Voorwoord

Bij deze neem ik graag de gelegenheid om een aantal mensen te bedanken. Ten eerste wil ik graag mijn promotor Prof. Dr. Frank Neven bedanken voor de begeleiding bij het maken van deze thesis. Bij hem kon ik altijd terecht met een vraag of een probleem. Ook Dirk Leinders wil ik graag bedanken voor zijn hulp bij problemen met DB2. Ook Wim Janssen verdient een dankwoord voor zijn hulp met LaTeX .

Een speciaal woord van dank gaat uit naar mijn ouders, die mij de mogelijkheid hebben gegeven om deze studies te volgen. Ook Sven Knapen wil ik bedanken voor het typen van een stuk van deze thesis. Deze mensen wil ik ook graag bedanken voor hun morele steun, alsook vrienden, familie en mijn medestudenten Wim, Ward, Erwin, Peter en Vanessa.

Samenvatting

Wanneer het om opslag en ondervraging van grote hoeveelheden XML-documenten gaat, is de XML-wereld verdeeld in twee kampen. Eén groep vindt dat XML-gegevens het best in relationele of object-relationele databases wordt opgeslagen. Uiteraard worden nog altijd XML-ondervragingstalen zoals XQuery gebruikt. Deze worden dan vertaald naar SQL, waarna de SQL op de relationele representatie van de XML-documenten wordt uitgevoerd. Deze aanpak heeft als voordeel dat de bestaande technologie ten volle benut kan worden. De andere groep is van mening dat de huidige systemen te weinig flexibiliteit toelaten en pleiten ervoor pure, oftewel native, XML-databasesystemen met eigen ondervragingstalen en indexsystemen te ontwikkelen.

Deze thesis bestaat erin de twee aanpakken met elkaar te vergelijken, zowel door een studie van de literatuur als door het testen van de systemen.

Inhoudsopgave

Hoofdstuk 1

XML en Databases

1.1 Inleiding

Deze thesis is als volgt georganiseerd. De rest van dit hoofdstuk geeft een korte inleiding over het belang van XML in databases. In het tweede hoofdstuk wordt elke soort database in het kort ingeleid. Hierin komen de zwakten en de sterkten van elk soort database aan bod. In hoofdstuk drie zijn de native databases aan de beurt. Hierin worden eerst de theoretische ideeën besproken zoals fysieke opslagstrategieën, indexstructuren en het beheer van transacties. Daarna worden enkele systemen onder de loep genomen. Het vierde hoofdstuk grijpt terug naar relationele databases. Hierin worden ook eerst de theoretische ideeën uit de doeken gedaan. Zo wordt er over opslagstrategieën en indexen gehandeld, maar ook over manieren om ondervragingen in XQuery en XSLT om te zetten naar SQL. Ook worden hier de bestaande relationele systemen besproken die XML-gegevens ondersteunen. Vervolgens wordt in het kort het topic XML-publishing aangeraakt. In hoofdstuk vijf komen de tests aan bod. Hierin wordt eerst uitgelegd wat getest zal worden, waarom en hoe. Daarna wordt het testen van native en relationele systemen besproken en daarna wordt een kort overzicht gegeven van de tests. In hoofdstuk zes zal een korte conclusie geformuleerd worden.

1.2 Wat is XML

XML[27, 28] staat voor Extensible Markup Language. Het is een flexiebel tekstformaat, afgeleid van SGML en ontworpen om gegevens te bevatten en beschrijven. Dit formaat beschrijft gegevens in een boomstructuur. Het formaat bestaat uit tags en tekstuele gegevens. De tags in XML zijn niet voorgedefinieerd, de gebruiker maakt zijn eigen tags.

Een XML-document bevat een structuur die gedefinieerd is door de tags. Omdat het aantal tags en het nesten ervan door de gebruiker mag gekozen worden, wordt gezegd dat XML-gegevens semi-gestructureerd is. Een XML-document heeft een structuur, maar twee verschillende XML-documenten hebben doorgaans niet dezelfde structuur. Gestructureerde gegevens zijn gegevens die in een tabel gegoten kunnen worden en dus een rigide structuur hebben. Elk gegeven heeft door de structuur ook een stricte annotatie. Ongestructureerde gegevens zijn gegevens die geen structuur bevatten. Voorbeelden hiervan zijn gewone tekstbestanden of bestanden die figuren beschrijven. Semi-gestructureerde gegevens zijn een middenweg tussen de twee. De XML-gegevens hebben geen rigide structuur, maar er is wel een structuur die van bestand tot bestand afhangt.

De structuur van een XML-bestand kan beschreven worden door een schema. Een schemadocument wordt opgesteld in een schemataal. De twee meest bekende schematalen van het moment zijn ongetwijfeld DTD en XML Schema. Het bestaan van een schema van de XML-gegevens is niet verplicht. Voor meer informatie over DTD, zie [1]. Voor meer informatie over XML Schema, zie [30].

XML-gegevens kunnen ondervraagd worden. Niet alleen de tekstuele inhoud tussen de tags, maar ook de structuur en de tags zelf mogen ondervraagd worden. Een ondervraging op XML-gegevens wordt een query genoemd. Zo een query wordt opgesteld in een ondervragingstaal. Een gangbare ondervragingstaal is XQuery. Voor meer informatie over XQuery, zie [29].

1.3 Belang van XML

XML is een wijdverspreide en geaccepteerde standaard. XML wordt vooral gebruikt bij gegevensuitwisseling tussen verschillende systemen. Door gegevens in een XML-formaat uit te wisselen, kan de incompatibiliteit van verschillende systemen overbrugd worden. De dag van vandaag is er een groot aantal bedrijven dat XML-gebaseerde Webdiensten voorziet om gegevens te publiceren. Net omdat XML zo wijdverspreid is en omdat er ondertussen heel wat XML-gegevens bestaan, is het niet mogelijk om XML als gegevensformaat te negeren.

1.4 Waarom XML in Databases?

Een belangrijke vraag die men zich kan stellen is waarom men XML-gegevens in een database wil plaatsen. Bij alle vormen van gegevens speelt er zich een evolutie af in de manier waarop deze gestockeerd worden. In het begin zijn er weinig gegevens voorhanden, dus worden die opgeslagen in enkele bestanden in een map op harde schijf. Na een tijdje wordt het aantal bestanden redelijk groot en moet er structuur komen in de manier van opslaan om dat bestand nog terug te vinden. Uiteindelijk wordt de hoeveelheid gegevens zo groot dat zelfs een goede mappenstructuur niet meer kan helpen. Dan wordt er een toevlucht tot databases genomen. De gegevens worden uit de bestanden gehaald en in de database geplaatst. Het zoeken in de enorme berg gegevens wordt gemakkelijk gemaakt door het DBMS, Database Management System. Dit is een programma dat het mogelijk maakt een database te creëren, updaten en beheren. Deze zorgt dus voor een interface om de gegevens zo efficiënt mogelijk aan te passen en te ondervragen, zonder dat de gebruiker zich zorgen maakt over plaatsing van de gegevens.

Ook bij XML heeft deze evolutie zich afgespeeld. Het aantal XML-gegevens is nu zo groot geworden dat er nood is aan een DBMS dat XML opslaat en efficiënt ondervraagt. Het volume aan XML-gegevens is namelijk zo groot geworden dat het opslaan in bestanden en ondervragen met gewone XPath of XQuery verwerkers niet meer voldoende efficiënt is.

XML-gegevens zijn verschillend van gegevens die voorheen in databases ondergebracht werden omdat XML-gegevens semi-gestructureerde gegevens zijn.

1.5 Belangrijke aspecten

Bij het kiezen van een DBMS zijn er een aantal belangrijke eigenschappen die in overweging genomen dienen te worden.

1.5.1 XML-gegevens

Ten eerste is het belangrijk hoe de XML-gegevens die opgeslagen zullen worden eruit zien. Zijn het gegevens met een klein of groot schema, is het schema vast of veranderlijk? Als de gegevens een klein schema hebben, komen er veel dezelfde tags voor en is de structuur nog redelijk rigide. Als de gegevens daarentegen een groot schema hebben zijn er veel meer tags te vinden en de structuur is al complexer. Het schema kan veranderlijk zijn maar ook vast, hetgeen belangrijk is om weten om een goede keuze te maken van DBMS.

1.5.2 Query behoeften

Het is ook van vitaal belang om weten welke queries uiteindelijk op het systeem zullen losgelaten worden. Zullen er vooral queries op de structuur gesteld worden „geef alle afstammelingen van `<item>`", of eerder op de inhoud „geef alle `<name>`'s"? En in het laatste geval kan er ook nog onderscheid gemaakt worden naar de aard van de queries. Zo zijn er queries die relationele queries uitdrukken, zoals „tel alle `<student>`-tags" of „geef alle `<title>`'s". Er zijn ook complexere queries die structuur en inhoud bevragen „geef alle `<name>`'s die afstammeling zijn van `<researcher>` en waarvan de sibling `<publications>` meer dan 3 `<items>` bevat".

Hoofdstuk 2

Bestaande Technologieën en Nieuwe Technologieën

2.1 Relationele databases

Een relationele database is een verzameling van gegevens, georganiseerd in tabellen. Het is mogelijk toegang te verkrijgen tot de gegevens, of de gegevens op verschillende manieren voor te stellen zonder de tabellen te veranderen. De relationele database is uitgevonden door E. F. Codd in 1970.

Voor relationele databases bestaan er relationele DBMS oftewel RDBMS. Omdat relationele databases en dus RDBMS al een hele tijd bestaan en veel bestudeerd zijn, zijn ze erg efficiënt geworden. Studies hebben ervoor gezorgd dat het plaatsgebruik van de gegevens in een relationele database tot een minimum beperkt wordt. Op dezelfde manier is ook gezorgd dat het ondervragen en aanpassen van de gegevens snel verloopt. Om het ondervragen van de gegevens sneller te laten verlopen, zijn er in de loop der tijd indexstructuren ontwikkeld. Om een query over de relationele database uit te drukken, gebruiken de meeste RDBMS Structured Query Language (SQL) als ondervragingstaal.

Om een database te kunnen gebruiken in een bedrijf of andere instelling zijn er een aantal dingen die in rekening gebracht moeten worden. In een bedrijf kan het mogelijk zijn dat twee of meerdere personen tegelijk toegang willen tot de gegevens. Als één persoon nu gegevens wil aanpassen, terwijl de andere ze wil lezen, kan het zijn dat de persoon die de gegevens wil lezen inconsistente gegevens als resultaat krijgt. Voor dit probleem zijn voor RDBMS methodes ontwikkeld, zodat de gegevens in de database ten alle tijde consistent zijn. Deze methodes worden methodes voor beheer van transacties genoemd.

Als het systeem waar het RDBMS op draait crasht in het midden van een aanpassing, kan het ook zijn dat er inconsistente gegevens in de database staat. Aangezien dit duidelijk niet gewenst is, zijn er recovery methodes ontwikkeld die ervoor zorgen dat wanneer het systeem terug draaiende is, de gegevens weer in een consistente staat worden gebracht en dat de gebruikers weten welke aanpassingen gebeurd zijn en welke niet. Voor een overzicht van zulke methodes, zie [35].

2.1.1 Voordelen

De voordelen van een relationeel DBMS zijn duidelijk. Door het jarenlange bestaan van deze systemen, zijn ze erg efficiënt. Ook methodes voor gelijktijdig gebruik en recovery methodes zijn optimaal. Er lijkt geen reden om een nieuw soort database en een daarbij horend DBMS te ontwikkelen.

2.1.2 Nadelen

Spijtig genoeg zijn de XML-gegevens die nu opgeslagen moeten worden niet gestructureerd, zoals reeds werd uitgelegd in Hoofdstuk 1. Ze zijn semi-gestructureerd. Het is dus niet mogelijk om XML-gegevens zonder meer in een relationele database te stoppen, zodat de informatie over één enkele knoop in een tupel af te lezen is.

Een XML-knoop kan namelijk een willekeurig aantal kinderen hebben, met willekeurig gekozen namen. Als er een schema van de XML-gegevens bestaat is er al meer informatie over de structuur van de XML-gegevens bekend, maar zoals vermeld in Hoofdstuk 1 is een schema niet verplicht. Over de gegevens is dus niets geweten, behalve dat ze een boomstructuur bevatten. Dit is niet genoeg informatie om de gegevens in een relationele database te stoppen. Er zijn twee mogelijkheden om de gegevens toch in een relationele database te plaatsen. Een eerste mogelijkheid is de semi-gestructureerde gegevens behandelen als ongestructureerde gegevens. Dit houdt in dat een volledig XML-document in één kolom en rij van de tabel geplaatst wordt. De gegevens worden opgeslagen als Character Large Object (CLOB). Dit zorgt er evenwel voor dat de gegevens ondervraagd worden met bestaande XQuery evaluators, hetgeen als gevolg heeft dat gegevens niet efficiënt ondervraagd worden. Het RDBMS laadt bij het ondervragen het XML-document volledig in het geheugen en het ondervragen gebeurt dan door een bestaande XQuery of XPath evaluator. Het grote probleem hiermee is dat hierdoor alle voordelen van een RDBMS niet gebruikt worden, omdat een volledig XML-document als eenheid van gegevens wordt bekeken. Een tweede mogelijkheid is om de semi-gestructureerde XML-gegevens in een

gestructureerde vorm te gieten. Dit „in een gestructureerde vorm gieten" wordt een mapping genoemd. Deze gestructureerde gegevens kunnen wel in een relationele database opgeslagen worden. Een overzicht van mappings wordt gegeven in Hoofdstuk 4.

2.2 Native databases

Native databases zijn ontstaan toen er nood was aan een database om XML op te slaan. Omdat er aan het opslaan van XML-gegevens in relationele databases toch wat nadelen verbonden bleken te zijn, waren er mensen die probeerden hiervoor een betere oplossing te vinden. Deze oplossing moest de semi-gestructureerde gegevens in eigen boomformaat kunnen opslaan, om zo de nadelen van het gebruik van een RDBMS te elimineren. Deze oplossing werd native database genoemd.

Een native database is dus een verzameling XML-gegevens die georganiseerd zijn in hun eigen boomstructuur. Net zoals bij relationele databases is het mogelijk om toegang te verkrijgen tot de gegevens of de gegevens op verschillende manieren voor te stellen zonder de boomstructuur van de gegevens te veranderen. Er bestaan native DBMS die werken met een native database.

Deze native DBMS bestaan slechts een korte tijd, dus er is nog niet zoveel onderzoek naar gedaan, maar de efficiëntie van native DBMS is aan het verbeteren door de vele studies die momenteel aan de gang zijn en al uitgevoerd zijn. Native DBMS zijn momenteel erg efficiënt als de gegevens in het geheugen passen, maar als de gegevens groter zijn, missen ze nog de ervaring en het onderzoek dat relationele DBMS hierin zo efficiënt hebben gemaakt.

Om queries op XML-gegevens uit te drukken wordt dikwijls XQuery gebruikt. Het is de „de facto standaard" ondervragingstaal voor XML-gegevens. XQuery gebruikt XPath expressies om doorheen de boomstructuur te navigeren. Toch is er geen standaard ondervragingstaal voor XML-gegevens. Native DBMS kunnen dus zelf ook een ondervragingstaal ontwerpen. Aangezien XQuery geen updates ondersteunt, maken native DBMS vaak een uitbreiding op XQuery.

Ook bij native DBMS worden vaak indexstructuren gebruikt om de queries sneller uit te voeren. Een overzicht van indexstructuren bij native DBMS wordt gegeven in sectie 3.2.

Omdat native DBMS nog in de kinderschoenen staan, zijn ook de methodes voor beheer van transacties nog niet erg ontwikkeld.

2.2.1 Voordelen

Doordat de native DBMS de XML-gegevens in eigen formaat opslaan, moet niet heel het XML-document in het geheugen geladen worden om meeste queries uit te voeren, mits een goede keuze van fysieke opslagstructuur. Hierdoor worden de nadelen van de relationele DBMS met XML-gegevens geëlimineerd.

2.2.2 Nadelen

Omdat de native DBMS nog niet zo lang bestaan, zijn het beheer van secundaire opslag, indexstructuren, methodes voor gelijktijdige toegang en recovery methodes nog niet optimaal.

Hoofdstuk 3

Native Databases – Bestaande mogelijkheden

3.1 Fysieke Opslagstrategieën

In alle relationele databases worden de gegevens op één manier opgeslagen, namelijk in tabellen. Native databases hebben als voornaamste kenmerk dat de semi-gestructureerde XML-gegevens in hun eigen boomformaat worden opgeslagen. Er kan een volgens [32] onderscheid gemaakt worden in manieren waarop native DBMS de gegevens opslaan.

Een goede keuze van opslagstrategie is belangrijk om de performantie van een relationeel DBMS te evenaren. Bij relationele DBMS is een pagina meestal de eenheid van gegevensoverdracht tussen het geheugen en de harde schijf. Het geheugen bevat buffers ter grootte van een pagina. Bij een aanvraag wordt de nodige pagina van de harde schijf gehaald en in een buffer van het geheugen geladen. Als de pagina niet meer nodig is in het geheugen kan ze terug naar de harde schijf geschreven worden. Door dit mechanisme moeten niet alle gegevens in het geheugen geladen worden, maar slechts de nodige stukken. Dit spaart schijftoegangen, die erg tijdrovend zijn. De gegevens worden bij relationele databases in tabellen georganiseerd. Een rij uit zo een tabel wordt een tupel genoemd.

De eenheid van opslag bij native databases is een record. Dit is vergelijkbaar met tupels in relationele databases. De eigenlijke inhoud van het record hangt af van de opslagstrategie. Een knoop in de boomstructuur van het XML-document wordt een knoop genoemd.

Records worden in blokken van vaste grootte in en uit het secundair geheugen gehaald. Deze blokken worden ook hier pagina's genoemd. Om te kunnen weten waar een bepaald record kan teruggevonden worden, moet er

worden bijgehouden in welke pagina een bepaald record zich bevindt en de offset in de pagina zelf. Dit kan gemakkelijk in een tabel bewaard worden.

Om te bepalen welke pagina's naar schijf worden geschreven en welke in het geheugen blijven, wordt meestal ofwel de Most Recently Used (MRU) of de Least Recently Used (LRU) methode gebruikt. De MRU methode schrijft de meest recent gebruikte pagina's terug naar de harde schijf als er een plaatstekort is in het geheugen. Deze methode gaat er van uit dat de pagina's die pas gebruikt zijn, niet meer nodig zullen zijn. De LRU methode schrijft de pagina die het langst niet meer gebruikt is terug naar schijf. Deze methode gaat er van uit dat de pagina's die het langst geleden zijn gebruikt niet meer nodig zullen zijn en dus best naar schijf geschreven kunnen worden, terwijl de pagina's die pas gebruikt zijn misschien nog opnieuw nodig zullen zijn voor het uitvoeren van de query. Ook de keuze van methode om te bepalen welke pagina's in geheugen blijven is belangrijk. Als een verkeerde methode gebruikt wordt, bestaat de kans dat er constant pagina's in en uit het geheugen geladen worden. Dit vertraagt het uitvoeren van de query aanzienlijk, aangezien er veel meer schijftoegangen nodig zijn.

Wanneer toegang tot een record gevraagd wordt, wordt een slot of lock op het record geplaatst. Er wordt bijgehouden hoeveel locks er op het record geplaatst zijn. Een pagina kan pas worden teruggeschreven naar de harde schijf als alle records in die pagina vrij zijn van sloten.

De rest van dit hoofdstuk wordt opgedeeld als volgt. Eerst worden de verschillende fysieke opslagstrategieën besproken, de invloed van het schema op de keuze van fysieke opslagstrategie en de fysieke opslagstrategie van Natix. Vervolgens worden de verschillende indexstructuren uitgelegd. Hierna wordt het beheer van transacties bij native DBMS behandeld en ten laatste worden er enkele bestaande native DBMS besproken.

In de rest van deze thesis zal het volgend lopend voorbeeld, Voorbeeld 3.1 gebruikt worden.

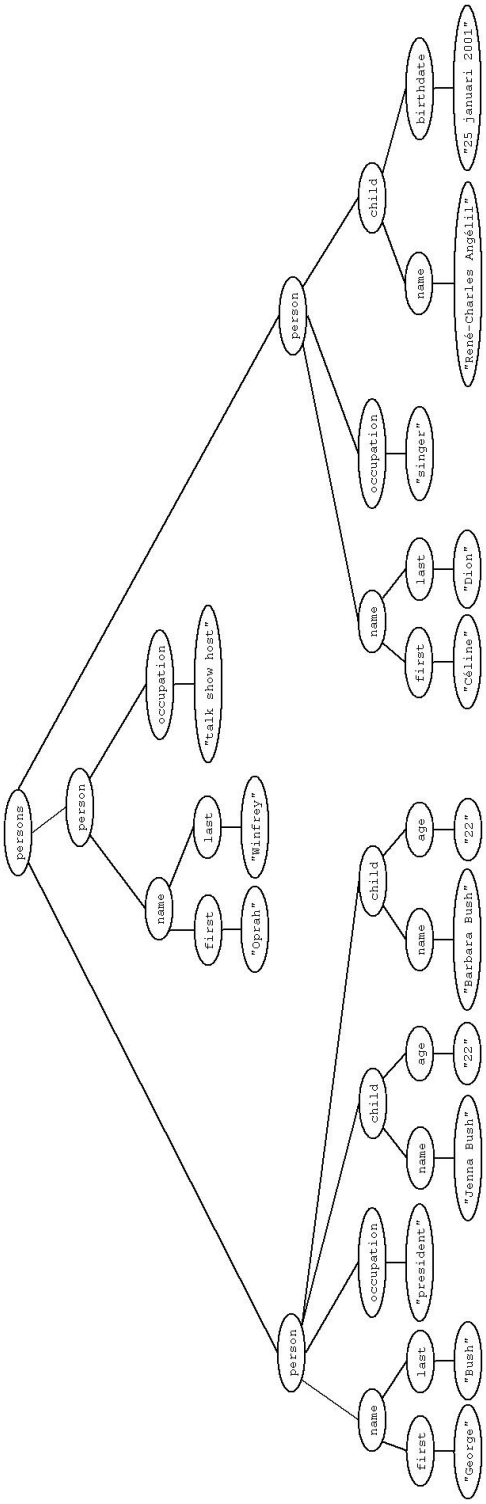
Voorbeeld 3.1. In Figuur 3.1 staat het XML-document voor dit voorbeeld. De root tag `persons` heeft drie `person` kinderen. Elke `person` heeft een `name`, een `occupation` en een aantal `child`-kinderen. Een `name` heeft op zijn beurt een `first` en een `last` en een `child` heeft op zijn beurt ook een `name` en een `age` of een `birthdate`.

Aangezien XML-gegevens een boomstructuur hebben, is het mogelijk om deze boomstructuur grafisch voor te stellen. De knopen van de boom zijn tags ofwel tekstuele inhoud. Tekstuele inhoud is te onderscheiden aan de quotes. Figuur 3.2 geeft grafisch de boomstructuur weer van het XML-document in Figuur 3.1.

In Figuur 3.3 staat het schema van het XML-document in Figuur 3.1, in

```
<persons>
  <person>
    <name>
      <first>George</first>
      <last>Bush</last>
    </name>
    <occupation>president</occupation>
    <child>
      <name>Jenna Bush</name>
      <age>22</age>
    </child>
    <child>
      <name>Barbara Bush</name>
      <age>22</age>
    </child>
  </person>
  <person>
     <name>
       <first>Oprah</first>
       <last>Winfrey</last>
     </name>
     <occupation>talk show host</occupation>
  </person>
  <person>
      <name>
       <first>Celine</first>
       <last>Dion</last>
      </name>
      <occupation>singer</occupation>
      <child>
        <name>Rene-Charles Angelil</name>
        <age>4</age>
      </child>
  </person>
</persons>
```

Figuur 3.1: XML-document lopend voorbeeld



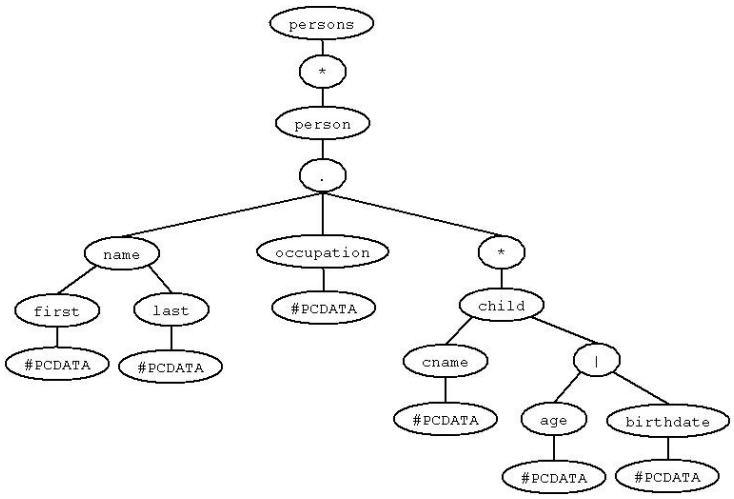
Figuur 3.2: Grafische boomvoorstelling lopend voorbeeld

```
<!ELEMENT persons    (person*)/>
<!ELEMENT person     (name,occupation,child*)/>
<!ELEMENT name       (#PCDATA|(first,last))/>
<!ELEMENT first      (#PCDATA)/>
<!ELEMENT last       (#PCDATA)/>
<!ELEMENT occupation (#PCDATA)/>
<!ELEMENT child      (name,(age|birthdate)/>
<!ELEMENT age        (#PCDATA)/>
<!ELEMENT birthdate  (#PCDATA)/>
```

Figuur 3.3: DTD lopend voorbeeld

DTD formaat.

Ook van het schemadocument kan een boomvoorstelling gemaakt worden. Deze boomvoorstelling wordt een schemagraaf genoemd. De knopen in deze schemagraaf bevatten ofwel tagnamen ofwel operatoren voor de reguliere expressies die in een DTD gebruikt worden. Deze operatoren zijn '.', '$*$', '|', '?' en '+'. Een grafische boomvoorstelling voor de DTD in Figuur 3.3 is te zien in Figuur 3.4.

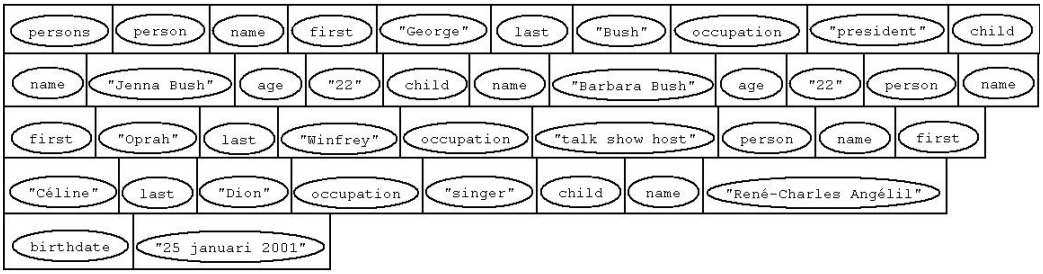


Figuur 3.4: Schema graaf lopend voorbeeld

▲

3.1.1 Depth-first Element-based

De eerste opslagstrategie is gemakkelijk te begrijpen. Elk element van het XML-document wordt in een record geplaatst en de records worden vervolgens in depth-first volgorde gestockeerd in een pagina. De records vormen tesamen een boomstructuur. Voorbeeld 3.2 geeft een voorbeeld van een stuk XML en hoe dat wordt opgeslagen, gebruik makend van de depth-first element-based opslagstrategie.



Figuur 3.5: Depth-first element-based opslagstrategie

Voorbeeld 3.2. De boom uit Figuur 3.2 wordt in depth-first manier doorlopen. Dit voorbeeld loopt in preorde doorheen de boom. Merk op dat ook in postorde of inorde doorheen de boom kan gelopen worden. Telkens een knoop voor het eerst tegengekomen wordt, wordt er een record gemaakt en de knoop wordt erin geplaatst. Het resultaat is een lange reeks records. Deze reeks records is afgebeeld in Figuur 3.5 Hoe deze records uiteindelijk in pagina's wordt onderverdeeld is hier niet voorgesteld, aangezien dit afhankelijk is van de grootte van de records en de grootte van de pagina's. ▲

Als er toegang tot een knoop wordt gevraagd, wordt de pagina met het record van die knoop in het geheugen geladen als die er nog niet in stond. Van zodra de knoop niet meer nodig is en er is plaats nodig in het geheugen, kan de pagina terug naar schijf worden geschreven.

Bij deze depth-first methode zouden depth-first path traversals zoals „geef het eerste kind van een bepaalde knoop" iets sneller kunnen gaan, aangezien de knopen die dicht bij elkaar staan in het XML-document ook dicht bij elkaar staan op een pagina. Spijtig genoeg zijn de meeste path traversals van een breadth-first aard zoals „doorzoek systematisch alle kinderen". Deze knopen staan dikwijls op andere paginas, vooral als de (sub)tree waarin gezocht wordt erg groot is.

Bij het invoegen van een nieuwe knoop of subtree in de boom kan het zijn dat er nieuwe pagina's gecreëerd moeten worden. Neem het geval dat de

pagina waarin de records toegevoegd moeten worden, vol is. Dan zijn er 2 mogelijkheden: ofwel verhuizen er records van die pagina naar buur-pagina's totdat alle pagina's vol zitten, ofwel worden er één of meerdere nieuwe pagina's gemaakt en de inhoud wordt over de nieuwe pagina's verdeeld. Meestal wordt de laatste methode gebruikt omdat die meer tijdseffeciënt is. De eerste manier zorgt voor zo vol mogelijke pagina's en is dus het meest plaatsefficiënt. Het nadeel van deze methode is dat bij haast elke keer er iets ingevoegd moet worden, er ook shift-operaties uitgevoerd moeten worden.

Een nadeel van de depth-first methode is dat als alle knopen met een bepaalde tagnaam gezocht moeten worden, er een scan van alle knopen moet uitgevoerd worden. Dit probleem kan wel verholpen worden door indexstructuren te gebruiken. Een overzicht van mogelijke indexstructuren wordt gegeven in sectie 3.2.

3.1.2 Depth-first Subtree-based

De tweede strategie is iets gesofistikeerder. Het XML-document wordt in subtrees verdeeld en elk zulke subtree vormt een record. Dikwijls heeft het record dan een vaste grootte. De records worden ook hier in depth-first volgorde in een pagina geplaatst. Voorbeeld 3.3 toont hoe het XML-document uit het lopend voorbeeld wordt opgeslagen, gebruik makend van de depth-first subtree-based opslagstrategie.

Voorbeeld 3.3. Figuur 3.6 toont hoe de boomvoorstelling uit Figuur 3.2 in subtrees wordt verdeeld.

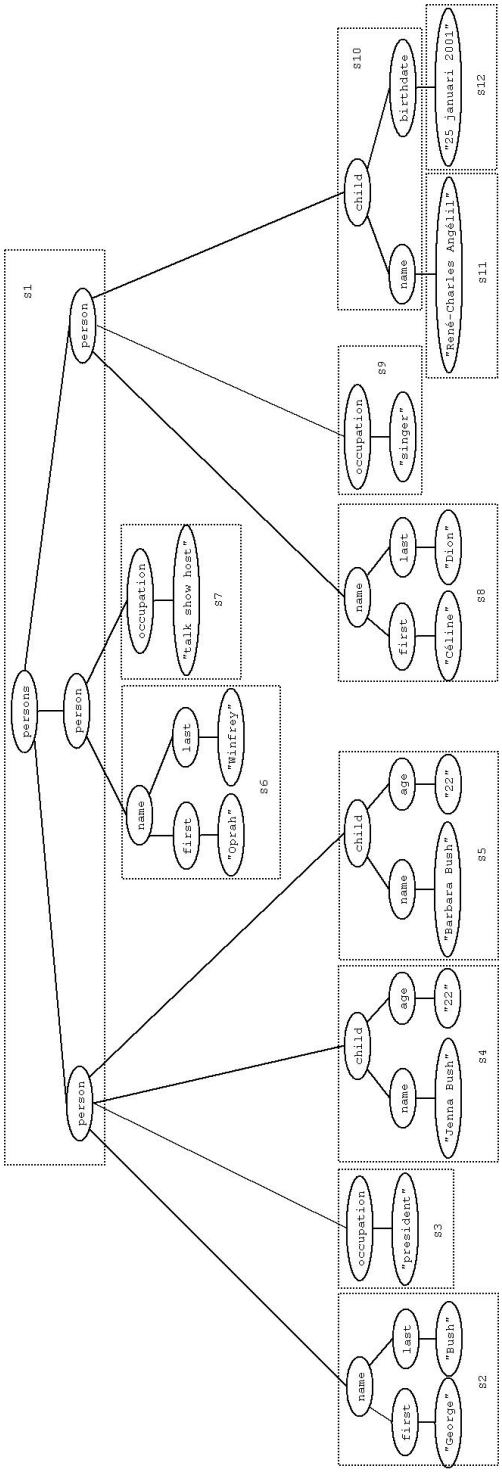
Als de XML-boom verdeeld is in subtrees die in een record passen, wordt elke subtree in een record geplaatst. De records onderling vormen nog steeds een boomstructuur. Deze boomstructuur is te zien in Figuur 3.7.

Nu kan deze boomstructuur ook in depth-first volgorde doorlopen worden. Elke keer een record tegengekomen wordt dat nog niet eerder doorlopen was, wordt het in de reeks records geplaatst. Het resultaat hiervan is net als bij de depth-first element-based opslagstrategie een lange reeks records. Een grafische voorstelling van deze reeks records is te zien in Figuur 3.8.

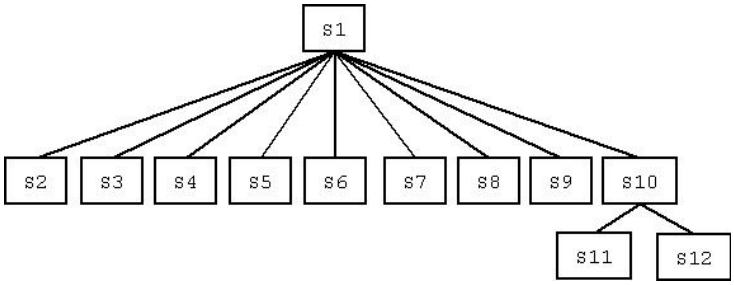
▲

Net zoals bij de depth-first element-based methode worden er records in het geheugen geladen die niet gevraagd waren, maar bij path traversals zou een deel van de knopen in de subtree uiteindelijk toch in het geheugen gevraagd worden. Zo wordt het aantal schijftoegangen minimaal gehouden.

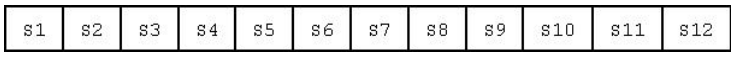
Het voordeel van subtrees te gebruiken in plaats van een serie van knopen, is dat de kinderen van een bepaalde knoop allemaal tot hetzelfde record behoren, als deze knoop teminste geen blad is van de subtree. Dit is meestal



Figuur 3.6: Indeling van de boomvoorstelling in subtrees voor de depth-first subtree-based opslagstrategie



Figuur 3.7: Boomstructuur van de records bij de depth-first subtree-based opslagstrategie



Figuur 3.8: Geserializeerde records voor de depth-first subtree-based opslagstrategie

niet het geval met de depth-first element-based methode, waar dikwijls alleen het eerste kind van de knoop in hetzelfde record zit. Het hebben van alle kinderen in hetzelfde record levert vooral voordeel op als alle kinderen bezocht moeten worden bij het uitvoeren van een query. Dit is vaak het geval met path traversals.

