknopen.

Bovendien spreken we af dat elk element van een DTD slechts één keer in deze graaf voorkomt, terwijl attributen en operatoren evenveel voorkomen in de graaf als in de DTD waarop deze graaf gebaseerd is. We zullen deze graaf herwerken tijdens het zoeken naar constraints. Het resultaat van dat herschrijven zal een geannoteerde DTD graaf zijn.

Definitie 2 *Een geannoteerde DTD graaf (of opslaggraaf) is een koppel (V,E), waarbij $V = V_A \cup V_E$, de verzameling van knopen en $E \subseteq V \times V \times Card$ de verzameling van takken. $Card = \{[0,1], [0,\infty[, [1,1], [1,\infty[\}$ [1] is de verzameling van de verschillende mogelijke cardinaliteiten.*

In het vervolg van de thesistekst zullen we soms over elementknopen spreken, wanneer we eigenlijk elementknopen en attribuutknopen bedoelen.

2.1.3 Elementgraaf

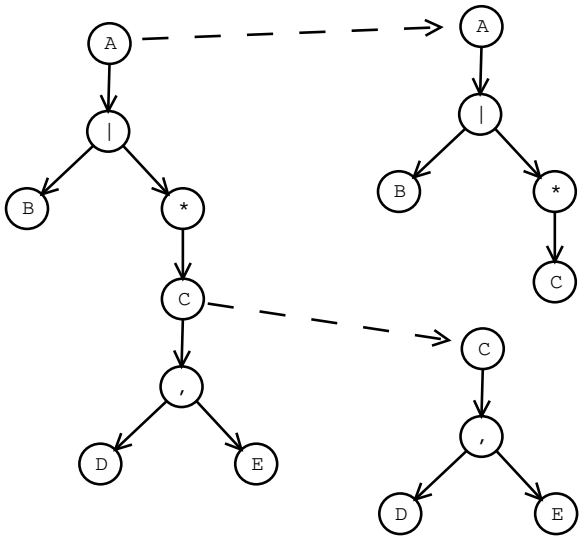
Een elementgraaf is een deelgraaf van een DTD graaf en wordt als volgt gedefinieerd:

Definitie 3 *Een elementgraaf voor een element e in de graaf $G = (V, E)$ is de deelgraaf van G die de unie is van alle paden vanuit e tot een elementknoop, zonder dat er een andere elementknoop in dit pad zit.*

Een gevolg hiervan is dat de elementknopen enkel als blad of als wortel kunnen voorkomen in een elementgraaf en dat alle takken van de elementgraaf in de graaf G ook in de oorspronkelijke graaf G moeten voorkomen. Bovendien zijn alle tussenknopen een operator.

In figuur 2.1 vindt u een voorbeeld DTD graaf, waarbij we de elementgrafen van de elementen `A` en `C` hebben getekend. De elementgrafen voor de andere elementen bestaan slechts uit één knoop, nl. de elementknoop van het element zelf.

[1] Om intervallen voor te stellen, gebruiken we de Belgische notatie. Dit wil zeggen dat indien we $[a, b[$ noteren, dit overeenkomt met het half-open interval $[a, b)$.



Figuur 2.1: DTD graaf met bijhorende elementgrafen voor elementen A en C

2.1.4 Opslagtabel

Tijdens de omzetting (en ook daarna) zullen we iets nodig hebben om te bewaren welke onderdelen van het XML document we op welke plaats bewaren. Hiervoor zullen we de opslagtabel definiëren, samen met een aantal operaties. De opslagtabel is een tabel waarin we in elke rij één attribuut hebben staan van een relatie die we bewaren en waarin de kolommen overeenkomen met het volledige pad, de variabelenaam, de relatie en het veld waarin we het element/attribuut uit het XML document zullen bewaren, samen met nog een kolom waarin een vlag kan staan. Deze vlag kan PK (voor primaire sleutel), FK (voor foreign key) of niets zijn.

Definitie 4 *Een opslagtabel* $\mathcal{T}$ *is gedefinieerd als* $\mathcal{T} \subset Path \times Var \times Table \times Field \times Flag$*, waarbij* $Flag = \{PK, FK, 0\}$.
Volgende operaties zullen gedefinieerd worden op de opslagtabel:
$\rho(n,t) =$ *naam van het attribuut in tabel* t *dat overeenkomt met het pad* n.
$\tau(n) = \{t | t$ *bewaart* n *in één van zijn attributen* $\}$

De operaties ρ en τ zullen we later gebruiken bij de vertalingsprocedure. De verzameling $Path$ komt overeen met de verzameling van alle padnamen. Een

padnaam is een concatenatie van een aantal elementnamen (gescheiden door een puntje), dat overeenkomt met een pad van de wortel tot het element of attribuut dat we willen bewaren, wat we (bijna) rechtstreeks uit de STORED queries kunnen halen.

2.2 Voorbereidende stappen

Alvorens we kunnen gaan zoeken naar constraints in de DTD en de STORED query vertalen naar een SQL statement, zullen we eerst nog een aantal voorbereidende stappen ondernemen. Zo zullen we een DTD en een STORED query uit een bestand moeten kunnen lezen en in de gepaste structuur zetten. Voor een DTD zal dit een graaf zijn, die later herschreven zal worden om zo vervolgens aan de hand van een geannoteerde DTD graaf proberen te zoeken naar constraints. Deze constraints worden dan later in de gegenereerde SQL statements opgenomen. Die SQL statements zullen de nodige relaties en constraints definiëren en de omzetting van het XML document naar een relationele database vervolledigen.

2.2.1 Parsen van DTD

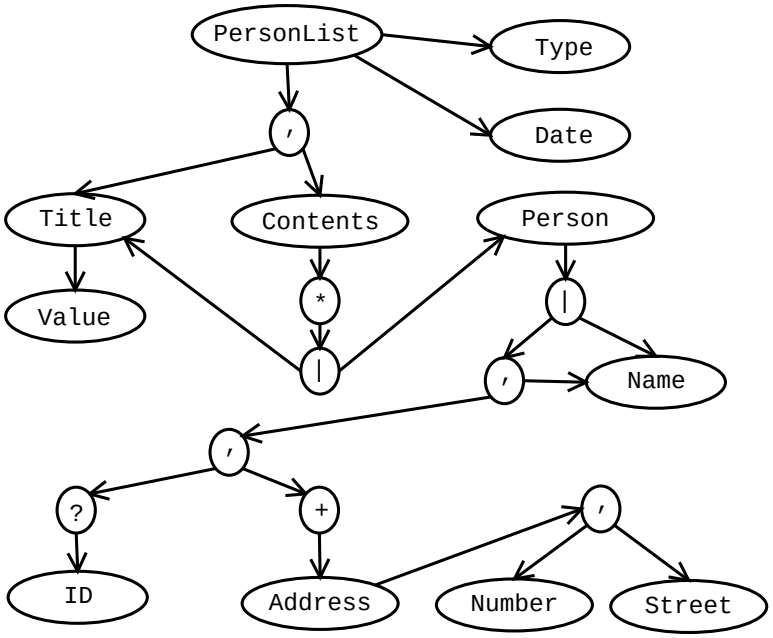
We zullen ons beperken tot een subset van DTD's die voldoende krachtig is om te illustreren wat er mogelijk is met deze techniek. De EBNF syntax van DTD's vindt u in bijlage A. Als u deze grammatica bekijkt, zal het direct opvallen dat `#PCDATA` enkel kan voorkomen als enige inhoud van een element. Dit is bewust gedaan, om de zaken niet te complex te maken. Hierdoor kunnen we echter niet het volgende uitdrukken: `<a>tekst <b>...</b></a>`, maar wel `<a><t>tekst</t> <b>...</b></a>`.

Tijdens het parsen, zullen we een DTD graaf opstellen. Het opstellen van deze graaf is eenvoudig en kan in één pass gebeuren tijdens het parsen. Telkens we een operator, attribuut of element in de DTD tegenkomen, maken we een nieuwe knoop hiervoor aan. Bij het zien van een referentie naar een element zullen we een tak aanmaken van het refererende element naar het gerefereerde element.

Een voorbeeld DTD die voldoet aan de beperkte grammatica vindt u in figuur 2.2. Deze zal na het parsen van de DTD omgezet worden naar de DTD graaf die u vindt in figuur 2.3.

```
<!DOCTYPE PersonList [
  <!ELEMENT PersonList (Title, Contents)>
  <!ELEMENT Title EMPTY>
  <!ELEMENT Contents ((Person|Title)*)>
  <!ELEMENT Person ((Name, Id, Address?)|(Name))>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Id (#PCDATA)>
  <!ELEMENT Address (Number+, Street)>
  <!ELEMENT Number (#PCDATA)>
  <!ELEMENT Street (#PCDATA)>
  <!ATTLIST PersonList Type CDATA #IMPLIED
                       Date CDATA #IMPLIED>
  <!ATTLIST Title Value CDATA #REQUIRED>
]>
```

Figuur 2.2: Voorbeeld DTD



Figuur 2.3: DTD graaf voor het voorbeeld van figuur 2.2

2.2.2 Parsen van STORED query

Naast de DTD moeten we ook nog een aantal STORED queries parsen. Het resultaat van dit parse-proces zal een opslagtabel zijn, zoals beschreven in sectie 2.1.4. Tijdens het parsen van de STORED queries zullen we het '`OPT`' keyword echter negeren. De reden hiervoor is de volgende: het '`OPT`' keyword zegt of een bepaald attribuut in een relationele database al dan niet NULL mag zijn. We gebruiken echter een DTD om deze constraints uit het schema te halen. Indien we het '`OPT`' keyword uit een STORED query wel zouden gebruiken, dan zou het zo mogelijk zijn om een aantal constraints uit de DTD te breken, wat dus tegen onze bedoeling indruist.

Het keyword '`KEY`' zouden we volgens voorgaand principe eigenlijk ook moeten negeren, omdat het namelijk hierdoor mogelijk zou zijn dat een gevalideerd XML document niet kan opgeslagen worden in de relationele database. Indien het keyword '`KEY`' niet voorkomt, zal de sleutel standaard het eerste attribuut van de relatie zijn. Omdat er dus steeds sowieso een sleutel zal zijn, zullen we het keyword '`KEY`' niet negeren tijdens het verwerken van de STORED query. In de meeste STORED queries wordt als sleutel gekozen voor een object identifier, een unieke identifier voor elke knoop uit de documentboom. Indien echter het attribuut dat we primaire sleutel willen maken geen object identifier is, dan zullen we het enkel als sleutel aanvaarden indien we in de DTD terug vinden dat het een attribuut is van het type ID.

De grammatica waaraan de STORED queries moeten voldoen, vindt u in bijlage B. Na het parsen van deze STORED queries is het in principe al mogelijk om SQL queries te genereren voor de opslag van het document, zei het dan wel zonder het bewaren van constraints.

In figuur 2.4 vindt u vier voorbeeld STORED queries, die samen een verliesloze opslag garanderen. De eerste zal een naam en een ID, samen met het eerste adres, bewaren in een relatie. Merk hierbij op dat het element '`Id`' niet als sleutel kan gekozen worden, daar dit enkel mogelijk is bij attributen van het type ID. We houden in deze relatie ook een referentie bij naar '`Contents`', zodat we weten uit welk '`Contents`'-element dit element afkomstig is. De tweede query zal eventuele extra adressen bewaren in een overflow graaf. Vervolgens zal in relatie '`Title`' alle titels die behoren tot de '`Contents`'-elementen bewaard worden. Merk hierbij op dat we naast de '`Value`' ook de object identifier van '`Title`' bewaren. Dit komt doordat we een sleutel nodig hebben voor deze relatie en '`Value`' hier niet geschikt voor zou zijn. De laatste query zal een element in de '`PersonList`' bevatten.

```
FROM PersonList.Contents.Person: $X
{  Name: $N, Id: $I,
   Address: {Number: $R, Street: $S}
}, PersonList.Contents: $C
STORE Person($X, $N, $I, $R, $S, $C)

FROM PersonList.Contents.Person: $X
{ Name: $N, Id: $I, Address: $A, $L: _ }
WHERE $L = Address
OVERFLOW G1($L)

FROM PersonList.Contents.Title: $T
{  Value: $V
}, PersonList.Contents: $C
STORE Title($T, $V, $C)

FROM PersonList.Contents: $C
   PersonList.Title.Value: $T1,
   PersonList.Type: $T2,
   PersonList.Date: $D
STORE PersonList($C, $T1, $T2, $D)
```

Figuur 2.4: Voorbeeld STORED queries

Omwille van de complexiteit die komt zien bij het bewaren van een overflow graaf, hebben we uiteindelijk besloten om dit aspect van STORED queries links te laten liggen (cfr. sectie 2.1.1).

2.3 Domain constraints

De eerste soort constraints die we uit de DTD gaan halen zijn de zogenaamde domain constraints of domeinbeperkingen. Hieronder verstaan we beperkingen die aan het domein (de mogelijke waarden) van een attribuut en/of element worden opgelegd. Dit domein kan beperkt zijn door een keuze tussen een aantal waarden te geven en door het feit of instanties van dit domein al dan niet de NULL-waarde mogen aannemen. We zullen hier enkel domain constraints voor attributen beschouwen, omdat de eerste soort do-

meinbeperkingen niet bestaan voor elementen in een DTD (wel in XML Schema) en we de tweede soort constraints er op een andere manier zullen uithalen.

2.3.1 Enumeration

Indien de verschillende waarden die een attribuut mag aannemen, opgesomd zijn, dan zullen we voor de omzetting naar SQL een domein creëren dat exact die toegelaten waarden bevat.

Definitie 5 *Als $\eta(a)$ de lijst is van alle mogelijke waarden die het attribuut a kan aannemen, dan zal ϵ het domein definiëren als volgt:*
Als $t \in \tau(a)$ dan geldt:
$\epsilon(a,t) =$ `CREATE DOMAIN` $(\rho(a,t))$.“*Domain*″ `(VALUE IN (`$\eta(a)$`))`.
Anders is het resultaat van ϵ de lege string.

De operatie η zal dus uitgevoerd worden voor elke combinatie van een attribuut en een tabel (a,t). Bovendien zullen we bij het omzetten naar SQL erop moeten letten dat deze attributen aan hun domein worden gekoppeld. Deze operaties en alle andere die in dit hoofdstuk nog zullen gedefinieerd worden, kan u terug vinden in een overzicht in bijlage F.

2.3.2 Required of Implied

Voor elk attribuut moet in het DTD steeds gezegd worden of in een instantie van het element steeds dit attribuut moet ingegeven worden of niet. Dit gebeurt met de keywords `#REQUIRED` en `#IMPLIED`. Voor de vertaling van deze constraints definiëren we de vertalingsoperator δ.

Definitie 6 *De vertalingsoperator δ voor domain constraints wordt als volgt gedefinieerd: Als $t \in (\tau(e) \cap \tau(a))$, dan:*

$\delta(e,a,t,required) =$ `CHECK ((`$\rho(e,t)$ `NULL AND` $\rho(a,t)$ `NULL) OR`
`((`$\rho(e,t)$ `NOT NULL AND` $\rho(a,t)$ `NOT NULL))`

$\delta(e,a,t,implied) =$ `CHECK (`$\rho(e,t)$ `NOT NULL OR` $\rho(a,t)$ `NULL)`

Anders is het resultaat van δ de lege string.

We voeren nu δ uit voor elk koppel element-attribuut dat in een attribuutlijst voorkomt, per tabel waarin dit koppel bewaard zal worden. Stel dus

dat we volgende attribuutlijst hebben, waarbij de a's de namen zijn van het attribuut en de r's de waarden *implied* of *required* kunnen aannemen.
`<!ATTLIST e a1 d1 r1 ... an dn rn>`
Dan zullen we voor elk koppel (e, a_i) en voor elke tabel t, de vertaling $\delta(e, a_i, t, r_i)$ doen. Het resultaat hiervan voegen we aan de lijst van constraints voor tabel t toe.

2.4 Constraints van choice operatoren

De choice operatoren kunnen (samen met andere cardinaliteitsoperatoren) vrij ingewikkelde constraints induceren. De eenvoudigste choice constraint die we ons kunnen inbeelden, is diegene die we terugvinden in de documentbeschrijving `(a|b)`. Deze constraint zou vertaald kunnen worden naar volgende SQL constraint:

```
CHECK ((A IS NOT NULL AND B IS NULL) OR
       (A IS NULL AND B IS NOT NULL))
```

Alvorens we de constraints kunnen bepalen die aan de relaties worden opgelegd door de choice operatoren uit de DTD, zullen we een aantal transformaties van de DTD graaf moeten doorvoeren. We zullen per STORED query een beperkte DTD graaf maken door een zekere vorm van "pattern matching" toe te passen en vervolgens op deze DTD grafen enkele herschrijfregels uit te voeren. In de volgende twee subsecties worden deze voorbereidingen uitgewerkt. De derde subsectie zal dan de eigenlijke vertaling van de choice constraints omhelzen.

2.4.1 Pattern Matching tussen DTD graaf en opslagtabel

Nu we de STORED queries en de DTD graaf hebben ingelezen, kunnen we voor elke STORED query gaan bepalen welk gedeelte van het XML document, dat met een deel van de DTD graaf overeenkomt, we zullen bewaren in de relatie die met de STORED query geassocieerd wordt. Het resultaat hiervan is een DTD graaf per STORED query waaruit de overbodige informatie voor de relatie die de STORED query bewaard weggewerkt is. Bovendien zullen de knopen nu geen elementnamen maar padnamen bevatten. De reden hierachter is dat er in STORED queries wordt gezegd welk element waar in welke relatie bewaard wordt aan de hand van de padnamen.

Het algoritme dat deze pattern matching verwezenlijkt, vindt u in figuur 2.5, waarbij de twee parameters g en q respectievelijk de DTD graaf en de

STORED query zijn.

Ter illustratie van dit algoritme zullen we de derde query uit figuur 2.4 matchen met de DTD graaf uit figuur 2.3. Het resultaat hiervan vindt u in figuur 2.6.

2.4.2 Herschrijven van DTD graaf naar geannoteerde DTD graaf

De DTD graaf die we nu per STORED query hebben opgesteld, bevat nog niet genoeg informatie om op een eenvoudige manier constraints uit de DTD te halen. Zo kunnen er ingewikkelde constructies in de reguliere expressies van de documentbeschrijving zijn.

We zullen daarom ook de DTD graaf voor een STORED query herschrijven in drie stappen:

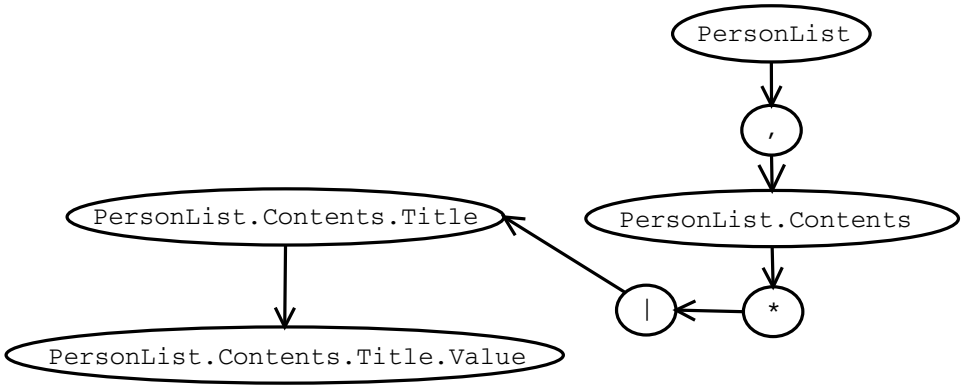
- Wegwerken van haakjes en niet-bewaarde interne elementknopen
- Naar boven verplaatsen van keuze operatoren
- Omzetten naar geanoteerde DTD graaf

Het eerste gedeelte van het herschrijven bestaat uit het wegwerken van de haakjes. Bovendien kunnen we in deze fase ineens de interne elementknopen verwijderen die we niet meer nodig hebben. Het betreft hier de knopen voor elementen of attributen die niet bewaard worden. Als een element zelf (i.e. z'n object identifier) niet bewaard wordt, maar wel (enkele van) z'n attributen, dan moet er minstens één van z'n bewaarde attributen steeds voorkomen wanneer het element zelf voorkomt. Dit komt overeen met het *required* zijn van minstens één van de bewaarde attributen. Indien dit niet het geval zou zijn, zouden we een slechte STORED query opgesteld hebben, omdat gegevens bij het bewaren verloren zouden gaan. Hoe we dit eerste gedeelte doen, vindt u in figuur 2.7.

Een tweede vereenvoudiging die we gaan doorvoeren is het verplaatsen van de keuze operatoren, zodanig dat de twee deelbomen van keuzeknopen zelf steeds vrij zijn van keuzeknopen. Hiervoor gebruiken we het migrateChoice algoritme, zoals dit wordt voorgesteld in [19]. We zullen dit algoritme echter een beetje aanpassen, zodat het in het meer algemene geval kan werken (dus waarbij ook '+' en '?' operatoren kunnen voorkomen in de expressies). Het resultaat van deze uitbreiding vindt u in figuur 2.8.

```
algorithm patternMatch(g, q);

input:
  g: DTD graaf
  q: STORED query
output:
  result: resultaat van het "matchen" van g met q

algorithm:
  result = ({root(g)}, {});
  while (not all pathnames(variables(q)) in result) do
    n = an element of leafs(result) and some of its descendants
        are stored by q;
    s = suffix of n;
    e = rename(elementTree(s, g), s, n);
    foreach (leafnodes(e) as l) do
      e = rename (e, l, n.l);
    result = merge(result, e);

  remove all leafs from result that are not saved by q;

  while (exists(edge(s,t) in result with target not in result))
    remove edge(s,t) from result;
    if (exists(node n in result with no adjecent edges)) then
      remove node(n);
```

Figuur 2.5: Het patternMatch algoritme



Figuur 2.6: Resultaat van het pattern matching algoritme (voorbeeld)

Het derde deel van het hervormen van de DTD graaf naar een geannoteerde DTD graaf bestaat eruit om de knopen die behoren tot een cardinaliteits-operator weg te werken. We zullen de cardinaliteit dan bijhouden in de takken, die dus deze informatie zullen bevatten samen met de padnaam die we in het eerste deel hebben toegevoegd aan de takken. Het omzetten zelf gebeurt door het algoritme 'annotate(e_1, e_2, v)' herhaaldelijk uit te voeren voor elke operatorknoop v samen met de tak naar zijn ouder en de tak naar een kind. Indien een operator meerdere kinderen heeft, dan voeren we dit uit voor elke tak naar een kind van die operatorknoop. Het algoritme zal een nieuwe tak e als output geven. We kunnen na het beschouwen van een operatorknoop, deze verwijderen uit de graaf en de takken die we met het algoritme zijn bekomen aan de graaf toevoegen. Het algoritme 'annotate' zal beslissen aan de hand van de knoopoperator en de cardinaliteit van de twee takken welke de cardinaliteit zal zijn van de nieuwe tak. Deze tak zal de bronknoop uit de eerste tak verbinden met de doelknoop uit de tweede tak. De beslissingen voor de cardinaliteit van de tak die het algoritme 'annotate' zal nemen, vindt u in figuur 2.9. Het algoritme zelf wordt niet in deze tekst uitgewerkt. Dit algoritme is echter wel geïmplementeerd en voor meer informatie verwijzen we u naar de bijlagen.

Na het uitvoeren van annotate zal de DTD graaf als enige operatorknopen slechts '|' en ',' hebben. Bovendien zal de operatorknoop '|' enkel mogelijk zijn als rechtstreeks kind van eeen element en een ','-knoop enkel als rechtstreeks kind van een '|'-knoop. Het resultaat hiervan voor het voorbeeld ziet u op figuur 2.10.

Onze DTD graaf bevat op dit moment alle informatie die we nodig hebben

```
algorithm simplifyGraph(g, v);

input:
  g: DTD graaf
  v: root van de DTD graaf
output:
  result: DTD graaf zonder haakjes en enkel interne elementknopen voor
          elementen of attributen die bewaard worden

algorithm:
  if (type(v) = '()') then
    foreach (children(v) as child) do
      remove edge(v, child);
      foreach(parent(v) as parent) do
        add edge(parent, child);
        remove edge(parent, v);
    remove vertex(v);

  else if (type(v) = Element) then
    if (exists(parent(v)) and (not elementSaved(v))) then
      swapped = false;
      foreach (children(v) as child) do
        if ((type(child) = Attribute) and (not swapped)) then
          if (required(child)) then
            swapped = true;
            newv = child;
      if (swapped) then
        remove edge(v, newv);
        foreach (parent(v) as parent) do
          remove edge(parent, v);
          add edge(parent, newv);
        foreach (children(v) as child) do
          if (child <> newv) then
            remove edge(v, child);
            add edge(newv, child);
        v = newv;
      else
        foreach (children(v) as child) do
          foreach (parent(v) as parent) do
            remove edge(parent, v);
            add edge(parent, child);
        remove vertex(v)

  foreach (children(v) as child) do
    simplifyGraph(g, child);
```

Figuur 2.7: Het simplifyGraph algoritme

```
algorithm migrateChoice(r);

input:
  r: expressie van een operator en z'n kinderen uit de DTD graaf
output:
  result: expressie equivalent met r, maar waarbij de eventuele
          "|" operator vanboven staat

algorithm:
  switch r do
   case r does not contains "|" operator
     return r;

   case r = (r1)*
     migrateChoice (r1) = (a1 | ... | an);
     return (a1*, ..., an*)*;

   case r = (r1)+
     migrateChoice (r1) = (a1 | ... | an);
     return (a1, (a1*, ..., an*)*) | ... |
            (an, (a1*, ..., an*)*);

   case r = (r1)?
     migrateChoice (r1) = (a1 | ... | an);
     return (a1? | ... | an?);

   case r = (r1 | r2)
     migrateChoice (r1) = (a1 | ... | an);
     migrateChoice (r2) = (b1 | ... | bm);
     return (a1 | ... | an | b1 | ... | bm);

   case r = (r1, r2)
     migrateChoice (r1) = (a1 | ... | an);
     migrateChoice (r2) = (b1 | ... | bm);
     return ((a1, b1) | ... | (a1, bm) |
             (a2, b1) | ... | (a2, bm) | ...
             (an, b1) | ... | (an, bm));
```

Figuur 2.8: Het migrateChoice algoritme

,	?	_	+	*
?	?	?	*	*
_	?	_	+	*
+	*	+	+	*
*	*	*	*	*

+	?	_	+	*
?	*	*	*	*
_	*	+	+	*
+	*	+	+	*
*	*	*	*	*

?	?	_	+	*
?	?	?	*	*
_	?	?	*	*
+	*	*	*	*
*	*	*	*	*

*	?	_	+	*
?	*	*	*	*
_	*	*	*	*
+	*	*	*	*
*	*	*	*	*

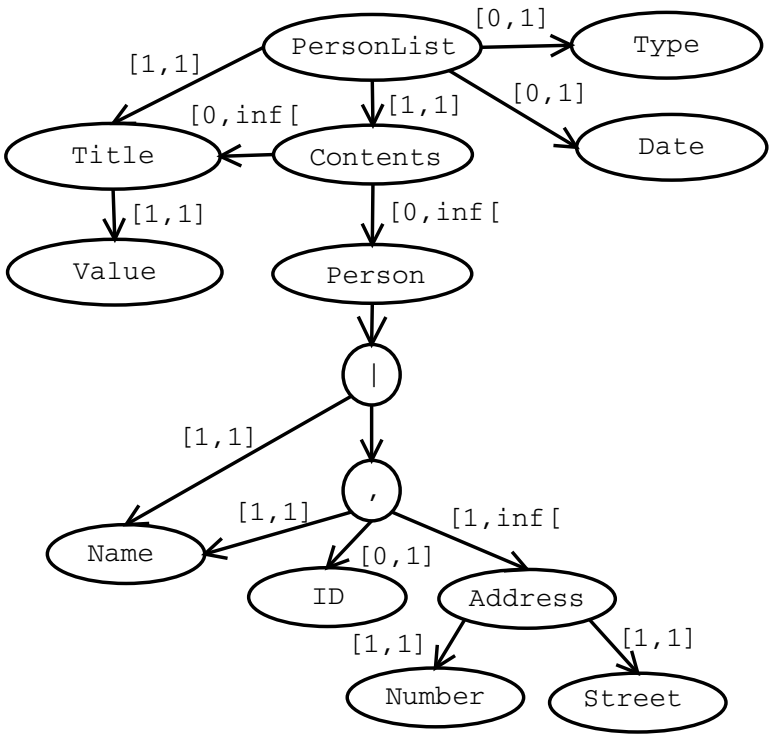
Figuur 2.9: Omzetregels voor annotate()

om de constraints die de DTD legt op ons relationeel schema eenvoudig te gaan zoeken.

2.4.3 Zoeken van choice constraints

Nadat we voorgaande stappen hebben doorlopen, kunnen we nu voor elke relatie de choice constraints bepalen die hierin zullen voorkomen. We gaan in onderstaand algoritme ervan uit dat alle voorgaande vereenvoudigingsstappen reeds gebeurd zijn. We zullen dan ook het algoritme (dat u verder in deze subsectie vindt) uitvoeren op deelbomen (we noemen deze elementgrafen), waarbij enkel de wortel en de bladeren elementknopen zijn en alle andere knopen operatorknopen zijn. Deze operatorknopen kunnen ofwel keuze knopen zijn (als direct kind van de wortel), ofwel sequentieknopen (als kind van de wortel indien er geen keuzeknoop is, anders als kind van de keuzeknoop).

We zullen ook de cardinaliteitsoperatoren '?', '+' en '*' op de takken zetten, in plaats van knopen te voorzien voor deze operatoren. De '+' operator kan echter weggewerkt worden door het toepassen van de eenvoudige herschrijfregel '$a+ == (a, a*)$'. Indien er dan nog een '*' operator voorkomt in de elementgraaf, dan zullen we deze elementgraaf niet beschouwen om choice constraints te bepalen, aangezien we dan zeker weten dat ofwel minstens één element van de sleutel een bestemming moet zijn van de '*'-tak, ofwel de '|'-knoop de bestemming zijn van een '*'-tak. In het eerste geval zullen we dus geen choice constraints binnen deze elementgraaf moeten uit-



Figuur 2.10: DTD graaf (voor het voorbeeld) na het uitvoeren van annotate

drukken. In het tweede geval zal de sleutel minstens alle kinderen van de '|' node moeten bevatten, maar dit zou tot gevolg hebben dat we een set moeten bewaren in één tupel. Bijgevolg zou deze situatie dus als niet geldig beschouwd moeten worden, aangezien de maker van de STORED query een query heeft gemaakt die niet kan resulteren in iets wat we kunnen opslagen in een relationele database. Er worden dus geen choice constraints voor zo'n elementgraaf gegenereerd. Om deze reden kunnen we in het algoritme ons beperken tot de cardinaliteitsoperator '?'.

We kunnen de constraints die opgelegd worden door het gebruik van choice operatoren dan als volgt vinden:

- Bepaal het resultaat van χ voor de elementgraaf.
  $\chi(x_1|...|x_n) = (\sigma(x_1) \times \mu(x_2) \times ... \times \mu(x_n))$
  $+(\mu(x_1) \times \sigma(x_2) \times \mu(x_3) \times ... \times \mu(x_n)) + ...$
  $+(\mu(x_1) \times ... \times \mu(x_{n-1}) \times \sigma(x_n))$

  $\sigma(y_1, ..., y_n) = \alpha(y_1), ..., \alpha(y_n)$
  `(als geen tak met '?' naar (y_1, ..., y_n))`

  $\sigma(y_1, ..., y_n) = (\alpha(y_1), ..., \alpha(y_n)) + (\beta(y_1), ..., \beta(y_n))$
  `(anders)`
  $\mu(y_1, ..., y_n) = \beta(y_1), ..., \beta(y_n)$

  $\alpha(z) = z$
  `(als geen tak met '?' naar z)`

  $\alpha(z) = true$
  `(anders)`

  $\beta(z) = \neg(z)$

- Herschrijf χ dat de vorm $x_1 + ... + x_n$ heeft, zo dat er geen '+'-operator meer voorkomt in $x_i = b_1 \times ... \times b_m$. Het resultaat hiervan is $\chi_1 = \zeta(x_1) + ... + \zeta(x_n)$.

  $\zeta(b_1 \times ... \times b_m) =$
  Als $\exists b_i : b_i = (a_1 + a_2)$
  Dan $\zeta(b_1 \times ... \times b_m) = \zeta(b_1 \times ... \times b_{i-1} \times a_1 \times b_{i+1} \times ... \times b_m) + \zeta(b_1 \times ... \times b_{i-1} \times a_2 \times b_{i+1} \times ... \times b_m)$

Anders $\zeta(b_1 \times ... \times b_m) = b_1 \times ... \times b_m$

- Als $\chi_1 = y_1 + ... + y_n$
  Stel dan $\pi = \gamma(y_1) + ... + \gamma(y_n)$.
  In π kan dan geen '$\times$'-operator meer voorkomen.

  Stel $A = \{a_i | a_i$ is de naam van een attribuut$\}$
  Stel $B = (b_{1,1}, ..., b_{1,m_1}) \times ... \times (b_{n,1}, ..., b_{n,m_n})$
  Als $\exists b_{i,j} : \exists h : (1 \leq h \leq |A|) \wedge (b_{i,j} = a_h)$
  Dan $\gamma(B) = a_h, \gamma(\nu((b_{1,1}, ..., b_{1,m_1}), a_h), ..., \nu((b_{n,1}, ..., b_{n,m_n}), a_h))$
  Anders als $\exists b_{i,j} : \exists h : (1 \leq h \leq |A|) \wedge (b_{i,j} = \neg a_h)$
  Dan $\gamma(B) = \neg a_h, \gamma(\nu((b_{1,1}, ..., b_{1,m_1}), a_h), ..., \nu((b_{n,1}, ..., b_{n,m_n}), a_h))$
  Anders $\gamma(B) = ()$

  De operatie ν zal indien mogelijk 1 keer a_h schrappen uit $(b_{i,1}, ..., b_{i,m_i})$. Indien dit niet mogelijk is, dan zal 1 keer $\neg a_h$ geschrapt worden. Als ook dit niet gaat, schrappen we niets.

- Verwijder *true* uit π.

- Als $\pi = x_1 + ... + x_n$, doe dan voor elke x_i afzonderlijk volgende hernoeming van de atomen: Overloop alle atomen in x_i en hernoem deze naar de naam van het atoom, gevolgd door het getal j voor het j^{de} voorkomen van dit atoom in x_i.

- Vervang in π de literals z door `z IS NOT NULL` en de literals $\neg z$ door `z IS NULL`.

- Vervang vervolgens , door `AND` en + door `OR`

- Zet voor deze uitdrukking "`(`$\rho(root, t)$ `IS NULL) OR`". Hierbij is *root* de wortel van de elementgraaf waarvoor we de choiceconstraint bepalen. Indien dit element namelijk niet voorkomt, dan moet de rest van de expressie niet meer gelden (dus de kinderen moeten niet voorkomen). Bovendien mag de rest van de expressie dan niet meer gelden, dus de bladeren mogen niet voorkomen, maar dit volgt uit onze vertaling van de TGD, zoals die in sectie 2.7 te vinden is.

- Zet nu rond deze uitdrukking `CHECK(...)`.

We zullen eerst trachten de logica die acher deze vertaling schuilt uit te leggen. Intuïtief gezien zegt een constraint '$x_1 \mid \ldots \mid x_n$' dat er juist één van de x_i's het geval is in een instantie van het element met van het type $x_1 \mid \ldots \mid x_n$ en alle andere x_i's niet voorkomen. De operatie σ zegt dus dat een bepaalde x_i het geval is, terwijl de operatie μ juist zegt dat een bepaalde x_i niet voorkomt. Deze operaties worden tot op een lager niveau uitgewerkt, waarbij we opmerken dat indien een 'a?' voorkomt, we niets kunnen en moeten zeggen over het al dan niet voorkomen van het element 'a'. Vervolgens gaan we met de operatie ζ ons tussenresultaat vereenvoudigen, zodat we op een som van producten uitkomen. Een product komt overeen met een soort conjunctie, waarbij we opmerken dat indien er een contradictie voorkomt tussen de verschillende termen van de conjunctie dit wordt opgelost door de operatie γ. Meer bepaald zal de operatie γ het product uitrekenen, met als resultaat een tupel van atomen en/of negaties van atomen. Een atoom is in deze context een element of een attribuut. Merk ook op dat indien minstens één term zegt dat een element moet voorkomen, dit element ook effectief moet voorkomen. Dit volgt uit de definitie van deze operatie. Tot slot kunnen we de bekomen expressie eenvoudig omzetten naar het gewenste SQL statement.

In bijlage D vindt u een volledig uitgewerkt voorbeeld, waarin de hele vertaling van een XML document naar een relationele database wordt uitgewerkt. Indien u dus het choice constraint algoritme eens aan het werk wil zien, kan u daar terecht. Eveneens kan u dit algoritme zelf testen met de implementatie, die in bijlage E wordt beschreven.

2.5 Inclusion Dependencies

Inclusion Dependencies zijn een veralgemening van referentiële integriteit: een inclusion dependency zegt dat de waarden die op een bepaalde plaats in het document voorkomen, ook ergens anders moeten voorkomen, of, om het in het relationele model uit te drukken, waarden die in bepaalde kolommen voorkomen moeten ook voorkomen als waarden in kolommen in andere relaties. In een XML document kunnen we twee soorten inclusion dependencies tegenkomen, nl. diegene die veroorzaakt worden door het gebruik van de object identifiers (en dus eigenlijk slechts een resultaat zijn van de vertaling) en diegene die overeenstemmen met attributen van het type ID, IDREF en IDREFS.

Definitie 7 *Een inclusion dependency tussen twee verzamelingen a en b*

betekent dat alle elementen van a ook elementen moeten zijn van b.
Notatie: $a \subseteq b$.

2.5.1 Object identifiers

Doordat we het XML document opsplitsen in verschillende relaties, hebben we nood aan primaire sleutels. Aangezien we aan de hand van DTD's nooit de inhoud van een element als sleutel kunnen gebruiken voor dat element, zullen we de unieke identifier gebruiken die met de knoop van het element overeenkomt. Indien we ergens anders naar hetzelfde element verwijzen, moeten we natuurlijk dezelfde object identifier gebruiken. Er kunnen echter verschillende complicaties zich hierbij voordoen. Zo is het namelijk mogelijk dat de object identifier in verschillende relaties een primaire sleutel is. We zullen dan echter concluderen dat de totale verzameling van sleutelwaarden de unie is van de verzamelingen met waarden voor de object identifier als primaire sleutel.

Definitie 8 *De operatie* ω *die gegeven een element en een tabel de eventuele inclusion dependency die met dit element geassocieerd wordt als resultaat heeft, wordt als volgt gedefinieerd:*
Stel $R = \tau(e)$ *en* $T = \{p | p \in R \wedge p$ *heeft* e *als primaire sleutel*$\}$.
Dan geldt voor elke $t_i \in (T - R)$*:*
Indien $|T| = 1$ *(en dus* $T = \{p\}$*) dan:*
$\omega(e, t_i) =$ `FOREIGN KEY (`$\rho(e, t_i)$`) REFERENCES` p`.`$\rho(e, p)$
Indien $|T| > 1$ *(en dus* $T = \{p_1, \ldots, p_n\}$*) dan geldt:*
$\omega(e, t_i) =$ `CHECK` $\rho(e, t_i)$ `IN ((SELECT` $\rho(e, p_1)$ `FROM` p_1`) UNION ...`
`UNION (SELECT` $\rho(e, p_n)$ `FROM` p_n`))`

Anders heeft ω *de lege string als resultaat.*

In voorgaande definitie hebben we voor het eerste geval een inclusion dependency van de vorm $t_i \subseteq p$ en in het tweede geval hebben we $t_i \subseteq (p_1 \cup \ldots \cup p_n)$.

2.5.2 Attributen van het type ID/IDREF(S)

In de DTD's zelf komt ook een vorm van inclusion dependendies voor, meer bepaald tussen verschillende attributen. Attributen van het type IDREF(S) verwijzen naar attributen van het type ID. Merk hierbij op dat indien meerdere attributen het type ID hebben, men niet kan zeggen naar welk attribuut

van het type ID een IDREF juist verwijst. Voor de eenvoudigheid van de (constraint) vertaling gaan we hier enkel de vertaling bepalen voor de constraints voor attributen van het type IDREF (dus niet voor IDREFS).

De vertaling van de constraint zullen we doen aan de hand van 2 operaties die we zullen definieren. De eerste operatie zal gegeven een tabel, een SQL statement genereren die alle waarden van instanties van attributen van het type ID die in die tabel bewaard worden als resultaat hebben.

Definitie 9 *De operatie θ zal gegeven een tabel t volgende SQL statement genereren:*
Zij $A = \{a_1, \ldots, a_n\}$, de verzameling van attributen van het type ID die in tabel t bewaard worden en $I = \{i_k \mid (\rho(a_k, t) = i_k) \wedge (a_k \in A)\}$.
Indien $|I| = 1$, dan :
$\theta(t) =$ `(SELECT` i_1 `FROM` t`)`
Indien $|I| > 1$, dan :
$\theta(t) =$ `(SELECT` i_1 `FROM` t`) UNION ... UNION (SELECT` i_n `FROM` t`)`
Anders heeft θ de lege string als resultaat.

Met deze operatie kunnen we dan uiteindelijk de inclusion dependency voor een gegeven attribuut van het type IDREF en een tabel t als volgt uitdrukken:

Definitie 10 *De operatie ι die gegeven een attribuut a van het type IDREF en de naam t van een tabel waarin a bewaard wordt, de constraint voor de relationele database die de inclusion dependency uitdrukt als resultaat heeft, is als volgt gedefinieerd:*
Als T de verzameling is van alle tabellen t_i waarvoor geldt dat $\theta(t_i)$ niet de lege string als resultaat heeft, dan geldt:
$\iota(a, t) =$ `CHECK` $\rho(a, t)$ `IN (`$\theta(t_1)$ `UNION ...UNION` $\theta(t_n)$`)`

2.6 Equality Generating Dependencies

De equality generating dependencies die we zullen beschouwen, zeggen eigenlijk dat een bepaald element slechts 1 kind van een bepaald type kan hebben. Indien dat element toch twee kinderen van dat type zou hebben, dan zouden deze dezelfde zijn.

Definitie 11 *Men spreekt over een Equality Generating Dependency (EGD) als voor een element x van type X geldt dat als twee subelementen y_1, y_2 van type Y zijn, geldt dat $y_1 = y_2$. Men noteert deze afhankelijkheid als $X \rightarrow Y$.*

In een XML document, dat vergezeld is van een DTD, kan men deze constraint terugvinden door een [0,1] of [1,1] cardinaliteit op de tak naar een elementknoop. Bovendien moet van het element waarin de tak aankomt, de object identifier bewaard worden, omdat we anders uitspraken zouden doen over de inhoud van het element in plaats van het element zelf.

We zullen ons in wat volgt beperken tot het omzetten van EGDs binnen één relatie. Het algemene geval kan immers niet bevredigend omgezet worden in SQL, aangezien we moeilijk (in het algemeen) een verband kunnen leggen tussen tupels en elementen en bijgevolg dus moeilijk kunnen zeggen of één element zich nu in één of meerdere tupels manifesteert.

De vertalingsoperator κ zal onderscheid maken tussen het algemene en een specifiek geval. Het specifieke geval wordt efficiënter vertaald, maar in feite is deze vertaalregel overbodig en volstaat dus enkel het algemene geval.

Definitie 12 *De operator κ, die de equality generating dependency uitdrukt voor een gegeven element e in de tabel t, ziet er als volgt uit:*
Als er een pijl van e naar c loopt in de DTD graaf met cardinaliteit [0,1] of [1,1], $t \in \tau(c)$ en van c de object identifier bewaard wordt, dan kunnen er zich twee gevallen voordoen:

- *Ofwel is $\rho(e,t)$ de primaire sleutel in de tabel t. In dit geval kunnen we deze EGD vertalen als volgt:*
  $\kappa(e,c,t) =$ `PRIMARY KEY` $\rho(e,t)$`, UNIQUE` $\rho(c,t)$
- *In het andere geval gebeurt de vertaling als volgt:*

```
κ(e,c,t)  =  CHECK (
                SELECT COUNT(DISTINCT ρ(e,t))
                FROM t
                WHERE ρ(e,t) IS NOT NULL
             ) = (
                SELECT COUNT(DISTINCT (ρ(e,t),ρ(c,t)))
                FROM t
                WHERE ρ(e,t) IS NOT NULL
             ))
```

Anders is het resultaat van κ de lege string.

Dat het tweede geval voor κ geldig is om de EGD uit te drukken, zullen we nu eerst bewijzen.

Bewijs
We moeten bewijzen dat $X \rightarrow Y \Leftrightarrow \#(disinct\ X) = \#(distinct\ (X, Y))$. We zullen hiervoor het bewijs in twee delen opsplitsen:

- $X \rightarrow Y \Leftarrow \#(disinct\ X) = \#(distinct\ (X, Y))$

  Indien we voor de tupels in r een nieuwe relatie f definiëren die het verband legt tussen de X-waarde in een tupel en de (X, Y)-waarde in dat zelfde tupel, dan kunnen we door het Pigeon-Hole principe aantonen dat f een bijectie is als we vinden dat f een surjectie is, aangezien $\#(disinct\ X) = \#(distinct\ (X, Y))$. Het is duidelijk dat f een surjectie is, aangezien er per verschillende X-waarde x in r ook minstens 1 (X, Y) waarde (x, y) in r bestaat. Nu we weten dat f een bijectie is, weten we dat er met elke verschillende X-waarde juist 1 vercshillende (X, Y)-waarde overeenkomt, dus $(((x, y_1) \in r) \wedge ((x, y_2) \in r)) \rightarrow y_1 = y_2$. Dit is dus juist datgene wat de EGD $X \rightarrow Y$ wil zeggen.

- $X \rightarrow Y \Rightarrow \#(disinct\ X) = \#(distinct\ (X, Y))$.

  Stel dat we weten dat de EGD $X \rightarrow Y$ geldt. We zullen aantonen dat per verschillende X-waarde, er juist 1 verschillende (X, Y)-waarde in de relatie r zit.

  - *Minstens 1:*
    Eerst tonen we aan dat per verschillende X-waarde er minstens 1 verschillende (X, Y)-waarde in r zit. Stel dat x een veschillende X-waarde in r is, waarvoor we geen (X, Y)-waarde in r kunnen vinden. Aangezien x in r voorkomt, bestaat er ook een waarde y voor Y die in hetzelfde tupel in r als x zit. Bijgevolg is (x, y) een (X, Y)-waarde in r en aangezien er een waarde is voor (X, Y), is er dus ook minstens 1 verschillende waarde voor (X, Y).

  - *Hoogstens 1:*
    We hadden verondersteld dat de EGD $X \rightarrow Y$ geldt. Als we nu voor een gegeven X waarde, 2 verschillende (X, Y) waarden in de relatie r hebben, dan geldt dat $\exists x : \exists y_1, y_2 : ((x, y_1) \in$

$r) \wedge ((x, y_2) \in r) \wedge (y_1 \neq y_2)$. Dit is echter duidelijk in contradictie met de EGD.

Bijgevolg hebben we nu aangetoond dat er juist 1 verschillende (X, Y)-waarde per verschillende X-waarde in r. Dit heeft tot gevolg dat $\#(distinct\ X) = \#(distinct\ (X, Y))$.

Door de twee richtingen bewezen te hebben, hebben we bewezen dat de tweede regel voor κ geldig is om een EGD uit te drukken.

□

We zullen deze vertaling doen voor alle koppels elementen voor elke tabel, waarbij we wel opmerken dat er niet steeds de `PRIMARY KEY` constraint moet toegevoegd worden als die reeds bestaat. Meer info hierover vindt u in sectie 2.8, waar de generatie van de SQL queries wordt besproken.

2.7 Tuple Generating Dependencies

Een tuple generating dependency (zoals wij die zullen gebruiken) is een afhankelijkheid die ervoor zorgt dat een bepaald element minstens 1 kind heeft.

Definitie 13 *Een Tuple Generating Dependency (TGD) zegt dat als er elementen van type X aanwezig zijn, er ook elementen van type Y moeten aanwezig zijn. Men noteert deze afhankelijkheid als $X \twoheadrightarrow Y$.*

Concreet bestaan er in een XML document twee soorten TGDs. De eerste zijn child constraints, welke zeggen dat indien er pijlen met cardinaliteit [1,1] of [1,∞[gaan van de ouder p naar de kinderen $c_1, \ldots, c_n$, de TGD '$p \twoheadrightarrow \{c_1, \ldots, c_n\}$' geldt. De tweede soort zijn de parent constraints, die zeggen dat elk element van het type c (dat het type is van het kind) een ouder moet hebben van het type p. In een XML document geldt dit laatste echter niet altijd en het is moeilijk om te beslissen wanneer dit al dan niet geldt. Maar aangezien STORED queries ervoor zorgen dat we met padexpressies werken en er steeds '$c \twoheadrightarrow p$' geldt, kunnen we deze parent constraint toch beschouwen bij onze vertaling.

De vertaling van child constraints gebeurt door de operator ξ. We bouwen echter wel de beperking in dat we deze constraints enkel binnen 1 relatie beschouwen. De vertalingsoperator ξ is nu als volgt gedefinieerd:

Definitie 14 *De operator ξ die de mogelijke child constraint van p naar c (in t) uitdrukt, is als volgt gedefinieerd:*
Indien $(p, t, n_1) \in \rho$, $(c, t, n_2) \in \rho$ en er een edge van p naar c is met cardinaliteit 1 of $[1, \infty[$, dan geldt:

- *Ofwel is $\rho(p, t)$ de primaire sleutel in de tabel t. In dit geval kunnen we deze TGD vertalen als volgt:*
  $\xi(p, c, t) = \rho(c, t)$ `IS NOT NULL, PRIMARY KEY` $\rho(p, t)$
- *In het andere geval gebeurt de vertaling als volgt:*
  $\xi(p, c, t) =$ `CHECK ((`$\rho(p, t)$ `IS NULL) OR (`$\rho(c, t)$ `IS NOT NULL))`

Anders heeft ξ de lege string als resultaat.

De tweede vertaling is geldig omdat deze juist zegt dat als voor een bepaald tupel het attribuut dat overeenkomt met p aanwezig is (i.e. niet `NULL`), dan moet ook het attribuut overeenkomend met c veschillend zijn van `NULL`. Om nu alle child constraints voor een tabel te bekomen zullen we op een analoge manier te werk gaan als bij het bepalen van de EGDs in sectie 2.6.

De parent constraints zullen we ook uitsluitend binnen één relatie beschouwen. De operatie λ zal deze constraint naar SQL vertalen als volgt:

Definitie 15 *De operator λ die de mogelijke parent constraint van c naar p (in t) uitdrukt, is als volgt gedefinieerd:*
Indien $(p, t, n_1) \in \rho$, $(c, t, n_2) \in \rho$, weten we dat c een kind is van p. We bepalen λ dan als volgt:

$\lambda(c, p, t) =$ `CHECK ((`$\rho(c, t)$ `IS NULL) OR (`$\rho(p, t)$ `IS NOT NULL))`

Anders heeft λ de lege string als resultaat.

Deze vertaling spreekt voor zich. De constraint zegt immers dat als het kind c voorkomt, de parent p ook moet voorkomen. De logica achter deze vertaling en de werkwijze om de parent constraints te extraheren uit de DTD graaf en de STORED query voor een bepaalde tabel is analoog aan de werkwijze voor child constraints.

2.8 Generatie van de SQL statements

Het omzetten van het XML document naar een relationele database wordt beëindigd met het genereren van SQL statements die deze omzetting doen en het uitvoeren van deze queries in een RDBMS. Deze verzameling queries kunnen we in twee categoriëen opdelen, namelijk die queries die de structuur bepalen van de relationele database en die queries die de gegevens van het XML document invoeren (via '`INSERT`' statements) in de database volgens de door het omzettingsalgoritme bepaalde structuur. We zullen enkel de eerste soort bekijken.

Zoals reeds gezegd kon in principe na het inlezen van de STORED queries de nodige queries gegenereerd worden, maar dan wel zonder constraints te bewaren. Deze statements zijn van de vorm `CREATE TABLE (...)` en elk SQL statement van die vorm zal overeenkomen met juist één STORED query.

Vervolgens zullen we de gegenereerde SQL constraints samenvoegen met de reeds gegenereerde `CREATE TABLE` query van de respectievelijke relatie.

2.8.1 Zonder constraints

Zonder constraints te bewaren, is de vertaling eenvoudig te doen nadat we de STORED queries hebben ingelezen. We hebben op dat moment immers de opslagtabel gegenereerd en moeten nu per STORED query een SQL statement maken die de gevraagde tabel aanmaakt. De vertaling gebeurt als volgt:

- Begin de constraint met "`CREATE TABLE` t `(`", waarbij t de tabel is die we willen aanmaken.
- Voor alle $(a, t, n) \in \rho$, genereer het SQL fragment "n `TEXT NULL`". Scheidt deze fragmenten met een komma.
- Voeg een lijn toe die de primaire sleutel constraint, die impliciet of expliciet uit de STORED query volgt, gaat toevoegen in een statement van de vorm `PRIMARY KEY`.
- Sluit de SQL statement af met "`)`".