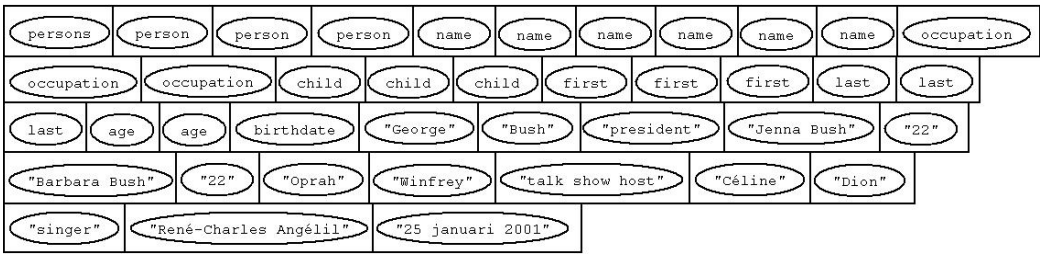
Bij het invoegen van een element of subtree in een record zijn er drie mogelijkheden. Een eerste mogelijkheid is dat het element of de subtree nog bij in het record past waar het in moet komen. In dat geval wordt dit gewoon toegevoegd. Een tweede mogelijkheid is dat de subtree niet meer in een record past maar wel nog bij in de pagina past. In dat geval wordt er één of meerdere record(s) met die subtree in gemaakt en in de pagina toegevoegd. Een laatste mogelijkheid is dat de subtree niet meer in het record past en dat er in de pagina ook niet genoeg plaats is. In dat geval wordt de pagina gesplitst zoals in de depth-first element-based methode.

3.1.3 Clustering Element-based

Een derde fysieke opslagstrategie bestaat erin om de elementen die dezelfde tagnaam hebben, bij elkaar te plaatsen in een pagina. Een element vormt bij deze strategie een record, net zoals bij de depth-first element-based strategie. Een voorbeeld van een XML-document en de manier waarop die wordt opgeslagen is terug te vinden in Voorbeeld 3.4.

Voorbeeld 3.4. De boom uit Figuur 3.2 wordt doorlopen en telkens een nieuwe knoop wordt tegengekomen, wordt die in een record geplaatst en

gesorteerd toegevoegd. Het resultaat is ook hier weer een reeks records, die te vinden is in Figuur 3.9. ▲



Figuur 3.9: Geserializeerde records voor clustering element-based opslagstrategie

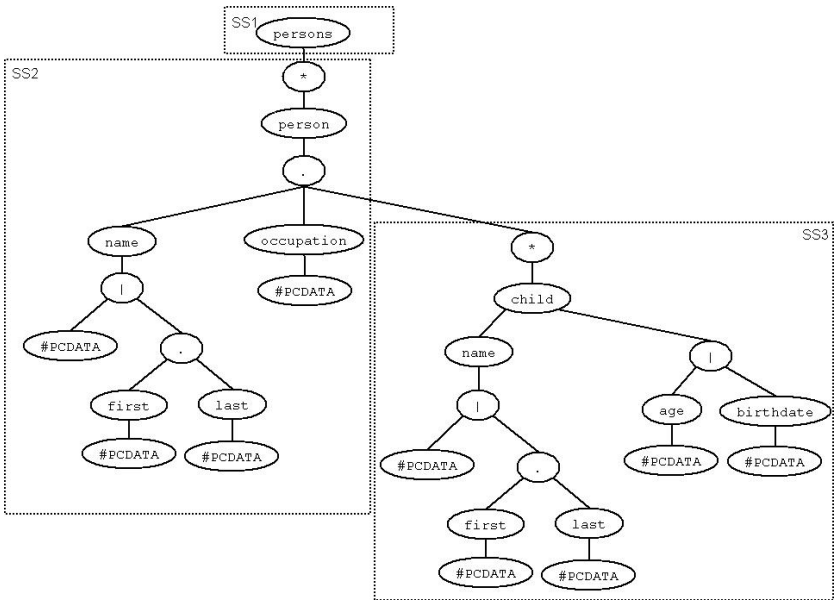
Een ander aspect bij deze opslagstrategie is dat best bijgehouden wordt waar de elementen beginnen met een bepaalde naam. Dit kan gemakkelijk met een tabel. Een nadeel van deze clustering methode is dat als path traversals uitgevoerd worden, er veel schijftoegangen moeten gebeuren. Path traversals vragen namelijk meestal achtereenvolgens knopen met verschillende tagnamen. Path traversals zijn veel voorkomende queries op XML-gegevens en dus wordt deze strategie niet zo vaak gebruikt. Als de gegevens enkel in XML-vorm zijn opgeslagen in een native database om zo de gegevens in de files te kunnen uitwisselen, zullen er niet veel path traversals voorkomen in ondervragingen en kan deze strategie efficiënter zijn. Tegelijk kan dan de bedenking gemaakt worden waarom het XML-document dan in een native database wordt opgeslagen. Bij deze opslagstrategie wordt er namelijk van uit gegaan dat knopen worden opgevraagd door middel van hun tagnaam. Doordat de records zo gegroepeerd zijn dat de elementen met dezelfde tagnaam bij elkaar staan op één pagina, moeten er minder schijftoegangen plaatsvinden.

Als het document dikwijls verandert kunnen element-based methodes beter zijn. Als er namelijk iets aan de structuur van het document verandert moeten enkel pointers in bepaalde records veranderd worden.

In tegenstelling tot element-based methodes, is het bij subtree-based methodes ook mogelijk dat subtrees opgesplitst worden. Dit brengt extra werk met zich mee. Bij depth-first strategieën kan het zijn dat bij aanpassingen het nieuwe record in het midden van een pagina toegevoegd of verwijderd moet worden om de depth-first volgorde te behouden. Ook dit kan nadelig zijn doordat dan pagina's gesplitst of samengevoegd worden.

3.1.4 Clustering Subtree-based

Een laatste strategie neemt net als de depth-first subtree-based strategie ook subtrees als record. De onderverdeling van deze subtrees gebeurt door middel van een algoritme dat eerst de schemagraaf verdeelt in schema-subtrees. Een schema-subtree wordt gekozen volgens een aantal regels. Een knoop van de schemagraaf is de root van een schema-subtree als het ofwel de root is van de schemagraaf, ofwel kardinaliteit '*' of '+' heeft én kinderen heeft. Na het verdelen van de schemagraaf worden de subtrees in de XML-boomstructuur zo gekozen dat elke subtree een instantie is van een schema-subtree. De instanties van een bepaalde schema-subtree worden dan bij elkaar geclusterd in de pagina's. In Voorbeeld 3.5 wordt de onderverdeling in subtrees van de schemagraaf uit het lopend voorbeeld getoond en hoe de records worden opgeslagen.



Figuur 3.10: Verdeling van de schemagraaf voor de clustering subtree-based strategie

Voorbeeld 3.5. Figuur 3.10 toont hoe de schemagraaf uit 3.4 in schema-subtrees wordt verdeeld. De schemagraaf wordt van onder naar boven doorlopen. Als een '$*$'- of '+'-knoop wordt tegengekomen waarvan het kind ook kinderen heeft dan wordt dit de root van een nieuwe schema-subtree. Deze schema-subtree bevat alle afstammelingen van de pasgekozen root die nog niet in een schema-subtree vervat waren. Zo wordt eerst de schema-subtree SS3 van onder naar boven doorlopen. Er wordt een '$*$'-knoop tegengekomen.

Deze '∗'-knoop heeft als kind de *child* knoop en die *child* knoop heeft op zijn beurt weer kinderen. De '∗'-knoop wordt dus de root van een nieuwe schema-subtree. Zo wordt er verder omhoog geklommen in de schemagraaf tot een nieuwe '∗'-knoop tegengekomen wordt. Dit is de ouder van *person* knoop, die op zijn beurt ook weer kinderen heeft. De '∗'-knoop wordt root van de schema-subtree SS2. SS2 bevat alle afstammelingen van die '∗'-knoop behalve de knopen in SS3, omdat die knopen al in een schema-subtree vervat waren. Ten laatste wordt ook de *persons* knoop de root van een nieuwe schema-subtree SS1 omdat het de root is van de schemagraaf.

Figuur 3.11 toont de verdeling van de XML-boomstructuur uit Figuur 3.2 in subtrees. Elke subtree is een instantie van een schema-subtree. Zo zijn subtrees S2, S5 en S6 instanties van schema-subtree SS2.

In Figuur 3.12 staat een mogelijke plaatsing van de records op een pagina. Instanties van eenzelfde schema-subtree worden bijeen geclusterd. Zo zijn subtrees S2, S5 en S6 bij elkaar geclusterd.

▲

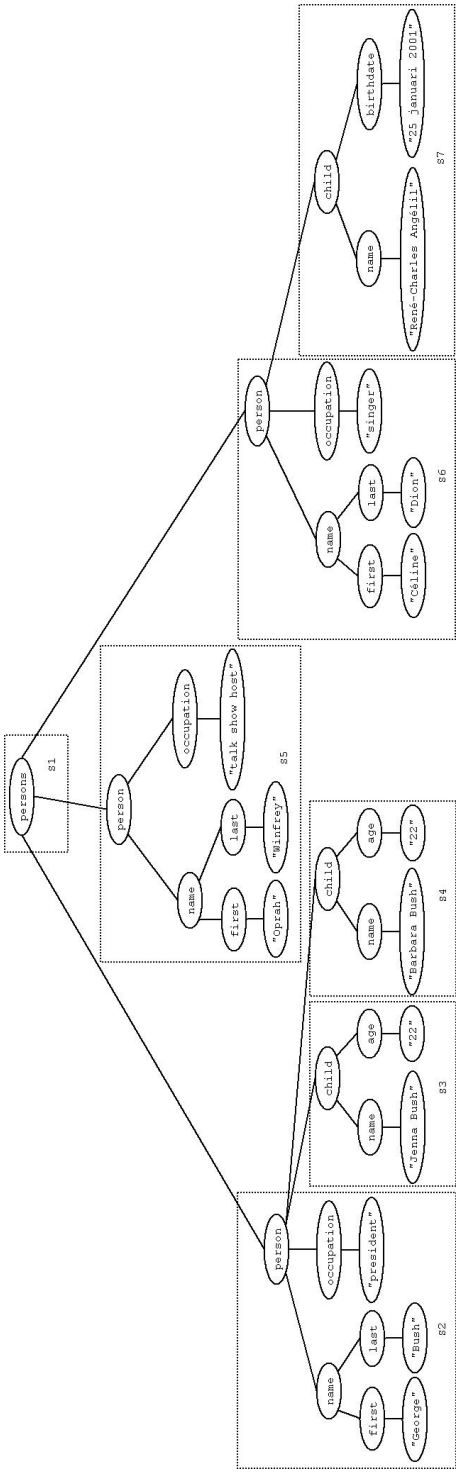
Een groot voordeel van deze strategie is dat de subtrees die samen geclusterd zijn, allemaal dezelfde vorm hebben omdat ze instanties van hetzelfde semantisch blok zijn. Als dan de instanties van een bepaald semantisch blok gevraagd worden, zijn ze sneller teruggevonden omdat ze bij elkaar staan.

Bij path traversals kan deze strategie zijn nut bewijzen. De path traversal kan namelijk uitgevoerd worden op de schemagraaf en het resultaat hiervan is een verzameling van één of meerdere schema knopen die behoren tot één schema-subtree. Dan kunnen de instanties van die schema-subtree, die bij elkaar geclusterd zijn in pagina's geladen worden en daar kan dan mee verder gewerkt worden. Het grote voordeel hiervan bij een path traversal is dat niet de XML-gegevens moeten doorlopen worden. Het grootste nadeel van deze methode is dat er een schema van de gegevens moet bestaan.

Het maken van path traversals doorheen gegevens, opgeslagen met een subtree-based strategie gaat meestal sneller dan bij element-based strategieën. Dit komt vooral doordat bij subtree-based strategieën de kinderen van een knoop vaak in hetzelfde record staan.

3.1.5 Schema

Elke strategie heeft duidelijk zijn voor- en nadelen. Om de beste strategie te kunnen kiezen is het best kennis te hebben van de gegevens die dienen opgeslagen te worden. Op die manier kan onrechtstreeks ook info ingewonnen worden over welke queries vaak zullen voorkomen. Dan kan een strategie



Figuur 3.11: Verdeling van de boomstructuur in subtrees door de clustering subtree-based strategie

Figuur 3.12: Geserializeerde records voor de clustering subtree-based strategie

gekozen worden die de voordelen en nadelen van de strategieën tegen elkaar afweegt voor die gegevens en queries.

Als de gegevens een grootte hebben die vergelijkbaar is met het schema dan kunnen subtree-based methodes in verhouding voor veel ongebruikte ruimte in de records zorgen. In dat geval zijn element-based methodes zuiniger met schijfruimte. Dit plaatsefficiëntie probleem is niet ondenkbaar maar wel onwaarschijnlijk. De originele probleemstelling was immers dat er zoveel XML-gegevens waren dat die niet meer te beheren was in gewone bestanden en te ondervragen met bestaande query evaluatoren. Als de gegevens daarentegen veel groter zijn dan het schema, zullen deze subtree-based methodes in verhouding voor niet zo veel ongebruikte ruimte zorgen. In dat geval speelt vooral tijdsefficiëntie een grote rol. De tijd die een aanpassingsoperatie of een query inneemt is dan belangrijker. Dit neemt natuurlijk niet weg dat plaatsefficiëntie ook een aspect is dat zorgvuldig in de gaten gehouden moet worden. Omdat de gegevens op zich al plaats genoeg innemen, is een hoog percentage van extra plaatsverbruik onwenselijk.

Een andere belangrijke factor om de keuze van fysieke opslagstrategie te bepalen zijn de querybehoeften. Als er vele locatie-queries gesteld zullen worden en bijgevolg meerdere path traversals uitgevoerd zullen worden, zijn subtree-based methodes sneller. Als er daarentegen meer over de gegevens in het bestand dan de structuur ervan ondervraagd wordt, zal een clustering methode sneller zijn. In sommige gevallen kunnen deze querybehoeften uit het schema afgeleid worden. Als er namelijk overwegend tekstuele gegevens in de XML-gegevens staat en het schema niet erg groot is, is de kans groter dat er over de inhoud zal ondervraagd zal worden, dan dat er vragen over de structuur zullen gesteld worden. Omgekeerd als het schemadocument erg groot is en er een doorgedreven structuur in zit, is de kans groter dat er meer structurele queries gesteld zullen worden.

Spijtig genoeg is er niet altijd kennis over het schema beschikbaar. In dat geval worden meestal aannames gemaakt over de XML-gegevens of de oplossing met het beste algemeen resultaat gekozen.

Om toch over schema-informatie te kunnen beschikken houden sommige native DBMS een descriptief schema bij. Een descriptief schema is een dynamisch schema van de gegevens dat, telkens de gegevens aangepast worden, zelf ook aanpast wordt als dat nodig is. Dit staat in tegenstelling tot een

prescriptief schema, dat statisch is. Een prescriptief schema is een conventioneel schema. Het is een op voorhand opgesteld schema waaraan alle XML-gegevens zal moeten voldoen. Een descriptief schema is enkel een structurele samenvatting van de huidige gegevens. Een voordeel van een descriptief schema ten opzichte van een prescriptief schema is ten eerste dat het schema zo klein mogelijk is, omdat het namelijk geen informatie bevat die niet van de gegevens zelf komt. Een tweede voordeel is dat de gegevens zoveel veranderd mogen worden als gewenst, omdat het schema niet statisch is. Een nadeel is dat het schema bij aanpassingen altijd nagekeken moet worden en eventueel moet aangepast worden. Dit kan het aanpassen van de gegevens zelf vertragen.

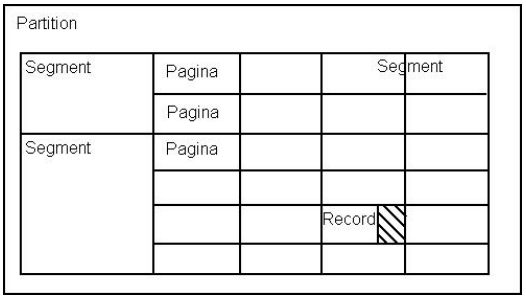
3.1.6 Natix Storage Engine

Een interessante physical Storage Engine is die van Natix [31]. Deze Storage Engine gebruikt namelijk een aantal datastructuren om de toegang tot knopen zoveel mogelijk gescheiden te kunnen houden van het beheer van secundaire opslag.

De kleinste eenheid van opslag is een record. Deze records bevatten elk één subtree. Een record is hier van variabele lengte, maar de lengte heeft wel een bovengrens, namelijk de grootte van een pagina. Een pagina heeft een vaste grootte en bevat records. Pagina's zijn op hun beurt gegroepeerd in segmenten. Er zijn verschillende soorten segmenten, die elk een andere soort gegevens opslaan. Segmenten zijn de hoofdinterfaces naar de XML-gegevens.

Een ander aspect dat de Natix Storage Engine interessant maakt, is hoe een logische view van de boom gesplitst wordt van de fysieke splitsing in records. De Natix Storage Engine gebruikt een subtree-based aanpak. De keuze van de grootte van een subtree wordt bepaald door een drempelwaarde, de threshold. Een subtree wordt in een record geplaatst en de records worden ongeordend opgeslagen in segmenten. Deze records hebben onderling ook een boomstructuur. Deze onderlinge structuur van de records wordt apart in andere segmenten opgeslagen. Figuur 3.13 toont het samenspel van datastructuren in Natix Storage Engine. Deze figuur laat zien dat een pagina eigenlijk niet meer is dan een interne datastructuur die gebruikt wordt voor het beheren van het secundair geheugen.

Omdat de meeste XML-bomen niet op één pagina passen moeten grote bomen gesplitst worden. De logische view van de boom blijft uiteraard dezelfde maar de onderliggende fysieke boomstructuur is in subtrees opgesplitst. Natix Storage Engine heeft een mooie oplossing voor het splitsen van bomen in subtrees. Deze oplossing gebruikt facade objecten, proxy objecten en scaffolding objecten. Figuur 3.14 toont een logische view van een XML-



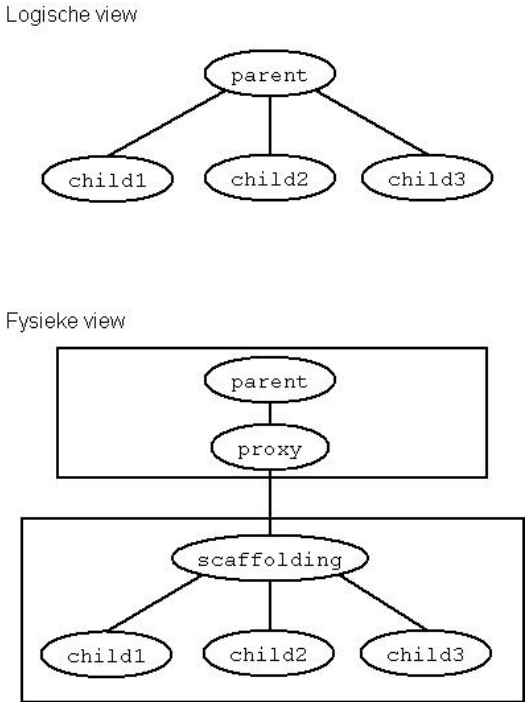
Figuur 3.13: Gebruikte datastructuren in Natix

boomstructuur en zijn effectieve fysieke structuur. Als de logische view van de boom opgesplitst dient te worden in twee subtrees, wordt in de ouder subtree een proxy object ingevoegd als kind. In de kind subtree wordt een scaffolding object ingevoegd die wijst naar alle logische kinderen van de ouder van het proxy object. Een proxy object wijst naar scaffolding objects in andere records dan het zijne. Een scaffolding object wordt eigenlijk enkel gebruikt om meerdere kinderen in het record onder één enkele root te kunnen scharen. Zo hoeft het proxy object enkel te wijzen naar scaffolding objects. Het vervangen van de proxy en scaffolding objecten door hun subtrees heeft als resultaat weer de logische boom.

Bij het aanpassen van de boom kan het zijn dat er zich records moeten splitsen of dat records moeten samengevoegd worden tot één enkel record. Voor het algortime dat in de Natix Storage Engine gebruikt wordt bij het invoegen van een knoop wordt verwezen naar [31].

3.2 Indexstructuren

Net zoals bij relationele DBMS worden er in native DBMS indexstructuren gebruikt om het uitvoeren van een query sneller te laten verlopen. Een overzicht van mogelijke indexstructuren bij relationele databases wordt gegeven in [35]. Het idee achter een indexstructuur is meestal om te zorgen voor een kleinere verzameling gegevens of een geordende verzameling gegevens die referenties bevatten naar de uiteindelijke gegevens. In die kleinere of geordende verzameling gegevens kan dan gezocht worden en dan kan een referentie gevolgd worden naar de uiteindelijke gegevens. Dit gaat veel sneller dan de hele verzameling ongeordende gegevens te doorzoeken. Merk op dat fysieke opslagstrategieën de gegevens wel in een volgorde groeperen, maar dat de volgorde gekozen is zodat records die dikwijls samen worden opgevraagd ook dicht bij elkaar staan in de pagina's. De verzameling gegevens is dus eigen-



Figuur 3.14: Logische en fysieke view van een boom in Natix

lijk niet compleet ongeordend, maar niet geordend zodat het zoeken optimaal verloopt.

Voor XML-gegevens kan er een onderscheid gemaakt worden tussen twee soorten indexen, namelijk de index op waardes en de index op paden. Een index op waardes is een index die het zoeken naar tekstuele waardes versnelt. Deze tekstuele waardes kunnen tagnamen zijn of de tekstuele inhoud van tags. Deze indexen sorteren meestal alle waardes om er zo snel één te kunnen vinden. Een index op paden is een index die path traversals sneller laat verlopen. Deze indexstructuren houden een kleinere verzameling gegevens bij, meestal een samenvatting van de structuur.

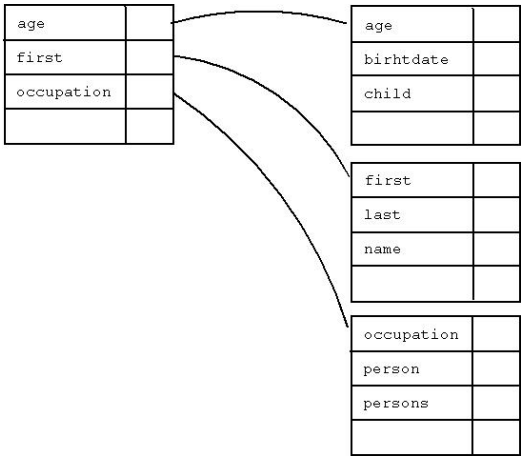
3.2.1 B-tree Indexen

Bestaande relationele indexen kunnen in veel gevallen ook gebruikt worden als indexen voor XML-gegevens. Een welgekende indexstructuur voor relationele gegevens is de B-tree. Deze indexstructuur wordt beschreven in [35]. Deze structuur heeft zijn nut duidelijk bewezen bij relationele databases. Deze indexstructuur behoeft geen grote aanpassingen om compatibel te zijn met XML-gegevens. Enkel de pointers naar tupels in een relationele database

worden vervangen door pointers naar knopen in een native XML-database. Deze B-tree indexen vallen onder de indexen op waardes.

Een B-tree is een indexstructuur die een geordende verzameling van gegevens beheert. Een B-tree wordt voorgesteld door een boom, die blokken als knopen heeft. Zo een blok bevat een record. Een record in een B-tree is erg verschillend van een record als eenheid van native opslag, beschreven bij de fysieke opslagstrategieën. Een record in een blok van een B-tree bevat (key, ptr) koppels, waarin key de sleutelwaarde is en ptr een pointer naar een ander blok of een verzameling knopen. De key is meestal een tagnaam of de tekstuele inhoud van een tag. Voorbeeld 3.6 toont hoe het XML-document uit het lopend voorbeeld geïndexeerd kan worden met een B-tree index.

Voorbeeld 3.6. Deze B-tree index indexeert de tagnamen van het XML-document uit 3.1. Ze heeft blokken van grootte vier. Dit wil zeggen dat er in elk blok vier records bijgehouden kunnen worden. Het eerste record uit het linkse blok bevat als key waarde de key waarde van het eerste record van het blok waar het naar wijst, namelijk $article$. De „lege" vakjes aan de rechterkant van een blok stellen de ptr waardes voor. Deze worden voorgesteld door pijlen waar mogelijk. Pijlen naar de boomstructuur, weergegeven in Figuur 3.2 zouden het geheel alleen maar onoverzichtelijk maken. Het is niet moeilijk in te beelden dat er vanuit de ptr-vakjes in de blokken pijlen vertrekken naar knopen met als tagnaam de corresponderende key waarde. ▲



Figuur 3.15: B-tree indexstructuur

Het zoeken in een B-tree gaat als volgt. Als een knoop gezocht wordt met een bepaalde key waarde, wordt eerst gekeken naar het root record. Het

record wordt doorlopen. Voor elk koppel wordt gecheckt of de *key* waarde van het volgende koppel groter of gelijk is aan de gezochte *key* waarde. Telkens dit zo is, wordt naar het volgend koppel overgegaan. Als dit niet zo is zijn er enkele mogelijkheden. Een eerste mogelijkheid is dat de huidige *key* waarde gelijk is aan de gezochte *key* waarde en de *ptr* wijst naar een set knopen. Dan is de gezochte set knopen gevonden. Een tweede mogelijkheid is dat *ptr* naar een set knopen wijst, maar de *key* waardes toch niet hetzelfde zijn. In dat geval bestaat er geen knoop met de gezochte *key* waarde. De laatste mogelijkheid is dat de *ptr* naar een ander record wijst. In dat geval wordt de pointer gevolgd. Vervolgens wordt het nieuwe record op dezelfde manier doorlopen. Voor meer informatie over B-tree indexen bij relationele DBMS, zie [35].

De *key* waarde kan in het geval van XML-gegevens een tagnaam zijn, tekstuele gegevens die een knoop bevat of zelfs een structurele identifier. De B-tree index kan dus veel soorten queries versnellen. Als de *key* een tagnaam is, zullen queries die alle knopen vragen met een bepaalde tagnaam sneller uitgevoerd kunnen worden. Ook queries die knopen met een bepaalde tekstuele waarde zoeken kunnen met een B-tree index versneld worden. Zelfs sommige structurele queries kunnen sneller uitgevoerd worden met een B-tree index. Hiervoor is echter bijkomstige informatie over de knopen nodig. Als een knoop een bepaalde identifier heeft, kunnen queries knopen adresseren met een identifier. Zelfs een range van knoop identifiers kan dan opgevraagd worden.

3.2.2 Schemagraaf

De schemagraaf van een XML-document kan op zich ook gezien worden als een indexstructuur, mits een kleine uitbreiding. Elke knoop uit de schemagraaf moet dan een verwijzing naar alle instanties van die knoop bevatten. De schemagraaf van het lopend voorbeeld kan teruggevonden worden in Figuur 3.4.

De aangepaste schemagraaf is dan een index op paden. Als de ondervraging een path traversal is, dan kan het pad in de schemagraaf gevolgd worden. Uiteindelijk wordt er terecht gekomen in één of meerdere knopen van de schemagraaf. Het resultaat van die ondervraging is dan alle instanties van die knopen. Deze instanties kunnen gevonden worden door de referenties in de knopen van de aangepaste schemagraaf te volgen.

3.2.3 SUPEX

SUPEX is een schema-gebaseerde index. Ze bestaat uit twee datastructuren, namelijk een structural graph en een element map. De structural graph is een index op paden en de element map een index op waardes.

De structural graph wordt opgebouwd door middel van het schema en stelt een structurele samenvatting van de XML-gegevens voor. In de structural graph worden enkel de elementen opgenomen die bovenaan staan in de boomvoorstelling. Alle mogelijke paden die beginnen in de root hebben zo baat bij de structural graph. Ze kunnen deze immers gebruiken bij de eerste stappen in het pad. De structural graph heeft één root knoop en elke knoop heeft een label dat gelijk is aan de tagnaam van de overeenkomstige knoop in het schema, E-label genaamd.

De element map is een kleine B-tree index. De *key* waarde van de element map is een E-label. De *ptr* waardes in de blad-records van de element map zijn pointers naar knopen in de structural graph. Voorbeeld 3.7 toont hoe het XML-document uit het lopend voorbeeld geïndexeerd kan worden met behulp van de SUPEX indexstructuur.

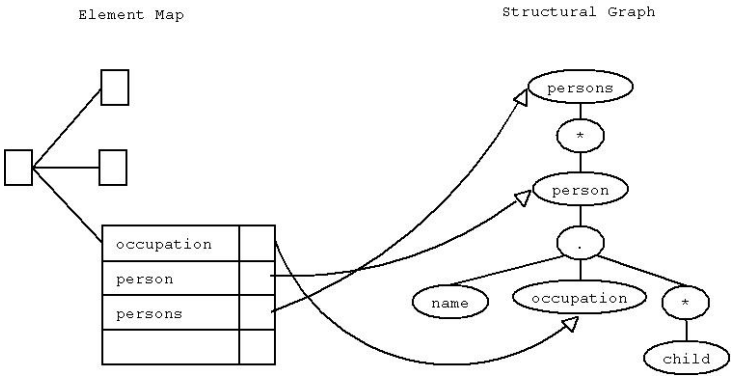
Voorbeeld 3.7. Figuur 3.16 toont de element map en de structural graph voor het XML-document uit het lopend voorbeeld 3.1. De element map heeft als *key* waardes de E-labels. De element map indexeert enkel de knopen van de structural graph, rechts weergegeven.

Figuur 3.17 geeft weer hoe de structural graph verwijst naar de boomvoorstelling van de XML-gegevens. Voor de duidelijkheid is slechts een deel van de boomvoorstelling uit Figuur 3.2 weergegeven. Het is niet moeilijk in te beelden hoe er verwijzingen naar de rest van de knopen in de boomvoorstellingen kunnen zijn.

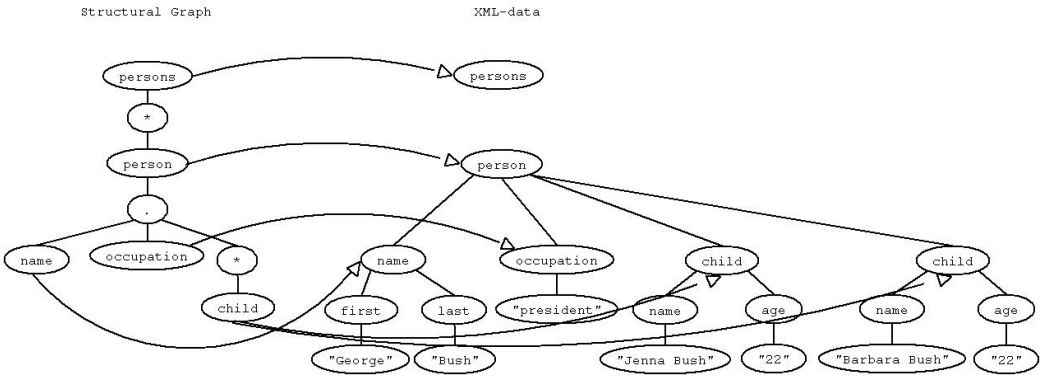
▲

De SUPEX indexstructuur ondersteunt twee soorten queries. De eerste soort zoekt elementen met een bepaalde tagnaam op door middel van de element map. De tweede soort zijn korte path traversals, die gevonden worden door doorheen de structural graph te traverseren. Ook queries zoals „$//E1/E2/.../En$" verlopen sneller als $E1$ tenminste in de structural graph staat. De tagnaam $E1$ kan dan namelijk eerst in de element map opgezocht worden. Dan kan de structural graph doorlopen worden om „$E1/E2/.../Em$" te vinden, waarbij Em een blad is in de structural graph. Zo kunnen de instanties van Em geladen worden en zo de boom verder doorlopen.

Deze indexstructuur toont hoe de twee mogelijke indexstructuren, index op waardes en index op paden, samen kunnen werken. Door alleen de bovenste knopen uit de schemagraaf in de structural graph te stoppen, blijven de



Figuur 3.16: SUPEX indexstructuur – Element map en stuctural graph



Figuur 3.17: SUPEX indexstructuur – Structural graph en XML-gegevens

element map en de structural graph klein. Hierdoor past deze indexstructuur gemakkelijk in het geheugen. Meer informatie over de SUPEX indexstructuur kan teruggevonden worden in [34]

3.2.4 Full Text Index Framework

De Full Text Index Framework indexstructuur is een index op waardes. Deze indexstructuur is gebaseerd op inverted files. Een inverted file is een manier om snel te zoeken op individuele termen. Het resultaat van die zoekopdracht is dan een lijst van document identifiers en eventueel offsets binnen het document.

Voorbeeld 3.8. Figuur 3.18 toont hoe het XML-document van het lopend voorbeeld geïndexeerd kan worden met de Full Text Index Framework indexstructuur. Hier is de tekstuele inhoud van de tags geïndexeerd. In de preprocessing stap wordt elk woord uit de tekstuele inhoud gehaald en apart in de index geplaatst. Zo wordt de string „talk show host" in drie woorden gesplitst. Elke entry uit de tabel wijst naar een knoop in de boomvoorstelling van het XML-document. Hier zal de entry „Dion" verwijzen naar de string „`Dion`" in de boomvoorstelling. De entry „Bush" zal verwijzen naar de string „`Bush`", de string „`Jenna Bush`" en de string „`Barbara Bush`" ▲

Het idee achter een inverted file is om de te doorzoeken gegevens te preprocessen zodat de zoektermen geordend in tabellen komen te staan. Hierdoor moet geen totale scan meer gebeuren van de documenten, enkel de geordende tabel moet doorzocht worden. Dit preprocessen gebeurt in een aantal stappen. Een eerste stap is het extraheren van de sleutelwoorden en het document waarin ze staan, en die sleutelwoorden in een tabel te plaatsen. Hierna wordt die lijst geordend. Ten laatste worden uit deze tabel de duplicaten samen gegroepeerd zodat elke zoekterm maar een keer voorkomt in de tabel. Als deze laatste stap gedaan is bevat de tabel de zoektermen en lijsten van document identifiers en offsets waarin de zoektermen voorkomen. Op deze tabel kunnen bestaande relationele indexstructuren geplaatst worden zoals een B-tree index. De *key* waarde is de zoekterm. Deze relationele index versnelt het zoeken in de tabel en zo het zoeken in de XML-gegevens.

Voor deze indexstructuur wordt dit concept zo aangepast dat het een aantal verschillende soorten contexten opslaat in plaats van document identifiers en offsets. Een simple knoop context bestaat uit een document identifier, knoop identifier en de posities van de zoekterm in de knoop. Een voorbeeld van een dergelijke index is gegeven in Voorbeeld 3.8. Er bestaan ook meer complexe contexten die ook structurele informatie bevatten zoals d_{min} en d_{max}. Deze waardes zullen uitgelegd worden bij de XSASR indexstructuur.

2001
22
25
Angélil
Barbara
Bush
Céline
Dion
George
host
january
Jenna
Oprah
president
René-Charles
show
singer
talk
Winfrey

Figuur 3.18: Full Text Index Framework indexstructuur

Ondervragingen die alle knopen zoeken met een bepaalde tagnaam of die een knoop die een bepaalde zoekterm bevat opvragen, zullen sneller gaan. Ook knopen met een bepaalde $docID$, d_{min} of d_{max} waarde zullen sneller opgevraagd kunnen worden. De indexstructuur is namelijk gericht op het vinden van knopen met één bepaalde tekstuele of numerieke zoekterm.

Een nadeel van deze indexstructuur is dat de XML-gegevens voorverwerkt moeten worden. De XML-gegevens moeten gescand worden voor zoektermen en de gegevens moeten gesorteerd toegevoegd worden in de lijst. Dit toevoegen kan versneld worden door relationele indexen. Dit preprocessen kost vooral tijd bij het aanpassen van de gegevens. De gepreproceste lijst neemt ook plaats in beslag. De hoeveelheid ruimte dat ze inneemt is vooral afhankelijk van het aantal zoektermen.

Het gebruik van deze indexstructuur verschilt in principe niet erg van een B-tree index. Het enige wat deze indexstructuur extra aanbrengt is de mogelijkheid om op termen te zoeken die zich in het midden van de tekstuele inhoud van een knoop bevindt. Met een B-tree index zou het bijvoorbeeld niet mogelijk zijn om alle elementen waarin de term „show" ergens voorkomt te vinden. Voor meer informatie over Full Text Index Framework, zie [31]

3.2.5 eXtended Access Support Relations (XSASR)

XSASR is een indexstructuur die queries over de structuur van de XML-gegevens ondersteunt. Het is een speciale soort index op paden. Voor elke knoop uit een bepaald document houdt deze indexstructuur een $docID$, een d_{min} waarde, een d_{max} waarde, een $parent\_d_{min}$ waarde en een $eType$ bij in een tabel. Hierin vormen ($docID$, d_{min}) een sleutelwaarde.

Het $docID$ is een document identifier. De d_{min} en d_{max} waardes worden berekend door de boom in depth-first traversal te doorlopen. Als een eerste keer langs een knoop wordt gekomen, krijgt die een d_{min} waarde. Als een laatste keer langs de knoop wordt gelopen, krijgt die een d_{max} waarde. De $parent\_d_{min}$ waarde is de d_{min} waarde van de parent knoop. Deze dient om de parent knoop te identificeren. Merk op dat het $docID$ van de parent knoop hetzelfde is als dat van de child knoop. De $eType$ waarde geeft het type van het element aan. Dit kan de tagnaam of attribuut naam zijn. Voorbeeld 3.9 geeft een XSASR-encodering van een stuk XML.

Voorbeeld 3.9. Figuren 3.20, 3.21 en 3.22 plaatsen bij elke knoop in de boomvoorstelling van het XML-document uit het lopend voorbeeld een d_{min} en een d_{max} waarde. De waardes staan tussen haakjes bij in de knoop. De boomstructuur is hier in stukken verdeeld om de nummers en tagnamen nog leesbaar te maken.

Figuur 3.23 toont hoe de knopen in een relationele tabel worden geplaatst. De corresponderende d_{min} en d_{max} waardes worden in een rij gezet met de tagnaam, $parent\_d_{min}$, $docId$ en $eType$. ▲

Al deze waardes komen in relationele tabellen te staan. Als een query gesteld wordt, wordt deze eerst omgezet in een SQL-query. Deze SQL-query wordt dan losgelaten op de relationele tabellen. Deze methode is erg gelijkend op de methode beschreven in het volgend hoofdstuk in sectie 4.2.1, die later besproken zal worden. Aangezien de waardes nu in relationele tabellen staan, is het ook weer mogelijk om er bestaande relationele indexen op te plaatsen. Deze laten toe om in de tabellen snel knopen te zoeken met bepaalde $DocID$, d_{min}, d_{max}, $parent\_d_{min}$ of $eType$ waardes. Voor meer informatie over de XSASR indexstructuur, zie [31].

3.2.6 Link Index (Lindex)

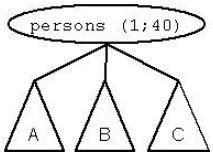
Lindex is een soort index op paden. Deze index indexeert enkel het pad naar de ouder. Het krijgt een *object_id* en een *label* van een bepaalde knoop en geeft de ouder knoop van die knoop terug. De *object_id*, de *label* waarde en de pointer naar de ouder knoop worden opgeslagen in een relationele tabel. Net zoals bij XSASR kunnen nu hierop bestaande relationele indexen geplaatst worden voor het snelle ophalen van een ouder knoop.

Deze index kan het zoeken versnellen als bij de opslagstrategie geen pointers worden voorzien naar ouders. Als dat wel het geval is, heeft de Lindex eigenlijk weinig nut. De meeste systemen van tegenwoordig voorzien al een pointer naar een ouder knoop en hebben dus geen behoefte aan een Lindex. De index zou enkel extra plaats innemen en geen wezenlijk verschil maken voor de query snelheid. Voor meer informatie over de Lindex indexstructuur, zie [33].

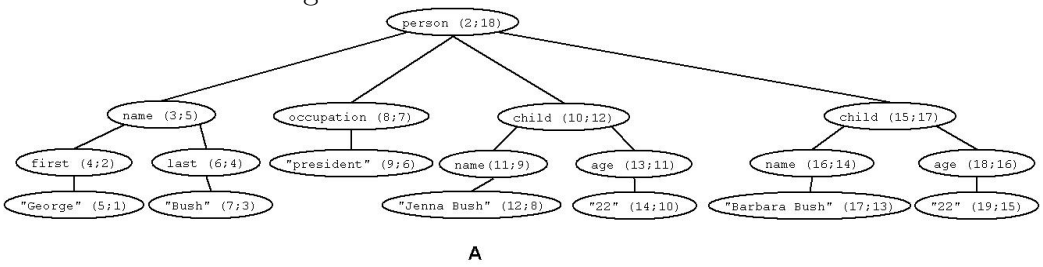
3.2.7 Value Index (Vindex)

De Value Index is een index op waardes. Vindex neemt een *label*, een *operator* en een *value*. Het geeft alle knopen terug met tagnaam *label* en een waarde die voldoet aan de conditie die gespecifieerd is door *operator* en *value* (bijvoorbeeld > 10).

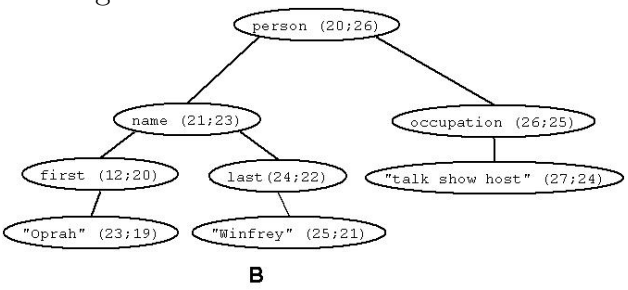
Deze indexstructuur is een speciaal geval van een B-tree index. Deze index kan strings met reals of integers vergelijken. Daarom moeten er drie indexen bijgehouden worden. De eerste is een String Vindex, die alle strings indexeert. De tweede is een Real Vindex. Deze bevat index entries voor numerieke waarden. De laatste is een String-coerced-to-real Vindex, die alle



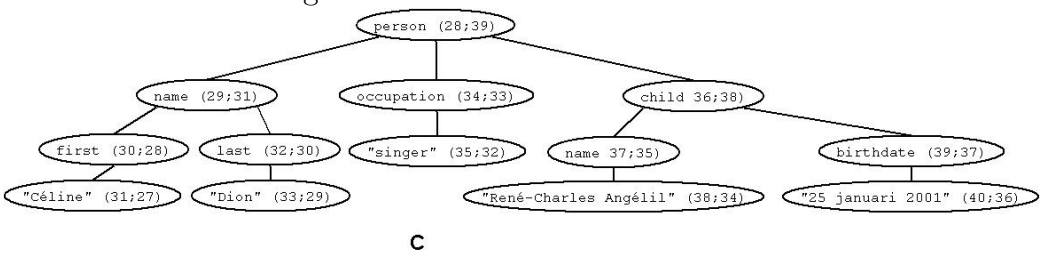
Figuur 3.19: XSASR indexstructuur



Figuur 3.20: XSASR indexstructuur



Figuur 3.21: XSASR indexstructuur



Figuur 3.22: XSASR indexstructuur

$knoop\_ptr$	$DocId$	d_{min}	d_{max}	$parent\_d_{min}$	$eType$
	1	1	40	null	persons
	1	2	18	1	person
	1	3	5	2	name
	1	4	2	3	first
	1	5	1	4	”George”
	1	6	4	3	last
	1	7	3	6	”Bush”
	1	8	7	2	occupation
	1	9	6	8	”president”
	1	10	12	2	child
	1	11	9	10	name
	1	12	8	11	”Jenna Bush”
	1	13	11	10	age
	1	14	10	13	”22”
	1	15	17	2	child
	1	16	14	15	name
	1	17	13	16	”Barbara Bush”
	1	18	16	15	age
	1	19	15	18	”22”
	1	...	...	...	...

Figuur 3.23: XSASR indexstructuur

string waardes bevat, omgezet naar numerische waardes. Voor elke Vindex wordt een B-tree Index gemaakt. In Voorbeeld 3.10 wordt getoond hoe het XML-document uit het lopend voorbeeld geïndexeerd wordt aan de hand van de Vindex indexstructuur.

Voorbeeld 3.10. Figuur 3.24 toont hoe de Vindex indexstructuur het XML-document indexeert. In dit XML-document wordt de waarde "22" als een string gezien terwijl het eigenlijk de leeftijd van de persoon uitdrukt. Een query van de vorm „geef alle personen met leeftijd kleiner dan 40" is dus niet ondenkbaar. Voor zo een query kan in dit geval de String-coerced-to-real Vindex helpen. Of voor deze query effectief de indexstructuur gebruikt wordt, hangt af van de implementatie in het systeem zelf. ▲

$String - coerced - to - real Vindex$
22

$StringVindex$
22
25 january 2001
Barbara Bush
Bush
Céline
Dion
George
Jenna Bush
Oprah
president
René-Charles
singer
talk show host
Winfrey

Figuur 3.24: Vindex indexstructuur

Als er een zoekopdracht binnenkomt en de *value* is een string waarde, dan wordt in de String Vindex gezocht. Als *value* kan omgezet worden naar een real, wordt de waarde omgezet en die omgezette waarde wordt opgezocht in de Real Vindex. Als *value* een numerische waarde is, wordt in de Real Vindex en daarna in de String-coerced-to-real Vindex gezocht.

Deze indexstructuur kan zijn nut bewijzen bij het numerisch vergelijken van strings met getallen. Aangezien XML-gegevens meestal ingelezen worden vanuit een tekstbestand, wordt de inhoud van een element soms zonder meer

in een knoop geplaatst als string. Als de waarde eigenlijk een numerische waarde is, wordt die dan niet naar een integer of real omgezet. In dat geval kan een Value index goed van pas komen omdat deze dan enkel de numerische waardes gaat doorzoeken, in plaats van alle string waardes.

Een nadeel van deze indexstructuur is dat er redundante informatie opgeslagen wordt. Alle strings die kunnen omgezet worden naar een numerische waarde, staan namelijk in de String-coerced-to-real Vindex én in de String Vindex. Voor meer informatie over de Vindex indexstructuur, zie [33].

3.3 Beheer van transacties

Net zoals bij relationele DBMS is het bij native DBMS belangrijk dat meerdere gebruikers gelijktijdig toegang kunnen hebben tot de database. Het zou erg inefficiënt zijn om slechts één gebruiker tegelijkertijd in de database toe te laten terwijl gelijktijdige toegang best mogelijk is. Om toch te zorgen dat bijvoorbeeld de ene gebruiker niets aanpast aan gegevens die een andere gebruiker net aan het lezen was worden er locking protocols gebruikt. Het is belangrijk dat deze locking protocols een zo hoog mogelijke graad van gelijktijdig gebruik toelaten, maar toch zorgen dat er geen conflicten kunnen optreden. Locking protocols voor relationele databases worden beschreven in [35].

De locking protocols voor relationele gegevens, die records locken, voldoen duidelijk niet voor XML-gegevens. Een actie in een knoop in de boomstructuur heeft namelijk gevolgen voor de kinderen van die knoop. Als bijvoorbeeld een knoop wordt gewist, worden alle kinderen van die knoop logischerwijs ook gewist. Locking protocols voor bomen geven al meer zekerheid, maar voldoen ook niet altijd. Door IDREF's te volgen kan er gesprongen worden naar een willekeurige andere knoop in de boom. Voorlopig is het enkel het beheer van transacties in Natix dat rekening houdt met sprongen doorheen de boom.

3.3.1 Beheer van transacties in Natix

Het locking protocol van Natix is gebaseerd op multi-granularity locking (MGL)[2] en strict two phase locking (2PL)[3]. Eerst wordt even vermeld wat multi-granularity locking en strict two phase locking inhouden.

Bij multi-granularity locking zijn er verschillende soorten stukken van gegevens die gelockt worden. Een stuk gegevens wordt een granule genoemd. Een granule kan een document zijn, een subtree of iets anders. Er wordt onderscheid gemaakt in groottes van granules en hierin is een hiërarchie terug

te vinden. Een knoop bijvoorbeeld is een kleine granule en een subtree een grotere granule. Het is duidelijk dat een subtree hoger staat in de hiërarchie, omdat een subtree een knoop kan bevatten maar niet omgekeerd. Naar granules die lager staan in de hiërarchie dan een bepaalde granule zal worden gerefereerd als afstammeling-granules van die bepaalde granule. Op eenzelfde manier wordt er gesproken over voorouder-granules. Op elke grootte van granule kan een lock geplaatst worden. Merk op dat hoe groter de gelockte granule is, hoe minder het gelijktijdig gebruik is. Tegelijk, als de granules groter zijn, zijn er in totaal minder granules die gelockt moeten worden om dezelfde hoeveelheid te locken. Bij grote granules wordt er dus minder tijd verspild aan het zetten van locks.

Er zijn verschillende soorten locks. Zo zijn er om te beginnen de read of gedeelde locks en de write of exclusieve locks. De read locks kunnen gelijktijdig op eenzelfde granule geplaatst worden, aangezien ze enkel geplaatst worden om gegevens te lezen. Lezen van dezelfde granule door twee transacties zorgt immers voor geen conflict. De write locks vragen exclusieve toegang omdat er geschreven wordt in de knoop en dit kan altijd voor conflicten zorgen. Een write lock laat dus geen ander soort locks gelijktijdig toe op dezelfde granule. Doordat de acties in een granule doorwegen naar alle elementen van die granule, is een lock dat expliciet op een granule geplaatst wordt ook impliciet geplaatst op alle afstammeling-granules. Om hiermee rekening te houden in het locking protocol zijn er intention read en intention write locks ontworpen.

Als op een bepaalde granule X een read lock geplaatst moet worden, moet er eerst een intention read lock geplaatst worden op al de voorouder-granules. Pas als op alle voorouder-granules een intention read lock geplaatst is kan op de te locken granule X het read lock geplaatst worden. Een intention write lock werkt op juist dezelfde manier maar dan voor write locks.

Een laatste soort lock is een read intention write lock. Dit houdt in dat de granule zelf gelockt wordt om te lezen en dat een afstammeling-granule een write lock houdt of er een wil. Om te weten of locks compatibel zijn met elkaar wordt in een lock matrix gekeken. De lock matrix kan worden teruggevonden in Figuur 3.25. Een lock matrix wordt als volgt gelezen. De kolommen stellen de locks voor die al geplaatst zijn. De rijen stellen de locks voor die geplaatst willen worden. Als bijvoorbeeld een read lock geplaatst wil worden op een granule en er wordt al een write lock gehouden op die granule, wordt er in de matrix in de kolom voor het write lock gekeken en in de rij voor het read lock. Als op deze plaats een *ja* staat, mag een read lock geplaatst worden op die granule. Als op deze plaats een *nee* staat, mag geen read lock geplaatst worden op die granule. In de lock matrix staat r voor read, w voor write en i voor intention. riw is dus een read intention write

lock.

	r	w	ir	iw	riw
r	ja	nee	ja	nee	nee
w	nee	nee	nee	nee	nee
ir	ja	nee	ja	ja	ja
iw	nee	nee	ja	ja	nee
riw	nee	nee	ja	nee	nee

Figuur 3.25: De lock matrix voor multi-granularity locking

Het multi-granularity locking protocol bepaalt de graad van gelijktijdig gebruik. Het zorgt ervoor dat er geen conflicten optreden bij het lezen of schrijven in een granule. Een ander aspect van locking is wanneer locks aangevraagd worden en losgelaten worden. Hiervoor zorgt het strict two-phase locking protocol. Het two-phase locking protocol zorgt ervoor dat locks aangevraagd worden wanneer ze nodig zijn en dat locks pas losgelaten worden op het einde van de transactie. Dit zorgt voor twee fases tijdens een transactie, namelijk een fase waarin het aantal locks toeneemt en een fase waarin de locks worden vrijgegeven. Dit verklaart de naam.

In Natix wordt er ook rekening gehouden met IDREF's. Dit houdt in dat er nog een ander soort lock wordt geïntroduceerd, namelijk het SPP lock. Als een transactie van één knoop naar een willekeurige andere knoop X springt, moeten de voorouders van die gerefereerde knoop X in principe ook intention locks krijgen. Maar de knopen die voorouders zijn niet gekend in de transactie. Dus wordt er omhoog gelopen in de boom en telkens wordt een SPP lock geplaatst op die voorouder-knopen. Als uiteindelijk de root bereikt wordt, wordt er weer naar knoop X genavigeerd en voor elke knoop die onderweg tegengekomen was wordt het SPP lock veranderd naar het gewenste lock. Dit kan een intention read of een intention write zijn. Nadat die intention locks geplaatst zijn kan het gewenste lock op knoop X geplaatst worden. In de lock matrix wordt het SPP lock met *spp* aangegeven.

Het update lock is een read lock, dat misschien later een write lock wordt. Dit lock kan worden gezet terwijl andere transacties nog read locks hebben staan, maar houdt nieuwe read locks en write locks tegen voor een eventuele write lock later. Als de transactie uiteindelijk toch wil schrijven, moet het wachten om het update lock in een write lock te kunnen veranderen tot alle read locks losgelaten zijn. In de lock matrix wordt een update lock met een u aangegeven. De lock matrix voor het locking protocol van Natix is weergegeven in Figuur 3.26.

Een ander aspect waar rekening mee gehouden wordt in Natix is lock

	r	*w*	*u*	*ir*	*iw*	*riw*	*spp*
r	ja	nee	nee	ja	nee	nee	ja
w	nee	nee	nee	nee	nee	nee	nee
u	ja	nee	nee	ja	nee	nee	ja
ir	ja	nee	nee	ja	ja	ja	ja
iw	nee	nee	nee	ja	ja	nee	ja
riw	nee	nee	nee	ja	nee	nee	ja
spp	ja	nee	nee	ja	ja	ja	ja

Figuur 3.26: De lock matrix voor Natix

escalation. Als er namelijk veel locks worden gehouden op kleinere granules ontstaat er een teveel aan locks die voor een extra hoop werk zorgen. Door het multi-granularity protocol biedt zich hier een mooie oplossing voor, namelijk het plaatsen van een lock op een grotere granule. In de praktijk betekent dit dat als er bijvoorbeeld veel subtrees gelockt worden van een document, dat de vele locks op subtrees vervangen worden door een enkele lock op het document.

Een ander aspect dat in rekening gebracht moet worden is een deadlock. Als een transactie een lock op een granule A houdt en wacht tot hij een lock mag plaatsen op granule B, terwijl een andere transactie een lock houdt op granule B terwijl hij wacht om een lock te zetten op granule A, ontstaat er een deadlock. Dit probleem wordt in Natix als volgt opgelost. Er wordt een waiting graph bijgehouden waarin terug te vinden is op welke transactie een andere wacht voor een bepaald lock. Als een transactie een bepaalde tijd wacht om een lock te kunnen zetten, wordt de waiting graph nagekeken voor lussen in de graaf. Als er effectief een lus te vinden is in de waiting graph, wordt de aanvraag van dat lock gewoon geweigerd. Meer informatie over beheer van transacties in Natix kan teruggevonden worden in [31].

3.3.2 Beheer van transacties in OrientX

OrientX laat meerdere clients tegelijk toe in de database. Hiervoor is dus beheer van transacties nodig. In OrientX is een uitbreiding op role-based acces control geïmplementeerd, namelijk knoop-mapping role-based access control. Role-based access control maakt beslissingen in verband met data-toegang aan de hand van één of meerdere rollen. Een gebruiker die toegang tot de gegevens wil, krijgt een rol toegewezen. Een rol bepaalt welke actie de gebruiker al dan niet op gegevens mag uitvoeren. Voor meer informatie over role-based access control, zie [36]. De knoop-mapping role-based access control methode maakt gebruik van de hiërarchie van de XML-gegevens om

de hiërarchie van de rollen te bepalen. Als rol A superieur is aan rol B, dan zijn de gegevens waartoe A toegang verkrijgt een superset van de gegevens waartoe B toegang verkrijgt. De rollen kunnen gemapt worden op knopen, zodat de knoop waarop A gemapt werd een voorouder is van de knoop waarop B gemapt werd. Een rol kan gezien worden als een verzameling triples $(knoop, Context, Action)$. Hierin stelt *knoop* de tagnaam van de root van de subtree in het XML-document voor. *Context* is een pad en geeft de unieke positie van de knoop in het schema. *Action* stelt een verzameling acties die toegelaten zijn op de knoop. Een gebruiker kan positieve of negatieve rollen krijgen. Meer over het beheer van transacties in OrientX is terug te vinden in [32].

3.3.3 Beheer van transacties in Sedna

Sedna beheert transacties op nog een andere manier. Het two-phase locking protocol wordt gebruikt. Dit protocol wordt uitgelegd bij het beheer van transacties in sectie 3.3.2. Er zijn twee aparte levels van locking, die gescheiden blijven. Zo zijn er de logische locks en de fysieke locks. De logische locks worden geplaatst voor conflicten op logisch level zoals dirty reads. De fysieke locks worden op een pagina geplaatst als er een fysieke operatie op die pagina wordt uitgevoerd.

Sedna gebruikt drie soorten locks, namelijk de read lock, de write lock en een update lock. Deze locks hebben dezelfde betekenis als bij het multi-granularity protocol, uitgelegd bij het beheer van transacties in sectie 3.3.2. De lock matrix voor de locks van Sedna is weergegeven in Figuur 3.27.

	r	w	u
r	ja	nee	nee
w	nee	nee	nee
u	ja	nee	nee

Figuur 3.27: De lock matrix voor Natix

Net zoals bij Natix wordt bij Sedna een waiting graph bijgehouden voor deadlock detection. Nadat een transactie een bepaalde tijd aan het wachten is om een lock te mogen zetten wordt in de waiting graph of er geen lussen aanwezig zijn. Als er een lus aanwezig is wordt de laatste aanvraag geweigerd. Meer informatie over beheer van transacties in Sedna kan gevonden worden in[40]

3.4 Systemen

Nu een paar algemene design aspecten van native databases aan bod zijn gekomen kunnen de bestaande native DBMS onderzocht worden. Hoe beheren zij de fysieke opslag? Gebruiken ze indexen en zoja, welke? Deze belangrijke vragen en nog meer worden hier beantwoord.

3.4.1 OrientX

Het eerste systeem dat besproken wordt is OrientX. OrientX is schema-gebaseerd, wat wil zeggen dat het gebruik maakt van een schema om belangrijke beslissingen in verband met opslag en ondervragingen te nemen. Dit schema is een descriptief schema, beschreven in sectie 3.1.5.

OrientX gebruikt volgens [32] een multi-granularity opslagstrategie. Dit houdt in dat er verschillende manieren geïmplementeerd zijn om de gegevens op te slaan. Tussen de vier fysieke opslagstrategieën, beschreven in sectie 3.1, wordt een keuze gemaakt aan de hand van het schema.

OrientX ondersteunt XQuery 1.0. Niet alle features worden ondersteund, maar wel de belangrijkste. De queries worden geëvalueerd aan de hand van de Xalgebra algebra. Voor meer informatie over XAlgebra, zie [4]. Het evalueren gaat in grote lijnen als volgt. Eerst wordt de input query geparst naar een syntax tree. Deze wordt getransformeerd naar een XAlgebra expressie. Daarna wordt die expressie door de query optimizer gebruikt om het originele query plan aan de hand van die expressie te optimalizeren. Het uiteindelijk resultaat hiervan is een optimaal fysiek query plan.

Als indexstructuur wordt SUPEX (zie 3.2.3) gebruikt. Voor het opstellen van de structural graph wordt het schema gebruikt.

Het systeem heeft geen ingebouwde security module die de database tegen indringers beschermt. Er is ook geen rekening gehouden met het feit dat het systeem kan crashen. Er worden geen backups genomen, noch zijn er andere recovery methodes voor na een system crash. Het is dus best mogelijk dat er inconsistente en dus onbruikbare gegevens in de database staan na de crash. Meer over OrientX is terug te vinden in [32].

3.4.2 Natix

De ontwerpers van Natix hebben getracht om een aantal doelen na te streven bij het maken van het systeem, namelijk efficiëntie, expressiviteit en flexibiliteit. Deze doelen worden tastbaar gemaakt door het plaatsefficiënt opslaan en het tijdsefficiënt toegang verkrijgen van de gegevens, ondersteuning van

XPath en XQuery, SAX en DOM interfaces en het creëren van een veilige multi-user omgeving.

De eerste twee doelen worden vooral gerealiseerd door de implementatie van de fysieke opslagstrategie. De Natix Storage Engine, beschreven in sectie 3.1.6 legt uit hoe de fysieke opslag gebeurt bij Natix.

Eén van de doelen van Natix was expressiviteit. Hiermee wordt bedoeld dat het systeem alle queries moet kunnen uitvoeren die uitdrukbaar zijn in een gangbare XML-ondervragingstaal. Natix ondersteunt XQuery 1.0 als ondervragingstaal.

Natix implementeert twee indexstructuren, namelijk de Full-text index framework index, beschreven in 3.2.4 en de XSASR index, beschreven in 3.2.5. Deze indexstructuren zijn tot stand gekomen door het volgen van twee denkpistes. Eén denkrichting is het verbeteren en aanpassen van een traditionele full-text index zodat die overweg kan met XML-gegevens. Een andere denkrichting is het maken van een volledig nieuwe indexstructuur, die gemaakt is om met XML-gegevens te werken. Voor verdere informatie over Natix, zie [31]

3.4.3 Lore

Lore, dat voor Lightweight Object Repository staat, is een systeem van een vorige generatie. Het is niet speciaal gemaakt voor XML-gegevens, maar voor semi-gestructureerde gegevens waarin een knoop meerdere ouders kan hebben. Natuurlijk kan het systeem ook XML-gegevens opslaan.

De fysieke opslag bij Lore gebeurt iets anders dan bij de beschreven strategieën. Er wordt ook gebruikt gemaakt van disk pagina's met een vaste grootte en objecten die in het stuk over fysieke opslagstrategieën records genoemd worden. Deze objecten bevatten een enkele knoop en zijn van variabele grootte. Ze worden in de pagina's geplaatst volgens een first-fit algoritme. Er is zelfs een mechanisme ontworpen om objecten te beheren die groter worden dan één enkele pagina.

Het ondervragen van de gegevens gebeurt in een taal speciaal geschreven voor Lore, namelijk Lorel, hetgeen staat voor Lore language. Lorel is een extensie van OQL, dat hier niet besproken wordt. De geïnteresseerde lezer wordt verwezen naar [6].

Lore kan slechts één gebruiker tegelijkertijd toelaten in de database. Er is geen bescherming van de gegevens, zodat elke indringer kan weten wat er in de database staat. Ook recovery bij system crashes ontbreekt bij Lore. Als het systeem crasht, bestaat er een reële kans dat de gegevens inconsistent zullen achterblijven.

Er worden dataguides gebruikt bij Lore. Een dataguide is een beknopte en accurate samenvatting van de structuur van de database. Het is dus een datastructuur die gelijkend is op een descriptief schema. Een dataguide wordt bijgehouden in de database zodat de dataguide ondervraagd kan worden en zo de structuur van de database bepaald kan worden. Voor meer informatie over Lore, zie [33].

3.4.4 Timber

Timber is gebouwd op Shore. Shore staat voor Scalable Heterogeneous Object REpository. Het is een systeem dat secundaire opslag beheert voor objecten. Het is een gulden middenweg tussen een object-geöriënteerde database en een filesysteem. Voor meer informatie over Shore, wordt verwezen naar [7]. Het Shore systeem zal verantwoordelijk zijn voor het beheer van secundair geheugen en voor het bufferen. Timber gebruikt een depth-first subtree-based strategie zoals beschreven in sectie 3.1.2.

Een doel bij het ontwerpen van Timber was om het systeem zo onafhankelijk mogelijk te maken van een ondervragingstaal. Er zijn parsers ontwikkeld voor XQuery, Quilt, XML-QL en XQL, die een query in een van die talen omzet naar een interne representatie, een tree algebra, van een query. De interne representatie van de query wordt uiteindelijk uitgevoerd op de gegevens. Spijtig genoeg wordt alleen de parser voor XQuery onderhouden. Timber ondersteunt dus XQuery en heeft de mogelijkheid ook andere ondervragingstalen te ondersteunen. De tree algebra ondersteunt aanpassingen, maar omdat die nog niet geïmplementeerd zijn in XQuery, zijn ze ook nog niet geïmplementeerd in Timber.

Timber steunt op Shore voor het beheren van gelijktijdige toegang tot gegevens. Aangezien Shore in de eerste plaats gebouwd is om objecten op te slaan, is de vraag of dit altijd goed zal verlopen. Er zijn geen bijkomstige algoritmes geïmplementeerd om gelijktijdige toegang voor XML-gegevens te beheren. Ook bij een systeem crash steunt Timber op de recovery componenten van Shore.

In Timber zijn B-tree indexen geïmplementeerd. Deze indexen hebben als key een tagnaam, attribuut waardes en dergelijke. De inhoud van het element kan ook geïndexeerd worden. Meer informatie over Timber is te vinden in [37].

3.4.5 Tamino

Tamino is een commercieel systeem en geeft weinig prijs over zijn fysieke opslagstrategie, enkel dat de gegevens in boomformaat wordt opgeslagen.

Tamino ondersteunt twee ondervragingstalen, Tamino X-Query en XQuery. XQuery is niet volledig geïmplementeerd, maar wel een groot deel. Zo worden de FLWOR expressies, pad expressies, vele functies en alle datatypes ondersteund. Ook is er een uitbreiding op XQuery geïmplementeerd die aanpassingen kan uitdrukken. Tamino X-Query is een taal gebaseerd op XPath en speciaal gemaakt voor Tamino. Deze taal heeft een aantal extra functionaliteiten voor het opzoeken van tekst en kan aanpassingen uitdrukken.

Tamino maakt gebruik van vele indexstructuren. Er zijn B-tree indexen geïmplementeerd, Full-Text Index Framework indexen en nog andere, waar de details niet van te verkrijgen zijn.

Er is zorg besteed aan het beheer van gelijktijdige toegang en het valideren van handtekeningen van gebruikers zorgt ervoor dat Tamino minder kwetsbaar is voor aanvallen. Systeem falingen worden ook opgevangen. Tamino voorziet hardware-gebaseerde backup mogelijkheden. Hoe dit precies werkt is niet gekend.

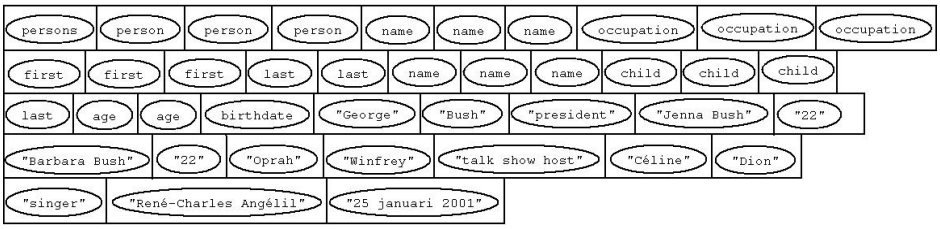
Tamino maakt gebruik van schema informatie. Deze wordt gebruikt om het ondervragen sneller te laten verlopen. Het schema is spijtig genoeg een prescriptief schema, dat de administrator eerst moet ingeven vooraleer er XML-gegevens kunnen ingevoerd worden. Voor meer informatie over Tamino, zie [38].

3.4.6 Sedna

Bij het ontwerp van Sedna is een descriptief schema opslagstrategie ontwikkeld. Het systeem maakt dus gebruik van een descriptief schema. Bij deze strategie worden knopen geclusterd volgens hun positie in het schema. Dit lijkt op een clustering element-based opslagstrategie. Het enige verschil is dat de elementen niet geclusterd worden volgens tagnaam, maar volgens hun plaats in het schema. Voorbeeld 3.11 toont hoe Sedna de elementen opslaat op de harde schijf. Merk op dat het feit dat knopen geclusterd worden volgens hun plaats in het schema, impliceert dat ze volgens tagnaam geclusterd worden als elke tagnaam slechts éénmaal in het schema voorkomt. Bij deze opslagstrategie wordt het structurele gedeelte en het tekstuele gedeelte van de knoop gescheiden. Het structurele gedeelte van de knoop bevat pointers naar tekstuele inhoud, ouders en kinderen. Omdat tekstuele inhoud een variabele lengte heeft wordt deze apart opgeslagen. Hierdoor kan de structurele inhoud geplaatst worden in records van een vaste lengte.

Voorbeeld 3.11. Net zoals bij de clustering subtree-based opslagstrategie kan gesproken worden van instanties van een schema. Zo is elke knoop in de boomstructuur in Figuur 3.2 van het XML-document een instantie van

een knoop in de schemagraaf, weergegeven in Figuur 3.4. Alle instanties van bijvoorbeeld de *person*-knoop in de schemagraaf worden bij elkaar geplaatst op een pagina. Figuur 3.28 toont hoe de records geserializeerd worden en gegroepeerd zodat instanties van dezelfde knoop in de schemagraaf bij elkaar staan. Merk op dat er twee groepjes van *name*-knopen zijn. Dit komt omdat er twee *name*-knopen in de schemagraaf zijn. Er zijn dus instanties gegroepeerd van twee verschillende schema knopen met dezelfde naam. ▲



Figuur 3.28: Fysieke opslagstrategie van Sedna

Sedna ondersteunt de functionaliteiten van de XQuery library en de XQuery Core. Ook is er een uitbreiding op XQuery geïmplementeerd die toelaat aanpassingen uit te drukken. Indexstructuren zijn nog niet geïmplementeerd.

Ook over het beheer van gelijktijdige toegang is nagedacht. Het gebruikt een locking protocol om verschillende synchronisatieproblemen op te lossen. Voorlopig is de eenheid waar een slot op geplaatst kan worden een heel XML-document. Omdat dit de efficiëntie naar beneden haalt, is er een nieuwe methode in de maak. Ondersteuning voor aanvallen van buitenaf is er niet. Ook in het geval van een systeem faling zal Sedna niets ondernemen om de gegevens in een consistente staat te brengen. Meer informatie over Sedna kan teruggevonden worden in [39].

Hoofdstuk 4

Relationele Databases – Bestaande mogelijkheden

4.1 Opslagstrategieën en Indexen

In Hoofdstuk 2 is de moeilijkheid besproken om XML-gegevens op te slaan in relationele databases. Een eerste mogelijkheid om XML-gegevens toch in een relationele database te plaatsen, is het invoegen van een compleet XML-bestand in een kolom van de database. De XML-gegevens worden als één CLOB opgeslagen. Dit is in feite niets meer dan het archiveren van XML-bestanden en hierbij worden de voordelen van een relationele database onbenut gelaten. Een andere manier, die in deze sectie aan bod komt is het versnipperen of „shredding" van het XML-bestand, om het zo toch in een relationele database te kunnen opslaan. Dit versnipperen is eigenlijk niet meer dan het mappen van XML-gegevens op relationele gegevens en dus kan een methode om XML-gegevens te versnipperen in relationele gegevens een mapping genoemd worden.

De boomstructuur van het XML-document wordt in sommige mappings versnipperd in bogen. Bogen zijn de „lijnen" die twee knopen van de boomstructuur verbinden. Om een boog te kunnen identificeren wordt er gebruik gemaakt van unieke nummers die de knopen identificeren. Aangezien een boog twee knopen verbindt, is ze op een unieke manier bepaald door de twee knopen die ze verbindt. In een XML-bestand kan het namelijk niet zijn dat er twee bogen zijn die dezelfde knopen met elkaar verbinden. Het gebruiken van twee nummers voor het identificeren van één bepaalde boog heeft als voordeel dat zo bogen die aan een bepaalde andere boog gelinkt worden door een knoop snel gevonden kunnen worden. In andere mappings worden alleen de tagnamen en de tekstuele inhoud in tabellen geplaatst. In dat geval

is het schema van de relationele tabellen zo opgesteld dat de structuur van het XML-document weer kan opgebouwd worden, of wordt er ergens apart bijgehouden hoe de structuur van de XML-gegevens was.