2.8.2 Constraints invoegen

Als we een SQL statement hebben voor een tabel t, dan kunnen we de constraints die we in dit hoofdstuk bepaald hebben, invoegen als volgt:

- **Domain Constraints**
  De enumeration constraints voegen we toe door voor alle $(a, t, n) \in \rho$ het resultaat van $\eta(a, t)$ toe te voegen aan de SQL statement. Bovendien vervangen we "n `TEXT NULL`" door "n n`Domain NULL`".

  De constraint die overeenkomt met het required of implied zijn van een attribuut, voegen we toe door voor elk element e met een lijst van attributen a_i (met overeenkomende r_i = required of implied), $\delta(e, a_i, t, r_i)$ uit te voeren. We scheiden de resulterende strings door komma's en zetten ze achteraan in het `CREATE TABLE` statement.

- **Choice Constraints**
  Nadat we pattern matching hebben gedaan tussen de STORED query en de DTD graaf, moeten we voor elke elementgraaf die we in het resultaat vinden, het choice constraint algoritme uitvoeren. Het resultaat hiervan is een aantal choice constraints, die we scheiden door komma's en achteraan in de `CREATE TABLE` query gaan toevoegen.

- **Inclusion Dependencies**
  De inclusion dependencies geïntroduceerd in een tabel t door de object identifiers worden in de SQL statement gezet door voor elk element e, waarvoor $(e, t, n) \in \rho$, het resultaat van $\omega(e, t')$ (voor elke tabel t') te bepalen en deze resultaten samen te voegen.

  Daarnaast zal ook $\iota(a, t)$ voor elke a waarvoor $(a, t, n) \in \rho$ uitgevoerd worden en de resultaten hiervan geconcateneerd (waarbij natuurlijk de diverse constraints gescheiden worden door komma's), zodat we ook de constraints die veroorzaakt worden door IDREF vertaald hebben.

- **Equality Generating Dependencies**
  Deze constraints zullen we toevoegen door voor de tabel t en alle combinaties elementen (e, c), het resultaat van $\kappa(e, c, t)$ te bepalen. Indien dit iets anders oplevert dan de lege string, dan zullen we dit (samen met een scheidende komma) vanachter toevoegen in het `CREATE TABLE`

statement. Indien in deze query meermaals (n keer) dezelfde `PRIMARY KEY` constraint voorkomt, dan mag dit $n - 1$ keer geschrapt worden.

- **Tuple Generating Dependencies**
  Analoog aan de EGDs zullen we deze constraints voor de tabel t bepalen door voor alle combinaties elementen (p, c), het resultaat van $\xi(p, c, t)$ te bepalen. Indien dit een niet-lege string als resultaat heeft, dan zullen we deze samen met een komma achteraan in het `CREATE TABLE` statement toevoegen. Ook zullen we de redundante voorkomens van de `PRIMARY KEY` constraint verwijderen.

Wanneer we de SQL statements die hiervan het resultaat zijn, uitvoeren, maken we dus een nieuwe relatie aan waarin we een aantal constraints die in dit hoofdstuk behandeld werden, bewaard worden. In het volgende hoofdstuk zullen we de kracht en beperkingen van deze methode om XML documenten te vertalen naar relaties bekijken.

Hoofdstuk 3

Kracht en beperkingen van vertaling

De vertaling die we in het vorige hoofdstuk hebben gedefinieerd heeft enkele tekortkomingen die we reeds vermeld hebben. In dit hoofdstuk zullen we uitzoeken in hoeverre we XML documenten kunnen omzetten in relationele databases en welke constraints we al dan niet kunnen vertalen, om zo de kracht en beperkingen van deze vertaling te ontdekken.

3.1 Gegevens

Eerst bekijken we welke XML documenten we kunnen omzetten naar een relationele database ongeacht het feit of we al dan niet de constraints bewaren. Zoals we zullen zien, is er maar één grote beperking op het input document.

3.1.1 Niet-recursieve documentbeschrijving

Omwille van het feit dat we STORED queries gebruiken om de mapping te definiëren en we het gebruik van een overflow graaf uitsluiten, weten we dat de documentbeschrijving geen cycles mag bevatten [26], of met andere woorden niet recursief mag zijn. Dit wil zeggen een bepaald element zichzelf nooit als nakomeling kan hebben. Volgende documentbeschrijving is dus niet geldig:

```
<!DOCTYPE A[
  <!ELEMENT A(B | C)>
  <!ELEMENT B(A | C)>
  <!ELEMENT C(#PCDATA)>
]>
```

Waarom kunnen we dit nu eigenlijk niet omzetten met STORED queries? Dit komt doordat we in STORED queries een variabele steeds met één volledig pad associeren. Voor de tekst in het element C, kunnen we nu echter oneindig veel paden vinden om hiertoe te komen: `A.C, A.B.C, A.B.A.C, A.B.A.B.C, ...`. Dit zou impliceren dat we oneindig veel variabelen zouden moeten kunnen bewaren en dus een relatie moeten maken met een oneindig aantal attributen. Dit kan echter niet. In STORED kan deze tekortkoming worden opgelost door vanaf een bepaalde nestingsdiepte alles te bewaren in de overflow graaf, waarin we eender welke vorm van semi-gestructureerde data kunnen bewaren, zei het wel op een minder efficiënte manier. Doordat we echter in onze vertaling de overflow graaf niet behandeld hebben, zullen we dus vrede moeten nemen met het feit dat we geen XML documenten met een recursieve documentbeschrijving kunnen omzetten naar een relationele database.

3.1.2 Gegevenstypes

Doordat we gebruik maken van DTD's als schema voor de XML documenten zullen we enkel tekst als type kunnen herkennen voor de gegevens. Dit impliceert echter dat wanneer we vragen gaan stellen op de omgezette gegevens, we geen enkele functie meer, die op getallen gedefinieerd is, kunnen gebruiken. Omdat deze beperking ook gevolgen heeft op de volgende fase van de vertaling, zullen we dit dan ook in hoofdstuk 6 verder bespreken.

3.1.3 Conclusie

Onze vertaling kan alle XML documenten met een niet-recursieve documentbeschrijving omzetten naar een relationele database. Een beperking hierbij is de omzetting van types voor de verschillende attributen. Dit komt omdat we ons baseren op een DTD, waarbij we niet over de nodige kracht bezitten om uit te drukken dat bvb. een bepaald element een vlottend komma getal is.

3.2 Constraints

In hoofdstuk 2 zagen we hoe we een aantal constraints konden bewaren tijdens het omzetten van ons XML document. Spijtig genoeg hebben we hier niet alle constraints kunnen bewaren. In deze sectie zullen we dan ook overlopen wat we wel en wat we niet hebben kunnen bewaren.

3.2.1 Domain Constraints

De eerste constraint die we kunnen vinden in een XML document zijn domain constraints. In het geval van een enumeration is het duidelijk dat we de constraint volledig bewaren. Ook de required en implied constraints worden bewaard door het al dan niet nullable zijn van het attribuut in de resulterende tabel. Deze twee zaken zorgen ervoor dat we de domain constraints voor attributen volledig bewaren bij de omzetting. Voor elementen heeft het enkel zin om te kijken of elementen met enkel een textnode als kind van een bepaald type zijn. Aangezien dit bij het gebruik van DTD's enkel `#PCDATA` kan zijn, ligt de beperking op de tekst die in zo'n veld staat enkel in het feit dat er bvb. geen "`<`" symbool mag in voorkomen (moet vervangen worden door "`<`"). Deze restrictie vertalen we niet expliciet in hoofdstuk 2, maar we veronderstellen dat er een automatische omzetting is van XML entities. Bijgevolg kunnen we besluiten dat alle domain constraints bewaard kunnen worden door onze omzetting.

3.2.2 Constraints van choice operators

We hebben in 2.4 een algoritme opgesteld dat de constraints die op het XML document gelegd worden door een documentbeschrijving, zal vertalen in een SQL statement dat gechecked moet worden. Uit de constructie van het algoritme kunnen we intuïtief begrijpen dat deze vertaling correct is indien alle elementen binnen een term van de "genormaliseerde" expressie (i.e. expressies van de vorm '$x_1 \mid \ldots \mid x_n$', waarbij de x_i's geen '|'-operator bevatten) niet twee of meer keer voorkomen. We zullen dit echter niet bewijzen in deze thesis. Nu is er nog de vraag hoe het zit indien een element wel meer dan één keer voorkomt in dezelfde term van die "genormaliseerde" expressie.

We zullen aan de hand van een voorbeeld ons vertrouwen in de correctheid en compleetheid van dit algoritme vergroten, zonder dit te bewijzen. Stel

dat we de choice constraints in de elementboom voor a wensen om te zetten, waarbij a als volgt gedefinieerd is:

```
<!ELEMENT a((b, c)|(b?, b)|(c, d)?)>
```

Indien we hierop de operatie χ uitvoeren, bekomen we het de expressie $((b, c) \times (\neg b, \neg b) \times (\neg c, \neg d)) + ((\neg b, \neg c) \times (true, b) \times (\neg c, \neg d)) + ((\neg b, \neg c) \times (\neg b, \neg b) \times ((c, d) + (\neg c, \neg d)))$.

Met de operatie ζ wordt dit herschreven naar $((b, c) \times (\neg b, \neg b) \times (\neg c, \neg d)) + ((\neg b, \neg c) \times (true, b) \times (\neg c, \neg d)) + ((\neg b, \neg c) \times (\neg b, \neg b) \times (c, d)) + ((\neg b, \neg c) \times (\neg b, \neg b) \times (\neg c, \neg d))$.

Deze expressie wil dus zeggen dat we vier mogelijkheden hebben, allen gescheiden door een '+'-teken. De eerste mogelijkheid wordt via γ herschreven als volgt:

$$\begin{aligned}
\gamma((b, c) \times (\neg b, \neg b) \times (\neg c, \neg d)) &= b, \gamma((c) \times (\neg b) \times (\neg c, \neg d)) \\
&= b, c, \gamma(() \times (\neg b) \times (\neg d)) \\
&= b, c, \neg b, \gamma(() \times () \times (\neg d)) \\
&= b, c, \neg b, \neg d, \gamma(() \times () \times ()) \\
&= b, c, \neg b, \neg d
\end{aligned}$$

Belangrijk hierbij is dat we de negaties pas op het allerlaatste naar buiten gaan brengen. Dit komt omwille van de semantiek van STORED. Stel dat er één element b voorkomt, dan zien we dat het geval $(b?, b)$ van toepassing is. Alhoewel het de tweede b is die we zien, zegt de padexpressie $a.b[1]$ dat dit de eerste b die voorkomt in dit element wordt gestopt. Er is echter geen tweede b element, dus $a.b[2]$ zal leeg zijn. We zullen met andere woorden onafhankelijk van de DTD gaan beslissen welk element we waar gaan bewaren, zuiver op basis van de STORED query. Dit is ook logisch, aangezien we anders op onbeslisbare problemen kunnen stuiten, bijvoorbeeld in het geval $((b?, b)|(b, b?))$.

We werken de hele expressie verder uit door het uitvoeren van γ op de verschillende termen en bekomen het volgende resultaat:
$(b, c, \neg b, \neg d) + (b, \neg c, \neg d) + (c, d, \neg b, \neg b) + (\neg b, \neg b, \neg c, \neg d)$

Het essentiële om nu een correcte vertaling te doen is dat we van links naar rechts nummers gaan toekennen aan de verschillende elementen van de termen. We zullen elk element of negatie ervan nummeren met het aantal voorkomens in hetzelfde element (of negatie) links van het huidige (in de beschouwde term) plus één. Zo wordt voorgaande expressie vertaald naar het volgende:
$(b_1, c_1, \neg b_2, \neg d_1) + (b_1, \neg c_1, \neg d_1) + (c_1, d_1, \neg b_1, \neg b_2) + (\neg b_1, \neg b_2, \neg c_1, \neg d_1)$

Indien we deze expressie even nader bekijken, dan is het eenvoudig om in te zien dat dit de correcte vertaling is van de choice constraint voor de elementgraaf van a.

Aangezien ik ook geen enkel tegenvoorbeeld heb kunnen vinden waaruit blijkt dat de vertaling niet correct is, kunnen we met voldoende vertrouwen zeggen dat we de choice constraints volledig bewaren.

3.2.3 Inclusion Dependencies

In sectie 2.5 hebben we gezien dat we bij het bewaren van een XML document twee soorten inclusion dependencies (INDs) kunnen tegenkomen.

De eerste soort betreft degene die veroorzaakt worden door het gebruik van object identifiers. Deze inclusion dependency dient om ervoor te zorgen dat we het originele XML document op een geldige wijze kunnen reconstrueren vanuit de relationele database. Het spreekt voor zich dat een object identifier, wanneer deze in een bepaalde relatie bewaard wordt, ofwel een primaire sleutel is in die relatie, ofwel geen primaire sleutel hierin is. In 2.5.1 hebben we dan ook beide van deze gevallen beschouwd. Indien de object identifier geen primaire sleutel is, dan moet deze een referentie zijn naar een voorkomen van die object identifier als een primaire sleutel. Het is nu redelijk te veronderstellen dat zo'n object identifier minstens één keer voorkomt als primaire sleutel en minstens één keer voorkomt als een (soort) foreign key. Er is echter niets dat de makers van de STORED queries hiertoe verplicht. Hierdoor kan de gebruiker, weliswaar met veel moeite, ervoor zorgen dat we een constraint genereren die niet geldig is doordat hij de object identifier niet in elke relatie waarin deze een primaire sleutel moet zijn, als primaire sleutel definieert. De fout ligt in dit geval echter bij de gebruiker die de STORED query heeft aangemaakt, waardoor we kunnen besluiten dat we in een normaal geval steeds de inclusion dependencies geïnduceerd door het gebruik van object identifiers correct bewaren.

De inclusion dependencies die voortkomen uit het gebruik van attributen met het type IDREF worden volledig bewaard door het algoritme in 2.5.2. In deze vertaling beschouwen we immers alle mogelijk gevallen voor zo'n voorkomen en bewaren we de semantiek volledig door het feit dat een attribuut van het IDREF ergens moet voorkomen in de unie van alle ID waarden. We hebben in sectie 2.5.2 echter geen IDREFS beschouwd. We zeiden toen dat we dat deden voor de eenvoudigheid, maar kunnen we eigenlijk wel IDREFS vertalen? Aangezien IDREFS het type is van een waarde van een

attribuut in een XML document, zullen STORED queries dit omzetten in één attribuut in een relatie. Dit attribuut kan echter meerdere waarden bevatten en zal dus in praktijk bewaard worden als een opsomming van allemaal waarden die voorkomen in een ID, gescheiden door een spatie of een ander scheidingskarakter. We kunnen echter doorgaans in SQL geen constraint definiëren die zegt dat alle substrings die gescheiden worden door een bepaald scheidingskarakter in een gegeven tekst (waarde van het attribuut overeenkomend met het IDREFS attribuut) moeten voorkomen in een lijst van mogelijke waarden. We zullen dus concluderen dat we in 't algemeen de semantiek van IDREFS niet kunnen bewaren.

De meeste inclusion dependencies worden bewaard in onze vertaling. Enkel deze constraints die voorkomen door het gebruik van IDREFS zullen we meestal links moeten laten liggen omdat we deze doorgaans niet kunnen uitdrukken in SQL.

3.2.4 Equality Generating Dependencies

Uit de constructie van de vertaling in 2.6 volgt onmiddellijk dat we EGD's enkel vertalen die binnen een relatie of tabel zich manifesteren. Bijgevolg zullen we onmiddellijk een heel aantal EGD's links laten liggen. Wanneer we de vertaling van EGD's nader bekijken dan zien we, geholpen door het bewijs dat we daar hebben geleverd, dat binnen één relatie de EGD's volledig bewaard worden.

Het feit dat we deze constraints enkel binnen één relatie beschouwen, legt echter geen beperking aan de kracht van deze vertaling op. Voor alle 2 elementen p en c die bewaard worden in de relationele database en waartussen er een parent-child relatie geldt, moet er immers normaal gezien een tabel bestaan waarin p en c samen bewaard worden, tenzij er een voorouder p' bestaat, die p uniek bepaalt. Enkel in dit geval is het mogelijk om p en c niet in dezelfde tabel te bewaren. In alle andere gevallen is dit nodig omdat we ons oorspronkelijk XML document aan de hand van de relationele data moeten kunnen reconstrueren. Aangezien de maker van de STORED query echter zelf in handen heeft hoe hij zijn relaties opbouwt, kunnen we zeggen dat deze vertaling voldoende krachtig is.

3.2.5 Tuple Generating Dependencies

Zoals we in sectie 2.7 reeds vermeld hebben, kunnen we in een XML document twee soorten TGD's onderscheiden. Voor de child-constraints hebben

we gezien hoe we deze kunnen vertalen binnen één relatie. Wanneer deze constraint echter over meerdere tabellen verdeeld is, is het heel moeilijk om deze constraint in SQL uit te drukken. Ook de tweede soort TGD's hebben we enkel beschouwd binnen één relatie.

Deze beperking tot één relatie legt in beide gevallen geen beperking op aan de kracht van deze vertaling. De reden hiervoor is hetzelfde als die bij de EGD's, namelijk dat de maker van de STORED queries er zelf voor kan zorgen dat alle twee elementen waarvoor een parent-child relatie geldt in minstens één tabel samen voorkomen.

3.2.6 Conclusie

De domain constraints en choice constraints worden volledig bewaard door onze vertaling. De inclusion dependencies kunnen we op IDREFS na ook volledig bewaren. Equality generating dependencies en tuple generating dependencies zullen we enkel vertalen wanneer deze zich binnen één relatie afspelen. Dit is echter voldoende om deze constraints volledig te kunnen bewaren wanneer de STORED queries goed zijn opgesteld.

Deel II

Vertalen van queries in XQuery naar SQL

Hoofdstuk 4

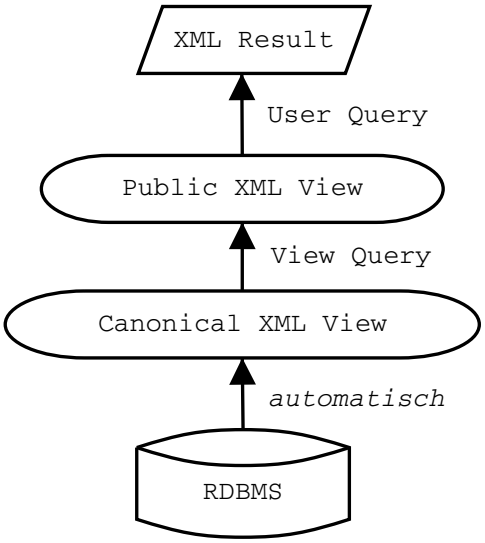
Vertaling van queries in SilkRoute

Het tweede deel van deze thesis bestaat, zoals reeds vermeld, uit het vertalen van XQuery naar SQL, waarvan het antwoord in XML formaat dient gegeven te worden. Er bestaan verschillende mogelijke vertaalstrategieën. Een eerste is XPERANTO [5, 6], waarbij we de grote lijnen hebben uitgelegd in 1.2.5. Een tweede methode is deze die voorgesteld wordt in de paper "Pushing XML Queries inside Relational Databases" [24]. Hierbij worden de gegevens van de XML view in de achterliggende relationele database bewaard volgens het Virtual Generic Schema (cf. 1.1.3). Deze methode kunnen we bijvoorbeeld gebruiken voor de gegevens die in de overflow graaf bewaard worden te vertalen, maar omwille van de complexiteit die bij het bewaren van de overflow graaf komt kijken (bv. beslissen of iets al dan niet in overflow graaf bewaard wordt), laten we deze vertaling naast ons liggen en verwijzen we naar [24] voor meer informatie hierrond.

Andere vertaalmethodes komen we tegen in [14, 29, 30, 31]. In dit hoofdstuk gaan we ons echter focussen op de methode die in kader van het SilkRoute project [12] werd voorgesteld.

4.1 Het SilkRoute project

Het hoofddoel van het SilkRoute project is om gegevens die in een relationele database staan te kunnen publiceren als XML, terwijl de eigenlijke gegevens toch in de relationele database bewaard worden. We zullen meer



Figuur 4.1: De verschillende views in SilkRoute

bepaald een virtual XML view aanbieden aan de gebruiker, dewelke deze kan ondervragen. Met virtual view bedoelen we dat de XML view niet persistent wordt gemaakt en dan ondervraagd, alhoewel we door een eenvoudige XQuery ondervraging wel de virtuele view kunnen bewaren in een file. Omdat het deze view is die ook publiek beschikbaar gesteld wordt, spreken we ook soms van public XML view. De hoofdgedachte die achter deze vertaling zit is om de structuur van het output XML document te scheiden van de berekening die dit document genereert.

4.2 Verschillende XML Views

Zoals reeds vermeld kan de gebruiker een query in XQuery opstellen, over de zogenaamde virtual XML view. Deze view zullen we ook de public XML view noemen, aangezien het deze XML view op de relationele database is, die de gebruikers (= het publiek) voor ogen hebben wanneer ze een query opstellen. Naast deze XML view, bestaan er nog twee andere views, waarvan er één overeenkomt met het eindresultaat van de gebruikersquery. Een illustratie van de relatie tussen de verschillende views vindt u in figuur 4.1. We zullen dit systeem in de volgende paragrafen verder trachten uit te leggen.

Persoon	RRN	Naam	Straat	Nr	Gemeente
	3456789	De Man Van Melle	Koekoekstraat	70	Melle

Hobbies	RRN	Hobby
	3456789	Vissen
	3456789	Tuinieren

Figuur 4.2: Tabellen voor lopend voorbeeld

```
<?xml version="1.0"?>
<CanonicalXMLview>
   <Persoon>
      <Tuple>
         <RRN>3456789</RRN> <Naam>De Man Van Melle</Naam>
         <Straat>Koekoekstraat</Straat> <Nr>70</Nr>
         <Gemeente>Melle</Gemeente>
      </Tuple>
   </Persoon>
   <Hobbies>
      <Tuple><RRN>3456789</RRN><Hobby>Vissen</Hobby></Tuple>
      <Tuple><RRN>3456789</RRN><Hobby>Tuinieren</Hobby></Tuple>
   </Hobbies>
</CanonicalXMLview>
```

Figuur 4.3: Canonical XML view voor het lopend voorbeeld

4.2.1 Canonical XML View

De eerste view is deze die door vele RDBMS systemen automatisch kan gegenereerd worden. We gaan gewoon onze platte relationele database in een platte XML view steken. Dit wil zeggen dat alle bladeren of elementen op eenzelfde nestingsdiepte zijn te vinden. Deze canonical XML view komt dus overeen met de Flat Translation die we in 1.2.1 hebben besproken.

In figuur 4.2 vindt u een voorbeeld relatie, die omgezet wordt naar de canonical XML view die u in figuur 4.3 vindt. We zullen die voorbeeld als lopend voorbeeld gedurende het hele hoofdstuk gebruiken.

In de tabel `Persoon` is het rijksregister nummer (`RRN`) de primaire sleutel; in de tabel `Hobbies` is dit het koppel `(RRN, Hobby)`.

```
<HobbyLijst> {
for $p in $CanonicalView//Persoon/Tuple
return
   <Persoon RRN="{$p/RRN}">
   {$p/Naam}
   <Adres>{$p/Straat}{$p/Nr}</Adres>
   {
      for $h in $CanonicalView//Hobbies/Tuple
      where $h/RRN = $p/RRN
      return $h/Hobby
   }
} </HobbyLijst>
```

Figuur 4.4: View query voor het lopend voorbeeld

4.2.2 Public XML View

De public XML view is die view die de gebruikers voor zich krijgen om te ondervragen. Deze view zal door de database administrator gemaakt worden door een view query uit te voeren op de canonical XML view. Deze view query wordt uitgedrukt in XQuery en zal eenmalig aangemaakt moeten worden. In deze query gebruiken kunnen we een variabele *$CanonicalView*, die de Canonical XML view van de relationele database weergeeft.

De view query voor ons voorbeeld vindt u in figuur 4.4. Merk op dat er in deze view query een join gebeurt van de twee tabellen uit de relationele database. Wanneer we deze query uitvoeren op de canonical XML view, dan krijgen we de public XML view die we zien in figuur 4.5. Merk bovendien op dat in SilkRoute de canonical XML view niet echt gegenereerd wordt, net als de public XML view. Bijgevolg zal niet enkel de gebruikersquery, maar de combinatie hiervan met de view query omgezet worden naar SQL.

4.2.3 Result XML View

De result XML view is de (niet-virtuele) view die de gebruiker als antwoord krijgt op de door hem gestelde XQuery. In deze query heeft de gebruiker de variabele *$PublicView* tot zijn beschikking, die overeenkomt met de public XML view, die dus virtueel is.

We zullen in het lopend voorbeeld de query uit figuur 4.7 loslaten op de

```
<HobbyLijst>
   <Persoon RRN="3456789">
      <Naam>De Man Van Melle</Naam>
      <Adres><Straat>Koekoekstraat</Straat><Nr>70</Nr></Adres>
      <Hobby>Vissen</Hobby>
      <Hobby>Tuinieren</Hobby>
   </Persoon>
</HobbyLijst>
```

Figuur 4.5: Public XML view voor het lopend voorbeeld

```
<NamenLijst Hobby="Vissen"> {
   for $p in $PublicView//Persoon
   where $p/Hobby = "Vissen"
   return <Persoon>{$p/Naam/text()}</Persoon>
} </NamenLijst>
```

Figuur 4.6: Gebruikersquery voor het lopend voorbeeld

public XML view uit figuur 4.5. Deze query stelt dus een lijst van alle personen die vissen als hobby hebben op. Het resultaat van deze query, dat u vindt in figuur 4.6 zal het antwoord zijn wanneer de gebruiker deze query stelt op de public XML view. Hij krijgt dus zelf niets te zien van de achterliggende vertaling van XQuery naar SQL en de omzetting tussen de verschillende views. Hoe deze vertaling nu juist gebeurt, zullen we in de volgende sectie uitleggen.

4.3 View Forests

De manier om de mapping te definiëren van het relationeel model naar XML in SilkRoute zullen view forests zijn. Als we naar figuur 4.1 zien, dan kunnen

```
<NamenLijst Hobby="Vissen">
   <Persoon>De Man Van Melle</Persoon>
</NamenLijst>
```

Figuur 4.7: Result XML view voor het lopend voorbeeld

we bijvoorbeeld zien dat, indien we een view forest hebben die overeenkomt met de canonical view, we deze kunnen "kortsluiten" met de view query om zo een view forest te bekomen die overeenkomt met de public view. Meer bepaald zullen we met view forests de berekening van het resultaat van de gebruikersquery op de public XML view kunnen scheiden van de structuur van dit resultaat.