Aangezien in de relationele database XML-gegevens opgeslagen worden, worden er op deze gegevens ondervragingen gedaan in XML-ondervragingstalen. Er kan namelijk niet verwacht worden dat gebruikers een query in SQL stellen, want een SQL-query op deze versnipperde gegevens is erg complex en onintuïtief. Dus moeten de ondervragingen kunnen gebeuren in een XML-ondervragingstaal. Die XML-query moet dan omgezet worden in een SQL-query. Het resultaat van deze SQL-query is in sommige gevallen een reeks tupels die tesamen een XML subtree vormen. In dat geval moet het resultaat van de query ook terug omgezet worden in XML-gegevens, aangezien een reeks tupels ook onintuïtief is en hier dikwijls niet meteen een boomstructuur in gezien kan worden. Het omzetten van een XML-query naar een SQL-query en het omzetten van een reeks tupels in één of meerdere boomstructuren kan gebeuren in het RDBMS. In andere gevallen bestaan er speciale middleware-applicaties. Deze middleware-applicaties zorgen dan voor een interface bovenop het RDBMS. Ze versnipperen de XML-gegevens en plaatsen de relationele gegevens in tabellen. Ze zetten XML queries om naar SQL-queries en stellen de SQL-query aan het RDBMS. Het resultaat van die SQL-query wordt hierna omgezet in het nodige formaat en in dat juiste formaat aan de gebruiker getoond.

Het mappen van zo een boomstructuur op relationele gegevens kan op verschillende manieren gebeuren. Er kan volgens [9] onderscheid gemaakt worden tussen die mappings door te bepalen wat nodig is om de mapping te bepalen. Dit kan een schema zijn, andere informatie over de XML-gegevens of zelfs een gebruiker die de mapping deels of volledig moet bepalen.

De rest van het hoofdstuk wordt als volgt ingedeeld. Eerst worden een aantal mappings besproken en daarna manieren om XQuery en XSLT naar SQL om te zetten. Hierna wordt in het kort uitgelegd hoe relationele systemen met XML-gegevens omgaan en in hoeverre ze XML-ondervragingstalen ondersteunen. Ten laatste wordt nog even aangehaald wat publishing is. In deze sectie over mappings wordt gebruik gemaakt van het lopend voorbeeld uit Hoofdstuk 3.

4.1.1 Generic Mapping

Dit soort mapping heeft geen speciale informatie over de XML-gegevens nodig om de mapping te kunnen bepalen. De mapping versnippert de boomstructuur in bogen en plaatst die bogen in één of meerdere tabellen. Een boog of edge geeft de relatie tussen twee knopen weer. Een ouder knoop en een kind

knoop worden in een boomstructuur immers altijd verbonden met een boog.

Er zijn meerdere manieren om bogen in één of meerdere tabellen te plaatsen. Ook hier kan dus een onderscheid gemaakt worden. Dit onderscheid kan gemaakt worden in de manier waarop de bogen zelf in tabellen geplaatst worden, en in de manier waarop tekstuele inhoud, oftewel waardes, in de tabellen geplaatst kunnen worden. De verschillende generic mappings worden hieronder verder uitgelegd. Voor meer informatie over generic mapping, zie [11].

4.1.1.1 Opslaan van Bogen

Er zijn drie manieren om bogen in tabellen op te slaan. Er is de edge aanpak, de binary aanpak en de universele tabel aanpak.

Edge aanpak De edge aanpak slaat alle bogen in één enkele tabel op. Deze tabel wordt de edge tabel genoemd. Deze edge tabel heeft als attributen *source*, *ordinal*, *name*, $flag$ en *target*. Hierin is *source* het unieke nummer van de knoop waar de boog vertrekt, de ouder knoop. Het *target*-attribuut is het unieke nummer van de knoop waar de boog aankomt, de kind knoop. Het *target*-attribuut kan ook een waarde bevatten, of een verwijzing naar een waarde. Hoe dit in zijn werk gaat zal verder uitgediept worden in het stuk over het opslaan van waardes. De *name* is de tagnaam van de knoop waarin de boog vertrekt. Het $flag$ attribuut geeft aan of de boog naar een andere interne knoop wijst, of anders geeft het $flag$ attribuut het type aan van de waarde van de knoop waarheen de boog wijst. Dit kan bijvoorbeeld een string zijn of een integer, maar ook een andere knoop. Het *ordinal* attribuut geeft aan naar het hoeveelste kind de boog wijst. Dit attribuut zorgt ervoor dat er rekening gehouden wordt met document orde. Een voorbeeld van een mapping met behulp van de edge aanpak wordt in Voorbeeld 4.1 weergegeven.

Voorbeeld 4.1. In Figuur 4.1 is te zien hoe de knopen genummerd zijn voor deze mapping. De manier van nummering is niet vastgelegd bij deze mapping. Als elke knoop een uniek nummer krijgt is dat voldoende. Hieronder wordt getoond hoe de mapping het XML-document in de edge tabel plaatst.

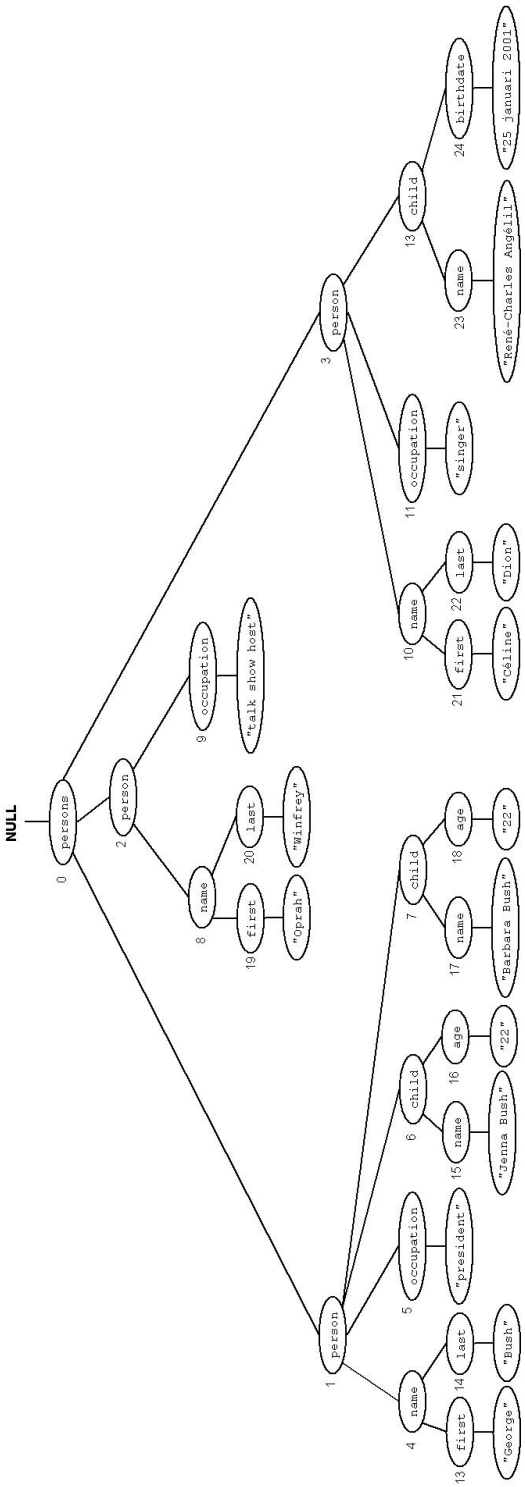
source	*ordinal*	*name*	*flag*	*target*
NULL	1	NULL	"ref"	0
0	1	"persons"	"ref"	1
0	2	"persons"	"ref"	2
0	3	"persons"	"ref"	3
1	1	"person"	"ref"	4
1	2	"person"	"ref"	5
1	3	"person"	"ref"	6
1	4	"person"	"ref"	7
4	1	"name"	"ref"	13
4	2	"name"	"ref"	14
6	1	"child"	"ref"	15
6	2	"child"	"ref"	16
7	1	"child"	"ref"	17
7	2	"child"	"ref"	18
2	1	"person"	"ref"	8
2	2	"person"	"ref"	9
8	1	"name"	"ref"	19
8	2	"name"	"ref"	20
3	1	"person"	"ref"	10
3	2	"person"	"ref"	11
3	3	"person"	"ref"	12
10	1	"name"	"ref"	21
10	2	"name"	"ref"	22
12	1	"child"	"ref"	23
12	2	"child"	"ref"	24
13	1	"first"	"string"	"George"
14	1	"last"	"string"	"Bush"
5	1	"occupation"	"string"	"president"
15	1	"name"	"string"	"Jenna Bush"
16	1	"age"	"int"	22
19	1	"first"	"string"	"Oprah"
20	1	"last"	"string"	"Winfrey"
9	1	"occupation"	"string"	"talk show host"
21	1	"first"	"string"	"Céline"
22	1	"last"	"string"	"Dion"
11	1	"occupation"	"string"	"singer"
23	1	"name"	"string"	"René-Charles Angélil"
24	1	"birthdate"	"string"	"25 january 2001"

▲

In [11] worden de volgende indexen voorgesteld. Om te beginnen wordt er een index op het *source* attribuut geplaatst. Deze index zorgt dat snel alle bogen opgezocht kunnen worden die starten in een bepaalde knoop. Hierdoor worden snel alle knopen gevonden die kinderen zijn van een bepaalde knoop. Een tweede index die voorgesteld wordt is een index op *name* en *target* tesamen. Deze index helpt bij het terugvinden van bogen die beginnen in een knoop met een bepaalde tagnaam en eindigen in een knoop met een bepaald knoopnummer of waarde. Dit heeft als gevolg dat er snel het *source* knoopnummer gevonden kan worden van een bepaalde boog. Als dit knoopnummer nu weer gebruikt wordt tesamen met een tagnaam van de knoop die één level hoger ligt kan zo opwaarts doorheen de boom gelopen worden met behulp van de index.

De edge aanpak heeft geen noemenswaardige problemen als er aanpassingen van de XML-gegevens doorgevoerd worden. Bij toevoegingen van subtrees of knopen moeten er nieuwe bogen in de tabel bijgevoegd worden en eventueel moeten er wat knoopnummers veranderd worden. Bij het verwijderen van subtrees of knopen moeten ook enkel de betreffende bogen verwijderd worden en er moeten eventueel ook enkele knoopnummers veranderd worden. Zelfs het verwisselen van hele subtrees kan gemakkelijk gebeuren door het veranderen van twee target waarden. Ook het aanpassen van een tagnaam kan heel snel doorgevoerd worden, aangezien hiervoor enkel het name attribuut van één enkel tupel moet aangepast worden.

Het omzetten van een query in een XML-ondervragingstaal in SQL is niet vanzelfsprekend. Om een path traversal uit te voeren moet er een join gebeuren op het *source* attribuut van de ene knoop met het *target* attribuut van de andere knoop. Dit zijn namelijk de enige attributen die ouder-kind relaties aanduiden. Een pad kan onbeperkt diep zijn, dus zou ook het aantal joins oneindig kunnen zijn. Natuurlijk is dit niet het geval, namelijk voor een bepaald XML-document is het pad van de root naar een blad van de boom altijd eindig. Maar het aantal joins in de SQL-query hangt af van het aantal keren dat van een ouder naar een kind, of van een kind naar een ouder moet gesprongen worden. Het aantal joins is dus niet voor elke XML-query hetzelfde, hetgeen het omzetten niet gemakkelijker maakt. Voor path traversals van de vorm „$/root/parent1/parent2/.../child$" is het aantal joins bepaald door de path traversal, namelijk voor elke kind-ouder sprong één. Voor path traversals die assen zoals descendant of ancestor gebruiken kan het aantal joins niet bepaald worden aan de hand van de query zelf, aangezien het aantal kind-ouder sprongen of ouder-kind sprongen niet gekend is. Dan kan er gekeken worden naar de maximale diepte van het document om het aantal joins te bepalen. Als er een schema is van de XML-gegevens, kan er gekeken worden op welke hoogte in de XML-boomstructuur de gewenste knopen zich



Figuur 4.1: Nummering van knopen XML-document lopend voorbeeld

kunnen bevinden en zo het aantal joins bepalen. Deze maximale diepte kan initieel wel gevonden worden, namelijk tijdens het versnipperen zelf. Deze waarde kan dan bijgehouden worden. Door aanpassingen kan deze waarde veranderen en moet ze dus altijd aangepast worden. Merk op dat er niet altijd juist geweten is in hoeverre de maximale diepte verandert, maar er wel altijd een bovengrens kan gevonden worden. Bijvoorbeeld, als twee subtrees verwisseld worden, moeten zoals gezegd enkel twee *target*-waardes veranderd worden. Deze aanpassing kan de maximale diepte ten hoogste vermeerderen met $m - 2$. Dit is het geval als een blad van de boom gewisseld wordt met een kind van de root. Het blad bevond zich op diepte m. Dit blad wordt dus eerst ergens anders geplaatst, dus de diepte is nu $m - 1$. Hierna wordt de subtree van lengte $m - 1$ toegevoegd en de maximale diepte is nu $2m - 2$. Het nadeel is wel dat die maximale diepte bij elke aanpassing veranderd moet worden.

Generic mappings hadden als kenmerk dat er niets nodig was - en dus ook geen schema - om de mapping te bepalen. We kunnen dus niet van de aanwezigheid van een schema uit gaan. Joins zijn bij deze aanpak trouwens niet erg snel aangezien de edge tabel erg groot kan worden en de tabel telkens met zichzelf moet gejoint worden. In Voorbeeld 4.2 wordt getoond hoe een query kan omgezet worden als de XML-gegevens met de edge aanpak in een relationele database opgeslagen worden.

Voorbeeld 4.2. Een path traversal wordt hier uitgedrukt in XPath. Een eerste query wordt gegeven door:
`/persons/person/*/name/last`
De omzetting van deze query naar SQL ziet eruit als volgt:

```
SELECT E1.target
FROM Edge AS E1, Edge AS E2, Edge AS E3, Edge AS E4, Edge AS E5,
Edge AS E6
WHERE E1.name = 'last' AND E1.source = E2.target
AND E2.name = 'name' AND E2.source = E3.target
AND E3.source = E4.target
AND E4.name = 'person' AND E4.source = E5.target
AND E5.name = 'persons' AND E5.source = E6.target
AND E6.target = NULL
```

De structuur uit Figuur 4.2 toont hoe deze query is opgebouwd. De `X`-knoop is de gezochte knoop. De bogen zijn gelabeld, ze stellen namelijk een instantie van de Edge tabel voor. `E1` komt overeen met de boog die gelabeld is met 1 enzovoort. Wat er in de query gebeurt is het volgende. Er wordt gezocht naar een knoop, dus een bron of een doel van een boog. De knopen die gelegen zijn op het pad van de root naar die knoop hebben een naam. Er worden

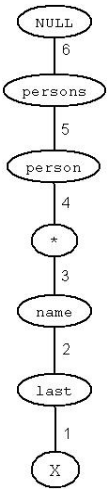
dus telkens bogen gezocht die gelinkt zijn door een bepaalde knoop, hetgeen de voorwaarden van de vorm `E(x).source = E(x+1).target` verklaart. De namen van de knopen kunnen we ook aflezen uit de edge tabel, hetgeen de voorwaarden van de vorm `E(x).name = 'naam'` verklaart.
Een tweede query wordt gegeven door de volgende XPath uitdrukking:
`//person//child/name`
De omzetting van deze query naar SQL ziet eruit als volgt:

```
SELECT E1.target
FROM Edge AS E1, Edge AS E2, Edge AS E3, Edge AS E4, Edge AS E5
WHERE E1.name = 'name' AND E1.source = E2.target
AND E2.name = 'child'
AND (E2.source = E3.target
OR ((E2.source = E4.target
OR (E2.source = E5.target AND E5.source = E4.target))
AND E4.source = E3.target))
AND E3.name = 'person'
```

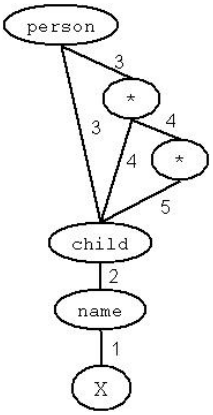
De structuur uit Figuur 4.3 toont hoe deze query is opgebouwd. Dit is gelijkaardig met hoe de eerste query is opgebouwd, met twee verschillen. Een eerste verschil is dat er een willekeurige boog met label `person` gekozen wordt. Merk op dat hiervoor geen paden moeten doorlopen worden, de boog kan zo uit de tabel worden geplukt. Een tweede verschil is dat er nu niet geweten is hoeveel bogen er tussen de `person` knoop en de `child` knoop liggen. Aan de hand van de maximale diepte van de boom is het wel mogelijk een maximum op dit aantal bogen te leggen. Het pad tussen de twee knopen kan dus een lengte 1, 2 of 3 hebben. Dit kan gesimuleerd worden door de drie mogelijke scenario's in de query te schrijven. Er worden dus 0, 1 of 2 willekeurige ongelabelde knopen tussen de `person` en de `child` knoop geplaatst. ▲

Binary aanpak De binary aanpak slaat de bogen op in meerdere tabellen. Voor elke tagnaam dat het XML-document bevat wordt een tabel gemaakt. Deze tabellen worden de binary tabellen genoemd. Elke tabel noemt B_{name}, waarin name een tagnaam is. Deze tabellen bevatten geen *name* attribuut omdat de naam van de knoop uit de tabel is af te lezen. De binary tabellen hebben als attributen *source*, *ordinal*, $flag$ en *target*. De betekenis van de attributen is dezelfde als bij de edge aanpak. Het aantal binary tabellen kan erg oplopen. Er is er namelijk één voor elke tagnaam. In Voorbeeld 4.3 wordt getoond hoe XML-gegevens in tabellen worden geplaatst met behulp van de binary aanpak.

Voorbeeld 4.3. Per *name*-attribuut uit de edge tabel wordt in de binary aanpak een nieuwe tabel gemaakt.



Figuur 4.2: Structuur van de eerste SQL-query



Figuur 4.3: Structuur van de tweede SQL-query

B_{noname}

source	*ordinal*	*flag*	*target*
NULL	1	"ref"	0

$B_{persons}$

source	*ordinal*	*flag*	*target*
0	1	"ref"	1
0	2	"ref"	2
0	3	"ref"	3

B_{person}

source	*ordinal*	*flag*	*target*
1	1	"ref"	4
1	2	"ref"	5
1	3	"ref"	6
1	4	"ref"	7
2	1	"ref"	8
2	2	"ref"	9
3	1	"ref"	10
3	2	"ref"	11
3	3	"ref"	12

B_{name}

source	*ordinal*	*flag*	*target*
4	1	"ref"	13
4	2	"ref"	14
8	1	"ref"	19
8	2	"ref"	20
10	1	"ref"	21
10	2	"ref"	22
15	1	"string"	"Jenna Bush"
23	1	"string"	"René-Charles Angélil"

B_{child}

source	*ordinal*	*flag*	*target*
6	1	"ref"	15
6	2	"ref"	16
7	1	"ref"	17
7	2	"ref"	18
12	1	"ref"	23
12	2	"ref"	24

B_{first}

source	*ordinal*	*flag*	*target*
13	1	`"string"`	`"George"`
19	1	`"string"`	`"Oprah"`
21	1	`"string"`	`"Céline"`

B_{last}

source	*ordinal*	*flag*	*target*
14	1	`"string"`	`"Bush"`
20	1	`"string"`	`"Winfrey"`
22	1	`"string"`	`"Dion"`

$B_{occupation}$

source	*ordinal*	*flag*	*target*
5	1	`"string"`	`"president"`
9	1	`"string"`	`"talk show host"`
11	1	`"string"`	`"singer"`

B_{age}

source	*ordinal*	*flag*	*target*
16	1	`"int"`	22

$B_{birthdate}$

source	*ordinal*	*flag*	*target*
24	1	`"string"`	`"25 january 2001"`

▲

Bij de binary aanpak moet er geen index zijn op het *name* attribuut. De naam is namelijk bepaald door de naam van de binary tabel. Er kan wel een index geplaatst worden op het *source* attribuut. Dit is om dezelfde redenen als bij de edge aanpak. Verder kan er een index geplaatst worden op het *target* attribuut van elke binary tabel. Dit is in wezen dezelfde index als de index op *name* en *target* tesamen bij de edge aanpak. Het plaatsen van deze index heeft dezelfde redenen als bij de edge aanpak. Aangezien het aantal binary tabellen kan oplopen, afhankelijk van het aantal tagnamen, kan het aantal indexen ook groot worden.

Aanpassingen aan de XML-gegevens kunnen voor wat meer problemen zorgen dan bij de edge aanpak. Als er namelijk een subtree of knopen toegevoegd worden met tagnamen die nog niet voorkwamen in het XML-document, dan moeten er binary tabellen bij komen. Ook moeten er dan nieuwe indexen voor die tabellen aangemaakt worden. Dit is niet echt bevorderlijk voor de snelheid van de aanpassing. Als er subtrees of knopen verwijderd worden waardoor een binary tabel leeg is, is de tabel overbodig geworden. In dat geval kan ze verwijderd worden, hetgeen de aanpassing wat vertraagt. De tabel kan ook gewoon blijven staan, maar dan ontstaat het risico dat er veel lege tabellen blijven staan als de XML-gegevens dik-

wijls aangepast worden. Naamsveranderingen kunnen ook problemen met zich meebrengen. Als de nieuwe naam al voorkwam in het XML-document, moet enkel het tupel van tabel verhuizen. Maar in het geval dat de nieuwe naam niet voorkwam, moet er een nieuwe binary tabel aangemaakt worden. Als door de naamsverandering een binary tabel leeg komt te staan, gelden dezelfde keuzes als bij het verwijderen van een knoop. Voor de rest zijn er bij aanpassingen van het XML-document dezelfde veranderingen in de tabellen als bij de edge aanpak.

Het omzetten van een query bij de binary aanpak verloopt ongeveer hetzelfde als bij de edge aanpak. Het verschil is dat de joins op kleinere tabellen uitgevoerd worden en dat er nu bewust een tabel met de juiste naam gekozen wordt om te joinen in plaats van telkens gewoon de edge tabel te kiezen. Bij de edge aanpak wordt de edge tabel steeds met zichzelf gejoined. Bij de binary aanpak worden er telkens de binary tabellen gejoined, wat niet zo kostelijk is omdat ze veel kleiner zijn dan de edge tabel. Een probleem bij de binary aanpak is dat bij paden waar een wildcard ($*$) in voorkomt, niet geweten is welke binary tabel nu eigenlijk onderzocht moet worden, dus moet elke binary tabel in de SQL-query opgenomen worden. De problematiek dat het niet geweten is hoeveel joins er moeten gebeuren voor bepaalde path traversals, is hetzelfde als bij de edge aanpak. In Voorbeeld 4.4 wordt getoond hoe een XML-query omgezet wordt in een SQL-query voor de binary aanpak.

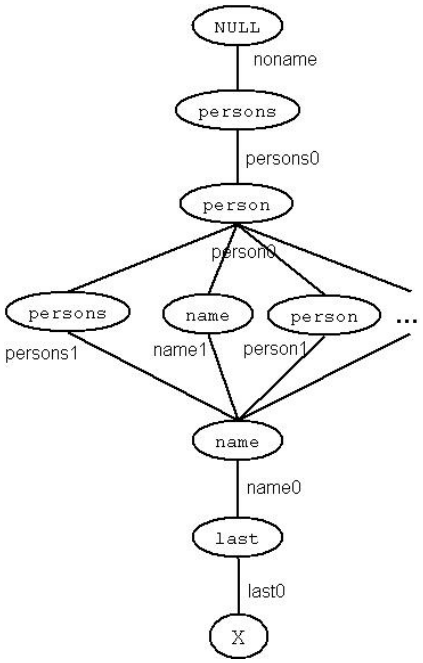
Voorbeeld 4.4. In dit voorbeeld worden dezelfde queries omgezet als in Voorbeeld 4.2. De eerste XPath uitdrukking wordt als volgt vertaald naar SQL als de binary methode gebruikt wordt.

```
SELECT B_last0.target
FROM B_persons AS B_persons0, B_person AS B_person0,
B_name AS B_name0, B_last AS B_last0, B_noname,
B_persons AS B_persons1, B_person AS B_person1,
B_name AS B_name1, B_child1AS B_child1, B_first AS B_first1,
B_last AS B_last1, B_occupation AS B_occupation1,
B_age AS B_age1, B_birthdate AS B_birthdate1
WHERE B_last0.source = B_name.target AND (
(B_name0.source = B_persons1.target AND
B_persons1.source = B_person0.target) OR
(B_name0.source = B_person1.target AND
B_person1.source = B_person0.target) OR
(B_name0.source = B_name1.target AND
B_name1.source = B_person0.target) OR
(B_name0.source = B_child1.target AND
B_child1.source = B_person0.target) OR
```

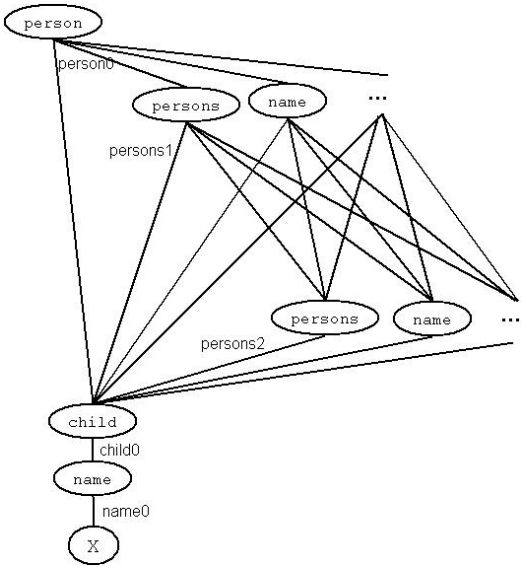
```
(B_name0.source = B_first1.target AND
B_first1.source = B_person0.target) OR
(B_name0.source = B_last1.target AND
B_last1.source = B_person0.target) OR
(B_name0.source = B_occupation1.target AND
B_occupation1.source = B_person0.target) OR
(B_name0.source = B_age1.target AND
B_age1.source = B_person0.target) OR
(B_name0.source = B_birthdate1.target AND
B_birthdate1.source = B_person0.target))
AND B_person0.source = B_persons0.target AND
B_persons0.source = B_noname.target
AND B_noname.source = NULL
```

Dit is een behoorlijk lange query. De boomstructuur uit Figuur 4.4 toont de structuur van deze SQL-query. Voor het bovenste en het onderste stuk van de graaf is de structuur hetzelfde als die in Figuur 4.2, voor de edge aanpak. Het verschil is namelijk dat er voor de wildcard '*' niet geweten is welke tabel er genomen moet worden, aangezien de tagnaam van de knoop niet bekend is. Daarom moeten alle mogelijke tagnamen ertussen geplaatst worden en dus moet er een instantie van elke mogelijke tabel zijn die de mogelijke knopen en bogen voorstellen.
De omzetting van de tweede query in SQL is nog langer dan de eerste en zou zeer onduidelijk zijn. Daarom wordt enkel de structuur van de SQL-query gegeven in Figuur 4.5. Net zoals in Figuur 4.3 zijn er drie verschillende lengtes van paden om tussen de `person` en de `child` knoop. Bij de edge aanpak was het mogelijk een knoop aan te duiden waarvan de naam niet gekend was. Bij de binary aanpak gaat dit niet zoals hierboven bij de omzetting van de eerste query duidelijk geworden is. Er zijn dus voor elke tussentijdse knoop, hier dus maximaal twee, instanties van alle mogelijke binary tabellen nodig. Die moeten met elkaar verbonden worden. Het is duidelijk dat de SQL-query erg lang wordt, aangezien elke tussenliggende knoop van de bovenste rij de ouder kan zijn van een andere tussenliggende knoop van de onderste rij. ▲

Universal table aanpak Bij de universal table aanpak worden net als bij de edge aanpak de bogen in één enkele tabel opgeslagen. Deze tabel is de universal tabel. Het verschil met de edge aanpak is dat bij de universal table aanpak voor elke tagnaam er een paar attributen zijn, waar er in de edge tabel voor alle namen dezelfde attributen gelden. Zo zijn er voor elke tagnaam „name" de attributen $name_{source}$, $name_{ordinal}$, $name_{flag}$, $name_{target}$. Deze universal tabel kan dus bekomen worden door een full-outer join te doen



Figuur 4.4: Structuur van de eerste SQL-query



Figuur 4.5: Structuur van de tweede SQL-query

van alle binary tabellen. Deze tabel kan een erg grote hoeveelheid attributen krijgen als er veel verschillende tagnamen in het XML-bestand voorkomen. Het aantal tupels in de universal tabel is hetzelfde als het aantal tupels in de edge tabel voor eenzelfde XML-bestand. Het nadeel bij deze aanpak is vooral het grote aantal NULL waardes. Alhoewel de relationele DBMS van tegenwoordig de NULL waardes efficiënt opslaan, is een grote hoeveelheid NULL waardes ongewenst. De volledige universal tabel zou erg veel ruimte innemen en is daarom niet weergegeven. Het is echter niet moeilijk om zich een universal table voor te stellen.

Er worden best indexen geplaatst op elk *source*-attribuut en op elk *target*-attribuut. Dit om dezelfde redenen als bij de binary aanpak. Aangezien dit er veel kunnen zijn, afhankelijk van het aantal tagnamen, zullen er ook veel indexen komen. Dit zijn er evenveel als bij de binary aanpak.

Bij het aanpassen van de XML-gegevens is er voor de universal table aanpak ook een probleem. Dit probleem ontstaat, net zoals bij de binary aanpak, als er een knoop of subtree wordt ingevoegd die tagnamen bevat die nog niet voorkwamen in het XML-document. Er moeten namelijk attributen bijkomen in de universal tabel, voor elke nieuw geïntroduceerde naam vier. Ook moeten er weer nieuwe indexen gecreëerd worden. Als een knoop of subtree verwijderd wordt waardoor een deel attributen onnodig worden kunnen er twee dingen gebeuren. Een keuze kan zijn om de attributen te laten staan, waardoor onnodig plaats verspild wordt in de tabel. Een andere keuze is om de attributen weg te halen, hetgeen de aanpassing vertraagt. Om te weten wanneer die attributen verwijderd mogen worden, moet er geweten zijn wanneer alle tupels een NULL-waarde hebben voor dat attribuut. Dit moet ook gecheckt worden en vertraagt dus nog meer de aanpassing. Als er een naamsverandering gebeurt moeten ook nieuwe attributen en indexen gecreëerd worden als de nieuwe naam nog niet voorkwam in de XML-gegevens. Als de oude naam het laatste voorkomen van die naam was in het XML-document dan gelden dezelfde problemen als bij het verwijderen van een subtree of knoop waardoor attributen overbodig worden. Voor de rest zijn de aanpassingen in de tabellen gelijkaardig als bij de edge tabel.

Bij het omzetten van een XML-query naar SQL moet er, net als bij de edge aanpak, een join gebeuren op de complete tabel als er een path traversal omgezet dient te worden. Net als bij de edge aanpak is dit een grote tabel en het joinen duurt dus langer dan bij de binary aanpak voor de meeste gevallen. Bij het gebruik van een wildcard is er wel geweten welke tabel onderzocht moet worden, omdat er maar één tabel is, maar welke kolommen onderzocht moeten worden is niet geweten. Er moet dus voor elke tagnaam de betreffende kolommen onderzocht worden. De problematiek dat het niet geweten is hoeveel joins er moeten gebeuren voor bepaalde path traversals,

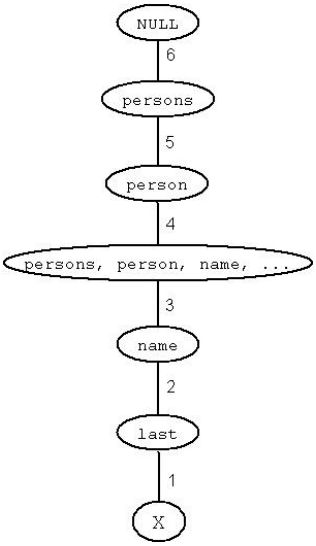
is hetzelfde als bij de edge aanpak en de binary aanpak. In Voorbeeld 4.5 wordt getoond hoe een XML-query omgezet wordt in een SQL-query voor de universal table aanpak.

Voorbeeld 4.5. Ook voor de universal table worden dezelfde queries omgezet. De eerste query wordt omgezet naar de volgende SQL-query:

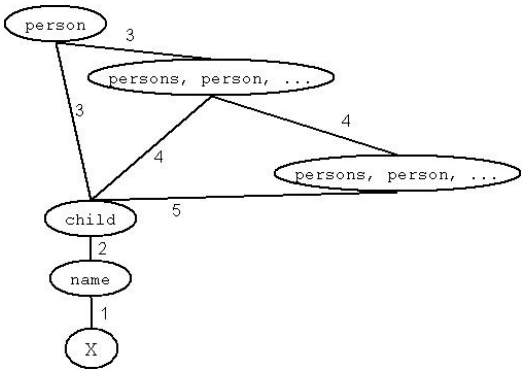
```
SELECT U1.last_target
FROM Universal AS U1, Universal AS U2, Universal AS U3, Universal
AS U4, Universal AS U5, Universal AS U5
WHERE U1.last_source = U2.name_target AND (
(U2.name_source = U3.persons_target AND
U3.persons_source = U4.person_target) OR
(U2.name_source = U3.person_target AND
U3.person_source = U4.person_target) OR
(U2.name_source = U3.name_target AND
U3.name_source = U4.person_target) OR
(U2.name_source = U3.child_target AND
U3.child_source = U4.person_target) OR
(U2.name_source = U3.first_target AND
U3.first_source = U4.person_target) OR
(U2.name_source = U3.last_target AND
U3.last_source = U4.person_target) OR
(U2.name_source = U3.occupation_target AND
U3.occupation_source = U4.person_target) OR
(U2.name_source = U3.age_target AND
U3.age_source = U4.person_target) OR
(U2.name_source = U3.birthdate_target AND
U3.birthdate_source = U4.person_target))
AND U4.person_source = U5.persons_target AND
U5.persons_source = U6.noname_target
AND U6.noname_source = NULL
```

Deze SQL-query lijkt heel sterk op die in Voorbeeld 4.4. Dit komt omdat de strategie hetzelfde is. Het feit dat we de tagnaam van de tussenliggende knopen niet weten speelt hier de grootste rol in. Om de juiste *target*- en *source*-kolommen te kunnen kiezen moet namelijk de naam geweten zijn van de knoop. De structuur van de query, weergegeven in Figuur 4.6 lijkt op die van Figuur 4.2. Dit komt omdat er voor de tussenliggende knopen maar één instantie van de universal tabel aangemaakt moet worden, net zoals dat het geval was bij de edge tabel. Dit kan enkel maar omdat als een knoop een bepaalde tagnaam heeft, die geen andere tagnaam kan hebben en dus de instantie van de universal tabel hergebruikt kan worden.

Voor de tweede XPath uitdrukking is net als bij de eerste de SQL-query gelijkaardig aan de SQL-query voor de binary aanpak. Om dezelfde redenen als bij de eerste XPath uitdrukking is de structuur van de query, weergegeven in Figuur 4.7 gelijkaardig aan de structuur van de query voor de edge aanpak, weergegeven in Figuur 4.3. ▲



Figuur 4.6: Structuur van de eerste SQL-query



Figuur 4.7: Structuur van de tweede SQL-query

4.1.1.2 Opslaan van Waardes

Er zijn twee manieren om waardes op te slaan. Waardes zijn de tekstuele inhoud in XML-gegevens. Deze inhoud kan een string zijn maar kan ook een

getal of datum voorstellen. Er is de inlining aanpak en de separate value tables aanpak om deze waardes op te slaan. Merk op dat bij het bespreken van manieren om bogen op te slaan er overal de inlining methode gebruikt is in de voorbeelden.

Inlining aanpak Bij de inlining aanpak worden de waardes mee opgeslagen in de tabellen waarin de bogen zijn opgeslagen. Voorbeelden van hoe de inlining aanpak de waardes opslaat zijn hierboven in 4.1.1.1 te vinden.

Bij het aanpassen van de tekstuele waardes en dus van het *target*-attribuut is er geen enkel probleem. De waarde wordt gewoon aangepast en er verandert niets aan een ander tupel in de database.

Het omzetten van een XML-query die een waarde bevat naar een SQL-query brengt geen extra moeilijkheden met zich mee. Merk op dat er hier verondersteld wordt dat er één *target*-attribuut is. Bij de meeste RDBMS is het niet mogelijk om integers en strings in eenzelfde kolom op te slaan. Dit kan worden omzeild door voor elk type gespecifieerd in $flag$ een andere *target*-kolom te maken. In Voorbeeld 4.6 wordt een voorbeeld gegeven van hoe een query omgezet wordt als de gegevens opgeslagen zijn met de inlining aanpak. De bogen zijn opgeslagen met behulp van de edge aanpak.

Voorbeeld 4.6. Om de omzetting naar SQL te verduidelijken voor het opslaan van waardes, zullen twee andere queries omgezet worden. Dit om het verschil tussen de inlining aanpak en de value aanpak te kunnen verduidelijken. Een eerste XPath uitdrukking ziet er als volgt uit.

```
//child[age>18]/name
```

De omzetting van deze uitdrukking naar een SQL-query is als volgt. Er wordt de edge aanpak gebruikt om bogen op te slaan. Dit is zo gekozen omdat de grootte van de SQL-query beperkt blijft. Het is natuurlijk ook mogelijk om deze uitdrukkingen om te zetten voor de inlining aanpak gecombineerd met de binary aanpak of universal tabel aanpak.

```
SELECT E1.target
FROM Edge AS E1, Edge AS E2, Edge AS E3, Edge AS E4
WHERE E1.name = 'name' AND E1.source = E2.target
AND E2.name = 'child' AND E3.source = E2.source
AND E4.source = E3.target AND E4.name = 'age'
AND E4.target > 18
```

De structuur van de query kan gevonden worden in Figuur 4.8.
Een tweede XPath uitdrukking wordt gegeven door.

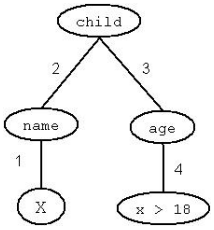
```
//person[last='Bush']/first
```

De omzetting van de SQL-query met de edge aanpak ziet er als volgt uit.

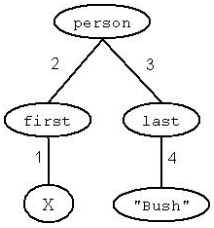
```
SELECT E1.target
```

```
FROM Edge AS E1,
WHERE E1.name = 'first' AND E1.source = E2.target
AND E2.name = 'person' AND E3.source = E2.source
AND E4.source = E3.target AND E4.name = 'last'
AND E4.target = 'Bush'
```

De structuur van de query wordt gegeven door Figuur 4.9. ▲



Figuur 4.8: Structuur van de eerste SQL-query



Figuur 4.9: Structuur van de tweede SQL-query

Separate value tables aanpak Een andere manier om de waardes op te slaan is om voor elk type van waarde een aparte tabel te maken. Dit worden value tabellen genoemd. De waardes worden dan in de tabel met het type van de waarde geplaatst. In de tabellen waar de bogen worden opgeslagen, kan het target attribuut ook een referentie naar een waarde in een value tabel bevatten. Een value table heeft als attributen een vid en een value. Het vid is een uniek nummer zodat het target attribuut in de tabel met de bogen naar deze waarde kan verwijzen. Het value attribuut bevat de waarde zelf. Een voorbeeld van hoe waardes kunnen opgeslagen worden met de separate value tables aanpak is terug te vinden in Voorbeeld 4.7. In het voorbeeld wordt de edge aanpak gebruikt om bogen op te slaan.

Voorbeeld 4.7. In plaats van expliciete waardes staan er nu v_{id} waardes in de *target* kolom.

source	*ordinal*	*name*	*flag*	*target*
NULL	1	NULL	"ref"	0
0	1	"persons"	"ref"	1
0	2	"persons"	"ref"	2
0	3	"persons"	"ref"	3
1	1	"person"	"ref"	4
1	2	"person"	"ref"	5
1	3	"person"	"ref"	6
1	4	"person"	"ref"	7
4	1	"name"	"ref"	13
4	2	"name"	"ref"	14
6	1	"child"	"ref"	15
6	2	"child"	"ref"	16
7	1	"child"	"ref"	17
7	2	"child"	"ref"	18
2	1	"person"	"ref"	8
2	2	"person"	"ref"	9
8	1	"name"	"ref"	19
8	2	"name"	"ref"	20
3	1	"person"	"ref"	10
3	2	"person"	"ref"	11
3	3	"person"	"ref"	12
10	1	"name"	"ref"	21
10	2	"name"	"ref"	22
12	1	"child"	"ref"	23
12	2	"child"	"ref"	24
13	1	"first"	"string"	v_1
14	1	"last"	"string"	v_2
5	1	"occupation"	"string"	v_3
15	1	"name"	"string"	v_4
16	1	"age"	"int"	vv_5
19	1	"first"	"string"	v_6
20	1	"last"	"string"	v_7
9	1	"occupation"	"string"	v_8
21	1	"first"	"string"	v_9
22	1	"last"	"string"	v_{10}
11	1	"occupation"	"string"	v_{11}
23	1	"name"	"string"	v_{12}
24	1	"birthdate"	"string"	v_{13}

V_{int}

v_{id}	*value*
v_5	22

V_{string}

v_{id}	*value*
v_1	"George"
v_2	"Bush"
v_3	"president"
v_4	"Jenna Bush"
v_6	"Oprah"
v_7	"Winfrey"
v_8	"talk show host"
v_9	"Céline"
v_{10}	"Dion"
v_{11}	"singer"
v_{12}	"René-Charles Angélil"
v_{13}	"25 january 2001"

▲

Er kan best een index geplaatst worden in elke tabel op het v_{id}. Zo kan rap een waarde teruggevonden worden waarnaar in het *target*-attribuut van de tabel met de bogen verwezen werd. Ook wordt er best een index geplaatst op het *value*-attribuut om snel in de waardes te kunnen zoeken. Dit is om dezelfde redenen als bij de inlining aanpak. Merk op dat bij de manieren om bogen op te slaan er telkens een index geplaatst werd op *target*. Dit komt nu mooi uit. Als er namelijk een knoop gezocht wordt die een bepaalde inhoud heeft, kan die inhoud in de value tabellen opgezocht worden en zo kan het v_{id} bekomen worden. Nu kan de index op het *target*-attribuut gebruikt worden om de knoop te zoeken die als *target* het gevonden v_{id} bevat.

Het aanpassen van waardes is bij de separate value table aanpak ook geen groot probleem. Als de nieuwe waarde een ander type heeft moet het tupel met de oude waarde simpelweg verwijderd worden uit de value tabel van het oude type en daarna moet een nieuw tupel ingevoegd worden in de value tabel van het nieuwe type. Dit tupel bevat dan de nieuwe waarde en hetzelfde vid als dat van het tupel dat verwijderd werd, om de referentiële integriteit te behouden.

Een XML-query die een waarde bevat omzetten naar een SQL-query brengt een beetje extra werk met zich mee. Er moet namelijk voor elke waarde die gerefereerd wordt, de value tabellen doorzocht worden. Nadat dit gedaan is moet er een join gebeuren op het vid attribuut van de value tabel en het target attribuut van de tabellen met de bogen in. Dit zorgt voor een extra join. In Voorbeeld 4.8 wordt een voorbeeld gegeven van hoe een query

omgezet wordt als de gegevens opgeslagen zijn met de separate value tables aanpak. De bogen zijn opgeslagen met behulp van de edge aanpak.

Voorbeeld 4.8. De omzetting voor de eerste XPath uitdrukking met de edge aanpak wordt gegeven door de volgende SQL-query.

```
SELECT V2.value
FROM V_int AS V1, V_string AS V2, Edge AS E1, Edge AS E2, Edge AS
E3, Edge AS E4
WHERE E1.name = 'name' AND E1.source = E2.target
AND E2.name = 'child' AND E3.source = E2.source
AND E4.source = E3.target AND E4.name = 'age'
AND E4.flag = 'int' AND E4.target = V1.vid AND V1.value > 18
AND E1.target = V2.vid AND E1.flag = 'string'
```

De structuur van de query is terug te vinden in Figuur 4.10. Een verschil met de query voor de inlining aanpak is ten eerste dat er in principe niet geweten is wat voor waarde het resultaat heeft. Dit kan wel afgeleid worden uit de waarde van $flag$ maar hier wordt verondersteld dat de waarde van de naam een string moet zijn. Deze restrictie kan gemakkelijk opgelegd worden door het $flag$ attribuut. Een ander verschil is dat de SQL-query nu een tussenstap maakt. Er wordt namelijk de referentie naar een waarde in een value tabel gevolgd.
De omzetting van de tweede XPath uitdrukking met de edge aanpak wordt gegegeven door de volgende SQL-query.

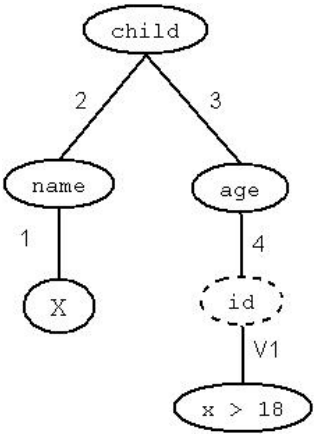
```
SELECT V2.value
FROM V_string AS V1, V_string AS V2, Edge AS E1, Edge AS E2, Edge
AS E3, Edge AS E4
WHERE E1.name = 'first' AND E1.source = E2.target
AND E1.flag = 'string'
AND E2.name = 'person' AND E3.source = E2.source
AND E4.source = E3.target AND E4.name = 'last'
AND E4.flag = 'string' AND E4.target = V1.vid
AND V1.value = 'Bush'
```

Figuur 4.11 toont de structuur van de SQL-query. Ook hier zijn er dezelfde verschillen met de inlining aanpak als hierboven. ▲

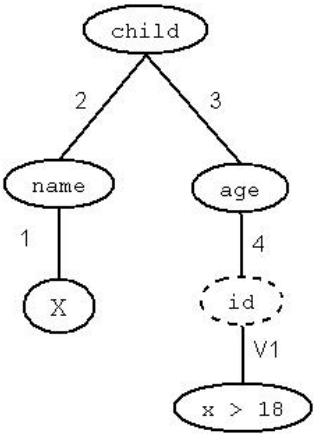
4.1.1.3 Voor- en nadelen

Nu de manieren besproken zijn waarop een generic mapping kan gebeuren, kan de vraag gesteld worden naar de sterktes en zwaktes van deze mapping.

Er is opgemerkt dat, telkens een path traversal moet uitgevoerd worden er joins moeten gebeuren. Aangezien dit de minst efficiënte operatie is die



Figuur 4.10: Structuur van de eerste SQL-query



Figuur 4.11: Structuur van de tweede SQL-query

een RDBMS kan uitvoeren, is een groot aantal joins nefast voor de snelheid van de ondervraging. Deze mapping zorgt dus dat path traversals niet snel uitgevoerd kunnen worden. Queries die relationele vragen uitdrukken daarentegen, gaan sneller. Als bijvoorbeeld de waardes van alle occupation knopen gezocht worden, kan deze ondervraging in een efficiënte SQL-query omgezet worden.

Deze mapping kan gemakkelijk geïmplementeerd worden door een middleware-applicatie of eventueel door het RDBMS zelf. Dit komt omdat er geen informatie over de XML-gegevens nodig is om de mapping te bepalen en om de omzetting van de queries te bepalen. Gebruikers kunnen dan met XML-gegevens werken en geen idee hebben dat er eigenlijk een relationele database achter het systeem zit.

Deze mapping is een goede keuze als gebruikers enkel met XML-gegevens willen werken en geen weet willen hebben van de relationele database die de gegevens beheert. Het is ook best dat de XML-gegevens eigenlijk vooral op de inhoud van de knopen worden ondervraagd. Het ondervragen over de structuur van het document gaat trager, zoals hierboven vermeld. Als de meeste vragen die over de XML-gegevens gesteld worden van een relationele aard zijn, kan deze mapping voor een redelijk efficiënte afhandeling van de vragen zorgen.

4.1.2 User-provided Mapping

De user-provided mapping gebruikt menselijke tussenkomst om de mapping te bepalen.

Een gebruiker, meestal een beheerder, bepaalt eerst hoe de XML-gegevens worden opgedeeld. Die gebruiker heeft normaal gezien weet van het schema of tenminste een deel van het schema van de XML-gegevens. Aan de hand van dat schema kan hij zijn keuzes maken. Dan bepaalt die gebruiker het schema van de relationele database en tenslotte kiest hij welke stukken XML-gegevens worden geplaatst in welke tabel en welke kolom. Merk op dat de gebruiker kan kiezen welke XML-gegevens hij in de tabellen plaatst en welke niet. Het kan dus zijn dat delen van de gegevens niet in de tabellen staan. Deze delen van de XML-gegevens zullen dus later ook niet ondervraagd kunnen worden. Ook de grootte van een te mappen stukje XML-gegevens wordt door de gebruiker bepaald. Hij kan bijvoorbeeld kiezen om enkel tekstuele inhoud van knopen te mappen, complete knopen of zelfs hele subtrees.

Er bestaan applicaties die de gebruiker helpen met het bepalen van een mapping, de user-provided mapping applicaties. Een voorbeeld van zo een user-provided mapping applicatie is Clio [22]. Deze applicaties hebben dikwijls een schema nodig om hun werk te kunnen doen. Ze kunnen meestal

voorstellen doen over de te kiezen mapping. Deze applicaties vergemakkelijken taken zoals het creëren van de relationele tabellen of eventuele indexen. De grootste taak van een dergelijke applicatie is dan natuurlijk om de uiteindelijke XML-gegevens in de juiste kolom te plaatsen. Zo hoeft de gebruiker zich daar niet mee bezig te houden.

Het nadeel van een user-provided mapping is dat ook de ondervragingen in een XML-ondervragingstaal door de gebruiker moeten omgezet worden naar SQL-queries. Een user-provided mapping applicatie kan deze taak voor de gebruiker wel gemakkelijker maken of zelfs van de gebruiker overnemen. Er dient zich hier wel een probleem als de gestelde ondervraging niet op te lossen is met de gegevens in de tabellen, omdat niet alle gegevens gemapt waren. Dit is een probleem dat enkel op te lossen is door de query te stellen op het originele document, of de mapping aan te passen zodat de gewenste gegevens wel in de tabellen staan. Doordat de gebruiker de mapping meestal wel intuïtief heeft opgesteld, zal het ondervragen van de gegevens in SQL trouwens ook veel gemakkelijker zijn.

Het plaatsen van indexen is ook een taak die voor de gebruiker is weggelegd. Aangezien deze de mapping heeft opgesteld en dus meestal weet heeft van de betekenis van de gegevens kan hij ook voorzien welke gegevens dikwijls opgevraagd zullen worden. De user-provided mapping applicaties kunnen ook voorstellen doen over het plaatsen van indexen.

Het aanpassen van de XML-gegevens zal niet voor noemenswaardige problemen zorgen. Enkel als er nieuwe gegevens worden toegevoegd die ook in de tabellen moet komen te staan, maar nog geen plaats hebben, moet de mapping herzien worden. Er moet namelijk meestal een tabel of een kolom bijkomen in de relationele database waar de gegevens in geplaatst kunnen worden. Als hier geen user-provided mapping applicatie is die deze taak op zich kan nemen, kan het invoegen van de gegevens in de tabellen een vervelende taak worden. Er moet namelijk meestal opgezocht worden waar die gegevens moeten bijkomen vooraleer ze op de juiste plaats kunnen ingevoegd worden. Het verwijderen van gegevens zal normaal gezien geen problemen opleveren als de gebruiker de juiste beperkingen gedefinieerd heeft zoals referentiële integriteit.

De user-provided mapping is een mapping die zorgt dat de gebruiker nauw in contact komt met de relationele database. Queries kunnen naderhand ook het gemakkelijkst in SQL zelf geschreven worden, aangezien queries in een XML-ondervragingstaal omzetten naar SQL door de gebruiker zelf moet gedaan worden of door de user-provided mapping applicatie. Soms zijn de queries zelfs niet om te zetten. Daarom wordt deze mapping best enkel gebruikt als de XML-gegevens enkel binnenkomen en rechtstreeks in een relationele database geplaatst moeten worden in bijvoorbeeld een bedrijf. De beheer-

der kan dan de gegevens mappen op de tabellen in de relationele database en andere werknemers uit het bedrijf kunnen hun ondervragingen doen in een taal die hun bekend is, namelijk SQL. Ook wordt best een user-provided mapping applicatie gebruikt om zo het werk van tabellen en indexen creëren, het vullen van de tabellen enzovoort door de applicatie te laten doen, terwijl de beheerder zijn tijd in belangrijker zaken kan steken.

4.1.3 Schema-inferred Mapping

De schema-inferred mapping gebruikt een schema van de XML-gegevens om de mapping te bepalen. Er zijn meerdere manieren om aan de hand van een schema XML-gegevens te mappen op relationele gegevens. Hier wordt de methode besproken die voorgesteld is in [12].

Eerst wordt van het schema van de XML-gegevens een ER-schema opgebouwd. Dit gebeurt als volgt. Eerst wordt het schema omgezet naar een ander, simpelER-schema dat equivalent is op de volgorde van elementen na. Om toch de document orde te kunnen onderzoeken wordt de volgorde later in de tabellen zelf bijgehouden. Van dat aangepast schema wordt een graafvoorstelling gemaakt en uit die graafvoorstelling wordt een elementgraaf gemaakt. Vanuit die elementgraaf kan dan een relationeel schema opgesteld worden. De volledige opbouw van een ER-schema kan gevolgd worden door middel van de omzetting van het lopend voorbeeld. Dit lopend voorbeeld adresseert spijtig genoeg niet alle aspecten van de mapping, maar het introduceren van een nieuw voorbeeld zou voor verwarring kunnen zorgen. Voorbeeld 4.9 geeft van het originele schema van het lopend voorbeeld een makkelijkere representatie.

Voorbeeld 4.9.

```
persons    ->  person*
person     ->  (name,occupation,child*)
name       ->  (first,last)
first      ->  #PCDATA
last       ->  #PCDATA
occupation ->  #PCDATA
child      ->  (name,(age|birthdate))
age        ->  #PCDATA
birthdate  ->  #PCDATA
```

Merk op dat de name-knoop nu enkel een first- en een last-knoop mag bevatten. Dit komt omdat het algoritme niet overweg kan met mixed content. ▲

4.1.3.1 Omzetten van het schema

Het omzetten van het schema naar een eenvoudigER-schema gebeurt volgens een aantal regeltjes. Deze zijn te vinden in Figuur 4.12. Er zijn drie types van regels te onderscheiden. Regels van het type I ontnesten geneste reguliere expressies. Zo kunnen er geen geen concatenatie of disjunctie operatoren voorkomen binnen een andere operator. Regels van het type II zorgen ervoor dat er geen unaire operators gelden op andere unaire operators. Het resultaat is dat er op elk element één enkele unaire operator overblijft. Regels van het type III groeperen elementen die meerdere keren voorkomen onder een *-operator.

Regels type I:
$(e_1, e_2)^* \rightarrow e_1^*, e_2^*$
$(e_1, e_2)? \rightarrow e_1?, e_2?$
$(e_1|e_2) \rightarrow e_1?, e_2?$

Regels type II:
$e_1^{**} \rightarrow e_1^*$
$e_1^*? \rightarrow e_1^*$
$e_1?^* \rightarrow e_1^*$
$e_1?? \rightarrow e_1?$

Regels type III:
$\ldots, a^*, \ldots, a^*, \ldots \rightarrow a^*, \ldots$
$\ldots, a^*, \ldots, a?, \ldots \rightarrow a^*, \ldots$
$\ldots, a?, \ldots, a^*, \ldots \rightarrow a^*, \ldots$
$\ldots, a?, \ldots, a?, \ldots \rightarrow a^*, \ldots$
$\ldots, a, \ldots, a, \ldots \rightarrow a^*, \ldots$

Figuur 4.12: Regels voor het omzetten van een schema

Het omzetten van het schema wordt in Voorbeeld 4.10 gegeven.

Voorbeeld 4.10.

```
persons    ->  person*
person     ->  (name,occupation,child*)
name       ->  (first?,last?)
first      ->  #PCDATA
last       ->  #PCDATA
occupation ->  #PCDATA
child      ->  (name,age?,birthdate?)
```

```
age        ->  #PCDATA
birthdate  ->  #PCDATA
```

▲

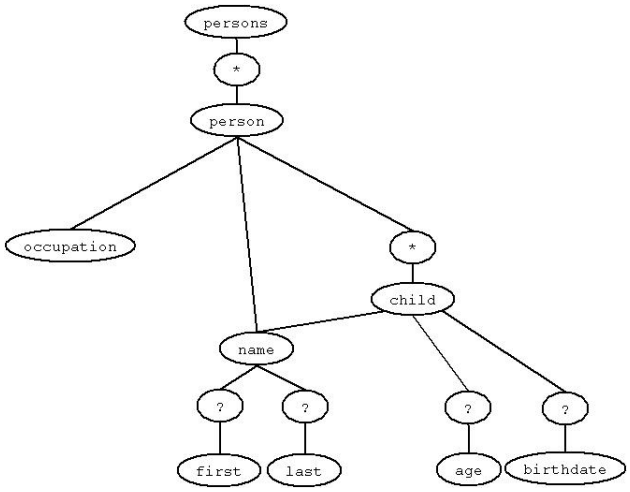
4.1.3.2 Genereren van een relationeel schema

Nu dat een eenvoudigER-schema is opgesteld kan een relationeel schema opgesteld worden. Er zijn drie technieken om dat te doen, namelijk de basic inlining techniek, de shared inlining techniek en de hybrid inlining techniek.

Een probleem bij het creëren van tabellen is het opsporen van recursie. Dit probleem wordt opgelost door middel van een graafvoorstelling van het schema. Deze wordt opgesteld als volgt. Voor elk element in het schema wordt een knoop aangemaakt. Daarna worden er bogen naar de knopen van kind-elementen getrokken. Als er kind-elementen zijn waarop een operator is geplaatst, wordt in de graaf een knoop gemaakt voor de operator en er wordt een boog getrokken van de knoop van het ouder element naar de knoop met de operator en een boog van de knoop met de operator naar de knoop met het kind-element. Elk element komt slechts één keer voor in deze graafvoorstelling, attributen en operatoren mogen meerdere keren voorkomen. Een lus in de graafvoorstelling wijst aan dat er recursie voorkomt in het schema. Nu is er geweten hoe recursie gedetecteerd kan worden. Voorbeeld 4.11 toont hoe de graafvoorstelling van het schema kan opgesteld worden.

Voorbeeld 4.11. Figuur 4.13 toont de graafvoorstelling van het aangepaste schema. Merk op dat dit geen boomstructuur is aangezien er twee knopen de name-knoop als kind hebben. In dit voorbeeld komt geen recursie voor aangezien geen lussen voorkomen in de graafvoorstelling. ▲

Om nu te bepalen welke tabellen er gecreëerd moeten worden, wordt de elementgraaf opgesteld. Deze elementgraaf bepaalt de tabellen die aangemaakt moeten worden voor een bepaald element. Voor welke elementen deze elementgraaf wordt opgebouwd hangt af van de gebruikte techniek. Eigenlijk is deze graaf het stuk uit de graafvoorstelling van het schema dat relevant is voor dat bepaald element. Deze elementgraaf wordt dus aan de hand van de graafvoorstelling van het schema opgebouwd. Voor het onderscheid zullen de knopen uit de graafvoorstelling van het schema graafknopen genoemd worden en knopen uit de elementgraaf elementknopen. Deze graafvoorstelling wordt in depth-first volgorde doorlopen, beginnend met de graafknoop van het element waarvoor er tabellen gecreëerd moeten worden. Een graafknoop wordt gemarkeerd als er een eerste keer doorheen gelopen wordt en de markering wordt teniet gedaan als alle kinderen van de graafknoop doorlopen werden. Als een ongemarkeerde graafknoop wordt tegengekomen, wordt een



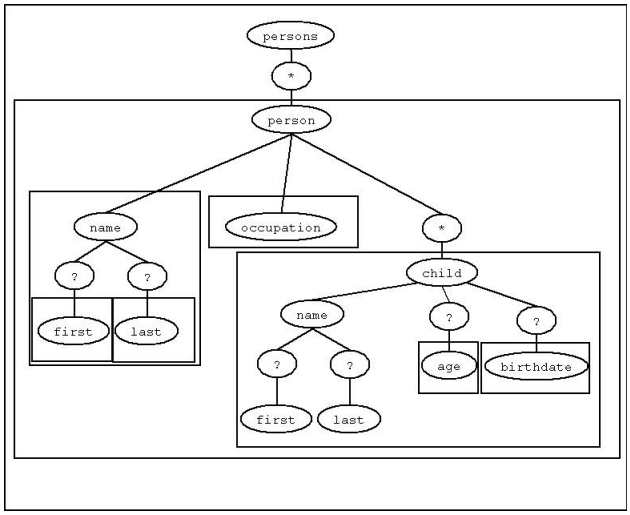
Figuur 4.13: Graafvoorstelling van het schema

nieuwe elementknoop in de elementgraaf aangemaakt met dezelfde naam. Dit geldt ook voor de graafknopen met operatoren. Dan wordt een gewone boog gemaakt van de elementknoop die overeenkomt met de ouder in de graafvoorstelling naar de huidige elementknoop. Als er geprobeerd wordt om doorheen een graafknoop te lopen die al gemarkeerd was, wordt er een zogenaamde backpointer boog gemaakt naar de elementknoop die al in de elementgraaf staat in plaats van een nieuwe elementknoop toe te voegen. Voorbeeld 4.12 toont hoe de elementgraaf kan opgesteld worden.

Voorbeeld 4.12. Figuur 4.14 toont de elementgrafen van elk element uit het schema van Voorbeeld 4.9. Elke rechthoek bevat een elementgraaf, namelijk degene van de hoogste knoop in de rechthoek. Merk op dat dit telkens wél een boomstructuur is. ▲

Basic inlining techniek De basic inlining techniek, of simpelweg basic, probeert zoveel mogelijk kind-elementen op één rij in een tabel te plaatsen. Er wordt een elementgraaf aangemaakt voor elk element omdat elk element in een DTD een mogelijke root kan zijn.

Aan de hand van de opgestelde elementgrafen kunnen nu tabellen worden gecreëerd volgens de basic inlining techniek. Er wordt eerst een tabel gemaakt voor het root element van de elementgraaf. Dan worden alle afstammelingen in dezelfde tabel geplaatst behalve twee uitzonderingen. Voor een afstammeling die kind is van een * wordt een nieuwe tabel gemaakt en de afstammelingen van deze knoop worden ook in die tabel geplaatst behoudens



Figuur 4.14: elementgrafen van het schema

dezelfde twee uitzonderingen. Een tweede uitzondering is als een knoop gevonden wordt waar een backpointer boog heen wijst. Ook voor zulk element wordt een nieuwe tabel aangemaakt. Dit wordt gedaan om het probleem van de recursie op te lossen. Voorbeeld 4.13 toont hoe een relationeel schema kan gegenereerd worden met behulp van de basic inlining techniek.

Voorbeeld 4.13. De elementgrafen voor elk element zijn terug te vinden in Figuur 4.14. Eerst worden tabellen gemaakt voor het element `persons`. Dit element heeft alleen een kind dat bereikbaar is door een '*'. Deze personstabel heeft dus enkel een personID. Nu wordt er ook een tabel gemaakt voor `persons.person` en `persons.person.child` omdat dat kinderen van een '*' zijn. Dan worden er tabellen gemaakt voor het element `person` en voor de rest van de elementen op dezelfde manier. Het totale ER-schema staat hieronder afgebeeld. Merk op dat dit schema erg veel overbodige gegevens bevat. De tabellen `persons.person.child` en `person.child` bevatten dezelfde informatie.

```
persons(personsID (int))
persons.person(persons.personID (int),
    persons.person.parentID (int),
    persons.person.name.first (string),
    persons.person.name.last (string),
    persons.person.occupation (string))
persons.person.child(persons.person.childID (int),
```

```
    persons.person.child.parentID (int),
    persons.person.child.name.first (string),
    persons.person.child.name.last (string),
    persons.person.child.age (int),
    persons.person.child.birthdate (string))

person(personID (int), person.parentID (int),
    person.name.first (string),
    person.name.last (string),
    person.occupation (string))
person.child(person.childID (int),
    person.child.parentID (int),
    person.child.name.first (string),
    person.child.name.last (string),
    person.child.age (int),
    person.child.birthdate (string))

name(nameID (int), name.first (string), name.last (string))

first(firstID (int), first (string))

last(lastID (int), last (string))

occupation(occupationID (int), occupation (string)

child(childID (int), child.age (int), child.birthdate (string))

age(ageID (int), age (int))

birthdate(birthdateID (int), birthdate (string))
```

▲

De basic inlining techniek is voor sommige queries niet goed genoeg. De gegevens worden op zo een manier opgeslagen dat er meer union's in een SQL-query zullen voorkomen dan gewenst. Ook wordt er voor elk element een tabel gemaakt en elementen kwamen ook nog eens voor als een attribuut in een andere tabel waardoor er redundantie ontstaat.

Shared inlining techniek De shared inlining techniek probeert minder tabellen te creëren en zo redundantie te vermijden.

Er worden enkel elementgrafen opgesteld voor bepaalde knopen uit de graafvoorstelling van het schema, die gekozen worden volgens het aantal bogen dat ernaar verwijst. Hier tellen uiteraard knopen met operatoren niet mee. Als er geen boog naar de knoop verwijst wordt een elementgraaf voor het element van die knoop opgesteld. Ook als er meerdere bogen naar de knoop verwijzen wordt een elementgraaf voor het element van die knoop opgesteld.

Net zoals bij de basic inlining techniek wordt een tabel aangemaakt voor het element waarvan een elementgraaf is opgesteld. De elementen waarvoor geen elementgraaf is opgesteld zullen vervat worden als attributen van andere tabellen. Knopen die kind zijn van een *-knoop worden ook hier in een aparte tabel geplaatst. Van de wederzijds recursieve elementen waarnaar één enkele boog wijst, wordt er maar een tabel gemaakt. Met wederzijds recursieve elementen wordt bedoeld dat de knopen van die elementen beiden in een lus van de elementgraaf staan. Doordat de knopen waarnaar één enkele boog wijst in een tabel van een voorouder kunnen staan hoeven er geen nieuwe tabellen voor die knopen gemaakt te worden. Voorbeeld 4.14 toont hoe een relationeel schema kan gegenereerd worden met behulp van de shared inlining techniek.

Voorbeeld 4.14. De elementgrafen voor elk element zijn terug te vinden in Figuur 4.14. Eerst worden tabellen aangemaakt voor het element `persons` omdat er geen pijlen naar wijzen in de graafvoorstelling van het schema. Nu wordt er ook een tabel gemaakt voor `name` omdat er in de graafvoorstelling van het schema twee bogen naar wijzen. Dit zijn de enige twee tabellen die expliciet aangemaakt worden. Nu worden er nog tabellen aangemaakt voor knopen die bereikt worden via een '*'. Dit zijn de tabellen `persons.person` en `persons.person.child`. Merk op dat dit schema heel wat korter is dan het schema in Voorbeeld 4.13.

```
persons(personsID (int))
persons.person(persons.personID (int),
    persons.person.parentID (int),
    persons.person.name.first (string),
    persons.person.name.last (string),
    persons.person.occupation (string))
persons.person.child(persons.person.childID (int),
    persons.person.child.parentID (int),
    persons.person.child.name.first (string),
    persons.person.child.name.last (string),
    persons.person.child.age (int),
```

```
    persons.person.child.birthdate (string))

name(nameID (int), name.parentID (int),
    name.first (string), name.last (string))
```

▲

Een voordeel ten opzichte van de basic inlining techniek is dat het aantal tabellen dat gecreëerd wordt veel kleiner is. Een nadeel ten opzichte van de basic inlining techniek is dat er meerdere joins moeten gebeuren als er een pad gezocht wordt dat in een bepaalde knoop begint.

Hybrid inlining techniek De hybrid inlining techniek probeert het aantal tabellen nog meer te beperken. Dit gebeurt door ook de knopen waarnaar meerdere bogen wijzen bij in tabellen van voorouders te plaatsen, als deze knoop tenminste niet bereikt wordt via een *-knoop of in een lus in de graaf staat. Voor de rest is deze techniek gelijk aan de shared inlining techniek. Voorbeeld 4.15 toont hoe een relationeel schema kan gegenereerd worden met behulp van de hybrid inlining techniek.

Voorbeeld 4.15. De elementgrafen voor elk element zijn terug te vinden in Figuur 4.14. Eerst worden tabellen aangemaakt voor het element `persons` omdat er geen pijlen naar wijzen in de graafvoorstelling van het schema. De tabel voor `name` wordt bij deze techniek niet aangemaakt. Merk op dat dit schema nog korter is dan het schema in Voorbeeld 4.14.

```
persons(personsID (int))
persons.person(persons.personID (int),
    persons.person.parentID (int),
    persons.person.name.first (string),
    persons.person.name.last (string),
    persons.person.occupation (string))
persons.person.child(persons.person.childID (int),
    persons.person.child.parentID (int),
    persons.person.child.name.first (string),
    persons.person.child.name.last (string),
    persons.person.child.age (int),
    persons.person.child.birthdate (string))
```

▲

4.1.3.3 Ondervragen

Aangezien deze mapping is afgeleid van het schema is er nu een schema document beschikbaar dat geraadpleegd kan worden bij het uitvoeren van queries. Dit kan vooral helpen bij het uitvoeren van path traversals. In [12] wordt uit de doeken gedaan hoe een query in de talen XML-QL[14] en Lorel[15] kan omgezet worden naar een SQL-query. Omdat deze talen geen echte XML-ondervragingstalen zijn, maar eerder aanleunen tegen een relationele ondervragingstaal zal de omzetting hier niet besproken worden. Een omzetting van een query in een XML-ondervragingstaal naar een query in SQL wordt niet voorgesteld in [12], maar het is natuurlijk wel mogelijk om zo een omzetting zelf te maken.

In [12] worden geen indexen voorgesteld. Indexen op alle $xxxID$ velden zouden wel handig kunnen zijn omdat deze heel efficiënt zijn geïmplementeerd door RDBMS en deze attributen de verwijzingen binnen de tabellen voorstellen. Deze verwijzingen kunnen door een index dus sneller gevonden worden.

Aanpassingen aan de XML-gegevens zijn in principe geen probleem. De gegevens moeten toch aan het schema voldoen en dus zullen ze altijd gemapt kunnen worden. Het probleem doet zich echter voor wanneer het schema verandert. Hierdoor kan het immers zijn dat het hele relationele schema omgegooid moet worden, omdat het relationele schema namelijk opgebouwd was aan de hand van het schema van de XML-gegevens.

4.1.3.4 Voor- en nadelen

Doordat het schema bekend is zullen queries die over de structuur gaan van de XML-gegevens sneller verlopen dan bij de generic mapping methode. Bij die methode was namelijk geen informatie over het schema beschikbaar. Afhankelijk van de gebruikte techniek wordt er veel of weinig redundante informatie opgeslagen. De basic inlining techniek zorgt duidelijk voor veel redundantie. De shared inlining en de hybrid inlining technieken zorgen voor een pak minder redundante gegevens. Deze methode zorgt ook voor een intuïtiever relationeel schema dan de generic mapping. Het enige nadeel aan deze mapping is dat er een schema van de gegevens aanwezig moet zijn. Dit schema moet ook stabiel blijven, het mag niet dikwijls aangepast worden, anders zal het relationeel schema dikwijls aangepast moeten worden en dat is niet gewenst. Voor flexiebele en vaak veranderende XML-gegevens is deze mapping methode dus niet zo gepast.