Een view forest is een graaf (meer bepaald een forest), waarbij de knopen voorzien zijn van een XML label en een SQL fragment. De edges in zo'n view forest representeren een ouder-kind relatie. Het XML label van een knoop is ofwel een elementtag, ofwel een attribuutnaam (die dus begint met een `@`), ofwel een elementair type, zoals string, integer, enz. Het SQL fragment bestaat ofwel uit een `FROM` en `WHERE` clause in het geval dat we een wortel of een interne knoop beschouwen van de view forest. Voor bladeren is het SQL fragment een `SELECT` clause.

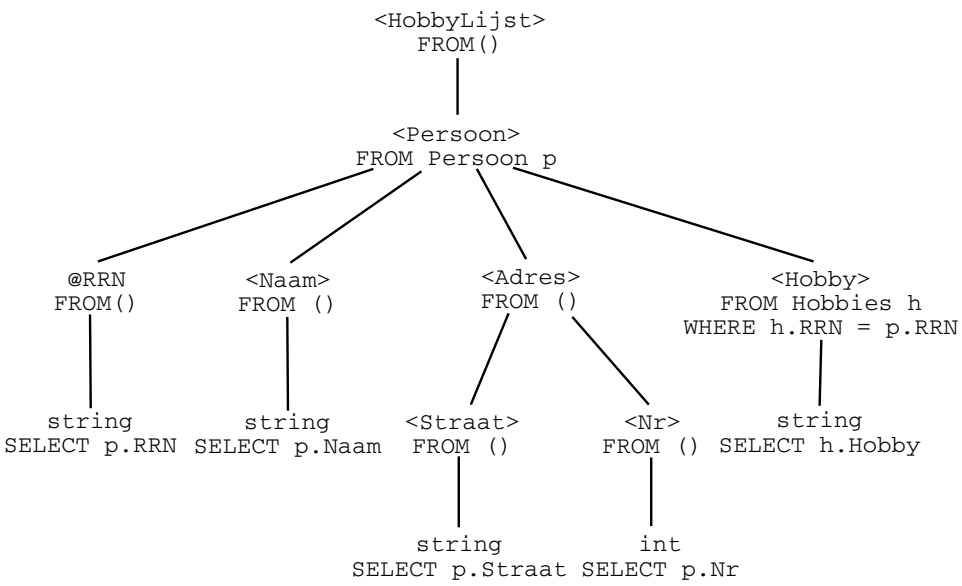
De vorm van de view forest suggereert ook de vorm van het document dat we krijgen indien we deze view forest gaan evalueren. We kunnen dit dus zien als een soort van template waarin we de resultaten van de SQL queries gaan invullen. Merk op dat dit een forest is omwille van het feit dat een XQuery een sequentie van elementen als resultaat kan hebben. In praktijk echter zal een View Forest meestal boom zijn in plaats van een forest, omdat we van een XQuery meestal verwachten dat het resultaat een nieuw XML document is, dat het antwoord is op onze vraag.

Een voorbeeld view forest die kadert in ons lopend voorbeeld vindt u in figuur 4.8. Deze view forest is meer bepaald een public view forest. Wat dit juist is, zullen we later in deze sectie bekijken. Belangrijk om weten is nu dat een view forest rechtstreeks kan omgezet worden naar een XML document door SQL queries te evalueren. Hoe dit gebeurt zullen we nu beschouwen.

4.3.1 Evaluatie van View Forests

Eens we een view forest hebben, is het moeilijkste gedeelte van de berekening van het resulterende XML document reeds gebeurd. We zullen in dat geval namelijk gewoon de nodige SQL fragmenten moeten combineren en uitvoeren, om vervolgens op de correcte plaatsen de resultaten hiervan in te vullen om het XML resultaat te bekomen.

Om de SQL query te bekomen die we met een bepaalde knoop van de view forest associëren, gaan we alle SQL fragmenten beschouwen van de huidige knoop en al zijn voorouders. De resulterende SQL query is de concatenatie



Figuur 4.8: Public View Forest voor het lopend voorbeeld

van de `FROM` clauses, de conjunctie van de `WHERE` clauses. Indien de knoop die we beschouwen een blad is, dan zal ook nog die ene `SELECT` clause die tot het SQL fragment van dat blad behoort, toegevoegd worden aan de SQL query.

Om dit voor het lopend voorbeeld een beetje concreet te maken, zullen we voor de knoop "`<Naam>`" (knoop 1.1.2), het "string" kind (knoop 1.1.2.1) van deze knoop en de knoop (1.1.4.1) die het kind is van de knoop "`<Hobby>`" (1.1.4) de SQL queries bepalen. De resulterende SQL queries voor deze knopen vindt u in figuur 4.9.

Nu we de uit te voeren SQL queries hebben bepaald, moeten we nog kort even bespreken hoe we deze resultaten nu in het template gaan invoeren. Dit doen we door eenvoudigweg voor elke rij in elk antwoord een verschillende XML node construeren. Voor het lopend voorbeeld betekent dit dat er voor elke rij in het resultaat van $C_{1.1.4}$ een XML knoop zal zijn en dat de label hiervan `<Hobby>` is. Er zal bovendien één XML knoop zijn per rij in het resultaat van $C_{1.1.4.1}$, gelabeld met de string '`h.Hobby`'.

Ter illustratie van de ouder-kind relatie beschouwen we de queries $C_{1.1}$ en $C_{1.1.4}$. Hierbij is een rij in het resultaat van $C_{1.1}$ een ouder van een rij in het

$C_{1.1.2}$ =
```
SELECT *
FROM Persoon p
```

$C_{1.1.4.1}$ =
```
SELECT h.Hobby
FROM Persoon p, Hobbies h
WHERE h.RRN = p.RRN
```

$C_{1.1.2.1}$ =
```
SELECT p.Naam
FROM Persoon p
```

Figuur 4.9: Enkele SQL queries voor knopen van public view forest uit het lopend voorbeeld

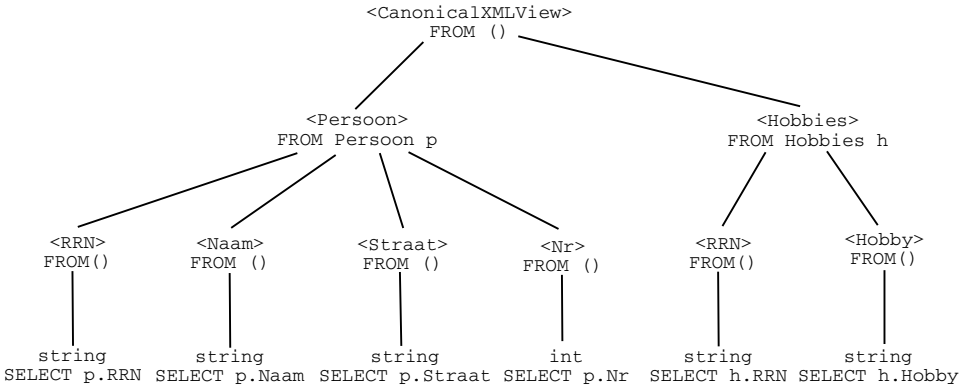
resultaat van $C_{1.1.4}$ als en slechts als deze beide een zelfde binding hebben voor de tupel variabele p. Met andere woorden zal dit in combinatie met de join-statement in de WHERE clause van $C_{1.1.4}$ ervoor zorgen dat de juiste hobbies bij de juiste persoon komen te staan. Wanneer we de resultaten volgens deze methode invullen, krijgen we het resultaat dat we in figuur 4.5 kunnen zien.

4.3.2 Canonical View Forest

Het canonical view forest zal, zoals de naam suggereert, de view forest zijn die evalueert tot de canonical view. Deze view forest kan dus automatisch gegenereerd worden. De rootnode geven we het XML label "`<CanonicalXMLView>`". Het overeenkomende SQL fragment met deze knoop is '`FROM ()`'. De kinderen van deze rootnode (dus de knopen met nestingsdiepte één), komen overeen met de verschillende relaties uit onze relationele database. Het overeenkomende SQL fragment voor deze knopen is dan ook van de vorm '`FROM` *tabelnaam*'. Op nestingsdiepte twee vinden we dan de elementen die overeenkomen met de verschillende attributen van de relaties, waarvan de inhoud op diepte drie wordt bewaard. Op nestingsdiepte (of niveau) twee vinden we dus enkel lege `FROM` clauses. Op niveau drie daarentegen treffen we de bladeren van de view forest aan, dewelke bestaan uit een eenvoudige `SELECT` clause.

Het dient wel opgemerkt te worden dat we de canonical view forest zelf nooit (rechtstreeks) zullen evalueren. In plaats van deze te evalueren, gaan we namelijk later zien hoe we een XQuery kunnen uitvoeren op zo'n view forest om een nieuwe view forest te krijgen, zodat we enkel de uiteindelijke view forest moeten gaan evalueren.

Als we de canonical view forest voor de tabel in figuur 4.2 gaan genereren, dan krijgen we het resultaat dat we in figuur 4.10 zien.



Figuur 4.10: Canonical View Forest voor het lopend voorbeeld

4.3.3 Public View Forest

Als we de view query uitvoeren op de canonical view forest hebben we als resultaat de public view forest, die dus evalueert tot de public view. Hoe die query wordt uitgevoerd op een view forest zullen we in de volgende sectie nog zien. Omdat de view query slechts één keer moet gedefinieerd worden en daarna steeds kan hergebruikt worden, zullen we in feite maar één keer de public view forest dienen te construeren. Voor elke gebruikersquery zullen we dan vertrekken van deze public view query, in plaats van op een naïeve manier te starten vanaf de canonical view forest.

Indien we voor het lopend voorbeeld de query uit figuur 4.4 uitvoeren op de canonical view forest van figuur 4.10, dan zullen we een view forest als resultaat hebben die eruit ziet zoals diegene die we reeds in figuur 4.8 hebben gezien.

4.4 View Forests combineren met XQuery

Nu we weten hoe we een view forest kunnen omzetten naar een aantal SQL queries, en het resultaat hiervan terug invullen in de template die overeenkomt met de structuur van de view forest, zullen we zien hoe we een view forest met een XQuery kunnen combineren tot een nieuwe view forest. Dit combineren van een view forest en een query zal handig zijn om de public view forest te genereren op basis van de canonical view forest en de view

query, alsook voor het combineren van de public view forest en de user query tot een nieuwe view forest, die bij evaluatie het antwoord zal geven op de gebruikersquery.

4.4.1 XQueryCore

De queries die gesteld worden in XQuery zullen we omzetten naar XQuery-Core, een subset die de categorie weergeeft van alle XQuery queries die we met SilkRoute kunnen vertalen naar SQL. Het voordeel van het omzetten van alle queries naar deze subset is dat we de omzetting van XQuery naar SQL kunnen vereenvoudigen naar een omzetting van XQueryCore naar SQL, zonder aan kracht in te boeten.

De XQueryCore zoals die in SilkRoute gebruikt wordt, is een subset van de XQuery Core [8], zoals deze door het W3C werd gedefinieerd. Dit wil dus zeggen dat onze queries die we kunnen stellen op de XML view beperkt zijn in kracht. De eerste beperking (in SilkRoute's XQueryCore) is dat we geen recursieve functies en operatoren (vb.: previous sibling (<<), next sibling (>>), is, isnot) kunnen gebruiken [12] in onze gebruikers- of view queries. We hebben echter nog geen sluitende cijfers gevonden over de hoeveelheid queries die deze operatoren gebruikt. Indien blijkt dat deze operatoren niet vaak gebruikt worden, dan mogen we concluderen dat dit in realiteit geen al te grote beperking is. De andere beperking zegt dat er geen functies die afhangen van de XML document volgorde mogen gebruikt worden in de gestelde XQuery queries.

We zullen in wat volgt een vertaling van XQuery naar SQL definiëren die gebaseerd is op deze notie van XQueryCore.

4.4.2 View Forest Composition Algorithm

Een compositie uitvoeren van een view forest en een XQuery resulteert in een nieuwe view forest. Hiermee hebben we ineens een methode ter beschikking om XQuery om te zetten naar SQL. Deze omzetting, die beschreven wordt in [12], is recursief gedefinieerd op de structuur van XQueryCore. In feite wordt de uitvoering van een query (omgezet naar XQueryCore) gesimuleerd, maar op een view forest in plaats van op XML forests. De grootste moeilijkheid zit hem hier dus in het correct genereren van de nieuwe SQL queries voor de resulterende view forest.

Het recursief algoritme VFCA krijgt parameters Env, Q, S mee. Env staat voor environment en is een verzameling van bindingen tussen variabelen en view forests. De parameter (Q) is de query die uitgevoerd dient te worden en S is een SQL fragment. De eerste twee parameters spreken voor zich, maar het nut van S dient eerst uitgelegd te worden. Deze parameter wordt gegenereerd door een `for` clausule en wordt onmiddellijk door de wortel van de overeenkomende `let` clausule geconsumeerd. In alle andere gevallen is S leeg. Ruw gezegd komt het erop neer dat we de hele view forest moeten herhalen voor elk antwoord in S. Voor de details hierrond verwijzen we de lezer naar [12].

In de rest van deze sectie zullen we nu de recursieve definitie van VFCA schetsen, zonder al te diep in details te gaan.

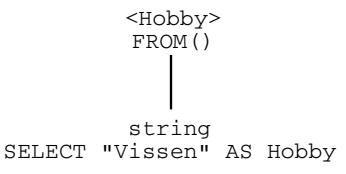
Letterlijke expressies

Indien Q van de vorm [Literal] is, dan zal het resultaat van de query een view forest zijn die uit één knoop bestaat. Deze knoop zal bovendien een blad zijn van de uiteindelijke view forest. De FROM en WHERE clause worden letterlijk overgenomen van S, terwijl de SELECT clause eruitziet als Literal `"AS AtomicValue"`, waarbij `AtomicValue` altijd een atomische waarde bevat.

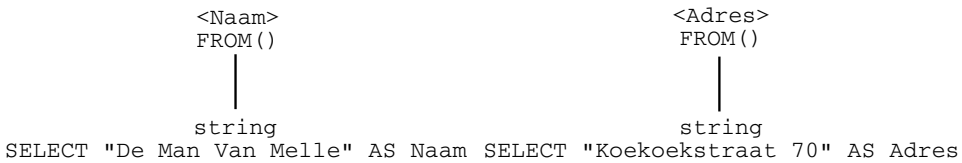
Een voorbeeld dat kadert in ons lopend voorbeeld is het vertalen van de query `[ "Vissen" ]` in context van de public view forest. Als we veronderstellen dat S leeg is, dan is het resultaat een view forest met 1 knoop. Deze knoop zal dan XML label "`string`" bevatten en het SQL fragment "`SELECT "Vissen" AS Hobby`".

Element en attribuut constructoren

Met deze constructoren kunnen we een nieuwe element- of attribuutknoop in de view forest aanmaken. Als Q de vorm heeft van [element $QName\{Expr\}$], dan wil dit zeggen dat de query een nieuwe elementknoop voor het element $QName$ genereert, waarvan de inhoud bepaald wordt door $Expr$. Het spreekt voor zichzelf dat we dan VFCA(Env, $Expr$, ()) gaan uitvoeren om het resultaat van de deelquery $Expr$ te bepalen. We zullen dan een nieuwe elementknoop aanmaken (die de wortel wordt van onze resultaat view forest) en de kinderen van deze knoop zullen de wortels van het resultaat van de deelquery zijn. De werking van attribuut constructoren is analoog aan die van element constructoren.



Figuur 4.11: Resultaat van voorbeeld omzetting van elementconstructor



Figuur 4.12: Resultaat van voorbeeld omzetting van sequentie

Ter illustratie van deze en de vorige omzettingsregel zullen we proberen om de query '`element HobbyLijst { "Vissen" }`' te vertalen in de context van de public view query, alhoewel dat dit nu eigenlijk nog geen rol speelt. We voeren gewoon de zo juist geziene regels uit en bekomen zo het resultaat dat je in figuur 4.11 kan zien. De onderste knoop is dus het resultaat van de deelquery `"Vissen"`. Dit zal een kind worden van onze nieuw gemaakte elementknoop met het XML label `<Hobby>`.

Sequentie expressies

Sequentie expressies van de vorm $(Expr1, Expr2)$ kunnen we vertalen door VFCA uit te voeren op de twee deelexpressies en vervolgens de view forest te construeren die bestaat uit de nodes van het eerste resultaat gevolgd door de nodes in het tweede resultaat.

Voor ons lopend voorbeeld illusteren we dit met volgende query:
`element Naam { "De Man Van Melle" },`
`element Adres { "Koekoekstraat 70" }` De vertaling van deze query levert de view forest uit figuur 4.12 als resultaat op.

Variabelen

Een variabele wordt vertaald door de view forest *vf* waaraan ze in de huidige environment *Env* gebonden is. Elk hierop volgend gebruik van de variabele moet het SQL fragment S bevatten. Daarom zullen we S "joinen" met de SQL fragmenten die geassocieerd worden met *vf*. Voor meer informatie over dit joinen wordt de lezer verwezen naar [12].

We zullen even een illustratie van deze regel doen aan de hand van ons lopend voorbeeld. Stel dat we volgende query willen uitvoeren op de public view forest uit figuur 4.8: '`for $p in $view/child::Persoon return $p`'. Zoals verwacht zullen we de variabele `$p` vertalen in de view forest die overeenkomt met alle nodes onder (en inclusief) "`<Persoon>`". Indien S een leeg SQL fragment is, volstaat dit om deze query te vertalen.

Binaire operatoren

Indien we een query Q van de vorm [*Expr1 BinOp Expr2*] wensen te vertalen, dan zullen we eerst de view forests *vf1* en *vf2* bepalen die overeenkomen met de view forests voor beide expressies. Deze view forests zullen eigenlijk view trees zijn. Vervolgens zullen we de SQL fragmenten van wortels van beide view forests "joinen" tot een nieuw SQL fragment S'. De SELECT clause van het S' zullen we bepalen door *BinOp* uit te voeren op de SELECT clauses van *vf1* en *vf2*.

Onder binaire operatoren verstaan we arithmetische, logische en vergelijkende operatoren. We zullen bij het voorbeeld van de voorwaardelijke expressies de vertaling van een vergelijkende operator illustreren.

Voorwaardelijke expressies

Om een `if-then-else` expressie te vertalen moeten we eerst de twee SQL fragmenten aanmaken die overeenkomen met de expressie na het `if` statement en z'n negatie. Elk van deze fragmenten dienen gecombineerd te worden met het SQL fragment S.

In plaats van verder hierop in te gaan zullen we dit illustreren aan de hand van ons lopend voorbeeld. Stel dat we volgende query proberen te vertalen: `if ($Hobby = "Vissen") then (element Visser {}) else ()`. Het mag duidelijk zijn dat dit slechts een deel van een volledige user query kan zijn. Om dit te vertalen zullen we eerst `$Hobby = "Vissen"` vertalen. Eerst

vertalen we `$Hobby` en `"Vissen"` naar twee view trees. We kunnen dan de hele vergelijking vertalen naar één node met een boolean waarde. Het XML label zal dan `boolean` zijn, terwijl het SQL fragment eruit ziet als
```
FROM Persoon p, Hobbies h
WHERE h.RRN = p.RRN
SELECT h.Hobby = "Vissen".
```

Vervolgens kunnen we de SQL fragmenten maken voor het `then` en het `else` gedeelte, die we respectievelijk $sqlTrue$ en $sqlFalse$ zullen noemen. Deze zien er als volgt uit:
```
FROM Persoon p, Hobbies h
WHERE (h.RRN = p.RRN) AND (h.Hobby = "Vissen")

FROM Persoon p, Hobbies h
WHERE (h.RRN = p.RRN) AND NOT (h.Hobby = "Vissen").
```

De vertaling gaat nu verder door de view forest *vf1* voor '`element Visser {}`' te bepalen met '$S = sqlTrue$' en *vf2* te bepalen voor '`()`' met '$S = sqlFalse$'. Voor *vf2* vinden we op triviale wijze de lege view forest. De view forest *vf1* daarentegen bestaat uit één elementknoop met SQL fragment $sqlTrue$ en XML label "`<Visser>`". Deze view forest zal tevens de vertaling zijn van de hele query.

For en Let expressies

Een `let` expressie wordt eenvoudig vertaald door de expressie rechts van het toekenningsteken te evalueren en de variabele te binden aan het resulterende view forest.

Een `for` expressie is echter een stuk moeilijker om te zetten. Zo'n expressie ziet er in 't algemeen uit als: [`for` *Var2* `in` *Var1/Axis::NodeTest* `return` *Expr*]. Eerst zullen we opvragen aan welke view forest *vf1* de variabele *Var1* gebonden is in *Env*. We zullen dan alle wortels $vn1$ van *vf1* overlopen en voor elke wortel de XPath stap op de input view-tree evalueren. Voor elke knoop die in het resultaat hier van zit, gaan we, afhankelijk of deze een voorouder of een afstammeling is van $vn1$, een nieuwe binding aanmaken voor $Var2$, zodat we een nieuwe environment Env' als resultaat krijgen. Tegen deze nieuwe environment zal dan de expressie *Expr* geëvalueerd worden. Voor meer details rond dit algoritme verwijzen we de lezer natuurlijk naar [12].

We zullen ter illustratie een `for` expressie proberen te evalueren in het kader

van het lopend voorbeeld. Stel dat we de volgende query moeten combineren met de public view forest uit figuur 4.8:
`for $p in $h/child::Persoon return $p`

Eerst moeten we de binding voor `$h` vinden, dewelke zal overeenkomen met de public view forest (`$h` staat immers voor `HobbyLijst`). We noemen de view forest die hiervan het resultaat is *vf1*. We beschouwen dan om beurt elke root van `$h`, dus enkel van de knoop $vn1$ met XML label "`<HobbyLijst>`". Vervolgens zullen we voor elke knoop $vn2$, die aan de expressie `HobbyLijst/child::Persoon` voldoet, nagaan of deze een voorouder of afstammeling is. Aangezien we als as `child` gebruiken, is het dus duidelijk dat dit een afstammeling is. De enige $vn2$ die we moeten overlopen is de knoop met het label "`<Persoon>`", die dus een kind is van die met label "`<HobbyLijst>`".

Dan zullen we een SQL fragment S' maken dat het resultaat is van het joinen van S met het SQL fragment van $vn2'$ ($vn2'$ is $vn2$ met hernoeming van de tupel variabelen). Vervolgens maken we een nieuwe elementknoop $vn2''$ aan met de naam en subforest van $vn2'$. We zullen dan een nieuwe environment Env' maken waarin we `$p` binden aan $vn2''$. Het resultaat van de compositie is dan het resultaat van $VFCA(Env', \$p, S'))$, waarvan we reeds gezien hebben hoe we dit moeten uitwerken.

Ingebouwde functies en operatoren

Ook voor de ingebouwde functies en operatoren `data()`, `distinct-values()`, `empty()` en de aggregate functions (zoals `count()`) zijn er vertalingsregels gedefinieerd. We gaan op deze regels niet in, maar verwijzen de lezer hiervoor naar [12].

Gebruikers-gedefinieerde functies

Enkel niet-recursieve gebruikers-gedefinieerde XQuery functies worden door VFCA ondersteund. De vertaling van zo'n functie voert eerst VFCA uit op elk van de actuele parameters van de functie, wat een aantal view forests oplevert. Vervolgens wordt een nieuwe environment Env' gemaakt die elk van de formele parameters van de functie bind aan zijn corresponderende view forest (dus binden aan de actuele parameter). Tenslotte voeren we VFCA recursief t.o.v. de functie body uit met de nieuwe environment Env en SQL fragment S.

De beperking tot recursieve gebruikers-gedefinieerde XQuery functies heeft zo zijn nodige repercursies op de kracht van deze methode, zoals we in hoofdstuk 6 zullen zien.

Hoofdstuk 5

SilkRoute en STORED combineren

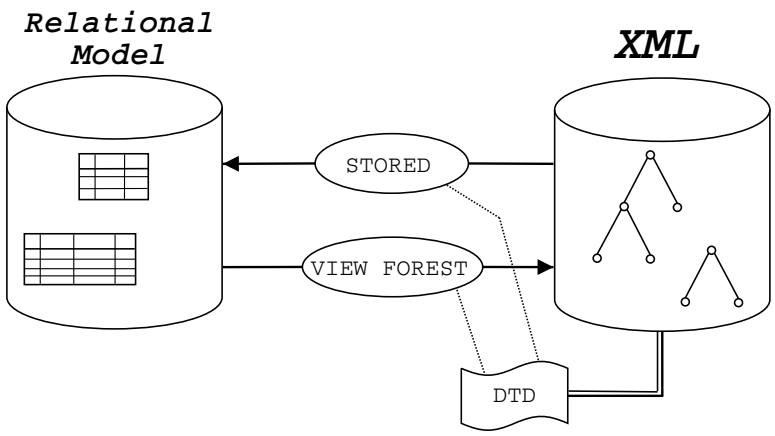
We hebben in deze thesis reeds gezien wat het SilkRoute en STORED project nu juist inhouden. In dit hoofdstuk gaan we beide methodes combineren om zo het mogelijk te maken om een XML document te bewaren in een relationele database en vervolgens hiervoor een view aan de gebruiker aan te bieden die overeenkomt met het orginele XML document. Deze view zullen we aanmaken, door via de DTD en STORED queries automatisch een public view forest te genereren, waarvan de evaluatie het gewenste resultaat oplevert.

5.1 Inleidende begrippen

De meeste begrippen die we in dit hoofdstuk nodig hebben, zijn reeds in 2.1 besproken geweest. In deze sectie zullen we dan ook slechts één begrip, dat eigenlijk reeds in hoofdstuk 4 gezien werd, verder bespreken in context van de omzetting die we zullen definiëren.

5.1.1 Public View Forest

Zoals we reeds in 4.3.3 gezien hebben zal de public view forest het resultaat zijn van het uitvoeren van de view query op de canonical view forest. In plaats van (zoals gebruikelijk) te vertrekken van de canonical view forest,



Figuur 5.1: Samenhang van STORED en SilkRoute

kunnen we echter onmiddellijk de public view forest aanmaken. We zullen hiervoor informatie uit de STORED queries en de DTD gebruiken.

Het is geweten dat een public view forest een graaf (meer bepaald een forest) is waarvan de knopen een XML label en een SQL fragment bevatten. Het XML label kan eenvoudig bepaald worden door de wortel van het originele XML document te bepalen uit de STORED queries en de rest van de graaf reeds te vullen met de XML labels die we vinden in de DTD. De STORED queries zullen dan verder gebruikt worden om de SQL fragmenten in te vullen. Alhoewel STORED ervoor zorgt dat een bepaald attribuut in een relatie duidelijk overeenkomt met een padnaam die een bepaald element of attribuut uitdrukt, is het niet triviaal om hieruit de SQL fragmenten te genereren. Daarom zullen we in de rest van dit hoofdstuk stap voor stap te werk gaan om een correcte view forest te bepalen die het gewenste resultaat oplevert bij evaluatie ervan.

5.2 Skelet van View Forest

We zullen steeds eerst een onvolledige public view forest maken die enkel maar XML labels bevat en nog geen SQL fragmenten. Zo verkrijgen we het skelet voor de gewenste view forest [1]. Vervolgens gaan we voor elke knoop

[1] Het skelet voor de view forest komt overeen met de DTD boom die kan afgeleid worden van de DTD graaf.

het passende SQL fragment bepalen. Het uiteindelijke resultaat zal dan de public view forest zijn die we zoeken.

Het root element van het orginele XML document kunnen we uit de DTD graaf halen. De tag voor dit element zal dan het root element zijn van de view forest. Vervolgens gaan we voor alle bladeren waarvan het XML label een tag is, alle subelementen en attributen bepalen. Voor elk subelement maken we een nieuwe knoop aan met als XML label de tag hiervoor. Voor elk attribuut van het beschouwde element maken we eveneens een knoop aan met als XML label de naam van het attribuut, voorafgegaan door een '`@`'. Indien een element geen subelementen heeft, dan is dit element van het type `EMPTY` of `#PCDATA`. Indien het tweede het geval is, zullen we een nieuwe knoop met het XML label '`text`' aanmaken. Alle nieuw aangemaakte knopen voor het beschouwde element zullen met dit element via een tak in de DTD graaf verbonden worden. We blijven dit herhalen totdat er geen bladeren meer zijn waarvan het XML label een tag is. Aangezien we in 3.1.1 hebben gezien dat de XML documenten die we beschouwen geen recursieve documentbeschrijving mogen hebben, ligt het voor de hand dat dit algoritme ooit zal eindigen.

Om het aanmaken van de gedeeltelijke public view forest (waarbij enkel de XML labels zijn ingevuld) te illustreren, zullen we gebruik maken van een lopend voorbeeld. Dit wil zeggen dat we op dit voorbeeld in de volgende secties zullen verderbouwen. De DTD en de STORED queries voor dit voorbeeld vindt u in figuren 5.2 en 5.3. Herinner u dat we in hoofdstuk 2 hebben gezien dat indien er geen waarde voor `KEY` gespecifieerd wordt, de eerst gedefinieerde variabele in de STORED query de primaire sleutel zal zijn. Indien u nu het voorgaande algoritme volgt, zal u zien dat u het resultaat uit figuur 5.4 bekomt.

5.3 SQL fragmenten

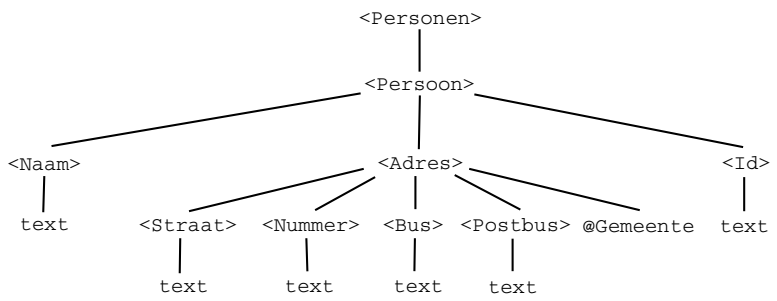
We hebben in de vorige sectie bekeken hoe we de XML labels aanmaakten om de vorm van de view forest te bepalen. We gaan nu aan deze knopen SQL fragmenten toekennen. Dit gaan we stap voor stap doen, door eerst de FROM clauses in te vullen, vervolgens de nodige WHERE clauses toe te voegen en tot slot de SELECT clauses in te vullen (wat enkel in de bladeren zal gebeuren). Wel zullen we omwille van een speciaal geval mogelijk nog enkele FROM clauses uitbreiden nadat we de SELECT clauses hebben toegevoegd. Het is belangrijk dat we deze volgorde hanteren, zoals later zal

```
<!DOCTYPE Personen [
  <!ELEMENT Personen (Persoon*)>
  <!ELEMENT Persoon ((Naam, Adres*) | Id)>
  <!ELEMENT Naam (#PCDATA)>
  <!ELEMENT Adres ((Straat, Nummer, Bus?)|(Postbus))>
  <!ELEMENT Straat (#PCDATA)>
  <!ELEMENT Nummer (#PCDATA)>
  <!ELEMENT Bus (#PCDATA)>
  <!ELEMENT Postbus (#PCDATA)>
  <!ATTLIST Adres Gemeente CDATA #REQUIRED>
]>
```

Figuur 5.2: DTD voor het lopend voorbeeld

```
FROM Personen: {
  Persoon: $P {
    Naam: $N, Id: $I
  }
}
STORE Persoon($P, $N, $I)

FROM Personen.Persoon: $P {
  Adres: $A {
    Straat: $S, Nummer: $N, Bus: $B, Postbus: $P, Gemeente: $G
  }
}
KEY ($A)
STORE Adres($P, $A, $S, $N, $B, $P, $G)
```

Figuur 5.3: STORED queries voor het lopend voorbeeld



Figuur 5.4: De XML labels van public view forest voor het lopend voorbeeld

blijken.

5.3.1 FROM clause

We beginnen dus met de de FROM clause in de SQL fragmenten van de interne knopen in te vullen. We doen dit als volgt:

- Voor elke tabel t waarvoor er een STORED query bestaat, doe het volgende:
  - Match de DTD graaf met de STORED query voor tabel t (cf. 2.4.1). Vereenvoudig het resultaat hiervan met het algoritme `simplifyGraph()` (cf. 2.4.2) en noem deze vereenvoudiging g.
  - Voor alle paden p in g die van de wortel naar een blad van g lopen, doe het volgende:
    - Indien er een element dat een deel is van de primaire sleutel van t op het pad p ligt, bepaal dan (indien mogelijk) de knoop k die als label de operator '*' of '+' bevat, een voorouder is van een knoop die overeenkomt met (een deel van) de primaire sleutel in t (noem deze knoop n) en waarvoor er geen andere '*'- of '+'-knopen meer op het pad tussen k en n liggen. Indien er verschillende mogelijkheden zijn voor k (omdat er meerdere delen van de primaire sleutel op het pad p liggen), kies dan voor de diepst geneste knoop die in aanmerking komt. Indien er geen waarde voor k kan gevonden worden, dan is k onbepaald.

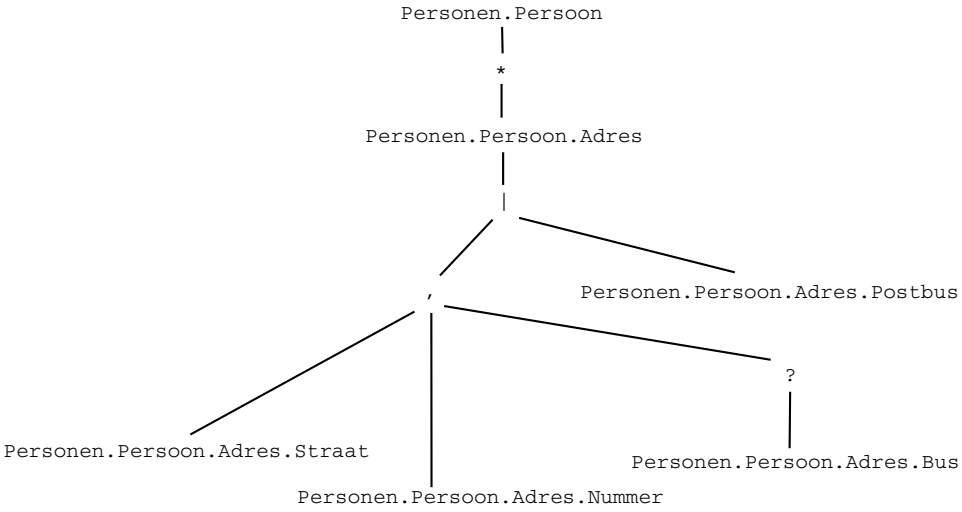
- Indien k bepaald is, laat dan E de verzameling zijn van de elementknopen e die in g nakomelingen zijn van k en waarvoor er geen pad bestaat tussen e en k waarop zich een element (verschillend van e) bevindt. Anders is $E = \phi$.
- Als $E \neq \phi$, voeg dan voor elke $e \in E$ de tabel t toe aan de FROM clause van de knoop in de public view forest die overeenkomt met de padnaam van e indien t nog niet in deze FROM clause staat.

Alvorens een voorbeeld te geven van de werking van dit algoritme gaan we de intuïtie achter het algoritme trachten te schetsen. Vervolgens zullen we de beperkingen zien van dit algoritme.

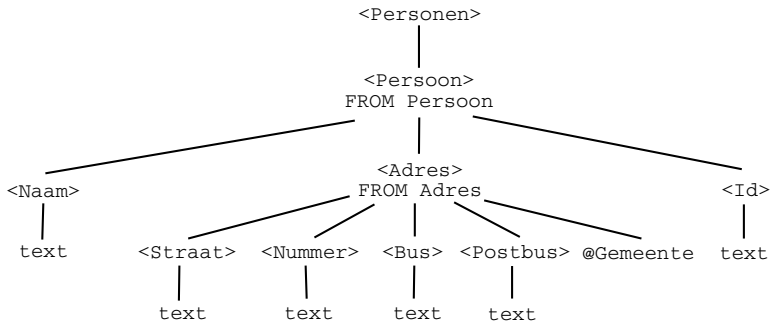
De sleutelwaarde voor een tabel bepaalt hoeveel tupels er in de tabel zitten en bijgevolg, door de tabel in de FROM clause van het sleutelelement te steken, hoeveel keer het sleutelelement voorkomt. Daarom moet bij evaluatie van de view forest in de knoop van het sleutelelement de tabel in de FROM clause zitten. Dit komt overeen met het vragen of de tabel in de FROM clause zit van de knoop van de sleutelwaarde of in een voorouder ervan. Indien er tussen een voorouder van het sleutel element en het sleutel element zelf geen '*'- of '+'-knoop zit, dan komt dit element evenveel keer voor als het sleutelelement en zetten we de tabel dus in de FROM clause van de knoop van dat element in plaats van die van het sleutelelement. In ons algoritme bepalen we zulke knopen door middel van de verzameling E.

Om het algoritme nog verder af te werken, zouden we er eigenlijk nog aan moeten toevoegen dat indien een tabel in de FROM clauses van twee knopen staat, waartussen er een voorouder/nakomeling relatie geldt, we deze in de diepst geneste knoop van de twee moeten schrappen uit de FROM clause. Om het algoritme echter min of meer leesbaar te houden, hebben we echter besloten om dit niet bij in de beschrijving ervan te zetten.

Alvorens de beperkingen van het algoritme te bekijken, zullen we de FROM clauses voor ons lopend voorbeeld trachten te bepalen. Daarvoor matchen we eerst de DTD graaf met de twee STORED queries doen. In figuur 5.5 vindt u het resultaat voor de tweede STORED query. We zien dat er slechts één '*'-operator hierin voorkomt, zodat we dus het oorspronkelijke document kunnen reconstrueren. Uit de query in figuur 5.3 kunnen we afleiden dat de primaire sleutel $Personen.Persoon.Adres$ is. Bijgevolg vinden we dat '$E = \{Personen.Persoon.Adres\}$'. We zullen dus '`Adres`' toevoegen aan de FROM clause van de knoop met het XML label "`<Adres>`". Het resultaat voor de volledig uitgewerkte FROM clauses ziet u in figuur 5.6.



Figuur 5.5: Resultaat van het matchen van STORED query voor Adres met DTD graaf



Figuur 5.6: Bepalen van FROM clauses in het lopende voorbeeld

Dit algoritme is echter niet altijd geldig. We zullen aantonen dat dit algoritme niet altijd geldig is als er meer dan één '*'- of '+'-operator in het resultaat zit van het matchen van de DTD graaf met de STORED query. Intuïtief gezien komt dit doordat er dan zaken worden opgeslagen met een verschillend aantal voorkomens, aangezien enkel bewaarde elementknopen (en de wortel) nog in de DTD graaf staan. Dit is echter niet steeds een probleem, zoals in het geval wanneer we een object identifier gebruiken waarop we joinen. We mogen dan wel de tabel zelf enkel gebruiken om de nakomelingen van de tweede '*'- of '+'-operator te reconstrueren en niet voor de reconstructie van de object identifiers. Maar aangezien de maker van de STORED query door het aanmaken van extra tabellen ervoor kan zorgen dat we die object identifiers in andere tabellen bewaren, die we dan wel kunnen gebruiken voor reconstructie, zijn we afhankelijk van de gemaakte STORED queries om al dan niet het originele document te kunnen reconstrueren.

We tonen nu aan waarom we meerdere '*'-operatoren een probleem kunnen vormen. Verduidelijking bij deze uitleg vindt u in figuur 5.7. Stel dat we een documentbeschrijving hebben van de vorm '`a = b*; b = c*`' en dat we in een tabel t de elementen $a.b, a.b.c$ wensen te bewaren. In het resultaat van het matchen van de STORED query voor t met de DTD graaf voor de documentbeschrijving zullen we dan twee '*'-operatoren krijgen. We weten ook dat $a.b$ een object identifier moet zijn. We zouden dan '`FROM t`' in drie verschillende knopen kunnen zetten in de public view forest, namelijk ofwel die knoop die overeenkomt met a, ofwel degene die overeenkomt met $a.b$, ofwel degene die overeenkomt met $a.b.c$. We weten dat er in t evenveel rijen zitten als er c's zijn. Als we nu '`FROM t`' zouden zetten in de knoop van $a.b.c$ dan zouden we bijgevolg ook het juiste aantal c's hebben. We hebben dan echter steeds juist één b, wat dus niet klopt met onze documentbeschrijving. Een andere mogelijkheid is om '`FROM t`' in de knoop overeenkomend met $a.b$ te zetten. We zouden dan echter evenveel b's als c's hebben, wat betekent dat er per b juist één c is, wat ook in conflict is met onze documentbeschrijving. Op analoge wijze kunnen we eenvoudig inzien dat ook het zetten van '`FROM t`' in de knoop overeenkomend met a niet geldig zou zijn. Het probleem is dus het feit dat we bij het gebruiken van '`FROM t`' in een bepaalde knoop er steeds voor zorgen dat er juist zoveel knopen zijn als er rijen zijn in het resultaat van de query die we krijgen door de public view forest daar te evalueren. Hierdoor kunnen we namelijk het aantal voorkomens van het middelste element b niet juist bepalen. Op het eerste zicht kunnen we dit oplossen door '`FROM t`' in de knoop die overeenkomt met $a.b$ te zetten en in deze knoop ook een '`SELECT DISTINCT t.b`' te doen. Hierdoor lossen we al

wel het probleem op dat we evenveel b's hebben als c's, maar er zijn nog b's die in het originele document kunnen voorkomen, maar niet in onze view, omdat ze geen kinderen van het type c hebben.

De andere mogelijkheid om twee '*'- of '+'-operatoren in een resulterende DTD graaf (na matchen en vereenvoudigen) te hebben, is doordat deze kinderen zijn van een keuze of sequentie. In die gevallen kunnen we echter eenvoudig inzien dat we dit onmogelijk door een STORED query in één relatie kunnen bewaren. Aan de hand van deze uitleg is het hopelijk duidelijk genoeg waarom we van de maker van de STORED queries eisen dat hij ervoor zorgt dat er niet meer dan één '*'- of '+'-operator zit in het matchen van de DTD graaf met eender welke STORED query die hij geschreven heeft.

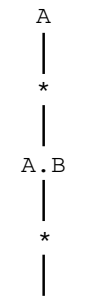
5.3.2 WHERE clause

Enkel knopen met een niet-lege FROM clause kunnen een niet-lege WHERE clause bevatten. In dit geval zal er namelijk een join gebeuren van twee of meerdere tabellen op die attributen waarin een zelfde element (object identifier) of attribuut bewaard wordt. We zullen de WHERE clauses als volgt gaan invullen:

- Overloop alle knopen in de public view forest met een niet-lege FROM clause
  - Laat L een lege lijst van gelijkheden zijn.
  - Overloop dan alle entries $(p, t, a) \in \rho$ (cf. 2.1.4). Als t in de FROM clause van de huidige knoop zit, voeg dan voor elk tupel $(p, t', a') \in \rho$ waarbij t' in een FROM clause van een voorouder van de huidige knoop of in de huidige knoop zelf zit (met $t \neq t'$) de expressie `(`t`.`a `=` t'`.`a'`)` toe aan L.
  - Indien L niet leeg is, zet dan " `AND` " tussen de verschillende termen en zet het resultaat hiervan in de WHERE clause van de huidige knoop.

We zullen dus steeds de restrictie op het cartesisch product van twee tabellen (wat overeenkomt met een join) zo vroeg mogelijk doen. Aangezien de joins gebeuren op attributen of object identifiers, verkrijgen we het gewenste resultaat, dat equivalent zal zijn met het originele XML document.

Door dit algoritme toe te passen op ons lopend voorbeeld, krijgen we als resultaat de public view forest die u in figuur 5.8 kan zien.

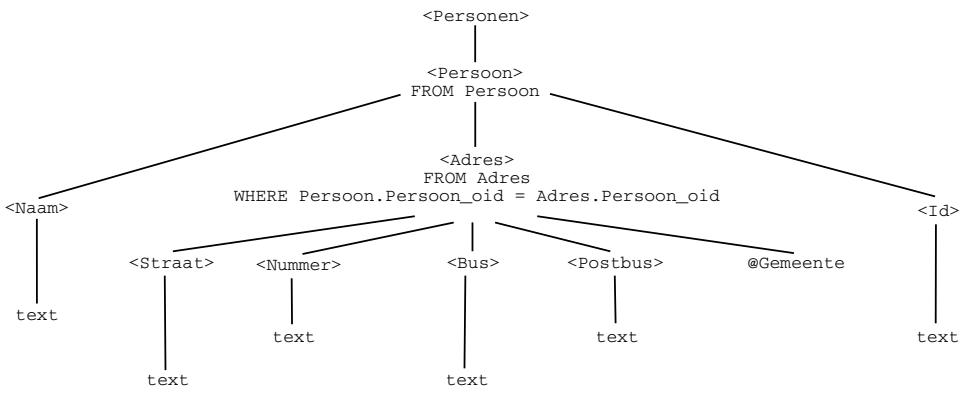


t	b	c
	1	1
	1	2
	2	3

```
<A><B><C>1</C></B></A>
<A><B><C>2</C></B></A>
<A><B><C>3</C></B></A>

<A><B>
  <C>1</C>
  <C>2</C>
  <C>3</C>
</B></A>
```

```
<A>
 <B><C>1</C></B>
 <B><C>2</C></B>
 <B><C>3</C></B>
</A>

<A>
 <B>
  <C>1</C>
  <C>2</C>
 </B>
 <B>
  <C>3</C>
 </B>
</A>
```

Figuur 5.7: Een voorbeeld probleem bij reconstructie, vanboven een mogelijke instantie van de relatie t en de DTD graaf voor A. Onderaan zien we de drie mogelijke gevolgen van reconstructie (met respectievelijk FROM clause in A, A.B en A.B.C), gevolgd door de gewenste reconstructie



Figuur 5.8: Bepalen van WHERE clauses in het lopende voorbeeld

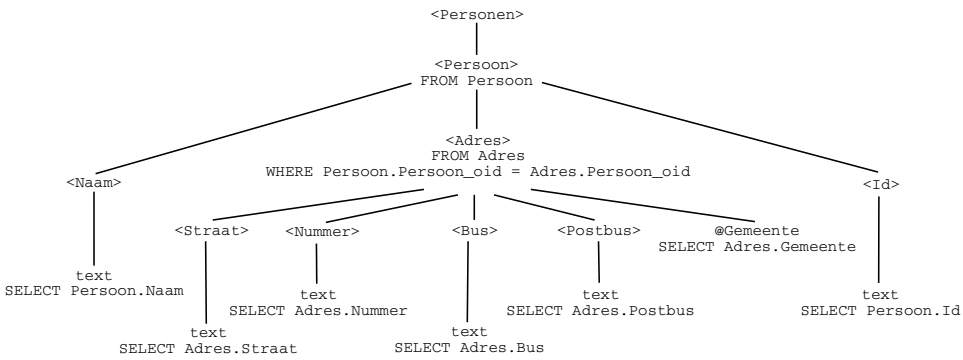
5.3.3 SELECT clause

De SELECT clauses zullen enkel voorkomen in de SQL fragmenten in de bladeren van de view forest. Deze SQL fragmenten zullen overigens alleen uit een SELECT clause bestaan, zoals we reeds gezien hebben in 4.3.

Een blad in de public view forest heeft ofwel een attribuutnaam ofwel '`text`' als XML label. Indien het '`text`' als XML label bevat, dan gaan we de padnaam zoeken in de public view forest van de ouder van deze knoop. In het andere geval is de XML label een attribuutnaam en zoeken we de padnaam van dit attribuut in de public view forest. Een padnaam kunnen we vinden door vanaf de root het pad te volgen tot de betrokken knoop en elke tag die je tegenkomt achteraan de padnaam toe te voegen. Nu we voor elk blad een padnaam hebben gevonden, kunnen we beginnen met het eigenlijke bepalen van de SQL fragmenten.

In de opslagtabel vinden we per padnaam één of meerdere tabel-attribuut paren. Voor een blad zullen we deze koppels opzoeken. Stel dat we voor een gegeven knoop de tabel-attribuut koppels noteren als (t_i, a_i). Indien er dan een koppel bestaat waarvoor t_i in de FROM clause staat van één van de voorouders, dan wordt het SQL fragment voor het blad in kwestie:
`SELECT` t_i`.`a_i

Omdat we alle tabellen die in FROM clauses zitten van de voorouders gejoined hebben, blijft het eender van welke tabel we de attributen gaan selecteren, zolang we maar zien dat de tabel in de FROM clause zit van een voorouder. We weten immers dat door de join die er is uitgevoerd, dit toch



Figuur 5.9: Eindresultaat view forest voor lopend voorbeeld

voor elke tabel hetzelfde resultaat zal opleveren.

Indien er echter geen koppel (t_i, a_i) bestaat waarvoor t_i in de FROM clause van een voorouder zit, selecteer dan een willekeurig koppel (t_i, a_i). Dit wil dus zeggen dat het SQL fragment van dat blad het volgende wordt:
`SELECT` t_i`.`a_i

In dit geval komt de tabel t_i echter niet in de FROM clause van de SQL query voor het blad zelf voor. Daarom zullen we in de ouder van het blad dat we beschouwen t_i toevoegen aan de FROM clause. In de praktijk zal dit enkel het geval zijn wanneer we tabellen hebben met hoogstens 1 tupel. Anders zal er immers steeds een '*'- of '+'-knoop aanwezig zijn in de graaf, en zal er dus steeds een deel van de sleutel een nakomeling moeten zijn van die '*'- of '+'-knoop, zodat de tabel reeds in een FROM clause zat.

Wanneer we dan de SELECT clauses gaan toevoegen aan ons lopend voorbeeld, bekomen we het resultaat dat u vindt in figuur 5.9. Deze view forest zal de view forest zijn die we uiteindelijk zochten.

Hiermee wordt dan ook dit hoofdstuk afgesloten, waarin we hebben gezien hoe we automatisch de public view forest kunnen aanmaken voor een relationele database die we gemaakt hebben met STORED queries, zodanig dat we als view terug een equivalent van ons originele XML document bekomen.

Hoofdstuk 6

Kracht en beperkingen van vertaling

In hoofdstuk 3 hebben we gezien welke beperkingen het eerste gedeelte van deze thesis oplegt aan de door ons voorgestelde methode. We zullen nu bekijken in hoeverre het tweede gedeelte nog extra beperkingen oplegt. Deze beperkingen kunnen zich op twee vlakken uiten. Het eerste waar we rekening mee moeten houden is of we een XML document volledig en correct kunnen reconstrueren vanuit de relationele database, zodat er voldaan wordt aan de DTD. Een tweede aandachtspunt is de kracht en beperking van queries, zoals we die reeds gedeeltelijk kort besproken hebben in hoofdstuk 4.

6.1 Reconstructie

De reconstructie van het XML document vanuit een relationele database zal gebeuren via een public view forest die gemaakt wordt aan de hand van de DTD en de STORED queries, zoals we gezien hebben in hoofdstuk 5. Indien de relationele database is gewijzigd sinds de automatische omzetting die beschreven werd in hoofdstuk 2, kunnen er zich extra complicaties voordoen bij de reconstructie omdat niet-bewaarde constraints kunnen gebroken worden. We zullen dan ook in wat volgt onderscheid maken tussen een relationele database waarin wijzigingen zijn gebeurd na creatie en die waarin dit niet is gebeurd.

6.1.1 Zonder wijzigingen in relationele database

Indien we geen wijzigingen in de relationele database aanbrengen, dan weten we dat alle gegevens in principe in een XML view kunnen gestoken worden zonder dat we hierbij de DTD schenden. Ons input XML document was hier immers een voorbeeld van. Wat we wel kwijt kunnen spelen is de volgorde waarin gegevens in een XML document stonden. De vraag is nu of de werkwijze die we in hoofdstuk 5 hebben gezien wel een XML document teruggeeft dat aan onze DTD voldoet.

Een eerste veronderstelling die we maken is dat de STORED queries zorgen voor een volledige opslag van de gegevens. Dit is een redelijke veronderstelling, aangezien het vanzelfsprekend is dat we anders niet het oorspronkelijke document kunnen reconstrueren (cf. 5.3.1). Verder veronderstellen we dat de STORED queries zo zijn opgesteld dat we alle constraints uit hoofdstuk 2 (buiten IDREFS) kunnen bewaren en dat ze resulteren in tabellen uit welke we het originele document kunnen reconstrueren. Uit hoofdstuk 5 kunnen we dan afleiden dat we zo een equivalent van ons originele XML document kunnen reconstrueren via onze methode.