4.1.4 Mapping op basis van gegevensanalyse

Een laatste mapping methode maakt gebruik van de XML-gegevens die dienen opgeslagen te worden om de mapping te bepalen. Er zijn aspecten waarmee rekening gehouden wordt bij deze mapping. Een aspect is dat er best zo weinig mogelijk plaats verbruikt wordt. Redundantie kan in sommige gevallen tot grotere performantie leiden bij het uitvoeren van een query, maar in de meeste gevallen is redundantie ongewenst. Een tweede aspect is dat de gegevens best zo weinig mogelijk in kleine stukjes geknipt worden en dan verdeeld over het totale schema. Dit zorgt later namelijk voor moeilijkheden als er ondervragingen op de gegevens worden gedaan. In [16] wordt een mapping op basis van gegevensanalyse besproken. Hierin wordt een datamining algoritme op de XML-gegevens losgelaten om een relationeel schema te genereren. Voor de details van deze mapping wordt verwezen naar [16].

Het voordeel van een mapping die alle gegevens analyseert vooraleer ze naar relationele gegevens gemapt worden is dat het een heel efficiënte mapping is. Dit geldt zowel voor tijd bij het uitvoeren van een query, als voor plaats in de relationele database. Het nadeel van deze mapping methode is duidelijk dat de gegevens gekend moeten zijn om een zo goed mogelijke mapping te bekomen. Als de XML-gegevens dikwijls aangepast worden is dit natuurlijk niet het geval. Er kan een heel goede mapping gevonden worden voor een bepaalde verzameling gegevens maar daarna kunnen de gegevens veranderen zodat de mapping veel minder performant is.

4.2 Query mogelijkheden

Als semi-gestructureerde gegevens in een relationele database worden geplaatst, kunnen deze gegevens enkel ondervraagd worden in relationele ondervragingstalen. Maar dit is meestal niet gewenst. Er zijn namelijk XML-gegevens in de database opgeslagen en die zouden best ondervraagd worden in een XML-ondervragingstaal. De bedoeling is namelijk om de effectieve opslagmethode van de XML-gegevens verborgen te houden van de gebruiker. Deze kan dan queries stellen in een XML-ondervragingstaal zoals XQuery of XPath, of XSLT-programma's op de XML-gegevens loslaten.

4.2.1 Van XQuery naar SQL

Er zijn meerdere manieren om XQuery ondervragingen om te zetten in SQL-queries. In [18] wordt een methode gegeven die hier niet besproken wordt. Een andere methode die hier wel wordt uitgelegd is beschreven in [20]. Er bestaan natuurlijk nog manieren en er zullen er ook nog bijkomen.

Om een XQuery-ondervraging correct om te kunnen zetten naar SQL is natuurlijk nodig dat de mapping van XML-gegevens naar relationele gegevens gekend is. Deze omzetting die hier gebruikt wordt is een soort generic mapping. De mapping plaatst, in tegenstelling tot de gewone generic mapping die bogen mapt, knopen in een tabel. Net zoals de edge aanpak bij generic mapping worden alle gegevens in één enkele tabel geplaatst. Bij deze omzetting noemt deze tabel de doctabel. Deze doctabel bevat alle knopen van de XML-gegevens en de knopen die gegenereerd worden tijdens het omzetten van de XQuery ondervraging. De doctabel heeft de volgende attributen. Een *pre*-attribuut dat een uniek preorde nummer geeft aan de knoop. Dit nummer wordt bekomen door in preorde doorheen de boomstructuur van de XML-gegevens te lopen. Een *size*-attribuut geeft aan hoeveel knopen de subtree onder de knoop bevat. Een *level*-attribuut geeft de lengte van het pad van de root tot de knoop weer. Het *kind*-attribuut bepaalt wat voor soort knoop het is. Dit attribuut kan maar twee waardes hebben, namelijk „elem" of „text". Dit betekent dat de knoop een interne knoop is of respectievelijk een tekstwaarde voorstelt. De betekenis van de inhoud van het *prop*-attribuut wordt bepaald door de waarde van het *kind*-attribuut. Als het *kind*-attribuut „text" is, dan bevat het *prop*-attribuut de tekstuele inhoud van de knoop. Als het *kind*-attribuut „elem" is, bevat het *prop*-attribuut de tagnaam van de knoop. Het laatste attribuut is het $frag$-attribuut. Dit attribuut bevat een identifier die aangeeft in welk document of welk gegenereerd fragment de knoop zich bevindt. Voorbeeld 4.16 geeft een voorbeeld van de mapping die gebruikt wordt bij deze omzetting.

Voorbeeld 4.16. Figuur 4.15 toont hoe de knopen van de XML-boomstructuur genummerd zijn voor ze in een tabel te kunnen plaatsen. De doctabel van het XML-document is hieronder weergegeven.

pre	*size*	*level*	*kind*	*prop*	*frag*
1	39	0	"elem"	"persons"	1
2	17	1	"elem"	"person"	1
3	4	2	"elem"	"name"	1
4	1	3	"elem"	"first"	1
5	0	4	"text"	"George"	1
6	1	3	"elem"	"last"	1
7	0	4	"text"	"Bush"	1
8	1	2	"elem"	"occupation"	1
9	0	3	"text"	"president"	1
10	4	2	"elem"	"child"	1
11	1	3	"elem"	"name"	1
12	0	4	"text"	"Barbara Bush"	1
13	1	3	"elem"	"age"	1
14	0	4	"text"	"22"	1
15	4	2	"elem"	"child"	1
16	1	3	"elem"	"name"	1
17	0	4	"text"	"Jenna Bush"	1
18	1	3	"elem"	"age"	1
19	0	4	"text"	"22"	1
20	7	1	"elem"	"person"	1
21	4	2	"elem"	"name"	1
22	1	3	"elem"	"first"	1
23	0	4	"text"	"Oprah"	1
24	1	3	"elem"	"last"	1
25	0	4	"text"	"Winfrey"	1
26	1	2	"elem"	"occupation"	1
27	0	3	"text"	"talk show host"	1
28	12	1	"elem"	"person"	1
29	4	2	"elem"	"name"	1
30	1	3	"elem"	"first"	1
31	0	4	"text"	"Céline"	1
32	1	3	"elem"	"last"	1
33	0	4	"text"	"Dion"	1
34	1	2	"elem"	"occupation"	1
35	0	3	"text"	"singer"	1
36	4	2	"elem"	"child"	1
37	1	3	"elem"	"name"	1
38	0	4	"text"	"René-Charles Angélil"	1
39	1	3	"elem"	"birthdate"	1
40	0	4	"text"	"25 january 2001"	1

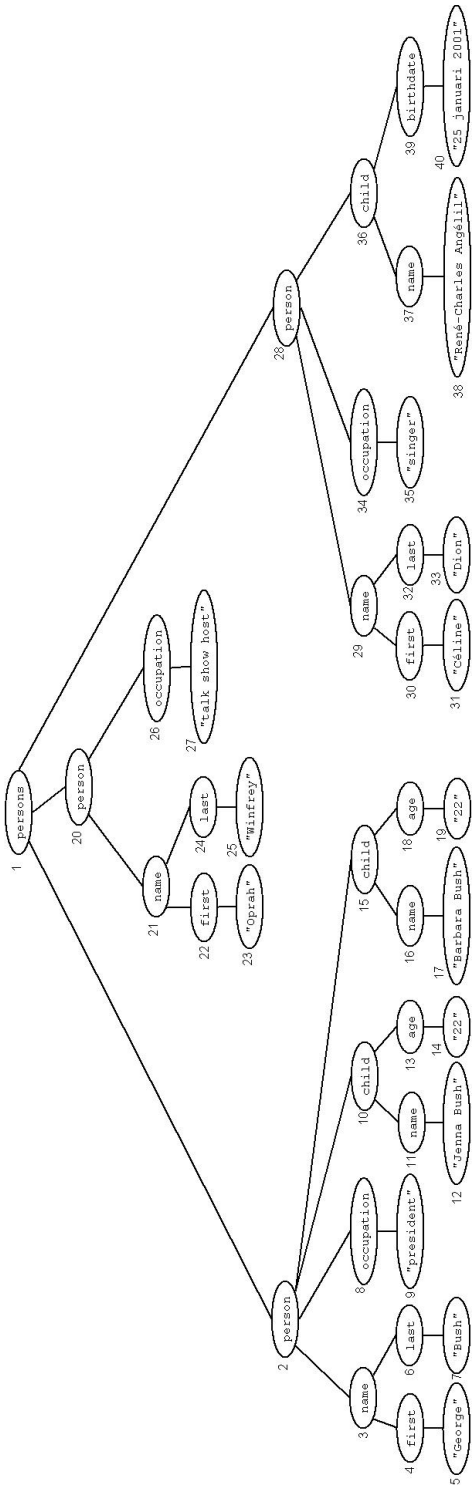
▲

Omzetting van assen Een eerste deel van deze omzetting is het omzetten van XPath-assen. Door middel van de attributen *pre*, *size*, *level* en *frag* kunnen assen naar SQL omgezet worden. In Figuur 4.16 worden de assen en hun omzetting naar SQL gegeven. Merk op dat andere assen geconstrueerd kunnen worden door het samenvoegen van twee omzettingen. Voorbeeld 4.17 kiest een as uit en legt de omzetting verder uit.

Voorbeeld 4.17. Als voorbeeld wordt de ancestor-as gekozen. Er wordt verondersteld dat er een tabel c bestaat waarin de knopen staan die al bereikt werden. Nu moeten de ancestor knopen van de knopen uit c gevonden worden. Een ancestor-knoop van c heeft een *pre*-waarde kleiner dan de *pre*-waarde van c. Dit vloeit voort uit de preorde nummering. Om geen knopen te kiezen die links van c staan, moet er ook nog voor gezorgd worden dat de *pre*-waarden van c kleiner of gelijk is aan de *pre*-waarde plus de *size*-waarde van de ancestor-knopen. Het is in te zien dat als het aantal knopen in de subtree onder een knoop wordt opgeteld met zijn *pre*-waarde, dat de *pre*-waarde van de knoop in het meest rechtse blad van die knoop bekomen wordt. Als nu deze waarde groter is dan de *pre*-waarde van c, komt deze knoop rechts van c voor en is de knoop een voorouder van c. Als deze waarde kleiner is dan de *pre*-waarde van c, betekent dat dat de knoop een knoop links van c was en geen voorouder. ▲

Omzetting van sequenties In XQuery kunnen eindige, geordende sequenties van items voorkomen. Zo een item kan een knoop zijn of een tekstuele waarde. Sequenties zijn eigenlijk lijsten en hebben geen boomstructuur. Ze kunnen dus gemakkelijk in een relationele tabel geplaatst worden. Deze tabel heeft attributen *pos*, *pre* en *val*. Het *pos*-attribuut stelt de positie voor in de sequentie. Het *pre*-attribuut is een knoopnummer als het item in de sequentie een knoop voorstelt. Dit knoopnummer komt overeen met een knoopnummer in de doctabel. Het *val*-attribuut stelt een waarde voor als het item geen knoop voorstelt, maar een gewone waarde. In de tabel wordt telkens de lexicografische voorstelling van de waarde gegeven. Voorbeeld 4.18 geeft een voorbeeld van een omzetting van een sequentie. Merk op dat de tabel tijdens de omzetting niet echt gegenereerd wordt, maar dat het in feite enkel het tussenresultaat is. Het wordt enkel even in een tabel gezet om de omzetting duidelijker te maken.

Voorbeeld 4.18. De sequentie $(10, 20, 30)$ van waardes wordt in de volgende tabel geplaatst. Merk op dat de *pre*-waarde NULL is omdat het geen knopen



Figuur 4.15: Nummering van de knopen van de XML-boomstructuur

node_test	$as(c, v, \alpha) : v \in c/\alpha?$
ancestor	v.pre $<$ c.pre AND c.pre $\leq$ v.pre + v.size
descendant	v.pre $>$ c.pre AND v.pre $\leq$ c.pre + c.size
child	$as(c, v, descendant)$ AND v.level = c.level+1
following	v.pre $>$ c.pre + c.size
preceding	v.pre + v.size $<$ c.pre

Figuur 4.16: Omzetting van XPath assen naar SQL

zijn in de sequentie.

pos	*pre*	*val*
1	NULL	10
2	NULL	20
3	NULL	30

▲

Omzetting van for-lussen Een tweede deel van de omzetting is het omzetten van for-lussen. Dit is iets moeilijker aangezien er hier variabelen in het spel komen. Deze variabelen hebben een omzetting naar SQL nodig. Ook moet er gedacht worden aan scopes waarin variabelen geëvalueerd worden. Variabelen stellen items of sequenties van items voor. Een enkel item wordt voorgesteld door een sequentie met slechts één element, zodat er voor variabelen één voorstellingswijze is. Variabelen worden ook in tabellen geplaatst.

Een voordeel van XQuery for-lussen is dat alle iteraties tegelijk kunnen geëvalueerd worden. Dit komt omdat een iteratie niets verandert aan de gegevens voor een volgende iteratie. Hier wordt dankbaar gebruik gemaakt van deze eigenschap. Er is een tabel die bijhoudt hoeveel keer de uitdrukkingen in de huidige scope worden uitgevoerd. Dit is de loop tabel, die één enkel attribuut bevat, namelijk *iter*.

Voor elke scope waarin de variabele zich bevindt, is er een andere omzetting van de variabele, maar dit wordt eerst even buiten beschouwing gelaten. Er wordt voorlopig verondersteld dat de variabelen opgeroepen worden in de scope waarin ze gedefinieerd zijn. De tabel waarnaar een variabele omgezet wordt heeft de volgende attributen. Het *iter*-attribuut geeft weer over welke iteratie van de for-lus het gaat. Het *pos*-attribuut geeft informatie over de positie in de sequentie die de variabele voorstelt. Dan zijn er nog de attributen *pre* en *val*. Deze stellen hetzelfde voor als bij de omzetting van een sequentie. Tenslotte is het nog belangrijk op te merken dat *pos* niet altijd de

exacte positie weergeeft in de sequentie, maar enkel een garantie geeft over de volgorde van items.

Om wat meer inzicht te krijgen in de omzetting van een for-lus, wordt eerst beschreven wat er gebeurt als een constante uitdrukking teruggegeven wordt in een for-lus. Dit proces heet loop-lifting en wordt beschreven in Voorbeeld 4.19.

Voorbeeld 4.19. Neem de XQuery uitdrukking

```
for $v in (1, 2, 3) return
 10
```

De lus van 3 iteraties wordt voorgesteld door de loop tabel die drie elementen heeft, namelijk 0, 1 en 2.

iter
0
1
2

Als er een constante uitdrukking in de body van een for-lus zit, wordt die uitdrukking „gelift" of naar boven gebracht. Dit is omdat de constante uitdrukking hetzelfde blijft voor elke iteratie. De SQL-query die dan de for-lus voorstelt is dan:

`SELECT` *iter*`,` 1 `AS` *pos*`, NULL AS` *pre*`,` *c* `AS` *val*
`FROM loop`

Hierin is c de constante uitdrukking. In dit geval zou het resultaat van de query de volgende tabel zijn:

iter	*pos*	*pre*	*val*
0	1	`NULL`	`"10"`
1	1	`NULL`	`"10"`
2	1	`NULL`	`"10"`

In het geval dat de constante uitdrukking een sequentie is van constanten, wordt de unie genomen van de twee SQL-queries. ▲

Nu wordt besproken hoe een variabele in elke iteratie wordt omgezet naar één enkele tabel. Er wordt verondersteld dat er een functie $row()$ bestaat die aan elke rij in de tabel een volgnummer toekent. Voorbeeld 4.20 geeft een voorbeeld van een omzetting van een for-lus naar SQL.

Voorbeeld 4.20. Neem de XQuery uitdrukking

```
for $v in (1, 2, 3) return
 (10, $v)
```

De for-lus van de drie iteraties wordt door net dezelfde tabel voorgesteld als in Voorbeeld 4.19. Ook de „gelifte" voorstelling van `10` kan daar teruggevonden worden. Een variabele in zijn eigen scope kan worden omgezet volgens

de volgende SQL uitdrukking.
`SELECT` *row*() `AS` *iter*, 1 `AS` *pos*, *pre*, *val*
`FROM` $q(e_x)$
`ORDER BY` *iter*, *pos*
Hierin is e_x de sequentie waarover de variabele itereert en $q(e_x)$ de omzetting van die sequentie. Variabele $v in de for-lus wordt dus voorgesteld door de volgende tabel.

iter	*pos*	*pre*	*val*
0	1	NULL	"1"
1	1	NULL	"2"
2	1	NULL	"3"

Het tussentijds resultaat is dan:

iter	*pos*	*pre*	*val*
0	1	NULL	"10"
0	2	NULL	"1"
1	1	NULL	"10"
1	2	NULL	"2"
2	1	NULL	"10"
2	2	NULL	"3"

Nu moet de for-lus nog ontnest worden. Het eindresultaat is dus:

iter	*pos*	*pre*	*val*
0	1	NULL	"10"
0	2	NULL	"1"
0	3	NULL	"10"
0	4	NULL	"2"
0	5	NULL	"10"
0	6	NULL	"3"

▲

Als er geneste for-lussen voorkomen in de XQuery uitdrukking, is het moeilijker om die naar SQL om te zetten. De omzetting van een XQuery uitdrukking wordt gedaan aan de hand van de regels in Figuur 4.17. De regels moeten als volgt gelezen worden. Boven de lijn staan de uitdrukkingen of tabellen die gekend moeten zijn om de uitdrukking om te zetten. Onder de lijn staat een uitdrukking die er zo uitziet: $\Gamma; loop; doc \| e \Rightarrow (q, doc')$. Γ is een verzameling van variabelen en hun omzetting naar SQL in de huidige scope. Γ ziet er uit als volgt: $\{\$v0 \rightarrow q_{v0}, \ldots, \$vi \rightarrow q_{vi}\}$ met q_{vx} de SQL omzetting van de variabele $\$vx$. *loop* beschrijft de huidige loop tabel en *doc* de huidige doctabel. e is de XQuery uitdrukking die dient omgezet te worden. q is de SQL-query van de uitdrukking en doc' is de nieuwe doctabel

na het uitvoeren van deze SQL-query. Deze doctabel verschilt enkel als er in de XQuery elementen of sequenties zijn aangemaakt. Bij het begin van de uitvoering heeft de tabel loop één enkel element, namelijk 0. De verzameling Γ is leeg en de doc-tabel bevat de knopen die in de XML-gegevens voorkomen.

$$\frac{}{\Gamma;\mathsf{loop};\mathsf{doc}\vdash c\mapsto\begin{pmatrix}\texttt{SELECT}\ l.iter,1\ \texttt{AS}\ pos,\\ \qquad\texttt{NULL AS}\ pre,c\ \texttt{AS}\ val\ ,\mathsf{doc}\\ \texttt{FROM loop AS}\ l\end{pmatrix}}\ \texttt{(1-CONST)}$$

$$\frac{}{\{\ldots,\$v\mapsto q_v,\ldots\};\mathsf{loop};\mathsf{doc}\vdash\$v\mapsto(q_v,\mathsf{doc})}\ \texttt{(2-VAR)}$$

$$\frac{\begin{array}{c}\Gamma;\mathsf{loop};\mathsf{doc}\vdash e_1\mapsto(q_1,\mathsf{doc}')\\ \Gamma+\{\$v\mapsto q_1\};\mathsf{loop};\mathsf{doc}'\vdash e_2\mapsto(q_2,\mathsf{doc}'')\end{array}}{\Gamma;\mathsf{loop};\mathsf{doc}\vdash\texttt{let}\ \$v\ \texttt{:=}\ e_1\ \texttt{return}\ e_2\mapsto(q_2,\mathsf{doc}'')}\ \texttt{(3-LET)}$$

$$\frac{\Gamma;\mathsf{loop};\mathsf{doc}\vdash e_1\mapsto(q_1,\mathsf{doc}')\qquad\Gamma;\mathsf{loop};\mathsf{doc}'\vdash e_2\mapsto(q_2,\mathsf{doc}'')}{\Gamma;\mathsf{loop};\mathsf{doc}\vdash(e_1,e_2)\mapsto\begin{pmatrix}\texttt{SELECT}\ iter,e_2.pos+m.pos\ \texttt{AS}\ pos,pre,val\\ q_1\ \texttt{UNION ALL}\quad\texttt{FROM}\ q_2\ \texttt{AS}\ e_2,\qquad\qquad,\mathsf{doc}''\\ \qquad\qquad(\texttt{SELECT MAX(}pos\texttt{) AS}\ pos\ \texttt{FROM}\ q_1)\ \texttt{AS}\ m\end{pmatrix}}\ \texttt{(4-SEQ)}$$

$$\frac{\begin{array}{c}\{\ldots,\$v_i\mapsto q_{v_i},\ldots\};\mathsf{loop};\mathsf{doc}\vdash e_1\mapsto(q_1,\mathsf{doc}')\qquad\mathsf{loop}'\equiv(\texttt{SELECT}\ iter\ \texttt{FROM}\ q_v)\\ q_v\equiv\begin{pmatrix}\texttt{SELECT}\ row()\ \texttt{AS}\ iter,1\ \texttt{AS}\ pos,pre,val\\ \texttt{FROM}\ q_1\\ \texttt{ORDER BY}\ iter,pos\end{pmatrix}\qquad\mathsf{map}\equiv\begin{pmatrix}\texttt{SELECT}\ iter\ \texttt{AS}\ outer,row()\ \texttt{AS}\ inner\\ \texttt{FROM}\ q_1\\ \texttt{ORDER BY}\ iter,pos\end{pmatrix}\\ \left\{\begin{array}{l}\qquad\texttt{SELECT}\ inner\ \texttt{AS}\ iter,pos,pre,val\\ \ldots,\$v_i\mapsto\quad\texttt{FROM map},q_{v_i}\qquad\qquad,\ldots\\ \qquad\texttt{WHERE}\ outer=iter\end{array}\right\}+\{\$v\mapsto q_v\};\mathsf{loop}';\mathsf{doc}'\vdash e_2\mapsto(q_2,\mathsf{doc}'')\end{array}}{\{\ldots,\$v_i\mapsto q_{v_i},\ldots\};\mathsf{loop};\mathsf{doc}\vdash\texttt{for}\ \$v\ \texttt{in}\ e_1\ \texttt{return}\ e_2\mapsto\begin{pmatrix}\texttt{SELECT}\ outer\ \texttt{AS}\ iter,\\ \qque_2.iter*m.pos+e_2.pos\ \texttt{AS}\ pos,\\ \qquad e_2.pre,e_2.val\\ \texttt{FROM map},q_2\ \texttt{AS}\ e_2,\qquad\qquad,\mathsf{doc}''\\ \qquad(\texttt{SELECT MAX(}pos\texttt{) AS}\ pos\\ \qquad\quad\texttt{FROM}\ q_2)\ \texttt{AS}\ m\\ \texttt{WHERE}\ inner=e_2.iter\end{pmatrix}}\ \texttt{(5-FOR)}$$

Figuur 4.17: Omzetting van andere constructs naar SQL

Nu zullen de regels even uitgelegd worden.

1. Deze regel geeft de omzetting van een constante zoals hierboven beschreven. Om de constante om te kunnen zetten moet niets gekend zijn behalve de loop tabel.

2. Deze regel geeft de omzetting van de variabele in de gedefinieerde scope. Als in de Γ verzameling $\$v \rightarrow q_v$ al is opgenomen, dan kan $\$v$ gewoon omgezet worden in qv.

3. Deze regel geeft de omzetting van een let-statement in XQuery. Om te beginnen is de omzetting van expressie e_1 nodig. Ten tweede is de omzetting van e_2 nodig. In e_2 moeten alle voorkomens van $\$v$ vervangen

worden door e_1 om aan de semantiek van de let-uitdrukking te voldoen. Dus moet in de Γ verzameling voor e_2 de omzetting van $\$v$ toegevoegd worden. De uiteindelijke omzetting van de let-uitdrukking is dan niet meer dan de omzetting van e_2.

4. Deze regel geeft de omzetting van een sequentie. De omzettingen van het eerste deel e_1 van de sequentie en het tweede deel e_2 van de sequentie zijn nodig om de omzetting van de hele sequentie te kunnen bepalen. Als een sequentie uit meer dan twee items bestaat, is het natuurlijk mogelijk om e_2 weer als een sequentie te gaan bekijken. Deze kan dan weer omgezet worden op dezelfde manier. De omzetting van de sequentie wordt gegeven door de unie van de omzetting van e_1 en de omzetting van e_2, waarin de posities worden verhoogd met het maximum van de posities in e_1.

5. Deze regel geeft de omzetting van een for-lus in XQuery. Er moet om te beginnen de omzetting gekend zijn van de uitdrukking waarover de variabele itereert. De v_i variabelen zijn de variabelen gedefinieerd in de scope buiten de for-lus. Hierna wordt q_v berekend. Dit is de omzetting van variabele $\$v$ in de scope binnen de for-lus. Omdat er binnen een for-lus gedoken wordt, moet er een nieuwe loop tabel berekend worden. Er moet ook een map tabel berekend worden, om de iteraties van de buitenste lus op iteraties van de binnenste lus te mappen. Dit is nodig zodat er geweten is in welke iteratie de buitenste lus is als de binnenste lus in een bepaalde iteratie zit en omgekeerd. Als die dingen gekend zijn, is er ook nog de omzetting nodig van e_2. Dit is de uitdrukking die geëvalueerd moet worden voor elke iteratie in de for-lus. Voor deze uitdrukking moeten de variabelen in de scope buiten de for-lus omgezet worden naar variabelen in de scope binnen de for-lus. Dit gebeurt voor elke $\$v_i$ van e_1 hierboven. Tenslotte is natuurlijk ook nodig dat de variabele $\$v$ gekend is in de uitdrukking e_2. Het resultaat van deze omzetting is het resultaat van de hele for-lus. Dit is dus niet meer „in" de for-lus, maar er net na. Daarom wordt terug de originele looptabel gebruikt. Ook $\$v$ is niet meer gekend en komt dus niet meer in Γ voor. Voor de variabelen $\$v_i$ worden terug de omzettingen voor de buitenste scope gebruikt. De omzetting van e_2 wordt terug ontnest in de uiteindelijke omzetting.

Deze regels voor het omzetten kunnen misschien erg theoretisch lijken, maar ze worden duidelijker met het uitwerken van een voorbeeld. Het hele proces van het omzetten van zo een uitdrukking aan de hand van de regels in Figuur 4.17 wordt dus uitgelegd in Voorbeeld 4.21.

Voorbeeld 4.21. Neem de XQuery uitdrukking

s | `for` $\$v_0$ `in ("a", "b") return`
s_0 | ($\$v_0$,
$s_{0.0}$ | `for` $\$v_{0.0}$ `in ("1", "2") return`
$s_{0.0}$ | (v_0, $\$v_{0.0}$)
s_0 |)

Er zijn duidelijk drie scopes, namelijk s, s_0 en $s_{0.0}$. In scope s zijn geen variabelen gedefinieerd, in scope s_0 is variabele $\$v_0$ gedefinieerd en in scope $s_{0.0}$ zijn variabelen $\$v_0$ en $\$v_{0.0}$ gedefinieerd. Het doorlopen van de regels heeft een recursief karakter, zoals zal blijken uit de uitwerking van het voorbeeld.

De totale uitdrukking is een for-lus. Eerst zullen alle tabellen berekend worden om de for-lus om te zetten, waarna de totale uitkomst berekend zal worden. Deze uitkomst zullen we in een tabel plaatsen die q1 genoemd zal worden.

- Volgens regel 5 is het eerste dat berekend moet worden voor een for-lus de uitdrukking waarover de variabele zal itereren. In dit geval is dat `("a", "b")`, een sequentie van twee items. De omzetting van deze uitdrukking zal in een tabel komen te staan die q2 genoemd wordt. Deze moet dus eerst berekend worden.
  - Volgens regel 4 is de eerste benodigde omzetting de omzetting van het eerste item. In dit geval is dit eerste item `"a"`, een constante uitdrukking. De omzetting van deze uitdrukking zal in een tabel q3 geplaatst worden.
    - Volgens regel 1 is er niets nodig om de constante uitdrukking om te zetten naar een SQL-query.
    - Deze SQL-query ziet er dus als volgt uit.
      `SELECT` $l.iter$`, 1 AS` pos`, NULL AS` pre`, "a" AS` val
      `FROM loop AS` l
      Merk op dat aan de `loop` tabel nog niets veranderd is sinds het omzetten van de query. Deze tabel, die hier $loop_{init}$ genoemd wordt, bevat enkel 1 tupel, namelijk (0). q3 is dus:

iter	*pos*	*pre*	*val*
0	1	NULL	"a"

  - Volgens regel 4 is nu de tweede benodigde omzetting die van het tweede item. In dit geval is dat `"b"`, een constante uitdrukking. De omzetting van deze uitdrukking zal in een tabel q4 geplaatst worden.
    - Volgens regel 1 is er niets nodig om de constante uitdrukking om te zetten naar een SQL-query.

* De SQL-query ziet er dus als volgt uit.
  SELECT $l.iter$, 1 AS pos, NULL AS pre, "b" AS val
  FROM loop AS l
  q4 is:

$iter$	pos	pre	val
0	1	NULL	"b"

– Nu de benodigde uitdrukkingen zijn omgezet kan q2 berekend worden uit q3 en q4. q2 is dus:
  $q3$ UNION
  (SELECT $iter$, $q4.pos + m.pos$ AS pos, pre, val
  FROM $q4$,
  (SELECT max(pos) AS pos FROM $q3$) AS m)
  q2 is:

$iter$	pos	pre	val
0	1	NULL	"a"
0	2	NULL	"b"

- Volgens regel 5 is het volgende dat moet berekend worden, de omzetting van de variabele in de for-lus. Die omzetting ziet er als volgt uit.
  SELECT $row()$ AS $iter$, 1 AS pos, pre, val
  FROM $q2$
  ORDER BY $iter$, pos
  q_{v0} is de omzetting van v_0 in scope s_0 en is dus:

$iter$	pos	pre	val
0	1	NULL	"a"
1	1	NULL	"b"

- Volgens regel 5 moet ook de loop relatie aangepast worden. Van zodra we in de lus springen, wordt er gewerkt met $loop_0$. Deze loop wordt berekend uit q_{v0} als volgt:
  SELECT $iter$
  FROM q_{v0}
  $loop_0$ is dus:

$iter$
0
1

- Volgens regel 5 moet nu ook de map tabel gemaakt worden. Deze wordt berekend als volgt.
  SELECT $iter$ AS $outer$, $row()$ AS $inner$
  FROM $q2$

`ORDER BY` *iter*`,` *pos*
$map_{x \to 0}$ is dus:

outer	*inner*
0	0
0	1

- Het laatste wat volgens regel 5 ook berekend moet worden, is de terug te geven uitdrukking. In dit geval is dat:
  (\$$v_0$,
  `for` \$$v_{0.0}$ `in („1", „2") return`
  (\$$v_0$, \$$v_{0.0}$)
)

  Dit is een sequentie. Merk op dat er in Γ, de verzameling omgezette variabelen, nog niets zat. Nu wordt hier $\$v_0 \to q_{v0}$ aan toegevoegd. De huidige `loop` tabel is veranderd in $loop_0$. q5 zal nu de mapping zijn van deze uitdrukking.

  - Volgens regel 4 is een eerste benodigdheid om een sequentie om te zetten het eerste item. Dit eerste item is $\$v_0$. Aangezien $\$v_0$ al in de verzameling variabelen zit, is de omzetting van deze variabele gewoon q_{v0}.
  - Volgens regel 4 is de tweede benodigdheid om een sequentie om te zetten het tweede item. Dit is in dit geval:
    `for` \$$v_{0.0}$ `in ("1", "2") return`
    (\$$v_0$, \$$v_{0.0}$)
    Dit is duidelijk een for-lus. De tabel $q6$ bevat de omzetting van deze uitdrukking.
    - Volgens regel 5 is het eerste dat gekend moet zijn om de for-lus om te zetten, de uitdrukking waarover de variabele itereert. In dit geval is dat `("1", "2")`. Dit is een sequentie van twee items. De omzetting van deze sequentie wordt in tabel $q7$ geplaatst.
      - Volgens regel 4 is het eerste dat gekend moet zijn om een sequentie om te zetten, het eerste item. Dit eerste item is `"1"`, een constante uitdrukking. Op eenzelfde manier als hierboven kan de tabel $q8$ bekomen worden die er als volgt uitziet:

iter	*pos*	*pre*	*val*
0	1	`NULL`	`"1"`
1	1	`NULL`	`"1"`

- Volgens regel 4 is het tweede dat gekend moet zijn om een sequentie om te zetten, het tweede item. Dit tweede item is `"2"`, een constante uitdrukking. Op eenzelfde manier als hierboven kan de tabel $q9$ bekomen worden die er als volgt uitziet:

iter	*pos*	*pre*	*val*
0	1	`NULL`	`"2"`
1	1	`NULL`	`"2"`

- Nu kan net als hierboven de sequentie omgezet worden, zodat $q7$ er als volgt uitziet:

iter	*pos*	*pre*	*val*
0	1	`NULL`	`"1"`
0	2	`NULL`	`"2"`
1	1	`NULL`	`"1"`
1	2	`NULL`	`"2"`

* Volgens regel 5 is het volgende dat berekend moet worden, de variabele. In dit geval is dat $\$v_{0.0}$. De omzetting van deze variabele in scope $s_{0.0}$ is $q_{v0.0}$. Dit kan net als hierboven worden berekend en ziet er als volgt uit:

iter	*pos*	*pre*	*val*
0	1	`NULL`	`"1"`
1	1	`NULL`	`"2"`
2	1	`NULL`	`"1"`
3	1	`NULL`	`"2"`

* Volgens regel 5 is het volgende dat berekend moet worden, de nieuwe `loop` tabel. Dit is nu $loop_{0.0}$. Deze tabel wordt ook weer net als hierboven berekend en ziet er als volgt uit:

iter
0
1
2
3

* Volgens regel 5 moet ook de `map` berekend worden. Dit kan ook net zoals hierboven en $map_{0\rightarrow 0.0}$ ziet er dan als volgt uit:

outer	*inner*
0	0
0	1
1	2
1	3

* Volgens regel 5 moet ook de teruggegeven uitdrukking gekend

zijn. Dit is in dit geval ($\$v_0, \$v_{0.0}$). Er wordt weer in een diepere for-lus gedoken. Dus de variabelen die gedefinieerd waren buiten de for-lus krijgen een nieuwe omzetting. De enige variabele die buiten de for-lus was gedefinieerd, is $\$v_0$. Deze krijgt een nieuwe omzetting, namelijk één voor in scope $s_{0.0}$. Deze omzetting gebeurt als volgt:
`SELECT` *inner* `AS` *iter*`,` *pos*`,` *pre*`,` *val*
`FROM` $map_{0\to 0.0}$`,` q_{v0}
`WHERE` *outer* = *iter*
En nu is de omzetting van $\$v_0$ in scope $s_{0.0}$ de tabel $q_{0\to 0.0}$.

iter	*pos*	*pre*	*val*
0	1	NULL	"a"
1	1	NULL	"a"
2	1	NULL	"b"
3	1	NULL	"b"

De omzetting van $\$v_{0.0}$ is een beetje hoger berekend en in de for-lus wordt $loop_{0.0}$ gebruikt. q10 is de omzetting van ($\$v_0, \$v_{0.0}$). Dit is een sequentie.