6.1.2 Na wijzigingen in relationele database

Wanneer we gegevens gaan toevoegen aan de relationele database, bestaat de kans dat we door het evalueren van de public view forest een XML view krijgen die niet meer voldoet aan de DTD. Dit kan echter enkel het geval zijn doordat we niet alle constraints bewaren bij de omzetting naar het relationele schema. In hoofdstuk 3 hebben we gezien dat, indien we onze STORED queries goed kiezen, we alle opgesomde constraints bewaren, behalve de inclusion dependencies die voortkomen door het gebruik van IDREFS. Nu kunnen we in attributen waarin normaal IDREFS worden bewaard, waarden insteken die niet in attributen van het type ID worden bewaard. Zo kunnen we de inclusion dependency breken en een XML view als resultaat krijgen die niet meer voldoet aan de DTD. Indien we echter in onze DTD geen attributen van het type IDREFS voorkomen en onze STORED queries goed opgesteld worden zullen we dus als resultaat steeds een XML view krijgen die voldoet aan de DTD.

6.1.3 Conclusie

We veronderstellen steeds dat de STORED queries zo zijn opgesteld dat we een public view forest hieruit kunnen maken met de geziene methode. De resulterende XML view voldoet aan de originele DTD indien we geen wijzigingen hebben aangebracht aan de relationele database. Meer zelfs, de resulterende XML view komt dan overeen met het originele XML document op volgorde van elementen na. Indien we wijzigingen hebben aangebracht in de relationele database, dan is het al dan niet voldoen van de resulterende XML view aan de DTD afhankelijk van hoe goed we de constraints bewaard hebben bij de omzetting van het oorspronkelijke XML document naar de relationele database. Het enige probleem dat zich bij de reconstructie zal stellen indien de STORED queries zelf goed zijn opgesteld, is dus de reconstructie van IDREFS, aangezien er nu ongeldige waarden kunnen zitten in dit attribuut.

6.2 Queries

We hebben in hoofdstuk 4 gezien hoe we XQuery queries kunnen omzetten naar SQL door het uitvoeren ervan op een view forest. Deze vertaling induceerde echter enkele beperkingen. Deze beperkingen zullen we nu kort bespreken.

6.2.1 Recursie

We hebben reeds gezegd dat al onze queries omzetbaar moeten zijn naar SilkRoute's XQueryCore. Recursieve features van XQuery behoren echter niet tot deze XQueryCore. Dit is te wijten aan het feit dat we eisen dat onze view forest als een eindige boom kunnen uitdrukken. Een recursieve functie kan echter niet uitgedrukt worden in een eindige boom. Door gebruik te maken van SQL-3 kunnen we bijgevolg de kracht van onze vertaling niet uitbreiden, omdat in dit geval de beperking ligt in het gebruik van de view forests.

6.2.2 Functies

Naast de recursieve functies zitten er ook geen functies die afhankelijk zijn van de volgorde van waarden in een XML document of die een volgorde

hieraan opleggen in XQueryCore. De eerste beperking is vanzelfsprekend, aangezien we in een relationele database geen strikte volgorde kennen. Bijgevolg kunnen we niets zinnig zeggen over de volgorde.

De tweede beperking die zegt dat we geen functies mogen gebruiken die een volgorde opleggen aan de waarden in een XML document (bvb. sort operatie) heeft vooral te maken met het feit dat we samengestelde queries kunnen maken, waarbij het voor ons onmogelijk is om gebruik te maken van de volgorde die we hebben opgelegd in een tussenresultaat.

Een derde beperking die niet afkomstig is van het gebruik van SilkRoute, maar zuiver afhankelijk is van het feit dat we DTD's gebruiken voor de omzetting is de beperking van het aantal types waarop we functies kunnen gebruiken. Zo kunnen we geen rekenkundige functies die werken op getallen gebruiken, aangezien we door onze mapping de inhoud van elementen van een XML document steeds typeren als tekst. We hebben deze beperking reeds eerder aangehaald in 3.1.2.

6.2.3 Conclusie

De queries die we kunnen stellen moeten steeds herleidbaar zijn tot SilkRoute's XQueryCore en mogen geen gebruik maken van een aantal types (zoals getaltypes e.d.). Het eventuele gebruik van SQL-3, waarin we over recursie mogelijkheden beschikken, zal niets veranderen aan de beperkingen opgelegd door SilkRoute's XQueryCore, aangezien de beperking afkomstig is van het gebruik van view forests, die we steeds wensen te evalueren naar een eindige boom.

Besluit

In deze thesis hebben we eerst enkele bestaande mappings tussen XML documenten en relationele databases bestudeerd. Vervolgens hebben we een methode gedefinieerd om een XML document aan de hand van STORED queries en een DTD in een relationele database te bewaren, waarbij het grootste deel van de constraints die de DTD oplegt aan het XML document, bewaard werden. In het tweede deel hebben we dan een methode geïntroduceerd om, voor de verzameling tabellen die we als resultaat kregen van de mapping uit het eerste deel, een XML view te genereren, die bovendien aan de oorspronkelijke DTD voldoet. Zolang de STORED queries een redelijke opslag van XML naar relaties definiëren, lukt het construeren van deze view.

Wat betreft het onderzoek naar de geldigheid van de voorgestelde algoritme's, zijn we meestal niet dieper op de zaak ingegaan, dan het geven van wat intuïtieve richtlijnen en voorbeelden, die ons vertrouwen in de algoritme's moeten geven. Ook de kracht en beperkingen van onze voorgestelde methode's zijn relatief kort besproken en geven slechts een ruwe schets van wat al dan niet mogelijk is.

De implementatie van het prototype dat een aantal constraints uit een DTD haalt en deze formuleert in SQL in functie van de verkozen opslag (afhankelijk van de STORED queries), was een indicatie voor enkele extra hinderpalen die we bij het verder uitwerken van deze theoretische resultaten kunnen ondervinden. Vele definities en algoritme's zijn immers niet volledig formeel en enkele ambiguïteiten in de specificatie voor zo'n algoritme's zijn haast onvermijdbaar.

Deel III

Bijlagen

Bijlage A

Grammatica voor DTD's

In het prototype dat tijdens deze thesis werd geïmplementeerd, wordt er een beperking opgelegd aan de DTD's die geparsed kunnen worden. Deze beperking heeft de bedoeling om voor de vertaling van XML documenten naar relationele databases ons niet te sterk te concentreren op details. De grammatica voor DTD's vindt u hieronder, in EBNF-formaat.

dtdDocument	=	"`<!DOCTYPE`" rootElement "`[`" declList "`]>`"
declList	=	decl {decl}
decl	=	elemDecl | attListDecl
elemDecl	=	elemDeclHead elemDeclTail
elemDeclHead	=	"`<!ELEMENT`" elementName
elemDeclTail	=	"`EMPTY`" "`>`" |
		"`(#PCDATA)`" "`>`" |
		"`(`" complexElemType "`)`" "`>`"
complexElemType	=	"`(`" complexElemType "`)`" |
		complexElemType "`,`" complexElemType |
		complexElemType "`|`" complexElemType |
		elemName |
		complexElemType card
card	=	"`?`" | "`*`" | "`+`"
attListDecl	=	"`<!ATTLIST`" elemName attDecl attDecl "`>`"
attDecl	=	attName attType ("`#REQUIRED`" | "`#IMPLIED`")
attType	=	"`ID`" | "`IDREF`" | "`IDREFS`" | "`CDATA`" | attDomain
attDomain	=	string "`|`" string

rootElement	=	elementName
elementName	=	identifier
attName	=	identifier
string	=	letterOrUnderscore | digit
identifier	=	letter letterOrUnderscore | digit
letterOrUnderscore	=	letter | "_"
letter	=	"a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
	=	"j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
	=	"s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" |
		"A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
	=	"J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
	=	"S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
digit	=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
		"8" | "9" |

Bijlage B

Grammatica voor STORED queries

In deze bijlage trachten we om een subset van de oorspronkelijke STORED query taal [10] te definiëren voor welke we de mapping van XML documenten naar een relationele database zullen verwezenlijken. In hoofdstuk 2 hebben we al vermeld waarom we geen overflow graaf ondersteunen. Verder zullen we ook de WHERE clause die soms in STORED queries voorkomt niet beschouwen, om de mapping niet te complex te maken. De eenvoudige syntax voor de door ons ondersteunde STORED queries ziet er dus als volgt uit (in EBNF-notatie):

storedquery	=	"`FROM`" varList key "`STORE`" name "`(`" attrList "`)`"
key	=	"`KEY`" keyList | ""
keyList	=	variable | variable "`,`" keyList
varList	=	dottedName "`:`" varSpecifier |
		varList "`,`" varList
varSpecifier	=	variable "`(`" varList "`)`" |
		"`(`" varList "`)`" |
		variable
variable	=	"`$`" name
dottedName	=	name | dottedName "`.`" dottedName
attrList	=	variable | variable "`,`" attrList
name	=	identifier
identifier	=	letter letterOrUnderscore | digit

letterOrUnderscore	=	letter | “_”
letter	=	“a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i” |
	=	“j” | “k” | “l” | “m” | “n” | “o” | “p” | “q” | “r” |
	=	“s” | “t” | “u” | “v” | “w” | “x” | “y” | “z” |
		“A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I” |
	=	“J” | “K” | “L” | “M” | “N” | “O” | “P” | “Q” | “R” |
	=	“S” | “T” | “U” | “V” | “W” | “X” | “Y” | “Z” |
digit	=	“0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” |
		“8” | “9” |

Bijlage C

Model-based mapping: voorbeelden

In deze appendix zullen we voor de model-based mapping methodes een aantal voorbeelden beschouwen. Deze voorbeelden heb ik tijdens het bestuderen van deze methodes gemaakt, om voor mezelf te controleren of het al dan niet de moeite waard was om deze methodes verder te beschouwen. Uiteindelijk heeft vooral het feit dat je steeds een zogenaamde vaste mapping hebt bij deze methodes, me van de model-based mapping technieken gehouden.

C.1 Table-based mapping

Deze mapping is het eenvoudigste, en wordt beschreven in 1.1.1. Stel dat we volgend XML document wensen om te zetten:

```
<A>
 <B>
  <D>
   <E>tekstje</E>
   <F>5.5</F>
  </D>
  <D>
   <E>andere tekst</E>
   <F>6.85</F>
```

```
  </D>
 </B>
 <C>
  <D>
   <E>-5</E>
   <G>nog iets</G>
  </D>
 </C>
</A>
```

We selecteren dan eerst de database A. Hierin voeren we volgende SQL statements uit om de tabellen B en C te creëeren.

```
CREATE TABLE B (
  E TEXT,
  F FLOAT
);

CREATE TABLE C (
  E INTEGER,
  G TEXT
);
```

Hiermee is de structuur van het XML document omgezet in een relationele database. Om de data in te voegen, kunnen we nog enkele `INPUT INTO ... VALUES (...)` queries uitvoeren, zodat niet enkel de structur, maar ook de inhoud van de database is omgezet.

C.2 Object-Relational Mapping

Daar waar de table-based mapping snel verworpen werd wegens de te grote beperking die aan de mogelijke XML-documenten werd opgelegd, is er bij de object-relational mapping een uitgebreidere waaier aan omzetbare XML documenten. Het omzetten gebeurt in 2 fasen. Eerst wordt het schema (DTD of XML Schema) omgezet naar een object schema, en vervolgens wordt dit object schema omgezet naar een relationeel schema.

In wat volgt zullen we enkele voorbeelden bekijken waarvoor we de mogelijke schetsing in 1.1.1 hebben beschouwd.

C.2.1 Mapping van DTD naar object schema

Omzetting van een eenvoudige DTD

Als voorbeeld beschouwen we volgende DTD:

```
<!DOCTYPE A[
  <!ELEMENT A (B, C)>
  <!ELEMENT B (#PCDATA)>
  <!ELEMENT C (D, E)>
  <!ELEMENT D (#PCDATA)>
  <!ELEMENT E (#PCDATA)>
]>
```

Om een DTD om te zetten, zullen we `#PCDATA` en `CDATA` steeds omzetten naar een `String`, aangezien er verschillende soorten data de inhoud kunnen zijn van elementen met dit type. Elk element dat van een ander type een instantie is, zal omgezet worden naar een klasse. Het resultaat van de omzetting zal dus het volgende zijn:

```
class A {
  String b;
  C c;
};

class C {
  String d;
  String e;
};
```

Indien het root element als type `#PCDATA` of `CDATA` heeft, zal dit worden omgezet naar een klasse met als enige member een `String`.

Omzetting van meer complexe DTD's

Als voorbeeld beschouwen we nu volgende DTD:

```
<! DOCTYPE A[
  <!ELEMENT A (B, (C, D)+)>
```

```
  <!ELEMENT B (#PCDATA | C*)>
  <!ELEMENT C (E?, F)>
  <!ELEMENT D (#PCDATA)>
  <!ELEMENT E (#PCDATA)>
  <!ELEMENT F (#PCDATA)>
]>
```

Voor de omzetting hiervan, hebben we 3 klasses nodig, namelijk A, B en C. Een instantie van klasse A zal steeds 1 element van type B hebben en vervolgens 1 of meerdere elementen van type C en van type D. Dit wordt weergegeven door een array van C, en een array van String die beide niet leeg mogen zijn. Een instantie van B bestaat ofwel uit een String, ofwel uit een array van elementen van klasse C. Klasse C bevat 2 keer een String, waarvan 1 mogelijk null kan zijn. De omzetting van vorige DTD naar object schema geeft volgend resultaat:

```
class A {
  B b;          // Not Nullable
  C[] c;        // Not Nullable
  String[] d;   // Not Nullable
};

class B {
  String b_1;   // Nullable
  C[] c;        // Nullable
};

class C {
  String e;     // Nullable
  String f;     // Not Nullable
};
```

De arrays c en d in klasse A moeten volgens de DTD steeds allebei dezelfde dimensie hebben. Dit blijkt echter niet uit het object schema. We kunnen dit oplossen door een nieuwe klasse A_1 te maken. Klasse A en klasse A_1 zouden er in dit geval als volgt uit zien:

```
class A {
  B[] b;        // Not Nullable
```

```
  A_1[] a_1;    // Not Nullable
};

class A_1 {
  C c;          // Not Nullable
  String d;     // Not Nullable
};
```

Een andere opmerking die we bij het vorige schema kunnen maken is dat het al dan niet null mogen zijn van een member van een klasse slechts kan worden aangegeven door de commentaar 'Nullable' en 'Not Nullable'. In SQL zullen we dit na de conversie van object shema wel kunnen weergeven door NULL en NOT NULL in de corresponderende tabellen.

C.2.2 Mapping van XML Schema naar object schema

Daar waar we in 1.1.1 de mapping van XML Schema naar Object Schema theoretisch beschouwd hebben, zullen we ze nu kort uitleggen aan de hand van een voorbeeld. We zullen volgend XML Schema afbeelden op een object schema.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:vb="http://xyz.edu/Voorbeeld"
          targetNamespace="http://xyz.edu/Voorbeeld">
   <element name="A" type="vb:complex"/>
   <complexType name="complex">
      <attribute name="att" type="string"/>
      <all>
         <choice>
           <element name="B" type="integer"/>
           <element name="C" type="float"
                    minOccurs="0" maxOccurs="unbounded"/>
         </choice>
         <element name="D">
            <simpleType>
               <restriction base="string">
                  <enumaration value="text1"/>
                  <enumaration value="text2"/>
                  <enumaration value="text3"/>
```

```
            </restriction>
          </simpleType>
        </element>
      </all>
   </complexType>
</schema>
```

De regels die we in 1.1.1 hebben geformuleerd, zeggen ons dat de omzetting van het element A uit de namespace `http://xyz.edu/Voorbeeld` kan worden omgezet tot volgend object schema.

```
class A {
   Integer  b;    // Nullable
   Float    c[];  // Nullable
   String   att;  // Not Nullable
   String   d;    // Not Nullable
};
```

C.2.3 Mapping van object schema naar relationeel schema

Voorbeeld

Beschouwen we volgend object schema, dat het resultaat was van de omzetting uit sectie 1.1.1. Het gaat dus om volgend object schema.

```
class A {
  B[] b;        // Not Nullable
  A_1[] a_1;    // Not Nullable
};

class A_1 {
  C c;          // Not Nullable
  String d;     // Not Nullable
};

class B {
  String b_1;   // Nullable
  C[] c;        // Nullable
};
```

```
class C {
  String e;     // Nullable
  String f;     // Not Nullable
};
```

De omzetting begint bij klasse A. Deze bevat een meerwaardige member, dus wordt de kolom ID toegevoegd als primaire sleutel. We zullen de tabellen die het resultaat zijn van de omzetting weergeven door de SQL statement die deze tabellen creëert. Klasse A zal dus het resultaat zijn van volgende query.

```
CREATE TABLE A (
  ID INTEGER PRIMARY KEY
);
```

De members van klasse A waarvoor we nu nog een omzetting moeten doen zijn b en a_1. De omzetting van b gebeurt door het omzetten van klasse B naar een tabel en vervolgens de foreign key toe te voegen aan deze tabel. Dit gebeurt dus als volgt:

```
CREATE TABLE B (
  ID INTEGER PRIMARY KEY,
  b_1 TEXT NULL,
  a INT REFERENCES A(ID)
);
```

Klasse A_1 wordt als volgt omgezet:

```
CREATE TABLE A_1 (
  ID INTEGER PRIMARY KEY,
  d TEXT NOT NULL,
  a INT REFERENCES A(ID)
);
```

Klasse C wordt omgezet in 2 verschillende tabellen, omdat het zowel gebruikt wordt binnen klasse A_1 als binnen klasse B. De vertaling van klasse C is dus de volgende:

```
CREATE TABLE C_1 (
  e TEXT NULL,
  f TEXT NOT NULL,
  a_1 INT PRIMARY KEY REFERENCES A_1(ID)
);

CREATE TABLE C_2 (
  e TEXT NULL,
  f TEXT NOT NULL,
  b INT REFERENCES B(ID)
);
```

Bij tabel C_1 is de foreign key eveneens de primaire sleutel van de tabel. Dit komt omdat in klasse A_1 er geen lijst van instanties van C zit, maar een instantie van C. Bijgevolg kan er niet meer dan 1 rij in tabel C_1 zitten die verwijst naar een bepaalde rij in tabel A_1.

Hiermee is nu de volledige omzetting van het voorbeeld van object schema naar relationeel schema afgelopen.

Bijlage D

Voorbeeld: van XML naar XML view

We behandelen in deze bijlage een uitgewerkt voorbeeld voor het volledige mappen van een XML document naar het relationele model, gevolgd door het creëren van een public XML view en het uitvoeren van een XQuery hierop.

D.1 Beginsituatie

Het XML document dat we in het voorbeeld gaan gebruiken ziet er als volgt uit:

```
<?xml version="1.0"?>
<BooksAndAuthors>
   <Auhtors>
      <Author>
         <Name>Ginsberg</Name>
         <BooksWritten/>
      </Author>

      <Author>
         <Name>Tanenbaum</Name>
         <BooksWritten>
            <BookWritten ISBN="0136386776"/>
```

```
            <BookWritten ISBN="0130888931"/>
         </BooksWritten>
      </Author>

      <Author>
         <Name>Woodhull</Name>
         <BooksWritten>
            <BookWritten ISBN="0136386776"/>
         </BooksWritten>
      </Author>
   </Authors>

   <Books>
      <Book ISBN="0136386776" Language="English">
         <Title>Operating Systems: Design and Implementation
                (Second Edition)</Title>
         <Chapter Title="Introduction"/>
         <Chapter Title="Processes"/>
         <Chapter Title="Input/Output"/>
         <Chapter Title="Memory Management"/>
         <Chapter Title="File Systems"/>
         <Chapter Title="Reading List and Bibliography"/>
 <Year>1997</Year>
      </Book>

      <Book ISBN="0130888931" Language="English">
         <Title>Distributed Systems: Principles and
                Paradigms</Title>
         <Chapter Title="Key principles of distributed systems"/>
         <Chapter Title="Real-world distributed case studies"/>
      </Book>
   </Books>
</BooksAndAuthors>
```

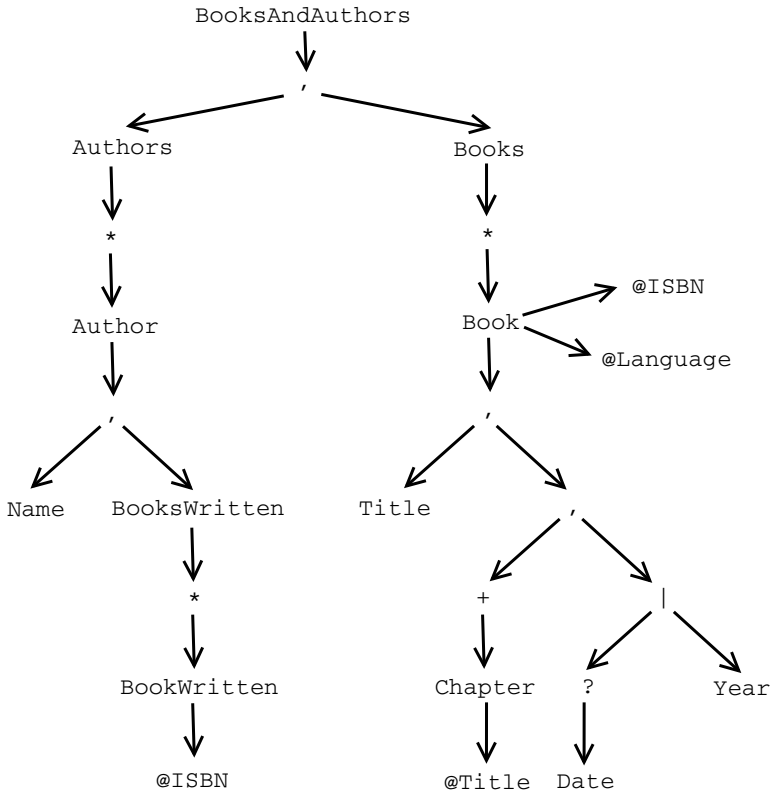
Dit XML document heeft volgende DTD:

```
<!DOCTYPE BooksAndAuthors[
   <!ELEMENT BooksAndAuthors(Authors, Books)>
   <!ELEMENT Authors (Author*)>
   <!ELEMENT Books (Book*)>
```

```
   <!ELEMENT Book (Title, Chapter+, (Date? | Year))>
   <!ELEMENT Title (#PCDATA)>
   <!ELEMENT Date (#PCDATA)>
   <!ELEMENT Year (#PCDATA)>
   <!ELEMENT Chapter EMPTY>
   <!ATTLIST Chapter Title CDATA #REQUIRED>
   <!ATTLIST Book ISBN ID #REQUIRED Language (English|Dutch) #IMPLIED>

   <!ELEMENT Author (Name, BooksWritten)>
   <!ELEMENT Name (#PCDATA)>
   <!ELEMENT BooksWritten (BookWritten*)>
   <!ELEMENT BookWritten EMPTY>
   <!ATTLIST BookWritten ISBN IDREF #REQUIRED>
]>
```

We zullen het XML document mappen naar het relationeel model door volgende STORED queries uit te voeren:

```
FROM BooksAndAuthors.Authors: {
   Author: $A {Name: $N}
}
STORE Author($A, $N)

FROM BooksAndAuthors.Authors.Author: $A {
   BooksWritten.BookWritten.ISBN: $B
}
KEY $A, $B
STORE BookWritten($A, $B)

FROM BooksAndAuthors.Books.Book: {
   ISBN: $B, Title: $T, Language: $L, Date: $D, Year: $Y
}
STORE Book($B, $T, $L, $D, $Y)

FROM BooksAndAuthors.Books.Book: {
   Chapter: $C {Title: $T}, ISBN: $B
}
STORE Chapter($C, $T, $B)
```



Figuur D.1: DTD graaf voor uitgewerkt voorbeeld

D.2 Mappen naar het relationele model

We zullen eerst de DTD graaf opstellen en de verschillende constraints trachten te halen uit de DTD. Vervolgens maken we een SQL statement aan om de nodige relaties aan te maken en om de waarden in de relationele database in te voeren. Merk op dat we dit laatste niet in de thesistekst zelf besproken hebben.

D.2.1 DTD graaf

Het eerste wat we gaan doen is het omzetten van de DTD naar een DTD graaf. Het resultaat hiervan vindt u in figuur D.1. In deze figuur worden de attributen aangegeven door een `@`.

D.2.2 Constraints bepalen

Het zoeken van de constraints in de DTD (graaf) gebeurt soort per soort. We zullen dan ook per constraintsoort zeggen welke resultaten we bekomen voor ons voorbeeld.

Domain constraints

Het eerste soort domeinbeperkingen zijn de enumeration constraints. Een enumeration komt enkel voor bij het attribuut `Language` van het element `Book`. Uit de DTD komen we het volgende te weten:
$\tau($`BooksAndAuthors.Books.Book.Language`$) = \{Book\}$
$\eta($`BooksAndAuthors.Books.Book.Language`$) = \{English, Dutch\}$

Bijgevolg vinden we voor $\epsilon($`BooksAndAuthors.Books.Book.Language`, `Book`$)$ het volgende:

```
CREATE DOMAIN LDomain (VALUE IN (English, Dutch))
```

Dit is het enige attribuut waarvoor ϵ niet leeg is. We moeten ook voor het attribuut `L` uit de tabel '`Work`' ervoor zorgen dat het type niet `TEXT` is, maar `LDomain`.

De tweede soort domeinbeperkingen voor attributen die we gaan opsporen zijn constraints die veroorzaakt worden door de keywords `#REQUIRED` en `#IMPLIED`. De operator δ geeft volgende resultaten wanneer we deze proberen te bepalen:
$\delta($`BooksAndAuthors.Books.Book.Chapter`, `Title`, `Chapter`$, required) =$

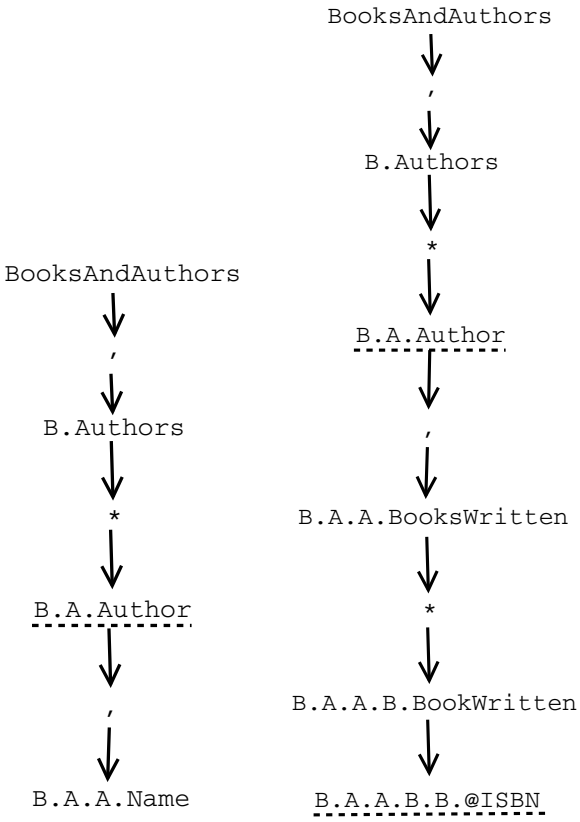
```
CHECK (C NULL AND T NULL) OR (C NOT NULL AND T NOT NULL)
```

Merk op dat de derde parameter van de operator δ bepaalt waar we de constraints gaan plaatsen in de SQL statements. Voor `ISBN` hebben we geen enkele domain constraint gevonden met deze methode, maar dit wordt later opgevangen.

Choice constraints

Om de choice constraints te ontdekken gaan we eerst voor elk van de vijf STORED queries een DTD graaf zoeken die het resultaat is van het mappen van een STORED query met de orginele DTD graaf.

De resultaten hiervan vindt u in figuren D.2 en D.3. Om te voorkomen dat deze tekeningen te groot werden, hebben we de elementnamen van de

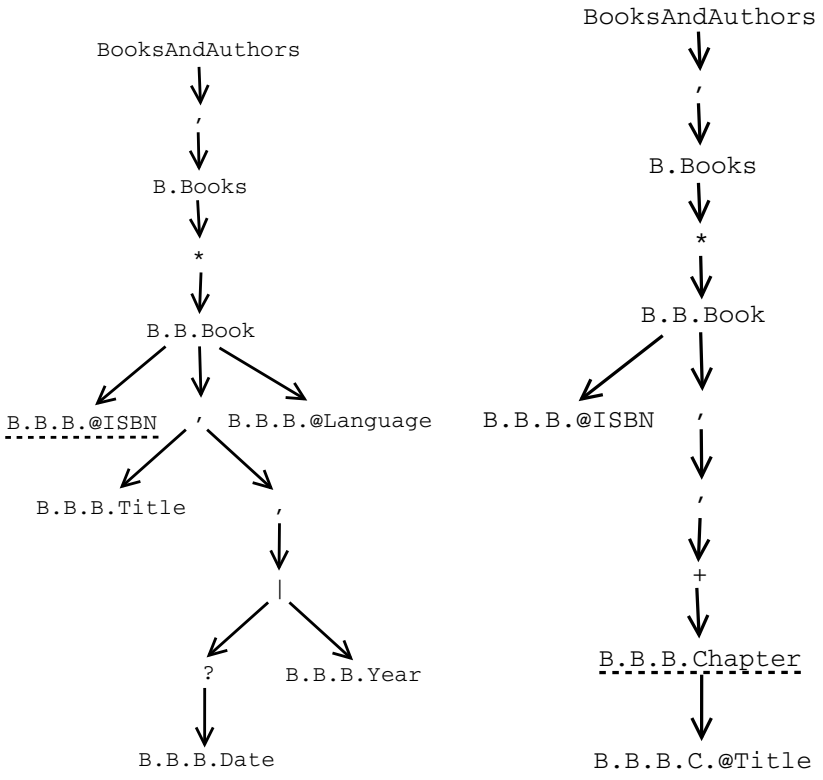


Figuur D.2: Resultaten van matchen van DTD graaf met de eerste twee STORED queries voor het uitgewerkte voorbeeld

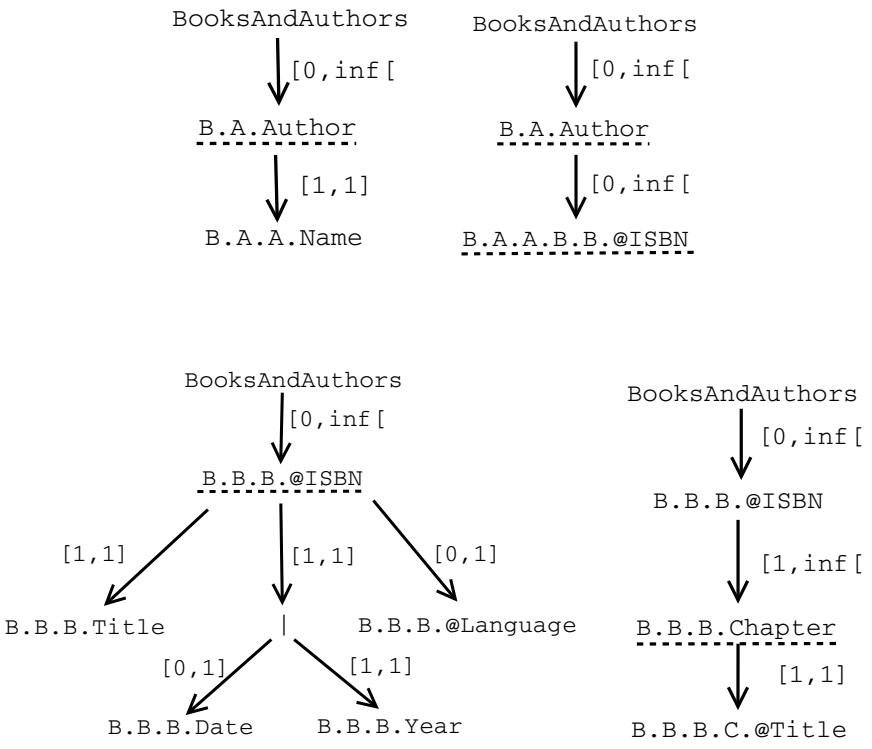
voorouders van een knoop ingekort tot één letter. Uit de figuur kan u afleiden waarvan deze letters afkomstig zijn. De elementen die tot de primaire sleutel behoren, zijn onderlijnd.

De volgende stap is het herschrijven van de verschillende DTD grafen naar geannoteerde DTD grafen. Het resultaat hiervan vindt u in figuur D.4.

De laatste stap in het zoeken naar de choice constraints is het bepalen van het SQL statement. We zien dat we enkel voor de derde STORED query een choice constraint moeten uitdrukken. Deze wordt als volgt bepaald:



Figuur D.3: Resultaten van matchen van DTD graaf met de laatste twee STORED queries voor het uitgewerkte voorbeeld



Figuur D.4: Resultaten van annotateGraph voor het uitgewerkte voorbeeld

$$\begin{aligned}
\chi(\texttt{B.B.B.Date? | B.B.B.Year}) &= (\sigma(\texttt{B.B.B.Date?})\times\mu(\texttt{B.B.B.Year})) + \\
&\quad (\mu(\texttt{B.B.B.Date?})\times\sigma(\texttt{B.B.B.Year})) \\
&= ((\alpha(\texttt{B.B.B.Date}) + \beta(\texttt{B.B.B.Date})) \\
&\quad \times(\beta(\texttt{B.B.B.Year}))) + \\
&\quad (\beta(\texttt{B.B.B.Date})\times\alpha(\texttt{B.B.B.Year})) \\
&= ((\texttt{B.B.B.Date} + \neg\ \texttt{B.B.B.Date}) \\
&\quad \times \neg\texttt{B.B.B.Year}) + \\
&\quad (\neg\texttt{B.B.B.Date} \times \texttt{B.B.B.Year})
\end{aligned}$$

We vertrekken hiervan om volgende verdere herschrijvingen te doen:

$$\begin{aligned}
\chi_1 &= \zeta((\texttt{B.B.B.Date} + \neg\texttt{B.B.B.Date}) \times \neg\texttt{B.B.B.Year}) + \\
&\quad \zeta(\neg\texttt{B.B.B.Date} \times \texttt{B.B.B.Year}) \\
&= \zeta(\texttt{B.B.B.Date} \times \neg\texttt{B.B.B.Year}) + \\
&\quad \zeta(\neg\texttt{B.B.B.Date} \times \neg\texttt{B.B.B.Year}) + \\
&\quad \zeta(\neg\texttt{B.B.B.Date} \times \texttt{B.B.B.Year}) \\
&= (\texttt{B.B.B.Date} \times \neg\texttt{B.B.B.Year}) + \\
&\quad (\neg\texttt{B.B.B.Date} \times \neg\texttt{B.B.B.Year}) + \\
&\quad (\neg\texttt{B.B.B.Date} \times \texttt{B.B.B.Year}) \\
\pi &= \gamma(\texttt{B.B.B.Date} \times \neg\texttt{B.B.B.Year}) + \\
&\quad \gamma(\neg\texttt{B.B.B.Date} \times \neg\texttt{B.B.B.Year}) + \\
&\quad \gamma(\neg\texttt{B.B.B.Date} \times \texttt{B.B.B.Year}) \\
&= (\texttt{B.B.B.Date}, \neg\texttt{B.B.B.Year}) + \\
&\quad (\neg\texttt{B.B.B.Date}, \neg\texttt{B.B.B.Year}) + \\
&\quad (\texttt{B.B.B.Year}, \neg\texttt{B.B.B.Date})
\end{aligned}$$

Als we dan tot slot deze uitdrukking omzetten naar een SQL statement (inclusief de controle om te zien of de wortel van de elementgraaf NULL is), dan bekomen we het volgende resultaat:

```
CHECK ((I IS NULL) OR
       (D IS NOT NULL AND Y IS NULL) OR
       (D IS NULL AND Y IS NULL) OR
       (Y IS NOT NULL AND D IS NULL))
```

Dit SQL statement wordt aan het CREATE TABLE statement van de tabel `Book` toegevoegd.

Inclusion Dependencies

De inclusion dependencies die we door de methode uit hoofdstuk 2 kunnen identificeren, zijn deze met als oorsprong object identifiers en attributen van

het type ID/IDREF.

Er zijn twee elementen uit het originele document waarvoor een object identifier bewaard wordt. Meer bepaald kunnen we volgende voorkomens vinden:
$R_1 = \tau(\texttt{BooksAndAuthors.Authors.Author}) = \{\texttt{Author, BookWritten}\}$
$R_2 = \tau(\texttt{BooksAndAuthors.Books.Book.Chapter}) = \{\texttt{Chapter}\}$
$T_1 = \{\texttt{Author}\}$
$T_2 = \{\}$

Merk op dat de tabel `BookWritten` niet in de verzameling T zit. Dit komt omdat niet '`B.A.Author`', maar `(B.A.Author, B.A.A.B.B.ISBN)` in deze relatie de primaire sleutel is. Aangezien T_2 een lege verzameling is, zal dit als resultaat hebben dat ω_2 een lege string is. Aangezien $(T_1 - R_1) = \{\texttt{BooksWritten}\}$, bepalen we enkel voor de tabel '`BooksWritten`' een SQL fragment om dit soort inclusion dependency uit te drukken. Bovendien merken we op dat $|T_1| = 1$. Het eindresultaat voor deze constraint is het volgende SQL fragment:

$\omega(\texttt{B.A.Author, BookWritten}) = \texttt{FOREIGN KEY A REFERENCES Author.A}$

De tweede en laatste soort inclusion dependency die we bepalen, is die voor attributen van het type ID/IDREF. Het enige attribuut dat bewaard wordt en als type ID heeft, is '`BooksAndAuthors.Books.Book.@ISBN`'. Dit attribuut wordt bewaard in de tabellen '`Book`' en '`Chapter`'. Er zijn dus twee tabellen waarvoor θ geen leeg resultaat oplevert, namelijk:

$\theta(\texttt{Book}) = \texttt{(SELECT B FROM Book)}$
$\theta(\texttt{Chapter}) = \texttt{(SELECT B FROM Chapter)}$

Het attribuut '`B.A.B.BookWritten.@ISBN`' is het enige attribuut van het type IDREF. Dit attribuut wordt enkel in de relatie '`BookWritten`' bewaard, waardoor we slechts één SQL statement moeten genereren om de ID/IDREF inclusion dependencies te karakteriseren.

```
ι(B.A.B.B.@ISBN, BookWritten) =
   CHECK (B IN ((SELECT B FROM Book) UNION
                (SELECT B FROM Chapter)))
```

Dit SQL fragment zullen we bij aan de CREATE TABLE statement voor de tabel `BookWritten` toevoegen.

Equality Generating Dependencies

Om equality generating dependencies (EGDs) te vinden binnen een relatie, moeten we voor elke tabel z'n geannoteerde DTD graaf inspecteren (nadat de keuze operatoren weggewerkt zijn). Voor elke twee elementen waartussen een pijl met cardinaliteit $[0, 1]$ of $[1, 1]$ loopt en waarvan de object identifier in het element waarin de tak aankomt, bewaard wordt, zullen we de EGD bepalen.

Er is slechts één element waarvoor een object identifier bewaard wordt, namelijk `B.B.B.Chapter`. Maar de enige tak die in dit element aankomt, heeft cardinaliteit $[1, \infty[$. Bijgevolg zullen we voor het uitwerken van κ steeds de lege string bekomen.

Tuple Generating Dependencies

Er zullen ook een aantal tuple generating dependencies (TGDs) uit de (geannoteerde) DTD grafen (zonder keuzeoperatoren) gehaald worden, waarbij we weer slechts binnen één relatie werken. Deze TGDs kunnen in twee soorten worden opgedeeld, namelijk child constraints en parent constraints.

Om child constraints te bepalen zullen we in de (geannoteerde) DTD grafen pijlen moeten zoeken met cardinaliteit $[1, 1]$ of $[1, \infty[$. De operator ξ zullen we laten werken op elk paar elementen (p, c) waarvoor vorige voorwaarde geldt. Het resultaat hiervan zetten we in de CREATE TABLE statement van de tabel in wiens DTD graaf we het koppel (p, c) vinden. Dit levert de volgende resultaten op:

ξ(`B.A.Author, B.A.A.Name, Author`) =
`N IS NOT NULL, PRIMARY KEY A`
ξ(`B.B.B.@ISBN, B.B.B.Title, Book`) =
`T IS NOT NULL, PRIMARY KEY B`
ξ(`B.B.B.@ISBN, B.B.B.Chapter, Chapter`) =
`CHECK((B IS NULL) OR (C IS NOT NULL))`
ξ(`B.B.B.Chapter, B.B.B.C.@Title, Chapter`) =
`T IS NOT NULL, PRIMARY KEY C`

De parent constraints zullen voor elke (geannoteerde) DTD graaf, waarin de keuzeknopen zijn weggewerkt, bepaald worden. Bijgevolg vinden we een heleboel parent constraints.

λ(`B.A.Author, B.A.A.Name, Author`) =
`CHECK ((N IS NULL) OR (A IS NOT NULL))`

```
λ(B.A.Author, B.A.A.B.B.@ISBN, BookWritten) =
   CHECK ((B IS NULL) OR (A IS NOT NULL))
λ(B.B.B.@ISBN, B.B.B.Title, Book) =
   CHECK ((T IS NULL) OR (B IS NOT NULL))
λ(B.B.B.@ISBN, B.B.B.Date, Book) =
   CHECK ((D IS NULL) OR (B IS NOT NULL))
λ(B.B.B.@ISBN, B.B.B.Year, Book) =
   CHECK ((Y IS NULL) OR (B IS NOT NULL))
λ(B.B.B.@ISBN, B.B.B.@Language, Book) =
   CHECK ((L IS NULL) OR (B IS NOT NULL))
λ(B.B.B.@ISBN, B.B.B.Chapter, Chapter) =
   CHECK ((C IS NULL) OR (B IS NOT NULL))
λ(B.B.B.Chapter, B.B.B.C.@Title, Chapter) =
   CHECK ((T IS NULL) OR (C IS NOT NULL))
```

D.2.3 SQL statements

Als we nu alle voorgaande constraints samen met de letterlijke vertaling van de STORED queries naar CREATE TABLE statements samenvoegen, dan krijgen we volgende SQL statements om de nodige relaties (met behoud van constraints) aan te maken:

```
CREATE DOMAIN LDomain (VALUE IN (English, Dutch))

CREATE TABLE Author (
   A TEXT NULL,
   N TEXT NULL,
   PRIMARY KEY A,
   N IS NOT NULL,
   CHECK ((N IS NULL) OR (A IS NOT NULL))
)

CREATE TABLE BookWritten (
   A TEXT NULL,
   B TEXT NULL,
   PRIMARY KEY (A, B),
   FOREIGN KEY A REFERENCES Author.A,
   CHECK (B IN ((SELECT B FROM Book) UNION
                (SELECT B FROM Chapter))),
```

```
   CHECK ((B IS NULL) OR (A IS NOT NULL))
)

CREATE TABLE Book (
   B TEXT NULL,
   T TEXT NULL,
   L LDomain NULL,
   D TEXT NULL,
   Y TEXT NULL,
   PRIMARY KEY B,
   CHECK ((C NULL AND T NULL) OR (C NOT NULL AND T NOT NULL)),
   CHECK ((I IS NULL) OR
          (D IS NOT NULL AND Y IS NULL) OR
          (D IS NULL AND Y IS NULL) OR
          (Y IS NOT NULL AND D IS NULL)),
   T IS NOT NULL,
   CHECK ((T IS NULL) OR (B IS NOT NULL)),
   CHECK ((D IS NULL) OR (B IS NOT NULL)),
   CHECK ((Y IS NULL) OR (B IS NOT NULL)),
   CHECK ((L IS NULL) OR (B IS NOT NULL))
)

CREATE TABLE Chapter (
   C TEXT NULL,
   T TEXT NULL,
   B TEXT NULL,
   PRIMARY KEY C,
   CHECK ((B IS NULL) OR (C IS NOT NULL)),
   T IS NOT NULL,
   CHECK ((C IS NULL) OR (B IS NOT NULL)),
   CHECK ((T IS NULL) OR (C IS NOT NULL))
)
```

Nadat we via deze SQL statements de nodige relaties hebben toegevoegd, gaan we ook de gegevens overzetten. Dit overzetten gebeurt via INSERT statements, welke vrij eenvoudig zijn op te stellen. De grote moeilijkheid om de gegevens in te voeren bestaat er echter in dat tijdens het invoeren van de gegevens de constraints steeds gecontroleerd worden, waardoor je de tupels van de verschillende tabellen in een weloverwogen volgorde moet invoeren. Zo mogen de tupels in de tabel '`BookWritten`' pas toegevoegd worden, nadat

de tupels in '`Book`' of '`Chapter`' zijn toegevoegd. Een mogelijke volgorde voor het toevoegen van de gegevens is de volgende:

```
INSERT INTO Author(A, N) VALUES ('1', 'Ginsberg')
INSERT INTO Author(A, N) VALUES ('2', 'Tanenbaum')
INSERT INTO Author(A, N) VALUES ('3', 'Woodhull')
INSERT INTO Book(B, T, L, D, Y) VALUES ('0136386776',
  'Operating Systems: Design and Implementation (Second Edition)',
  'English', NULL, '1997')
INSERT INTO Book(B, T, L, D, Y) VALUES ('0130888931',
  'Distributed Systems: Principles and Paradigms',
  'English', NULL, NULL)
INSERT INTO BookWritten(A, B) VALUES ('2', '0136386776')
INSERT INTO BookWritten(A, B) VALUES ('2', '0130888931')
INSERT INTO BookWritten(A, B) VALUES ('3', '0136386776')
INSERT INTO Chapter(C, T, B) VALUES ('4', 'Introduction',
  '0136386776')
INSERT INTO Chapter(C, T, B) VALUES ('5', 'Processes',
  '0136386776')
INSERT INTO Chapter(C, T, B) VALUES ('6', 'Input/Output',
  '0136386776')
INSERT INTO Chapter(C, T, B) VALUES ('7', 'Memory Management',
  '0136386776')
INSERT INTO Chapter(C, T, B) VALUES ('8', 'File Systems',
  '0136386776')
INSERT INTO Chapter(C, T, B) VALUES ('9',
  'Reading List and Bibliography', '0136386776')
INSERT INTO Chapter(C, T, B) VALUES ('10',
  'Key principles of distributed systems', '0130888931')
INSERT INTO Chapter(C, T, B) VALUES ('11',
  'Real-world distributed case studies', '0130888931')
```

Wanneer we deze INSERT statements hebben uitgevoerd, bekomen we uiteindelijk de set relaties die we beoogden. Het resultaat hiervan, ziet u in figuur D.5.

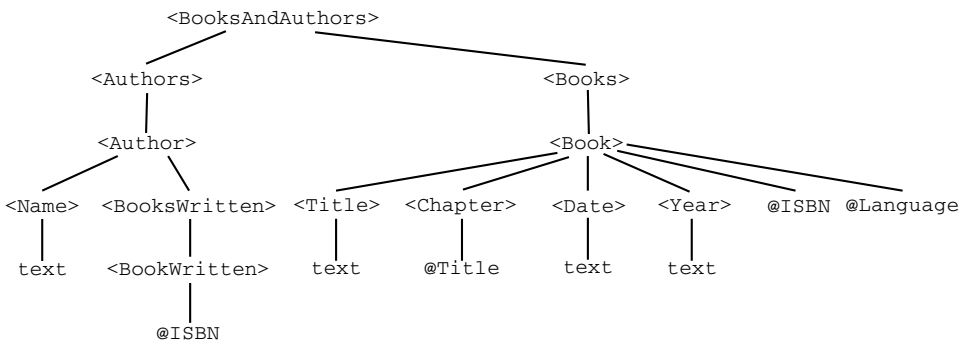
Author	*A*	*N*
	1	Ginsberg
	2	Tanenbaum
	3	Woodhull

BookWritten	*A*	*B*
	2	0136386776
	2	0130888931
	3	0136386776

Book	*B*	*T*	*L*	*D*	*Y*
	0136386776	Operating Systems...	English	`NULL`	1997
	0130888931	Distributed Systems...	English	`NULL`	`NULL`

Chapter	*C*	*T*	*B*
	4	Introduction	0136386776
	5	Processes	0136386776
	6	Input/Output	0136386776
	7	Memory Management	0136386776
	8	File Systems	0136386776
	9	Reading List and Bibliography	0136386776
	10	Key principles of distributed systems	0130888931
	11	Real-world distributed case studies	0130888931

Figuur D.5: Relaties voor het voorbeeld



Figuur D.6: XML labels voor voorbeeld

D.3 Creatie van public view forest

Nu we al onze gegevens hebben bewaard in relaties, moeten we de gebruikers van een XML view voorzien. Dit doen we door een public view forest te construeren, zoals we dit gezien hebben in hoofdstuk 5. We zullen eerst de XML labels aanmaken, zodat we een idee hebben over de vorm van de public view forest. Vervolgens bepalen we de SQL fragmenten in de bladeren en tenslotte de SQL fragmenten in de interne knopen.

D.3.1 XML labels

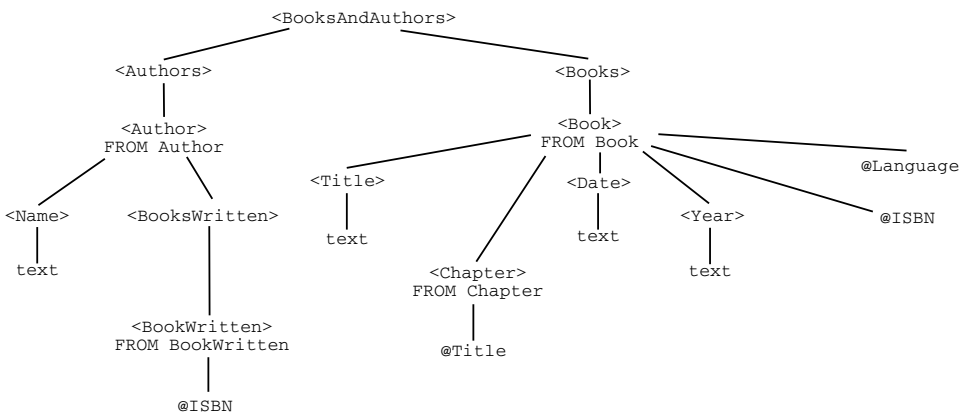
Het bepalen van de XML labels kunnen we rechtstreeks uit de DTD. Het resultaat hiervan vindt u in figuur D.6.

D.3.2 SQL fragmenten

Nadat we de public view forest z'n vorm hebben gegeven door de XML labels te bepalen, zullen we de SQL fragmenten bepalen. Hiervoor maken we gebruik van de DTD grafen uit figuren D.2 en D.3.

FROM clauses

We beginnen met het bekijken van de gematchte DTD graaf voor de tabel '`Author`'. Er is in deze DTD graaf slechts één pad p dat we moeten bekijken. De sleutelwaarde is '`B.A.Author`', waardoor we te weten komen dat k de



Figuur D.7: View forest na bepalen FROM clauses

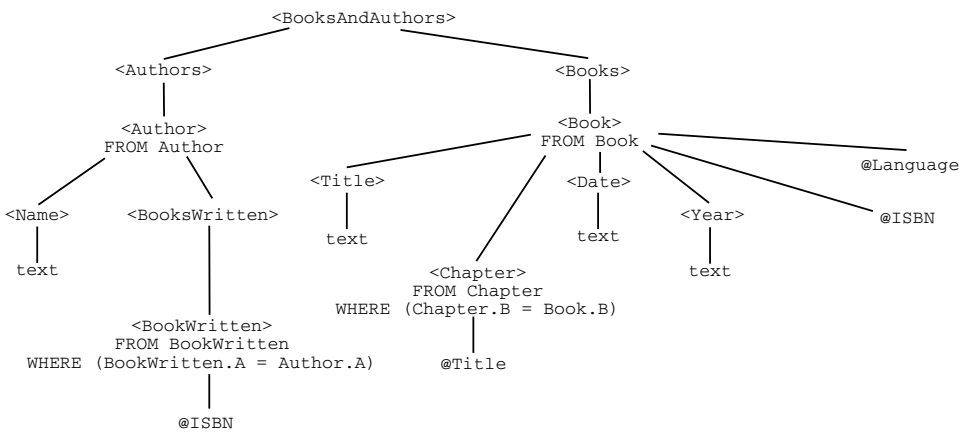
enige '*' knoop is uit de DTD graaf. Bijgevolg komen we zo ook te weten dat '$E = \{$`B.A.Author`$\}$' en voegen we de tabel '`Author`' toe aan de knoop van '`B.A.Author`' in de public view forest.

Vervolgens beschouwen we de DTD graaf voor de tabel '`BookWritten`'. We zien weeral slechts één pad, en vinden dat '$E = \{$`B.A.A.B.BookWritten`$\}$' (want k is de onderste '*' knoop). Bijgevolg voegen we '`BookWritten`' toe aan de FROM clause van de knoop '`B.A.A.B.BookWritten`'.

De meest complexe DTD graaf is ongetwijfeld die van de tabel '`Book`'. We vinden hierin namelijk verschillende paden, maar op slechts één pad ligt de primaire sleutel. Bijgevolg moeten we enkel dit pad beschouwen, aangezien we anders steeds geen k kunnen bepalen, en E dus ook leeg is. We vinden dat '$E = \{$`B.B.Book`$\}$' en voegen dus aan de knoop die hiermee overeenkomt de tabel '`Book`' aan de FROM clause toe.

Tenslotte bekijken we nog de laatste DTD graaf (voor de tabel '`Chapter`'). Aangezien in de DTD grafen in figuur D.3 nog niet de '+' operatoren zijn weggewerkt, moeten we dit eerst even doen. Als we dit doen, vinden we '$E =$ $\{$`B.B.B.Chapter`$\}$'. In de knoop "`<Chapter>`" voegen we dus '`Chapter`' toe aan de FROM clause.

Het resultaat van al deze toevoegingen vindt u in figuur D.7.



Figuur D.8: View forest na bepalen WHERE clauses

WHERE clauses

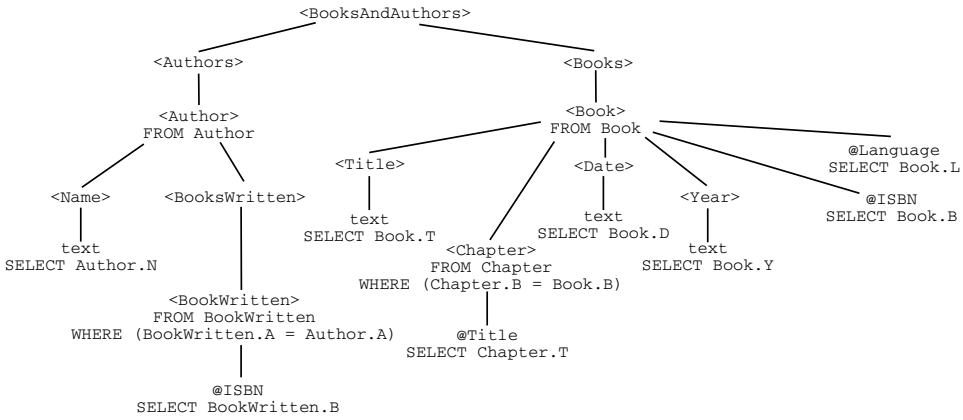
Er zijn vier knopen met een niet lege FROM clause, namelijk "`<Author>`", "`<BookWritten>`", "`<Book>`" en "`<Chapter>`". Enkel de tweede en vierde hebben een voorouder met een niet lege FROM clause.

Voor de knoop "`<BookWritten>`" vinden we dat '`B.A.Author`' zowel in de tabel '`Author`' als '`BookWritten`' bewaard wordt.
Bijgevolg is '$L = \{$`BookWritten.A = Author.A`$\}$'. De WHERE clause van de knoop "`<BookWritten>`" is dus bijgevolg '`WHERE (BookWritten.A = Author.A)`'.
De knoop "`<Chapter>`" zal op een gelijkaardige wijze de WHERE clause '`WHERE (Chapter.B = Book.B)`' krijgen.

Wanneer we al deze resultaten in de public view forest invullen, dan bekomen we de view forest uit figuur D.8.

SELECT clauses

In de laatste fase van het maken van de SQL fragmenten van de public view forest, introduceren we de SELECT clauses. Voor alle bladen is dit voor de hand liggend, behalve voor '`@ISBN`', aangezien dit in twee tabellen bewaard wordt. Aangezien echter enkel de tabel '`Book`' in de FROM clause zit van een voorouder, zetten we als SQL fragment voor het blad met XML label '`@ISBN`' de expressie '`SELECT Book.B`'. Het resultaat dat we nu bekomen, is het eindresultaat. Dit vindt u in figuur D.9.



Figuur D.9: Resultaat voor public view forest

D.3.3 XML view bepalen

Nu we de public view forest bepaald hebben, trachten we om de public XML view te construeren. Hoe dit gebeurt, is reeds beschreven in hoofdstuk 4. De SQL queries voor de verschillende knopen zijn de volgende:

$$
\begin{aligned}
R_B &= \textit{empty} \\
&= R_{B.A} \\
&= R_{B.B}
\end{aligned}
$$

$$
\begin{aligned}
R_{B.A.A} &= \texttt{SELECT *} \\
&\quad\ \texttt{FROM Author} \\
&= R_{B.A.A.N} \\
&= R_{B.A.A.B}
\end{aligned}
$$

$$
\begin{aligned}
R_{B.B.B} &= \texttt{SELECT *} \\
&\quad\ \texttt{FROM Book} \\
&= R_{B.B.B.T} \\
&= R_{B.B.B.D} \\
&= R_{B.B.B.Y}
\end{aligned}
$$

$$
\begin{aligned}
R_{B.A.A.B.B} &= \texttt{SELECT *} \\
&\quad\ \texttt{FROM Author, BookWritten} \\
&\quad\ \texttt{WHERE (BookWritten.A = Author.A)}
\end{aligned}
$$

$R_{B.B.B.C}$ =
```
SELECT *
FROM Book, Chapter
WHERE (Chapter.B = Book.B)
```

$R_{B.A.A.N.t}$ =
```
SELECT Author.N
FROM Author
```

$R_{B.A.A.B.B.I}$ =
```
SELECT BookWritten.B
FROM Author, BookWritten
WHERE (BookWritten.A = Author.A)
```

$R_{B.B.B.T.t}$ =
```
SELECT Book.Title
FROM Book
```

$R_{B.B.B.C.T}$ =
```
SELECT Chapter.T
FROM Book, Chapter
WHERE (Chapter.B = Book.B)
```

$R_{B.B.B.D.t}$ =
```
SELECT Book.Date
FROM Book
```

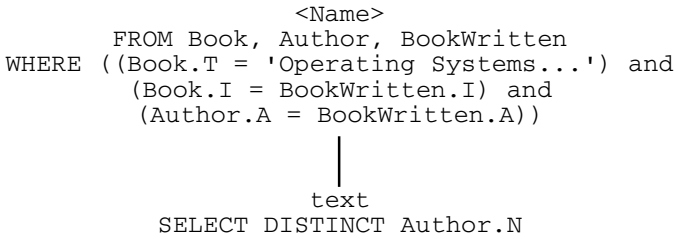
$R_{B.B.B.Y.t}$ =
```
SELECT Book.Year
FROM Book
```

$R_{B.B.B.L}$ =
```
SELECT Book.L
FROM Book
```

$R_{B.B.B.I}$ =
```
SELECT Book.B
FROM Book
```

We voeren al deze queries uit en het aantal lijnen in het resultaat zal overeenkomen met het aantal voorkomens van het desbetreffende element. Bijgevolg zullen er evenveel keer een "`<Name>`" element in de view voorkomen, als er namen van auteurs in de tabel `Author` zit.

Wanneer we volgens de regels uit hoofdstuk 4 te werk gaan, vinden we opnieuw het originele XML document, waarbij echter wel de volgorde gedeeltelijk verloren kan gaan. De XML view geeft dus een resultaat equivalent aan het originele XML document.



Figuur D.10: View forest na uitvoeren van query uit voorbeeld

D.4 Uitvoeren van een XQuery

We kunnen nu ook een query die gesteld wordt in XQuery uitvoeren op de XML view, door verschillende deelvragen te stellen in SQL op de verschillende relaties en deze antwoorden opnieuw te combineren tot een XML view, die dan als antwoord dient op de XQuery query. Stel dat we de namen van auteurs van het boek 'Operating Systems: Design and Implementation (second edition)' willen opvragen, dan doen we dit als volgt in XQuery:

```
distinct-values(
  for $b in $PublicView//Book,
      $a in $PublicView//Author,
      $w in $PublicView//BookWritten
  where (($b/Title = 'Operating Systems...') and
         ($b/@ISBN = $w/@ISBN) and
         ($w/Author = $a/Author))
  return $a/Name
)
```

Merk hierbij op dat we de titel niet volledig hebben geschreven om de query leesbaar te houden. Om nu het antwoord te bepalen, zetten we deze XQuery om naar SilkRoute's XQueryCore, alvorens de query met de public view forest te combineren. Het resultaat van deze combinatie levert ons een nieuwe view forest op, waarvan de evaluatie leidt tot een XML view op het antwoord. Deze view forest ziet u in figuur D.10.

D.5 Conclusie

Hiermee besluiten we dan ons globaal voorbeeld, waarin we verschillende aspecten van mappings tussen XML documenten en relationele databases hebben geïllustreerd. Hoewel het voorbeeld zelf vrij eenvoudig is, zou er aan de hand van dit voorbeeld toch duidelijk moeten geworden zijn hoe de verschillende mappings tot elkaar gerelateerd kunnen worden en gecombineerd tot één groot geheel.

Bijlage E

Implementatie: prototype

De implementatie van het zogenaamde prototype is volledig gebeurd in Objetive Caml. Deze taal is een functionele programmeertaal (afgeleid van ML), maar waarin ook een aantal features van imperatieve en object-georiënteerde programmeertalen ter beschikking zijn. De reden waarom we besloten hebben om voor deze programmeertaal te kiezen, is enerzijds dat Galax en de implementatie van SilkRoute die hieraan gekoppeld is, ook allebei geïmplementeerd zijn in deze taal, en anderzijds dat het maken van herschrijfregels vrij natuurlijk zou moeten overkomen. Spijtig genoeg is gebleken dat dit laatste niet betekent dat het eenvoudig is de regels te implementeren, vooral aangezien ik nog geen ervaring hiervoor had met functioneel programmeren, wat toch een andere wijze van redeneren vraagt.

In het vervolg van deze bijlage gaan we bespreken hoe de implementatie zelf in elkaar zit en wat de beperkingen en mogelijkheden zijn t.o.v. de theoretische uitleg uit hoofdstuk 2.

E.1 Algemene structuur

Het prototype toont hoe we aan de hand van een STORED query en een DTD een aantal relaties kunnen aanmaken, waarbij de choice constraints (cf. sectie 2.4) bewaard blijven. Andere constraints die we hebben beschreven in hoofdstuk 2 kunnen hier later eventueel nog bij aan worden toegevoegd.

We kunnen een vijf grote fasen onderscheiden in de implementatie van het prototype. Deze zijn de volgende:

- Parsen van DTD en STORED queries
- Pattern matchen van DTD graaf en STORED queries
- Uitvoeren van migrateChoice algoritme
- Zoeken van de choice constraints
- De SQL statements genereren

Deze 5 zaken zullen we in de volgende secties bekijken. De implementatie zelf bestaat uit een aantal files. In figuur E.1 vindt u welke bestanden overeenkomen met welk onderdeel van de implementatie. De files met extensie `.ml` zijn O'Caml programma's, terwijl die met extensie `.mli` interface bestanden zijn.

E.2 Parsers

De parsers voor de DTD's en STORED queries zijn geschreven door middel van de tools ocamllex en ocamlyacc. De werking van deze tools is analoog aan die van lex en yacc, zoals we die in unix omgevingen vaak zien. Ocamllex zal de lexical analysis doen, terwijl telkens iets is ingelezen een token wordt doorgegeven aan de feitelijke parser, dewelke door ocamlyacc wordt aangemaakt. Op de details van de parsers zelf gaan we niet ingaan. Voor meer informatie hierrond, verwijzen we de lezer naar [35] en [21].

Het programma ocamllex zet de regels die in een `.mll` bestand staan om naar een `.ml` bestand, dat een O'Caml programma bevat. Het programma ocamlyacc zal een `.mly` bestand omzetten in enerzijds een `.ml` bestand met de implementatie van de parser en anderzijds een `.mli` bestand dat de interface van de parser beschrijft. Hierin komen onder andere de tokens die de parser van de lexer moet krijgen, waardoor we dus eerst deze interface nodig hebben, alvorens we de lexer kunnen gaan compileren.

Voor STORED queries zullen we na het parsen de gegevens in instanties van het type `storedQuery` zetten. Dit type is als volgt gedefinieerd:

```
type storedQuery =
  {
    variables: varEntry list;
    relation: string;
    attributes: varName list;
```

`Makefile`	Indien u make uitvoert in een directory waar al deze bestanden staan, dan worden de executables aan de hand van deze regels gemaakt.
`choiceconstraints.ml(i)`	Deze module bepaalt adhv de geproceste DTD grafen (na `dsmatch`) de choice constraints.
`datamapping.ml`	Het hoofdprogramma dat de verschillende fases die moeten uitgevoerd worden zal oproepen.
`dsmatch.ml(i)`	Deze module doet de pattern matching tussen een DTD en een STORED query.
`dtd.ml`	Dit programma kan uitgevoerd worden om een DTD in te lezen en in een DTD graaf te zetten.
`dtdgraph.ml(i)`	Hier wordt de DTD graaf gedefinieerd, samen met enkele echte basis operaties.
`dtdgraphtools.ml(i)`	Hierin worden meer handige operaties voor DTD graphs gedefinieerd.
`dtdlexer.mll`	Deze file bevat de definitie voor de lexer voor DTDs.
`dtdparser.mly`	Dit bestand bevat alle herschrijfregels die in onze DTD parser zullen gebruikt worden.
`migratechoice.ml(i)`	Dit is de implementatie van het migrateChoice algoritme.
`stored.ml`	Dit programma kan uitgevoerd worden om een aantal STORED queries in te lezen en te parsen via de commandline
`storedlexer.mll`	Deze file bevat de definitie voor de lexer voor STORED queries.
`storedparser.mly`	Dit bestand bevat alle herschrijfregels die in onze STORED query parser zullen gebruikt worden.
`storedquery.ml(i)`	Hierin wordt de opslag structuur voor STORED queries gedefinieerd, samen met enkele operaties hierop.
`sqlgenerator.ml(i)`	Hierin worden de constraints met de rest van het schema gecombineerd.

Figuur E.1: Bestanden die bij de implementatie horen

```
    key: varName list;
  }
```

De lijst `variables` is een lijst met alle namen van variabelen, gekoppeld aan het pad (type `dotName`) dat we in deze variabelen steken. Merk op dat we enkel pad expressies bijhouden zonder daarbij rekening houden met de volgorde, hoewel we de implementatie kunnen uitbreiden om dit te ondersteunen. De variabele `attributes` bevat de lijst van namen van variabelen die we in de relatie gaan bewaren in de volgorde waarin deze in het `STORE` statement staan.

Een ingelezen DTD wordt in een DTD graaf gestoken. De definitie hiervan ziet er als volgt uit:

```
type vertex =
  {
    typename: string;
    tag: elementOrAttribute;
  }
;;

type edge =
  {
    source: elementName;
    destination: elementName;
  }
;;

type vertexList =
  Vertex of elementName * vertex * vertexList |
  NullVertex
;;

type edgeList =
  edge list
;;

type dtdGraph =
  {
    vertices: vertexList;
    edges: edgeList;
```

```
  }
;;
```

De structuur hiervan komt grotendeels overeen met die beschreven in sectie 2.1.2. We zullen dan ook niet verder hierop ingaan. Wel wensen we hier nog even te vermelden dat er een beperking inzit in het feit dat attributen niet dezelfde naam mogen hebben als andere attributen (van een ander element) of elementen. Deze beperking is echter niet eigen aan het algoritme, maar aan de implementatie, die we zo eenvoudig mogelijk trachten te houden.

Na het compileren kan u met de commando's `dtdparse` en `storedparse` beide invoerbestanden parsen. U kan deze commando's oproepen vanop de commandline, en ze verwachten hun input via standard input (`stdin`).

E.3 Pattern matching van DTD graaf en STORED query

De pattern matching van een DTD graaf en een STORED query, zoals beschreven in sectie 2.4.1, zal een nieuwe DTD graaf aanmaken per STORED query, die enkel de interessante delen van de volledige DTD graaf bevat. De functie die deze grafen gaat berekenen, is als volgt gedeclareerd:

```
val ds_match: Dtdgraph.dtdGraph -> Storedquery.storedQuery list
  -> Dtdgraph.dtdGraph list
```

Het matchen van een STORED query bestaat eruit om herhaaldelijk de elementboom voor een element dat een blad is in de nieuwe DTD graaf te bepalen in de originele DTD graaf, waarvoor er hier nakomelingen zijn die door de STORED query bewaard moeten worden. Telkens wanneer dit gebeurd is moeten elementknopen die bladeren zijn en waarvoor geen nakomelingen (incl. zichzelf) bewaard worden. Het bepalen van de elementboom gebeurt in `construction_part`, terwijl het verwijderen in de (recursieve) functie `remove_nonvariable_leafs` plaatsvindt.

E.4 MigrateChoice algoritme

Het migrateChoice algoritme werd in figuur 2.8 en sectie 2.4.2 beschreven. We zullen de DTD grafen omzetten zodat de choice operator zo hoog mogelijk wordt opgeschoven. De functie die we in O'Caml gedeclareerd hebben, heeft volgende typering:

```
val migrate_choice: Dtdgraph.dtdGraph list -> Dtdgraph.vertexList
  -> Dtdgraph.dtdGraph list
```

Ook hier volgt de implementatie vrij mooi de opzet van het algoritme, alhoewel er toch enkele complicaties bij kwamen zien door het feit dat het algoritme niet echt werkt met knopen.

E.5 Choice constraints zoeken

Wanneer we op deze plaats zijn gekomen van de implementatie, kunnen we voor het eerst in dit prototype echt op zoek gaan naar constraints. De operatoren die gedefinieerd staan in het algoritme van sectie 2.4 wordt hier mooi gevolgd, zoals u in `choiceconstraints.ml` kan zien. Per STORED query kunnen verschillende SQL zinnen die constraints voorstellen gegenereerd worden, wat zich ook in de declaratie van deze functie toont:

```
val choice_constraints: Dtdgraph.dtdGraph list
  -> Storedquery.storedQuery list -> string list
```

Een moeilijkheid was in dit geval om ervoor te zorgen dat je '*' knopen gaat negeren. Dit hebben we uiteindelijk opgelost met een functie die nagaat of er in de elementgraaf ergens zo'n knoop zit, alvorens de constraints te bepalen, opgelost. Ook de operaties ζ (`zeta`) en γ (`gamma`) waren niet eenvoudig te beschrijven, doordat we met een heleboel lijsten werkten waarin we een aantal structurele veranderingen moesten aanbrengen. Toch was dit waarschijnlijk eenvoudiger om in O'Caml te schrijven dan in C, maar het debuggen nam veel tijd in beslag in dit gedeelte van de implementatie.

E.6 Aanmaak van SQL statements

De constraints die in de `CREATE TABLE` queries komen, zijn reeds door de vorige fase bepaald. We zullen in deze fase dan vooral de `CREATE TABLE` query maken, en in de body hiervan het resultaat van de vorige fase inplakken. De functie die hiervoor wordt opgeroepen ziet er als volgt uit:

```
val generate_sql: Dtdgraph.dtdGraph -> Storedquery.storedQuery
  -> string -> string
```

Met het uitvoeren van deze fase, hebben we alle fases overlopen die in het prototype geïmplementeerd zijn. Merk op dat we de operator ρ, die padnamen omzet in namen van attributen in de relaties, niet geïmplementeerd hebben, waardoor de SQL statements niet rechtstreeks in te geven zijn in een relationele database. Door een eenvoudige zoeken/vervangen operatie kan u dit echter wel bewerkstelligen.

Bijlage F

Overzicht vertalingsoperaties

In deze bijlage zullen we de operaties die in hoofdstuk 2 gedefinieerd wordt kort overlopen.

Operator	Betekenis	Sectie	Pagina
α	Beeldt een expressie van de vorm `a[?]` af op een expressie die zegt dat deze expressie voorkomt.	2.4	39
β	Beeldt een expressie van de vorm `a[?]` af op een expressie die zegt dat deze expressie niet voorkomt.	2.4	39
γ	Bepaalt van een "genormaliseerde" expressie welke attributen nu wel en niet moeten vookomen volgens deze expressie.	2.4	40
δ	Bepaalt de domain constraint voor een attribuut die volgt uit het required of implied zijn van dit attribuut.	2.3	30
ϵ	Bepaalt de domain constraint voor een attribuut waarvoor er een opsomming (enumeration) is van de mogelijke waarden.	2.3	30
ζ	Herschrijft een expressie zodat de plustekens naar buiten worden gewerkt.	2.4	39
η	Geeft de lijst van mogelijke waarden voor een bepaald attribuut, indien deze waarden door een opsomming (enumeration) gegeven zijn.	2.3	30

Operator	**Betekenis**	**Sectie**	**Pagina**
θ	Geeft een lijst van alle IDs die in de opgegeven tabellen bewaard worden.	2.5	43
ι	Bepaalt de inclusion dependency voor een attribuut van het type IDREF.	2.5	43
κ	Bepaalt de equality generating dependencies binnen één tabel.	2.6	44
λ	Bepaalt de tuple generating dependencies die overeenkomen met parent constraints binnen één tabel.	2.7	47
μ	Beeldt een expressie waarin geen enkel elemenent, attributen, haakjes en '?' operatoren staan af op een expressie die zegt dat de eerste expressie niet voorkomt.	2.4	39
ν	Schrapt indien mogelijk bij voorkeur één keer een bepaald attribuut, anders een negatie van dit attribuut in een expressie.	2.4	40
ξ	Bepaalt de tuple generating dependencies die overeenkomen met child constraints binnen één tabel.	2.7	47
π	Resultaat van het uitvoeren van γ op de verschillende termen van een som.	2.4	40
ρ	Geeft de naam van een attribuut in een bepaalde tabel die overeenkomt met een gegeven element of attribuut.	2.1	25
σ	Beeldt een expressie waarin geen enkel elemenent, attributen, haakjes en '?' operatoren staan af op een expressie die zegt dat de eerste expressie voorkomt.	2.4	39
τ	Geeft alle tabellen terug waarin een bepaald element of attribuut bewaard wordt.	2.1	25
χ	Bepaalt een expressie die overeenkomt met een elementgraaf na het uitvoeren van het migrateChoice algoritme.	2.4	39
ω	Bepaalt de inclusion dependency die geïnduceerd worden door het gebruik van object identifiers in STORED queries.	2.5	42

Bibliografie

[1] Chaitanya K. Baru. XViews: XML Views of Relational Schemas. In *DEXA Workshop*, pages 700–705, 1999.

[2] Marijke Van Bergen. Algebra's voor XML Query Talen. Master's thesis, Universiteit Antwerpen, 2002.

[3] Ronald Bourret. Mapping DTDs to Databases, 1999.

[4] Ronald Bourret. Mapping W3C Schemas to Object Schemas to Relational Schemas, 1999.

[5] Michael J. Carey, Daniela Florescu, Zachary G. Ives, Ying Lu, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Publishing Object-Relational Data as XML. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.

[6] Michael J. Carey, Jerry Kiernan, Jayavel Shanmugasundaram, Eugene J. Shekita, and Subbu N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 646–648. Morgan Kaufmann, 2000.

[7] World Wide Web Consortium. XML Syntax for XQuery 1.0 (XQueryX), 2001.

[8] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics, 2002.

[9] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data in relations. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats.*, 1999.

[10] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, pages 431–442. ACM Press, 1999.

[11] Mary F. Fernandez, Wang-Chiew Tan, and Dan Suciu. SilkRoute: Trading between Relations and XML. In *Proceedings of Ninth International World Wide Web Conference*, 2000.

[12] "Mary Fernndez, Yana Kadiyska, Dan Suciu, Atsuyuki Morishima, and Wang-Chiew Tan". SilkRoute: A framework for publishing relational data in XML. *ACM Transactions on Database Systems (TODS)*, 27(4):438–493, 2002.

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

[14] Daniela Florescu and Donald Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[15] Juliana Freire and Jerome Simeon. Adaptive XML Shredding: Architecture, Implementation, and Challenges. In *EEXTT 2002*, pages 104–116, 2002.

[16] David Geleyn and Marijke Van Bergen. Een beknopte inleiding tot XML. Master's thesis, Universiteit Antwerpen, 2002.

[17] Meike Klettke and Holger Meyer. XML and Object-Relational Database Systems — Enhancing Structural Mappings Based on Statistics. *Lecture Notes in Computer Science*, 1997:151–??, 2001.

[18] Dongwon Lee and Wesley W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 323–338, 2000.

[19] Dongwon Lee and Wesley W. Chu. CPI: Constraints-Preserving Inlining algorithm for mapping XML DTD to relational schema. *Data Knowledge Engineering*, 39(1):3–25, 2001.

[20] Dongwon Lee, Murali Mani, Frank Chiu, and Wesley W. Chu. NeT and CoT: Translating Relational Schemas to XML Schemas using Semantic Constraints, 2001.

[21] Xavier Leroy. *The Objective Caml system (release 3.06): Documentation and user's manual*. 2002.

[22] Philip M. Lewis, Arthur Bernstein, and Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*, chapter 17: XML and Web Data, pages 537–622. Addison Wesley, 2001.

[23] Murali Mani and Dongwon Lee. XML to Relational Conversion using Theory of Regular Tree Grammars. 2002.

[24] Iona Manolescu, Daniela Florescu, and Donald Kossmann. Pushing XML queries inside relational databases. *Tech. Report no. 4112, INRIA*, 2001.

[25] Dongwon Lee Murali. Effective Schema Conversions between XML and Relational Models, 2002.

[26] Igor Nekrestyanov, Boris Novikov, and Ekaterina Pavlova. An Analysis of Alternative Methods for Storing Semistructured Data in Relations. In *ADBIS-DASFAA*, pages 354–361, 2000".

[27] Jan Paredaens, Toon Calders, Stijn Dekeyser, and Jan Hidders. Cursus Databases III: Objectgeorienteerde Databases. 2002.

[28] Albrecht Schmidt, Martin Kersten, Menzo Windhouwer, and Florian Waas. Efficient Relational Storage and Retrieval of XML Documents. *Lecture Notes in Computer Science*, 1997:137–??, 2001.

[29] Jayavel Shanmugasundaram, Jerry Kiernan, Eugene J. Shekita, Catalina Fan, and John Funderburk. Querying XML Views of Relational Data. In *The VLDB Journal*, pages 261–270, 2001.

[30] Jayavel Shanmugasundaram, Eugene J. Shekita, Jerry Kiernan, Rajasekar Krishnamurthy, Stratis Viglas, Jeffrey F. Naughton, and Igor Tatarinov. A General Techniques for Querying XML Documents using a Relational Database System. *SIGMOD Record*, 30(3):20–26, 2001.

[31] 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

VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK, pages 302–314. Morgan Kaufmann, 1999.

[32] Takeyuki Shimura, Masatoshi Yoshikawa, and Shunsuke Uemura. Storage and Retrieval of XML Documents Using Object-Relational Databases. In *Database and Expert Systems Applications*, pages 206–217, 1999.

[33] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems.* Computer Science Press, 1989.

[34] J.F.A.K. van Benthem, H.P. van Ditmarsch, J. Ketting, and W.P.M. Meyer-Viol. *Logica voor informatici.* Addison Wesley, 1994.

[35] Reinhard Wilhelm and Dieter Maurer. *Compiler Design.* Addison-Wesley, 1995.