- Volgens regel 4 moet eerst de omzetting van het eerste item van de sequentie gekend zijn. Dit is gekend want het is variabele $\$v_0$. De omzetting daarvan in scope $s_{0.0}$, $q_{0\to 0.0}$, is hier juist boven berekend.
- Volgens regel 4 moet dan de omzetting van het tweede item van de sequentie gekend zijn. Dat is ook gekend want het is variabele $\$v_{0.0}$. De omzetting daarvan, $q_{v0.0}$, is ook hierboven berekend.
- Volgens regel 4 kan nu de omzetting van de sequentie berekend worden. Dit geeft als resultaat $q10$.

iter	*pos*	*pre*	*val*
0	1	NULL	"a"
1	1	NULL	"a"
2	1	NULL	"b"
3	1	NULL	"b"
0	2	NULL	"1"
1	2	NULL	"2"
2	2	NULL	"1"
3	2	NULL	"2"

∗ Volgens regel 5 kan nu de omzetting van de for-lus gebeuren. Dit gebeurt volgens de volgende SQL-query:
`SELECT` *outer* `AS` *iter*`,` $q10.iter * m.pos + q10.pos$ `AS` *pos*`,`

$q10.pre$, $q10.val$
FROM $q10$, $map_{0\to 0.0}$, (SELECT max(pos) AS pos FROM $q10$) AS m
WHERE $inner = q10.iter$
$q6$ is dus:

iter	*pos*	*pre*	*val*
0	1	NULL	"a"
0	2	NULL	"1"
0	3	NULL	"a"
0	4	NULL	"2"
1	5	NULL	"b"
1	6	NULL	"1"
1	7	NULL	"b"
1	8	NULL	"2"

Merk op dat voor iteratie 1 de positie niet mooi bij 1 begint. Dat was ook niet gegarandeerd. De enige garantie die hierover bestaat is dat de volgorde van de posities juist is.

– Volgens regel 4 is het nu mogelijk om een sequentie op te bouwen van de twee items. Dit gebeurt op een manier als hierboven. Het resultaat van deze omzetting komt in $q5$ terecht en die ziet er uit als volgt:

iter	*pos*	*pre*	*val*
0	1	NULL	"a"
0	2	NULL	"a"
0	3	NULL	"1"
0	4	NULL	"a"
0	5	NULL	"2"
1	1	NULL	"b"
1	6	NULL	"b"
1	7	NULL	"1"
1	8	NULL	"b"
1	9	NULL	"2"

- Volgens regel 5 kan nu de for-lus omgezet worden. Dit gebeurt net als hierboven en $q1$ is dus:

iter	*pos*	*pre*	*val*
0	1	NULL	"a"
0	2	NULL	"a"
0	3	NULL	"1"
0	4	NULL	"a"
0	5	NULL	"2"
0	10	NULL	"b"
0	15	NULL	"b"
0	16	NULL	"1"
0	17	NULL	"b"
0	18	NULL	"2"

Het resultaat van deze omzetting is af te lezen uit tabel $q1$. Merk op dat de tussentijdse tabellen niet echt worden opgesteld. Ze dienen enkel ter verduidelijking zodat het resultaat van de SQL-queries duidelijker wordt. Voor meer informatie wordt verwezen naar [20].

▲

4.2.2 Van XSLT naar SQL

Het omzetten van XSLT-programma's naar SQL-queries is over het algemeen onmogelijk, maar de omzetting die hier besproken wordt zet enkel een deelverzameling van XSLT om naar SQL. Deze deelverzameling omvat recursieve templates, modes, parameters (weliswaar met enkele restricties), aggregatie, controle-uitdrukkingen en een groot deel van XPath. Delen van XPath die ondervragingen over de volgorde van elementen stellen zijn niet opgenomen. De as preceding wordt bijvoorbeeld niet ondersteund. Voorbeeld 4.22 geeft het voorbeeld dat verder uitgewerkt zal worden in de uitleg.

Voorbeeld 4.22. Als voorbeeld zal de omzetting van een XSLT-programma naar een SQL-query gemaakt worden voor de XML-gegevens uit het lopend voorbeeld. De root met tagnaam `persons` heeft 0 of meerdere kinderen met tagnaam `person`. Een knoop met tagnaam `person` heeft kinderen met de tagnamen `name`, `occupation` en `child`. Een knoop met `child` als tagnaam kan dan weer knopen met tagnamen `name` en `age` hebben.

Het XSLT-programma dat zal omgezet worden naar SQL is weergegeven in Figuur 4.18. Het programma selecteert in de eerste template eerst `person`-knopen die een `child`-knoop hebben met `name`-waarde 'John'. Daarna wordt de `age`-waarde van die `child`-knoop meegegeven als parameter aan een andere template, die opgeroepen wordt met alle `person`-knopen van het

```
<xsl:template match="person">
  <xsl:variable name="namevar">
    <xsl:value-of select="child/name"/>
  </xsl:variable>
  <xsl:if test="$namevar == 'John'">
    <xsl:apply-templates select="/persons/person" mode="1">
      <xsl:param name="age" select="child/age"/>
    </xsl:apply-templates>
  </xsl:if>
</xsl:template>

<xsl:template match="person" mode="1">
<xsl:param name="age"/>
  <xsl:variable name="agevar">
    <xsl:value-of select="child/age"/>
  </xsl:variable>
  <xsl:if test="$age < $agevar">
    <result>
      <xsl:value-of select="name"/>
    </result>
  </xsl:if>
</xsl:template>
```

Figuur 4.18: XSLT-programma

XML-document, namelijk `persons/person`. De mode van de andere template is '1'. Die template selecteert vervolgens `person`-knopen waarvan de `age`-waarde van de `child`-knoop groter is dan de meegegeven parameter. Van die geselecteerde `person`-knopen wordt dan de `name`-waarde uitgeschreven. Dit programma selecteert dus eigenlijk namen van ouders van kinderen die ouder zijn dan John.

▲

Deze omzetting van XSLT naar SQL werkt met een XML-view van relationele gegevens. Dit wordt uitgelegd in de volgende sectie 4.4 over XML-publishing. Daar wordt ook uitgelegd waarom dat recursieve XPath-uitdrukkingen geen probleem geven bij de omzetting. Nu wordt er even van uit gegaan dat de gegevens die in het XML-bestand staan, ook in een relationele database is geplaatst. De mapping van XML-gegevens op relationele tabellen wordt gegeven door een XML-viewtree. Een voorbeeld van zo een viewtree

kan gevonden worden in Voorbeeld 4.23.

Voorbeeld 4.23. De viewtree die zal gebruikt worden voor het omzetten van het XSLT-programma ziet er uit als volgt.

```
person(name, occupation) = Person(name, occupation)
child(name, age, personName) = Person(personName, -),
                               Child(name, age, parentName)
```

De viewtree dient gelezen te worden als volgt. De linkerkant van de vergelijking stellen knopen voor in de XML-boomstructuur. `child(name, age, personName)` betekent: de knoop `child`, met kinderen `name` en `age` en met ouder met `name = personName`. De rechterkant stellen tabellen voor in de relationele database. `Person(personName, -)` betekent dat er in de tabel `Person` een tupel is dat als eerste waarde `personName` heeft en als tweede waarde een willekeurige waarde. ▲

De omzetting werkt in twee delen. Het eerste deel is de parser. Deze parser ontleedt het XSLT-programma en zet het om naar een tussentijdse representatie. Deze representatie is een gerichte acyclische graaf met één enkele root. Elk pad van de root tot een blad van deze graaf kan daarna in het tweede deel omgezet worden tot een SQL-query. De unie van de queries die alle verschillende paden uitdrukken is de complete omzetting van het XSLT-programma.

Parsen Nu zal het parsen van de XSLT-templates besproken worden. Het eerste dat de parser doet is een functionele representatie maken van het XSLT-programma. Elke gedefinieerde template krijgt een functie. De naam van die functie is $f_{mode\text{-}naam}$. De functie krijgt als parameter een knoop mee waarin het programma zich bij de functieaanroep bevindt. Als er in de XSLT-templates parameters zijn gedefinieerd die meegegeven moeten worden, worden die ook in de functie als parameter geplaatst. Variabelen worden waar het kan vervangen door hun waarde. Constructs die voorwaarden uitdrukken zoals „if" worden ook naar een functionele representatie omgezet. Template aanroepen worden vervangen door functie-aanroepen. Maar er is nog niet geweten welke functie dat opgeroepen zal worden, er is enkel de mode geweten. Dus worden voorlopige functies f_x aangemaakt met x de mode. Deze tijdelijke functies worden later nog omgezet in de juiste definitieve functies. Ten laatste worden de stukken van de template die echte XML-gegevens genereren tussen return() geplaatst. Voorbeeld 4.24 toont hoe deze functies aangemaakt worden.

Voorbeeld 4.24. De templates worden omgezet in functies. De eerste template heeft mode '0', de default mode en matcht met een `person`-knoop. De tweede template heeft mode '1' en matcht ook een `person`-knoop.

$$f_{0\_person}(\$person) = \begin{array}{c} if(\$person/child/name =' John') \\ f_1(\$persons/person, \$person/child/age) \end{array}$$

▲

$$f_{1\_person}(\$person, \$age) = \begin{array}{c} if(\$person/child/age > \$age) \\ return(\$person/name) \end{array}$$

Een tweede taak van de parser is het toevoegen van functies die ingebouwde template-regels voorstellen. Er worden dus voor elke bestaande mode x de functie $f_x(\$default) = f_x(\$default/*)$ toegevoegd. Voorbeeld 4.25 toont hoe deze functies worden bijgevoegd.

Voorbeeld 4.25. De functies $f_0(\$def)$ en $f_0(\$def)$ worden toegevoegd.

$$\begin{array}{rcc} f_0(\$def) & = & f_0(\$def/*) \\ f_{0\_person}(\$person) & = & \begin{array}{c} if(\$person/child/name =' John') \\ f_1(\$persons/person, \$person/child/age) \end{array} \end{array}$$

$$\begin{array}{rcc} f_1(\$def) & = & f_1(\$def/*) \\ f_{1\_person}(\$person, \$age) & = & \begin{array}{c} if(\$person/child/age > \$age) \\ return(\$person/name) \end{array} \end{array}$$

▲

Een derde taak van de parser is het matchen van het huidige functionele programma tegen het schema van de XML-gegevens om zo de juiste functienaam te kunnen achterhalen. Dit gebeurt in Voorbeeld 4.26. Er wordt gekeken welke knopen in de XML-boomstructuur kinderen hebben en voor elk van die knopen wordt de functie f_x omgezet naar $f_{x\text{-}knoop}$. Let wel, dit is telkens aan de linkerkant van de functieaanroep.

Voorbeeld 4.26. De functies $f_x(\$def)$ worden omgezet naar $f_{x\_persons}(\$persons)$ en $f_{x\_child}(\$child)$.

$$\begin{array}{rcc} f_{0\_persons}(\$persons) & = & f_0(\$persons/*) \\ f_{0\_child}(\$child) & = & f_0(\$child/*) \\ f_{0\_person}(\$person) & = & \begin{array}{c} if(\$person/child/name =' John') \\ f_1(\$persons/person, \$person/child/age) \end{array} \end{array}$$

$$\begin{array}{rcc} f_{1\_persons}(\$persons) & = & f_1(\$persons/*) \\ f_{1\_child}(\$child) & = & f_1(\$child/*) \\ f_{1\_person}(\$person, \$age) & = & \begin{array}{c} if(\$person/child/age > \$age) \\ return(\$person/name) \end{array} \end{array}$$

▲

Hierna worden de wildcards geïnstantieerd. Dit wil zeggen dat $\$default/*$ vervangen wordt door relevante kinderen. Relevante kinderen zijn instanties van de * waarvoor aan de linkerkant een functie bestaat. Voor andere kinderen zal de functie aan de rechterkant nooit opgeroepen worden. Als geen instantie voor een wildcard kan gevonden worden, wordt die regel weggelaten. Voorbeeld 4.27 toont het resultaat hiervan.

Voorbeeld 4.27. Voor `persons` zijn de relevante kinderen beperkt tot `person`. Voor `child` zijn er zelfs geen relevante kinderen gevonden. Deze regel wordt dus weggelaten.

$$\begin{array}{rcc} f_{0\_persons}(\$persons) & = & f_0(\$persons/person) \\ f_{0\_person}(\$person) & = & if(\$person/child/name =' John') \\ & & f_1(\$persons/person, \$person/child/age) \end{array}$$

$$\begin{array}{rcc} f_{1\_persons}(\$persons) & = & f_1(\$persons/person) \\ f_{1\_person}(\$person, \$age) & = & if(\$person/child/age > \$age) \\ & & return(\$person/name) \end{array}$$

▲

Dan worden alle navigaties behalve ouder/kind navigaties omgezet naar simpele ouder/kind navigaties. Een XPath expressie van de vorm `x//y` zal omgezet worden naar een XPath expressie van de vorm `x/z1/.../y || x/z2/.../y`. De mogelijke paden worden gewoon met behulp van het schema geïnstantieerd. Alleen geldige navigaties worden behouden. Als er een functie met een ongeldige navigatie wordt opgeroepen, dan valt de functie-aanroep gewoon weg. Ongeldige navigaties in $return()$ statements geven ook lege resulaten. In het voorbeeld komen geen andere navigaties voor dan ouder/kind navigaties en dus kan deze stap voor dit bepaald voorbeeld overgeslagen worden.

Ten laatste worden de functie-aanroepen aan de rechterkant gematcht tegen het schema. Als de parameter van de functie-aanroep de naam *name* heeft, wordt de functie f_x vervangen door f_{x_name}. Het resultaat hiervan kan teruggevonden worden in Voorbeeld 4.28.

Voorbeeld 4.28. Hier worden voor f_0 en f_1 telkens enkel person knopen opgevraagd en de functienamen worden uitgebreid met person

$$\begin{array}{rcc} f_{0\_persons}(\$persons) & = & f_{0\_person}(\$persons/person) \\ f_{0\_person}(\$person) & = & if(\$person/child/name =' John') \\ & & f_{1\_person}(\$persons/person, \$person/child/age) \end{array}$$

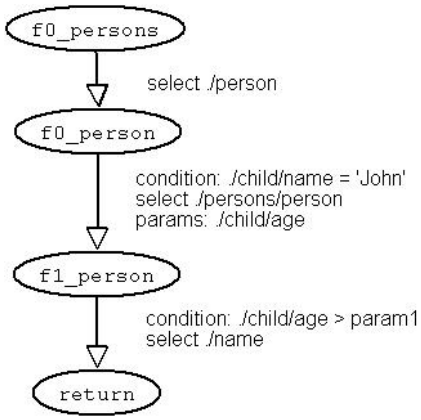
$$\begin{array}{rcc} f_{1\_persons}(\$persons) & = & f_{1\_person}(\$persons/person) \\ f_{1\_person}(\$person, \$age) & = & if(\$person/child/age > \$age) \\ & & return(\$person/name) \end{array}$$

▲

Het parsen heeft een functioneel programma gegenereerd. Dit functioneel programma kan nu tot een graaf omgezet worden. Elke functie is een knoop in de graaf. De functie die de root als parameter meekrijgt wordt bovenaan de graaf geplaatst. Elke keer een functie g wordt aangeroepen in een functie f, wordt er een pijl getrokken in de graaf van de knoop die f voorstelt naar de knoop die g voorstelt. Als er een voorwaarde rust op de aanroep van g wordt deze conditie op de pijl geplaatst. Hierna wordt deze graaf omgezet in verschillende single-call grafen. Zo een graaf kan slechts op één enkele manier doorlopen worden. Er wordt in de root begonnen en volgt het pad tot aan het blad. Voor elk verschillend pad in de tussentijdse graaf wordt een single-call graaf opgesteld. Voorbeeld 4.29 toont een tussentijdse graaf.

Voorbeeld 4.29. Figuur 4.19 toont de tussentijdse graaf. Tussen $f_{0\_persons}$ en f_{0_person} wordt er van `persons` naar `person` gesprongen in de XML-boomstructuur. Dit wordt aangegeven door `select ./person`. Er is geen voorwaarde geplaatst op de functieaanroep en er wordt ook geen parameter doorgegeven. Tussen f_{0_person} en f_{1_person} wordt er terug gesprongen naar de root en hieruit alle `person`-knopen geselecteerd. Dit wordt voorgesteld door `select /persons/person`. Op deze functie-aanroep is wel een voorwaarde geplaatst en deze wordt dus ook op de pijl geplaatst. De parameter die aan de functie meegegeven wordt, is ook vermeld langs de pijl. Tussen f_{1_person} en *return* wordt er gesprongen naar de `name`-knoop van de `person`-knoop. Dit wordt aangegeven door `select ./name`. Ook de voorwaarde is hier langs de pijl weergegeven. Deze aanroep heeft geen parameters. Merk op dat deze tussentijdse graaf al een single-call graaf is dus die hoeft niet meer omgezet te worden. ▲

QTree Voor elke single-call graph wordt er nu een QTree opgesteld. Een QTree is een boomstructuur die aan de hand van het schema en de single-call graaf wordt opgesteld. Een QTree bestaat eigenlijk uit drie delen, namelijk een boomstructuur, een verzameling voorwaarden en mappings voor parameters. De boomstructuur stelt de navigatie doorheen de XML-boomstructuur voor. De knopen die door de XPath uitdrukking bezocht worden in de XML-boomstructuur, komen ook in deze boomstructuur voor. De verzameling condities zijn voorwaarden die staan op het mogen traverseren van de QTree boomstructuur. Deze voorwaarden zijn niet alleen de if-constructs, maar ook predicaten in XPath uitdrukkingen. Een laatste deel is de verzameling mappings voor parameters. Een parameter kan een zelfgedefinieerde waarde hebben of kan een verzameling van één of meerdere knopen voorstellen. Een



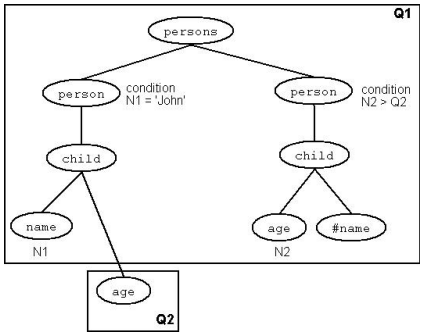
Figuur 4.19: Tussentijdse representatie en single-call graaf van het XSLT-programma

parameter kan zelf ook het resultaat zijn van een andere XSLT-query. Deze laatste soort parameters worden voorgesteld door geneste QTree's. Het opstellen van de QTree wordt duidelijker aan de hand van een voorbeeld. Voorbeeld 4.30 construeert een QTree aan de hand van een gegenereerde single-call graaf.

Voorbeeld 4.30. Figuur 4.20 toont de QTree voor het XSLT-programma. De query traverseert eerst van `persons` naar `child` en heeft daarvan de kinderen `age` en `name` nodig. Dus de QTree moet die pad bevatten. Daarna springt de query terug naar boven, naar de `persons`-knoop, om terug naar `child` te kunnen gaan. Ook van deze `child`-knoop zijn de kinderen `name` en `age` nodig. Er zijn twee verschillende instanties nodig van `person`, maar niet van `persons`. Er kan namelijk dezelfde `persons`-knoop gebruikt worden om alle kinderen op te vragen. De boomstructuur is gegeven door knopen en bogen in de QTree. Merk op dat een `age`-knoop in een aparte QTree zit. Dit komt omdat in de single-call graaf deze `age`-knoop als parameter meegegeven wordt. Dit is het resultaat van een andere XSLT-query en moest dus als geneste QTree voorgesteld worden. De verzameling voorwaarden wordt gegeven door de 'condition'-statements die bij de knopen staan. Deze kunnen afgelezen worden uit de single-call graaf en hierin worden bepaalde knopen gerefereerd door een variablenaam in plaats van het volledige pad, ofwel een QTree zoals dit bij Q2 het geval is.

▲

SQL genereren In het laatste stuk wordt de QTree omgezet naar SQL aan de hand van de viewtree, hierboven besproken. Eerst zullen de knopen in de



Figuur 4.20: QTree van het XSLT-programma

QTree gebonden worden aan instanties van de relationele tabellen. Daarna wordt de WHERE clause gegenereerd aan de hand van de binding die in de eerste stap gebeurde. Tenslotte worden de bindingen voor de knopen die teruggegeven moeten worden in de SELECT clause gezet. Een voorbeeld hiervan wordt gegeven in Voorbeeld 4.31.

Voorbeeld 4.31. De `person`- en `child`-knopen worden gebonden aan verschillende tabellen en er wordt gezorgd dat de `child`-knopen kinderen zijn van de juiste `person`-knopen. Dit gebeurt door de condities `c1.personName = p1.name AND c2.personName = p2.name`. Dan worden de expliciete condities in de WHERE-clause bij geplaatst. Voor de geneste QTree wordt een geneste SQL-query opgesteld op dezelfde manier. Uiteindelijk wordt de SELECT-clause gegenereerd aan de hand van de gemarkeerde knopen in de QTree.

```
SELECT p2.name
FROM person AS p1, person AS p2, child AS c1, child AS c2
WHERE c1.personName = p1.name AND c2.personName = p2.name
AND c1.name = 'John' AND c2.age <
(SELECT c3.age
FROM child c3
WHERE c3.name = c2.name)
```

▲

Het binden van een knoop in de QTree associeert die knoop met een tabel en een kolom. Deze tabel en kolom kunnen dan gezien worden als de waarde van die knoop. Dit binden gebeurt op een top-down manier, zodat elke knoop pas gebonden kan worden als ook zijn ouderknoop gebonden is. Voor elke andere knoop die met dezelfde tabel en kolom geassocieerd wordt, wordt een nieuwe instantie van de tabel aangemaakt in de SQL-query. Een uiteindelijke binding van een knoop is een verzameling bindingen van alle knopen van de root tot die knoop, een associatie met een tabel en kolom, een

lijst van tabellen die in de FROM clause moeten komen en de voorwaarden inherent aan het traverseren.

In de tweede stap wordt de WHERE clause gegenereerd. Eerst worden de voorwaarden inherent aan het traverseren in de WHERE clause geplaatst. Dan worden uit de QTree de expliciete voorwaarden gehaald en ook in de WHERE clause geplaatst. Uitdrukkingen in de voorwaardes worden als volgt vervangen. Een constante expressie wordt gewoon overgenomen. Een pointer naar een QTree knoop die een enkele knoop voorstelt wordt vervangen door zijn binding en een pointer naar een QTree die een verzameling knopen voorstelt wordt vervangen door een geneste SQL-query. Deze query wordt bekomen door het genereren van SQL recursief op te roepen op de QTree waarheen gewezen wordt.

In de derde en laatste stap wordt de SELECT clause gegenereerd. Uitdrukkingen of QTree knopen die in de output moeten komen staan in de QTree aangeduid. Deze uitdrukkingen worden op net dezelfde manier gegenereerd als bij het genereren van de WHERE clause.

Optimalizaties Nu dat de SQL-query gegenereerd is kan ze nog geoptimaliseerd worden. In [24] worden voorstellen gedaan om deze query nog te optimaliseren maar deze worden buiten beschouwing gelaten.

4.3 Systemen

Het is duidelijk dat er technieken bestaan om XML-gegevens in relationele tabellen op te slaan. In hoeverre zijn deze technieken nu te vinden in de bestaande RDBMS? MySQL en PostGreSQL hebben geen ondersteuning voor XML. DB2 daarentegen heeft wel ondersteuning voor XML, namelijk door de XML Extender.

4.3.1 DB2

De RDBMS DB2 heeft een XML Extender om met XML-gegevens overweg te kunnen. Deze XML Extender voorziet twee manieren om XML-gegevens op te slaan, namelijk de XML COLUMN en de XML COLLECTION.

4.3.1.1 XML column

Als XML-gegevens door middel van een XML column worden opgeslagen in DB2, wordt het hele XML-document in een kolom van een tabel opgeslagen. De gegevens staan gewoon in tekstformaat in een tabel. Om een fragment

XML-gegevens op te slaan in een kolom moet die kolom wel eerst XML-enabled zijn, zodat het RDBMS weet dat er XML-gegevens in geplaatst zijn en dat er dus bepaalde operaties op kunnen uitgevoerd worden. Een fragment XML-gegevens kan van het type XMLCLOB, XMLVARCHAR of XMLFILE zijn.

Een XMLVARCHAR wordt opgeslagen als VARCHAR in de tabel en kan dus enkel het maximum aan karakters van een VARCHAR bevatten. Een XMLCLOB wordt opgeslagen als CLOB in de tabel en kan dus ook maar het maximum aan karakters van een CLOB bevatten. Een XMLFILE is enkel een referentie naar een bestand op het lokale bestandssysteem.

Nadat de kolom XML-enabled is kunnen er operaties op de kolom uitgevoerd worden. Een eerste operatie is het invoegen van een XML-document. Een andere operatie is het opzoeken van een XML-document of opzoeken van gegevens in het XML-document. Dit wordt hieronder wat meer uitgelegd. De XML-documenten kunnen ook aangepast of verwijderd worden.

Er kan ook een Document Access Definition of DAD aangemaakt worden voor die kolom. Zo een DAD kan tekstuele inhoud van elementen en attributen mappen op side tables. Dit zijn aparte tabellen die de tekstuele inhoud bevatten. Op deze tabellen kunnen dan indexen geplaatst worden en zo kan de inhoud van het XML-document snel onderzocht worden. Een DAD werkt aan de hand van een DTD van de XML-gegevens. In de DAD wordt een pad aangegeven in het XML-document waar de gegevens te vinden zijn die uiteindelijk in de side tables moeten komen.

Het ondervragen van de XML-gegevens kan uiteindelijk op twee manieren. Een eerste manier is het ondervragen van de side tables. Deze ondervraging zal heel snel kunnen uitgevoerd worden omdat de gegevens op relationele tabellen gemapt zijn door middel van de DAD. De ondervraging zal dus enkel op relationele gegevens zijn en dit gaat snel door SQL. Een tweede manier is het ondervragen van de XML COLUMN zelf. Dit kan doordat SQL uitgebreid is met functies uit de XML Extender. Dit zijn extracting functies. Een SQL-query als volgt kan uitgedrukt worden. `SELECT extractVarchar(Order, /Order/Customer/Name) from sales_order_view`. Er zijn nog extracting functies, die kunnen gevonden worden in [25]. Extracting functies kunnen ook gebruikt worden in de WHERE-clause. Er is spijtig genoeg een nadeel aan deze extracting functies. Ze kunnen namelijk niet overweg met paden die meerdere resultaten opleveren voor een bepaald XML-document. Deze queries kunnen bijgevolg niet uitgedrukt worden.

Het plaatsen van XML-documenten in XML COLUMN's is eigenlijk niet meer dan het archiveren van XML-documenten in een relationele database. Als de gegevens niet vaak ondervraagd moeten worden en dikwijls enkel het gehele document wordt opgevraagd kan een XML COLUMN voordelen op-

leveren. Een ander voordeel aan deze methode is dat er geen DTD gekend hoeft te zijn. Het ondervragen van de gegevens gebeurt enkel door de XPath evaluator die ingebouwd is in de DB2 XML Extender. Eigenlijk worden er dus geen van de voordelen van een RDBMS gebruikt. Als die wel gebruikt willen worden, is een DAD nodig die de gegevens in het XML-document naar side tables mapt.

4.3.1.2 XML collection

Een andere methode om XML-documenten in DB2 op te slaan is de XML COLLECTION. Bij deze methode worden de XML-gegevens helemaal gedecomposeerd tot elke tekstuele waarde ergens in een tupel in een tabel staat. De tagnamen worden hier volledig wegggelaten. De XML-gegevens worden verdeeld over één of meerdere tabellen, afhankelijk van het schema van de XML-gegevens. Op dezelfde manier kunnen relationele gegevens als XML voorgesteld worden met een DAD. Dit wordt beschreven in het stuk over publishing.

Een dergelijke XML COLLECTION moet dus eerst een naam krijgen en enabled worden. Er zijn procedures voorzien in de DB2 XML Extender om XML-documenten te decomposeren. Een XML COLLECTION is eigenlijk een verzameling van tabellen die gegevens bevatten die naar XML-gegevens gemapt zullen worden. Op dezelfde manier als bij XML COLUMN's bestaat er een DAD. In dit geval is de DAD verplicht, wat niet het geval was bij XML COLUMN. Deze DAD heeft dezelfde functie als bij XML COLUMN, alleen bij de XML COLUMN werd de DAD enkel gebruikt voor side tables waarin enkel bepaalde gegevens werden gemapt terwijl bij XML COLLECTION de DAD gebruikt wordt om elk element en attribuut te mappen op een kolom in een tabel. Voor het invoegen, ondervragen, aanpassen en verwijderen in een XML COLLECTION kan gewoon SQL gebruikt worden.

Het feit dat het ondervragen van de XML-gegevens in SQL moet gebeuren is in feite een nadeel. Er kan namelijk niet in een XML-ondervragingstaal ondervraagd worden. Dit is dus niet gewenst, want de bedoeling was om queries te kunnen stellen in een XML-ondervragingstaal.

4.4 Publishing

XML-publishing is ook een onderwerp dat handelt over XML en RDBMS. Het is namelijk een manier om relationele gegevens uit een RDBMS in een XML-view te „gieten". Zo lijkt het voor de gebruikers alsof de gegevens in XML-formaat zijn opgeslagen, terwijl dit natuurlijk niet het geval is. Omdat XML-

publishing geen manier is om XML-gegevens op te slaan in een database, maar een manier om relationele gegevens een XML-uitzicht te geven, wordt het onderwerp slechts kort aangeraakt.

Als formaat om gegevens uit te wisselen is XML erg geschikt. Maar vele bedrijven hebben hun gegevens net op een relationele manier in een RDBMS opgeslagen. Het probleem ontstaat wanneer de gegevens uit een RDBMS uitgewisseld moeten worden in een XML-formaat. Hier kan XML-publishing helpen. Een XML-publishing applicatie kan dan dienst doen als een middleware-applicatie die de relationele gegevens in een XML-view giet. Voor bepaalde bedrijven kan XML-publishing ook nog andere voordelen hebben zoals bijvoorbeeld privacy. De relationele gegevens die niet voor publicatie bedoeld zijn, worden simpelweg niet in de XML-view opgenomen.

Er zijn een paar eigenschappen die een goed XML-publishing applicatie zou moeten bezitten. Ten eerste moet de mapping van relationele gegevens naar XML-gegevens flexiebel zijn. Een omzetting die relationele gegevens uit een willekeurige relationele database mapt op een XML-document dat aan een bepaald rigide schema voldoet, is niet voldoende. Zo is het mogelijk om een relationele database te mappen op een XML-document dat eruit ziet als in Figuur 4.21. Een tweede eigenschap is dat het systeem in staat zou moeten zijn zo een XML-voorstelling van de gegevens te geven die de gebruiker wilt. Een XML-voorstelling van relationele gegevens moet intuïtief zijn. Een mapping zoals in Figuur 4.21 geeft niet veel meerwaarde. Integendeel, een gebruiker van deze XML-gegevens moet de gegevens eerst terug omzetten naar een relationele vorm om er een duidelijk beeld over te krijgen. In het beste geval zorgt de mapping voor duidelijke tagnamen zodat een gebruiker in een oogopslag ziet wat met de gegevens bedoeld wordt. Ten derde is het ook belangrijk dat het mogelijk is dat niet alle gegevens uit de relationele database in het XML-view geplaatst worden. De XML-publishing applicatie moet dus selectief kunnen zijn. De applicatie moet in staat zijn die gegevens weg te filteren die voor de beheerder van de relationele database privaat beschouwd worden. Een laatste eigenschap is dat ondervragingen efficiënt uitgevoerd moeten kunnen worden.

Aangezien de gegevens aan de gebruiker in een XML-view gepresenteerd worden, is het best dat ondervragingen over deze gegevens in een XML-ondervragingstaal worden uitgedrukt. Maar omdat de gegevens in een RDBMS zijn opgeslagen kunnen enkel queries op de gegevens gesteld worden in SQL. Er moeten dus vertalingen van XQuery, XPath of XSLT naar SQL gebeuren. Deze mapping moet de query in een XML-ondervragingstaal in een zo efficiënt mogelijke SQL-query omzetten. Het resultaat hiervan wordt uiteraard weer op XML gemapt. Het probleem om XQuery en XSLT om te zetten naar SQL is al besproken hierboven. Merk op dat de omzetting in

```
<Database>
  <dbname>...</dbname>
  <table>
    <tablename>...</tablename>
    <column>
      <attributename>...</attributename>
      <row>...</row>
      <row>...</row>
      ...
    </column>
    <column>
      ...
    </column>
    ...
  </table>
  <table>
    ...
  </table>
</Database>
```

Figuur 4.21: Een voorbeeld van een mapping van relationele gegevens op XML-gegevens

het geval van XML-publishing minder complex is omdat de XML-gegevens gegenereerd werden uit relationele gegevens en dus niet onbeperkt diep zijn van structuur.

Een XML-publishing applicatie zal meestal bovenop een RDBMS geplaatst worden en rechtstreeks met het RDBMS communiceren. De gebruiker communiceert met de XML-publishing applicatie en zo is de publishing applicatie een interface tussen de gebruiker en het RDBMS.

Hoofdstuk 5

Tests

Nu dat de theorie van de XML-databases behandeld is, biedt zich de vraag of die theorie omgezet in praktijk wel zo goed werkt als de theorie lijkt. Met andere woorden, hoe goed zijn de bestudeerde DBMS nu eigenlijk? Dit wordt duidelijk met de uitvoer van tests.

5.1 Testcases

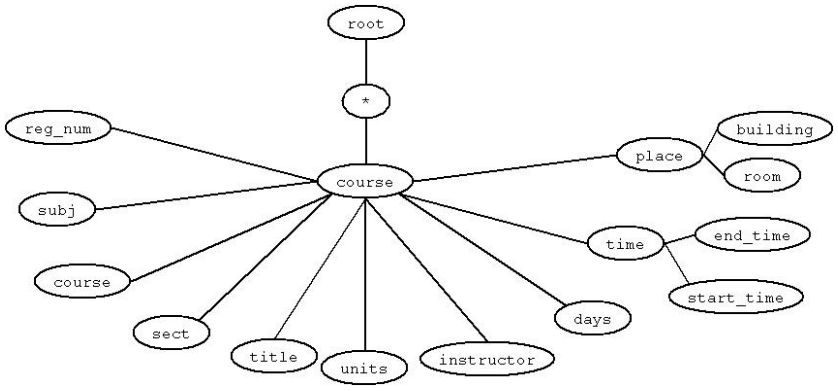
Een eerste benodigdheid bij het uitvoeren van test is een verzameling testcases. Een keuze die gemaakt moet worden is of bestaande testcases gebruikt worden of dat testcases zelf gemaakt worden. Een voordeel van het gebruik van bestaande, algemeen bekende testcases is dat anderen misschien dezelfde testcases gebruikt hebben en dus resultaten vergeleken kunnen worden. Een voordeel van het zelf maken van de testcases is dat juist de eigenschappen die getest moeten worden, beter getest kunnen worden omdat de testcases ernaar gemaakt zijn. Ook kunnen zelfgemaakte testcases met elkaar vergeleken en op elkaar afgestemd worden. Zo wordt er beter getest op bepaalde eigenschappen. Uiteindelijk kunnen hierdoor betere conclusies getrokken worden. Er werd gekozen om zelf testcases te genereren voor deze tests. De testcases bestaan uit een verzameling XML-bestanden en een verzameling ondervragingen.

5.1.1 XML-bestanden

De XML-gegevens voor het testen werden gegenereerd met een generator tool. Dit is een applicatie ontwikkeld door Ward Jans en Wim Janssen en werd aangepast naar de doeleinden. Deze applicatie kan XML-documenten genereren. In de programmacode wordt een boomstructuur opgebouwd voor een bepaald

schema. Een uitvoering van het programma genereert een aantal XML-documenten die instanties zijn van het schema. Als schema-documenten wordt een aantal DTD's van het internet [26] gehaald en eventueel aangepast naargelang de noden van de tests.

Er is een onderscheid gemaakt tussen drie verschillende groottes van XML-bestanden. XML-documenten die kleiner zijn dan 500kB worden kleine bestanden genoemd. XML-bestanden vanaf 500kB tot 100MB worden grote bestanden genoemd. Tenslotte zijn er bestanden groter of gelijk aan 100MB. Deze worden mega bestanden genoemd. In Voorbeeld 5.1 wordt een voorbeeld van een schema weergegeven.



Figuur 5.1: Schema-document

5.1.2 Ondervragingen

De ondervragingen op de gegevens worden in XQuery opgesteld. Voor het opstellen van de XQuery uitdrukkingen worden eerst een aantal ondervragingen in natuurlijke taal opgesteld om de bepaalde eigenschappen uit te drukken. Daarna wordt voor elke ondervraging in natuurlijke taal een XPath uitdrukking opgesteld voor elk schema. Voorbeeld 5.1 geeft de ondervragingen in natuurlijke taal en de XPath queries. De XQuery uitdrukkingen kunnen opgebouwd worden door een variabele te laten itereren over de resultaten van de XPath uitdrukking door middel van een for-lus en dan de variabele uit te schrijven in de body van de for-lus.

Voorbeeld 5.1. De ondervragingen worden in twee delen opgedeeld, namelijk de path traversals en de ondervragingen op naam. De path traversals zijn ondervragingen die forceren dat via een bepaald pad doorheen de boom gelopen wordt. De ondervragingen op naam zijn queries die een knoop selec-

teren in het bestand met een bepaalde naam. Hoe aan die knoop is gekomen mag door het systeem zelf gekozen worden.

- Path traversals
  - Geef alle bladeren van het langste pad.
    - Reed college: `/root/course/place/building`
    - PSDB: `/ProteinDatabase/ProteinEntry/reference/refinfo/year`
  - Neem het pad van hierboven en zoek een bepaalde voorouder.
    - Reed college: `/root/course/place/building/parent::*`
    - PSDB: `/ProteinDatabase/ProteinEntry/reference/refinfo/year/ancestor::*`
  - Vind alle siblings van een bepaalde knoop.
    - Reed college: `/root/course[1]/reg_num/parent::*/*`
    - PSDB: `/ProteinDatabase/*[1]/following-sibling::*`
- Zoeken op naam
  - Zoek alle knopen met een bepaalde naam die hoog-midden-laag in de boom voorkomt.
    - Reed college: `//course - //room - //room`
    - PSDB: `//ProteinEntry - // reference - //year`
  - Zoek naar alle knopen die een kind hebben met een bepaalde naam.
    - Reed college: `//course/reg_num`
    - PSDB: `//ProteinEntry/sequence`

▲

5.1.3 Problemen

Bij het genereren van de XML-bestanden ontstond na en tijdje een probleem omdat er maar een beperkte schijfruimte was. Dit is tijdelijk opgelost door het comprimeren van de bestanden. Op het probleem van de schijfruimte wordt later teruggekomen. De PC waarop de tests werden uitgevoerd, werden door de grote bestanden erg zwaar belast. Meerdere malen crashte explorer.exe in Windows2000.

5.2 XPath/XQuery evaluators

De originele probleemstelling was dat gewone XPath/XQuery evaluators niet voldoen voor het ondervragen van grote XML-bestanden. Er werd even getest of dit echt zo was, of dat misschien de XPath/XQuery evaluators een inhaalbeweging aan het maken zijn om er toch steeds grotere bestanden te kunnen evalueren.

5.2.1 QuizXopen

Een eerste XQuery evaluator die getest werd was QuizXopen. Deze XQuery evaluator had geen problemen met kleine bestanden, hetgeen verwacht was. Van zodra de bestanden echter 5MB of groter werden, kreeg deze evaluator al erge problemen met bepaalde ondervragingen. Het systeem gaf een „out of memory"-error bij bestanden vanaf 10MB.

5.2.2 Stylus Studio

Een andere XQuery evaluator die werd getest was Stylus Studio. Deze XQuery evaluator verraste doordat het openen van een mega-bestand lukte. Het ondervragen van de bestanden tot en met 50MB gaven geen noemenswaardige problemen. De meeste ondervragingen liepen af na 2 minuten. Van zodra echter de bestanden meer dan 75MB groot waren lukte het ondervragen niet meer, het systeem gaf een „out of memory"-error.

5.2.3 Saxon

Een XPath evaluator die getest werd is Saxon. Als deze evaluator een antwoord kan geven, doet hij dat vrijwel binnen de halve minuut. Als het systeem de grootte van het bestand echter niet aankan, geeft het een „out of memory"-error. Het is dus alles of niets bij dit systeem.

5.3 Native Systemen

5.3.1 OrientX

OrientX draait op Windows 2000. Het installeren van het systeem was geen enkel probleem. Bij de bestanden zaten ook voorbeeldgegevens. Dit waren enkele XML-bestanden, enkele bestanden met XQuery-ondervragingen en enkele bestanden met XPath-uitdrukkingen. Deze XML-bestanden waren

echter kleine bestanden, namelijk rond 100kB tot 200kB. De tekstuele interface werd gebruikt maar OrientX voorziet ook een grafische interface. Een korte overzicht van de tekstuele interface is gegeven in Voorbeeld 5.2.

Voorbeeld 5.2. De volgende commando's worden ondersteund:

```
Connect ip_address
Cr8DS -s dataSetName -t schemaFileName
DelDS -s dataSetName
Import -s dataSetName -d docName
Export -s dataSetName -d docName -o outputfilename
DelDoc -s dataSetName -d docName
XPath {-q queryexpr | -f input} -o resultfile
XQuery -f input -o resultfile
Update -f input
LstAllDS
LstDoc -s dataSetName
Exit
```

Deze commando's zijn achtereenvolgens het verbinden met de server, het maken van een dataset, het verwijderen van een dataset, het importeren en exporteren van een document in de dataset, verwijderen van een document uit een dataset, het ondervragen van de dataset aan de hand van een XPath of XQuery ondervraging, het aanpassen van een dataset, alle datasets weergeven, alle documenten uit een dataset weergeven en tenslotte het afsluiten van de applicatie. ▲

In eerste instantie werden de meegegeven voorbeelden uitgeprobeerd. Het uitvoeren hiervan bleek geen probleem. Dan is geprobeerd om de zelfgemaakte testcases te laden in de database. Dit leek te lukken, want er was geen probleem gesignaleerd bij het creëren van de dataset of bij het laden van een document. Maar er bleek een probleem te zijn bij het ondervragen van de gegevens. Een ondervraging had ofwel geen resultaat ofwel het resultaat van een vorige ondervraging die wel gelukt was ofwel crashte de OrientX server. De makers van OrientX lieten weten dat hoogstwaarschijnlijk het schemadocument fout was. Spijtig genoeg gaf het systeem geen foutmelding als het schemadocument niet correct kon geïnterpreteerd worden. Het maken van een dataset met een schemadocument leek dus gelukt te zijn, maar in feite was er een dataset aangemaakt zonder schema. Na het uitpluizen van het schemadocument bleken hier toch geen fouten in te zitten. Het xsd-document

is op verschillende manieren gegenereerd, namelijk handmatig en met behulp van een tool dat een DTD omzet in een XML Schema document. Voor de zekerheid is het xsd-bestand ook nog eens nagekeken met een XML Schema checker. Het XML Schema bestand was dus zeker juist. Toch lukte het niet om een andere dataset te ondervragen dan de dataset die was meegegeven met het programma. Het was dus uiteindelijk niet gelukt om OrientX op een degelijke manier te testen. De meegegeven dataset was namelijk maar een schamele 50kB groot. Dit is enkel een kleine file. Grote of mega bestanden zijn er dus niet getest.

5.3.2 Tamino

Tamino is een commercieel systeem en bijgevolg was alleen de evaluatieversie gratis beschikbaar. Deze evaluatieversie is erg beperkt in zijn mogelijkheden. Zo is de maximum grootte van de database gesteld op 2MB. De XML-gegevens moeten trouwens een schema hebben dat gedefinieerd moest zijn vooraleer er XML-gegevens in de database geplaatst kunnen worden. Ook is er maar een beperkte toegang tot de indexstructuren. Enkel de verplichte indexstructuren zijn verkrijgbaar in de evaluatieversie. Deze verplichte indexstructuren zijn dan ook weer niet af te zetten. Ook bij dit systeem waren voorbeeldgegevens meegeleverd in het evaluatiepakket.

Het testen van dit systeem was erg beknopt, omdat het systeem niet veel toelaat in zijn evaluatieversie. Het importeren van de gegevens ging zonder problemen en Tamino heeft een intuïtieve grafische interface. De ondervragingen op de gegevens werden ook in een mum van tijd uitgevoerd. Spijtig genoeg kon niet getest worden hoe Tamino overweg kan met grote of mega bestanden, aangezien deze bestanden te groot waren voor de evaluatieversie. Het beheer van het secundair geheugen werd dus niet getest, hetgeen het belangrijkste was wat er getest moest worden voor deze DBMS.

5.3.3 Timber

Timber is een systeem dat ook draait op Windows2000. Het installeren van het systeem verliep een beetje moeizaam, aangezien eerst Visual Studio 2003 bemachtigd moest worden. Na het installeren van Visual Studio 2003, hetgeen ook al erg moeizaam verlopen was, kon het Timber solution bestand gebuild worden. Ook dit verliep niet zonder hindernissen, maar de reactie op een paar mails naar één van de auteurs was erg verhelderend en Timber kon uiteindelijk geïnstalleerd worden op de PC.

Het laden van een klein bestand in Timber was geen probleem, alhoewel dit toch een drietal seconden duurde. Het ondervragen van deze kleine be-

standen verliep ook erg vlot. Alle ondervragingen liepen af in minder dan een seconde. Bij het laden van een groot bestand in Timber liep er wat mis. Het systeem gaf een melding dat er geen schijfruimte meer was. Na het uitdiepen van de documentatie bleek dat Timber tien maal de grootte van het XML-bestand reserveert om de datastructuren op te slaan op de harde schijf. Aangezien de ruimte op de harde schijf al beperkt was,zoals vermeld in 5.1.3, was het niet mogelijk om bestanden groter dan 30MB te testen. Om deze bestanden te kunnen testen waren reeds exotische oplossingen nodig. De tests op die bestanden verliepen allemaal vlot, alle ondervragingen gaven oplossingen na minder dan een seconde. Ook voor Timber zijn er dus geen mega bestanden getest kunnen worden.

5.3.4 Natix

Natix is een systeem waarvan binnenkort een commerciële versie van uitkomt. De evaluatie versie zou gratis zijn, maar die is ook nog niet beschikbaar. Om het systeem toch te kunnen bemachtigen is er een mail gestuurd naar de auteurs van de paper over Natix. Er werd geantwoord met een mailadres van de persoon die dat in orde kon brengen. Maar na 3 maanden aandringen en 5 mails later is het systeem nog niet bemachtigd. Dit is zeer spijtig, aangezien de structuur van Natix, besproken in de paper, erg veelbelovend leek. Het was dus niet mogelijk om Natix te testen.

5.3.5 Lore

Lore is een systeem van een vorige generatie. Na meerdere mislukte pogingen om het systeem te installeren, is er gemaild met de vraag hoe dit op te lossen. Het project Lore is al meerdere jaren geleden stopgezet en dat verklaart waarschijnlijk waarom er niet werd geantwoord op de mail. Aangezien het systeem niet geïnstalleerd geraakte kon het systeem niet getest worden.

5.3.6 Sedna

Sedna is ook een systeem dat draait onder Windows 2000. Het installeren van Sedna verliep erg vlot en het systeem is niet moeilijk om mee te werken. Om het systeem te testen werd de Java API gebruikt. Met een Java-programma werden eerst een aantal XML-bestanden in een database geplaatst. Daarna werden ondervragingen op de bestanden losgelaten. Voor de eerste kleine bestandjes werden queries snel uitgevoerd, ze waren namelijk allemaal uitgevoerd in minder dan 200milliseconden. Dan dook er ineens een probleem

op voor sommige bestanden. Eerst leek het alsof het systeem geen bestanden van een grootte van meer dan 500kB aankon, maar het ondervragen van sommige kleine bestanden lukte ook niet. Toen werd er contact opgenomen met de auteurs van de paper en de makers van het systeem. Die waren erg behulpzaam en na een weekend was het probleem opgelost. Na het onderzoeken van de geteste bestanden hadden ze een klein probleem gevonden in de driver voor de Java API. De Java API werd door de makers aangepast en teruggemaild. Na het vervangen van de driver kon nu ook geprobeerd worden de grote en de mega bestanden te testen. Na een paar succesvolle tests bleek er weer een probleem, maar dit keer met grotere bestanden. Er was nu ook het probleem dat die bestanden niet door te sturen waren via mail, dus moest een andere oplossing gevonden worden om de bestanden voor de makers beschikbaar te maken zodat ze het probleem konden localiseren. Uiteindelijk werd er een oplossing gevonden en de makers konden het probleem vinden. Na een nieuwe versie van de driver doorgestuurd te hebben werkte het programma goed en konden de testen allemaal uitgevoerd worden. Voorbeeld 5.3 toont een aantal resultaten van tests.

Voorbeeld 5.3. `Client started`

```
"psdb1.xml" has been loaded into collection "psdb1"
transaction took 140 milliseconds
"psdb4.xml" has been loaded into collection "psdb2"
transaction took 78 milliseconds

Executing query 1:
  document("psdb2")/*/*/*/*/*
Executing query 2:
  document("psdb2")/ProteinDatabase/ProteinEntry/reference
  /refinfo/year
Executing query 3:
  document("psdb2")/ProteinDatabase/ProteinEntry/reference
  /refinfo/year/parent::*
Executing query 4:
  document("psdb2")/ProteinDatabase/*[1]/parent::*/*
Executing query 5:
  document("psdb2")//ProteinEntry
Executing query 6:
  document("psdb2")//year
Executing query 7:
  document("psdb2")//ProteinDatabase
Executing query 8:
```

```
  document("psdb2")//ProteinEntry/sequence
Elapsed time:
query 1: 32 ms
query 2: 15 ms
query 3: 0 ms
query 4: 16 ms
query 5: 31 ms
query 6: 0 ms
query 7: 32 ms
query 8: 0 ms
```

(hoeveel van die resultaten moet ik hier laten zien) ▲

5.4 Relationele databases

5.4.1 Generic mapping

Om een idee te krijgen van de efficiëntie van de generic mappings, werden deze ook getest. Aangezien hier geen bestaande implementatie voor handen was, moest zelf gezorgd worden voor een implementatie. Dit werd weliswaar geen volledige middleware-applicatie. De applicatie is enkel in staat om XML-bestanden te versnipperen volgens één van de drie generic mappings, als reeds de juiste tabellen zijn aangemaakt. De implementatie is een klein java-programma dat gebruik maakt van een Xerces SAX-parser en de java API voor DB2. De ondervragingen worden handmatig omgezet naar SQL.

Het versnipperen van een klein XML-bestand verliep redelijk vlot. Een groot XML-bestand van toch maar 2MB duurde echter tien minuten om te versnipperen. Eens de bestanden versnipperd waren verliep het ondervragen erg vlot, mits een enkele uitzondering.

5.4.2 DB2

De DB2 XML Extender werd ook getest. Het versnipperen van het XML duurde telkens niet lang. Voor de grootste bestanden duurde dit een vijftal seconden. Het ondervragen van de XML-gegevens is voor XML COLLECTION niet getest omdat eigenlijk de gegevens in relationele tabellen zijn versnipperd en de ondervragingen in SQL moeten gebeuren. Het ondervragen van de XML-gegevens is voor XML COLUMN wel getest, namelijk met een extracting functie. Spijtig genoeg kunnen die extracting functions geen XPath expressies aan die meerdere knopen kunnen teruggeven dus zijn een paar XML-bestanden gemaakt die een root hadden met daaronder knopen,

die niet bereikt worden via een *. Spijtig genoeg geeft dit geen interessante gevallen om te testen. De resultaten van de tests waren positief. Dit is waarschijnlijk te wijten aan het feit dat de bestanden klein waren.

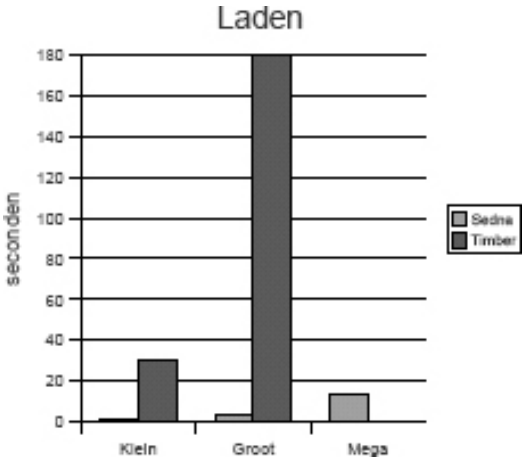
5.5 Resultaten

Om een juist beeld te kunnen vormen over de uitslagen van de tests, worden de resultaten in tabellen en grafieken geplaatst. Zo kunnen de verschillende systemen in een oogopslag vergeleken worden.

5.5.1 Laden van documenten

5.5.1.1 Native databases

De systemen Timber en Sedna zijn getest kunnen worden. De tijd nodig voor het laden van documenten, per systeem en per grootte-klasse van bestanden is weergegeven in Figuur 5.2. Het is duidelijk dat het laden van een bestand in Timber veel langer duurt dan in Sedna. De tijd nodig om een bestand te laden stijgt met de grootte van de bestanden die gebruikt zijn, hetgeen logisch is. Merk op dat het laden van een mega-bestand niet is gelukt in Timber, hetgeen de afwezigheid van een resultaat verklaart.

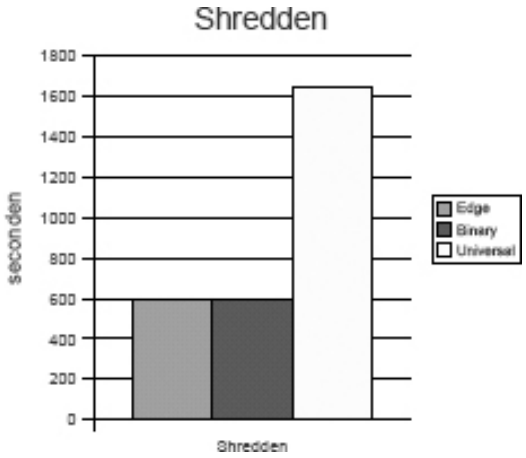


Figuur 5.2: Grafiek

5.5.1.2 Relationele databases

Om de generic mappings te testen werd een bestand van 2MB genomen. Dit is een bestand uit de klasse van de grote bestanden, maar is nog niet

erg groot. Deze grootte is gekozen om redelijke tijdstippen te bekomen bij het versnipperen van de XML-gegevens. De tijd nodig om dat bestand te versnipperen met elke aanpak is te vinden in Figuur 5.3. Het versnipperen van een bestand voor de edge aanpak en de binary aanpak duren ongeveer even lang, waar het versnipperen van datzelfde bestand voor de universal aanpak bijna driemaal zo lang duurt.



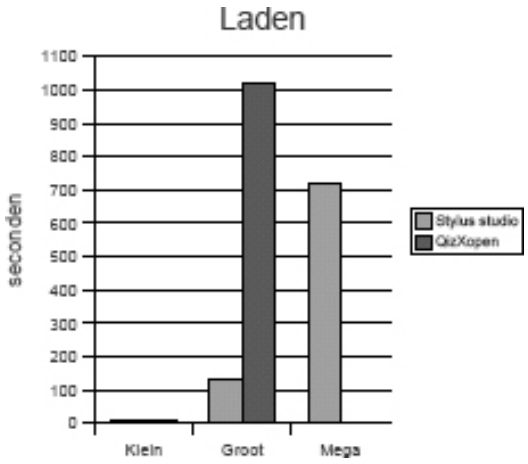
Figuur 5.3: Grafiek

5.5.1.3 XPath/XQuery evaluators

Voor het laden van bestanden zijn de XQuery evaluators getest. Met het laden wordt het openen van het bestand in de editor bedoeld. Het laden van een kleine bestanden ging ongeveer even snel bij beide systemen. Bij het laden van een groot bestand was er al een duidelijk verschil, QizXopen nam namelijk veel meer tijd hiervoor. Het laden van een mega bestand ging enkel in Stylus Studio, daarom staat er geen resultaat voor QizXopen vermeld voor een mega bestand.

5.5.1.4 Vergelijking

De tijd nodig voor het laden van bestanden is verschillend, afhankelijk welk type systeem gebruikt wordt. Bij native DBMS is het maximum aantal seconden voor het laden ongeveer 180, waar dat bij relationele systemen 1800 is en bij XQuery evaluators 1000 is. Hier valt op dat de native systemen erg weinig tijd nodig hebben in vergelijking met de andere systemen. Er dient wel gezegd dat bij relationele systemen enkel een groot bestand getest is en dat voor de andere systemen ook mega bestanden getest zijn. Het zou



Figuur 5.4: Grafiek

dus erg goed kunnen dat de gemiddelde tijdsduur voor het laden in een relationeel systeem veel groter is en bijgevolg de andere twee systemen nog meer overstijgt.

5.5.2 Ondervragen van documenten

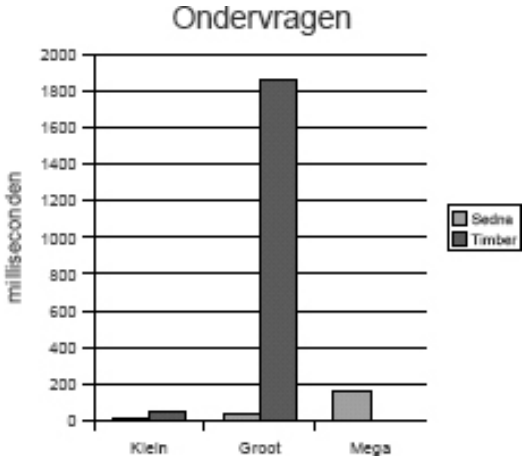
Nadat de XML-gegevens geladen zijn in de systemen, kunnen de gegevens ondervraagd worden. De resultaten van de ondervragingen zijn hier te vinden. Voor sommige systemen staan er resultaten per query. Deze queries zijn:

- Q1: `/*/*/*/*/*`
- Q2: `/ProteinDatabase/ProteinEntry/reference/refinfo/year`
- Q3: `//ProteinEntry`
- Q4: `//year`
- Q5: `//ProteinDatabase/count[*]`
- Q6: `//ProteinEntry/sequence`

5.5.2.1 Native databases

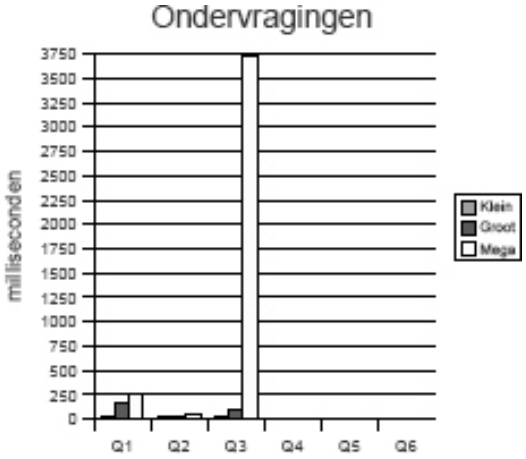
De gemiddelde tijdsduur van de uitvoer van een ondervraging, per systeem en per grootte-klasse van bestanden staat weergegeven in Figuur 5.5. Deze figuur lijkt sterk op de figuur die de tijdsduur van het laden bestanden in

native systemen weergeeft. Deze figuur heeft ook weer geen resultaat voor ondervragingen op mega bestanden voor Timber omdat hier geen resultaten van zijn.



Figuur 5.5: Grafiek

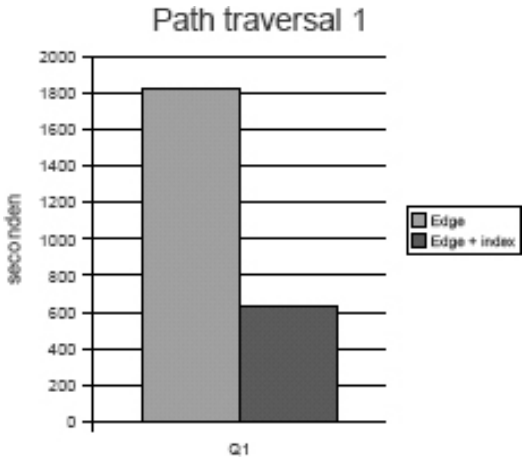
De uitvoer van verschillende queries in Sedna zijn in Figuur 5.6 weergegeven. Voor de uitvoer van de laatste drie ondervragingen is de uitvoertijd zo klein dat ze niet duidelijk op de figuur zijn aangeduid. Er is één query die eruit springt qua uitvoeringstijd. Dit is een query die veel gegevens als resultaat heeft. Dit verklaart de grotere tijdsduur van het uitvoeren van deze ondervraging. Voor de rest blijft de uitvoer van de ondervragingen onder de 250 milliseconden. Dit is een erg goed resultaat.



Figuur 5.6: Grafiek

5.5.2.2 Relationele databases

De duur van het uitvoeren van de ondervragingen is af te lezen uit de grafieken in Figuren 5.7, 5.8 en 5.9. Query Q1, oftewel path traversal 1 duurt erg lang. Dit is de enige query waarvan de tijdsduur in seconden is weergegeven. Deze query was enkel mogelijk om te zetten voor de Edge aanpak. De resultaten zijn af te lezen uit Figuur 5.7.



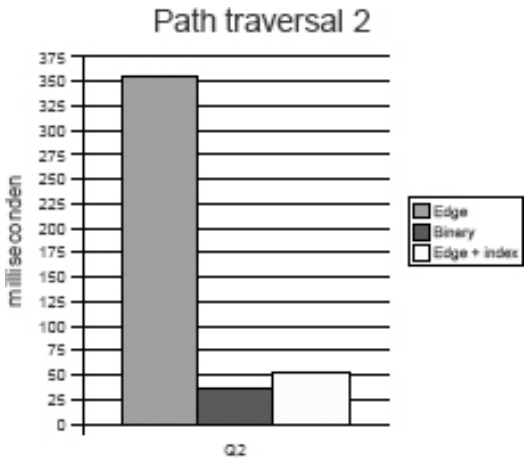
Figuur 5.7: Grafiek

De ondervraging Q2 is ook een path traversal die begint bij de root. De tijdsduur van het uitvoeren van Q2 staat weergegeven in Figuur 5.8. De tijdsduur voor de universal aanpak staat hier niet bij omdat die veel groter was, namelijk meer dan 10 keer de tijdsduur van de edge aanpak. Het toevoegen van deze tijdsduur zou het erg moeilijk maken om de andere aanpakken met elkaar te vergelijken.

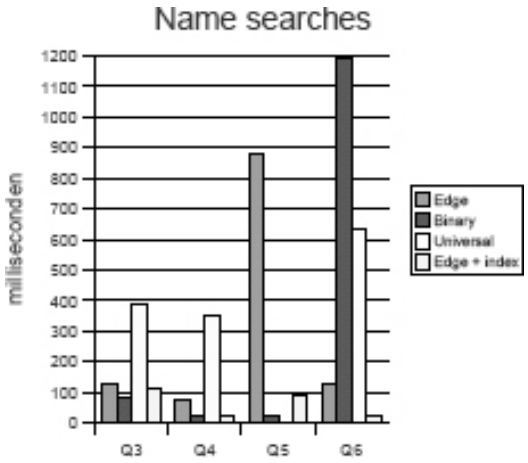
De ondervragingen Q3 tot en met Q6 zijn opzoekingen op naam. De tijdsduur van deze ondervragingen staat weergegeven in Figuur 5.9. Deze resultaten zijn veel kleiner dan de resultaten van de path traversals. Dit is ook logisch, aangezien een opzoeking op naam eerder een relationele query uitdrukt.

5.5.2.3 XPath/XQuery evaluators

De resultaten van het ondervragen door XPath/XQuery evaluators is weergegeven in Figuur 5.10. Zoals te zien is, gebruikt Stylus Studio het meeste tijd. Tegelijkertijd is Stylus Studio de enige evaluator die een mega bestand kan ondervragen. De andere evaluators konden dit niet en dus is ook geen

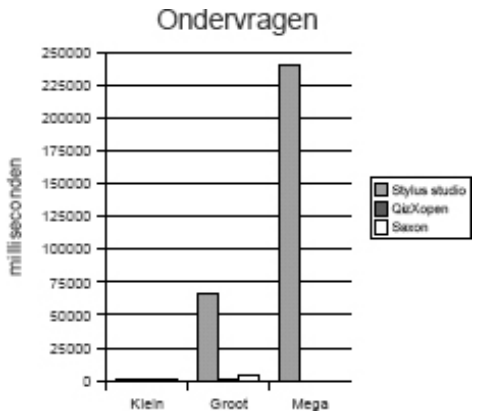


Figuur 5.8: Grafiek



Figuur 5.9: Grafiek

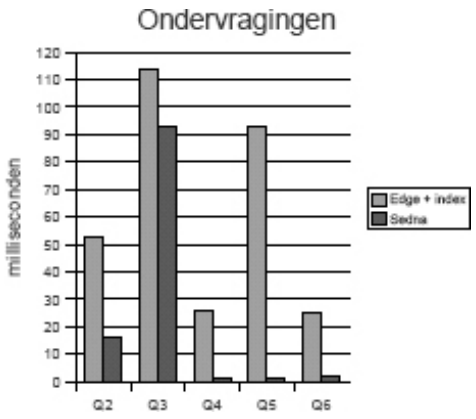
resultaat weergegeven. Van Saxon was het grootste bestand dat ondervraagd werd een 20MB.



Figuur 5.10: Grafiek

5.5.2.4 Vergelijking

Om het ondervragen van relationele en native systemen te kunnen vergelijken is de aanpak gekozen die het beste algemeen resultaat heeft voor de relationele systemen, en het beste systeem voor de native systemen. Dan zijn de resultaten voor grote bestanden genomen voor het native systeem, om te kunnen vergelijken met het geteste bestand voor het relationele systeem. De tijdsduur van ondervragingen Q2 tot en met Q6 zijn weergegeven in Figuur 5.11. Hier kan men zien dat het native systeem beter scoort op alle vlakken. Merk op dat ondervraging Q1 is weggelaten. Dit is gedaan omdat de uitkomst voor het relationele systeem erg groot was, veel groter dan het native systeem en veel groter dan alle andere uitkomsten. Voor de duidelijkheid is deze query dus weggelaten.



Figuur 5.11: Grafiek

Hoofdstuk 6

Conclusie

Nu dat er verschillende methodes besproken zijn en getest kunnen we hieruit een besluit trekken. De native databases lijken in theorie de beste oplossingen te bieden voor het opslaan van XML-gegevens. Toch bleek in dat in de praktijk het systeem minder goed of in sommige gevallen helemaal niet overweg kan met grote hoeveelheden gegevens. Dit wil niet zeggen dat alle native DBMS klaar zijn voor de prullenmand. In sommige systemen lukt het wél om de gegevens efficiënt te ondervragen. Hier moeten de mogelijkheden duidelijk nog verder onderzocht worden.

De manieren om de XML-gegevens in relationele databases te plaatsen zijn over het algemeen onintuïtieve constructies. De relationele DBMS doen hun best om de XML-gegevens op te slaan in een relationele database, maar hierdoor moet er gewrongen worden met de gegevens. Toch blijkt in de praktijk een heel aantal ondervragingen best snel te gaan als ze in relationele databases geplaatst zijn. Voor de ondervragingen op de structuur van de XML-gegevens worden deze relationele DBMS meestal toch ontmaskerd. Door het grote aantal joins is het simpelweg niet mogelijk om de ondervragingen efficiënt te laten verlopen.

Er kan gesteld worden dat vele native DBMS niet de efficiëntie halen van een RDBMS. Voor het enkele native systeem dat dat wel doet, is de efficiëntie van het ondervragen veel groter dan die van de relationele systemen. De native DBMS zijn dus zeker aan een opmars bezig. Het plaatsen van XML-gegevens in relationele databases zal binnen een korte tijd slechts een raar idee lijken.

Bibliografie

[1] Dtd tutorial. http://www.w3schools.com/dtd/default.asp.

[2] Vinay K. Chaudhri. *Transaction synchronization in knowledge bases: concepts, realization and quantitative evaluation.* PhD thesis, Toronto, Ont., Canada, Canada, 1996.

[3] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* John Wiley and Sons, Inc., 6th edition, 2002.

[4] Mary F. Fernández, Jérôme Siméon, and Philip Wadler. An algebra for xml query. In Kapoor and Prasad [5], pages 11–45.

[5] Sanjiv Kapoor and Sanjiva Prasad, editors. *Foundations of Software Technology and Theoretical Computer Science, 20th Conference, FST TCS 2000 New Delhi, India, December 13-15, 2000, Proceedings.*, volume 1974 of *Lecture Notes in Computer Science.* Springer, 2000.

[6] Oql tutorial. http://www.db.ucsd.edu/People/michalis/notes/O2/OQLTutorial.htm.

[7] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring up persistent applications. In Snodgrass and Winslett [8], pages 383–394.

[8] Richard T. Snodgrass and Marianne Winslett, editors. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.* ACM Press, 1994.

[9] Surajit Chaudhuri and Kyuseok Shim. Storage and retrieval of xml data using relational databases. In Apers et al. [10].

[10] Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors. *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. Morgan Kaufmann, 2001.

[11] Daniela Florescu and Donald Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[12] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In Atkinson et al. [13], pages 302–314.

[13] Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors. *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Morgan Kaufmann, 1999.

[14] Alin Deutsch, Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for xml. *Computer Networks*, 31(11-16):1155–1169, 1999.

[15] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[16] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing semistructured data with stored. In Delis et al. [17], pages 431–442.

[17] Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors. *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. ACM Press, 1999.

[18] David DeHaan, David Toman, Mariano P. Conses, and M. Tamer Özsu. A comprehensive xquery to sql translation using dynamic interval encoding. In Halevy et al. [19], pages 623–634.

[19] Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*. ACM, 2003.

[20] Torsten Grust, Sherif Sakr, and Jens Teubner. Xquery on sql hosts. In Nascimento et al. [21], pages 252–263.

[21] Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors. *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. Morgan Kaufmann, 2004.

[22] Mauricio A. Hernández, Lucian Popa, Yannis Velegrakis, Renée J. Miller, Felix Naumann, and Ching-Tien Ho. Mapping xml and relational schemas with clio. In *ICDE* [23], pages 498–499.

[23] *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*. IEEE Computer Society, 2002.

[24] Sushant Jain, Ratul Mahajan, and Dan Suciu. Translating xslt programs to efficient sql queries. In *WWW*, pages 616–626, 2002.

[25] Db2 xml extender. http://www-306.ibm.com/software/data/db2/ extenders/xmlext/.

[26] Xmldata repository. http://www.cs.washington.edu/research/ xmldatasets/.

[27] Extensible markup language. http://www.w3.org/XML.

[28] Franois Yergeau, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language 1.0. http://www.w3.org/TR/2004/REC-xml-20040204/, februari 2004.

[29] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, 2002.

[30] World Wide Web Consortium. XML schema. http://www.w3.org/XML/Schema.

[31] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele, and Till Westmann. Anatomy of a native xml base management system. *VLDB J.*, 11(4):292–314, 2002.

[32] Xiaofeng Meng, Daofeng Luo, Mong-Li Lee, and Jing An. Orientstore: A schema based native xml storage system. In *VLDB*, pages 1057–1060, 2003.

[33] Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[34] Xiaofeng Meng, Jing Wang, and Shan Wang. Supex: A schema-guided path index for xml data.

[35] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems, The complete book*. Prentice Hall, 2002.

[36] Role-based acces control. http://csrc.nist.gov/rbac/.

[37] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu, and Cong Yu. Timber: A native xml database. *VLDB J.*, 11(4):274–291, 2002.

[38] Tamino. http://www1.softwareag.com/corporate/products/tamino/ default.asp.

[39] Maxim Grinev, Andrey Fomichev, Sergey Kuznetsov, Kostantin Antipin, Alexander Boldakov, Dmitry Lizorkin, Leonid Novak, Maria Rekouts, and Peter Pleshachkov. Sedna: A native xml dbms. http://www.ispras.ru/ grinev/mypapers/sedna.pdf, 2004.

[40] Peter Pleshachkov and Leonid Novak. Transaction isolation in the sedna native xml dbms.

[41] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/.

[42] Michael Kay. saxon xslt-processor. http://saxon.sourceforge.net/.

[43] Michael Kay. saxon xslt-processor, betaalversie. http://www.saxonica.com/.

[44] J. Clark. XML Path Language (XPath). http://www.w3.org/TR/xpath.