De compilatie van NRC expressies naar SQL

Jan STULENS

promotor :
Prof. dr. Jan VAN DEN BUSSCHE

Eindverhandeling voorgedragen tot het bekomen van de graad
Licentiaat in de informatica afstudeervariant databases

universiteit hasselt

Abstract

Deze thesis handelt over het opslaan van complexe objecten in een database en het ondervragen van deze complex-object database. De complexe objecten worden gedecomponeerd en bewaard in een relationele database. De ondervragingstaal die gebruikt wordt, is Nested Relational Calculus (NRC). Elke NRC uitdrukking wordt vertaald naar een PSM programma. Het 'vertaalde' PSM programma wordt losgelaten op de relationele database. Dit PSM programma simuleert in de relationele database de NRC expressie over de complexe objecten. Het doel van deze thesis is een dergelijke applicatie te ontwikkelen.

Dankwoord

Graag zou ik even de mensen bedanken die mij gesteund en geholpen hebben tijdens mijn avontuur op het Limburgs Universitair Centrum (LUC) [1]. Er zijn zo veel personen die ik zou moeten bedanken, maar beperk me hier tot de voornaamsten. Vooraleerst zou ik mijn promotor Jan Van den Bussche en begeleidster Natalia Kwasnikowska willen bedanken, om me te begeleiden tijdens mijn master thesis en voor hun talrijke tips en raadgevingen. Bij problemen met DB2 tijdens de implementatie kon ik steeds terecht bij Dirk Leinders, waarvoor dank.

Veel dank ook aan mijn vrienden op de universiteit. Zoals Ben Hinssen en Jan Bollen, waar ik ettelijke projecten succesvol mee heb afgewerkt. Gijs Vermeulen dank ik voor zijn theoretisch inzicht en de verschillende discussies die we over gerelateerde onderwerpen gevoerd hebben. Ook niet te vergeten, is zijn aangenaam en humoristisch gezelschap doorheen de jaren. Nicolas Barbier wil ik niet vergeten te bedanken voor al zijn hulp. Ik heb mogen mee genieten van zijn gedrevenheid in de informatica.

Tenslotte wil ik mijn ouders en zus bedanken voor de sponsering, het geduld en de steun tijdens mijn studies aan de universiteit.

[1] Inmiddels is deze naam gewijzigd naar Universiteit Hasselt: http://www.uhasselt.be

Inhoudsopgave

Hoofdstuk 1

Inleiding

1.1 Relationeel model

Sinds het ontstaan van database systemen, systemen om grote hoeveelheden gegevens te beheren, is er een actief debat aan de gang over welk datamodel zo een dergelijk systeem zou moeten gebruiken. Deze vraag leek beantwoord, toen Ted Codd in 1970 een baanbrekend artikel [1] publiceerde, waarbij hij de begrippen uit de relationele algebra toepaste op het probleem van het beheren van grote hoeveelheden gegevens. Dit was het begin van een ontwikkeling in de databasewereld die binnen enkele jaren zorgde voor de definitie van het relationeel datamodel.

In een relationele database worden de gegevens opgeslagen in tabellen waarin de rijen de records vormen en de kolommen de gegevens die voor elk record worden bewaard. Tabellen kunnen met elkaar verbonden worden, door een kolom te laten verwijzen naar een record in een andere tabel. In een relationeel model is elke rij een record en kan bijgevolg geïdentificeerd worden door de waarde in een kolom of een combinatie van kolommen. Deze kolom(men), die het record identificeren noemen we de sleutel.

Voorbeeld:
In Figuur 1.1 is aangegeven dat er twee studenten in de database zitten, nl. 'Piet Vermeulen' en 'Sylvie Hoessels'. Zij hebben beiden een uniek id. Id is dan ook de sleutel in deze tabel.

De tabel 'Cursus' staat voor een verzameling cursussen die gevolgd kunnen worden door de studenten. De cursussen die hier gegeven zijn, zijn 'Economie', 'Basis Wiskunde 1', 'Basis Wiskunde 2' en 'Logica'. Voor elke cursus is er een moeilijkheidsgraad gegeven. Hier is de sleutel eveneens het id dat overeenkomt met de cursus.

In tabel 'Deelname' wordt bijgehouden welke student welke les volgt. We zien dat student met id 's0017203' één cursus volgt, nl. 'c003'. Als we deze gegevens opzoeken in de tabellen, vinden we dat het hier gaat om student Piet Geelen en hij volgt de cursus Basis Wiskunde 2. De overige records geven aan dat Sylvie Hoessels de cursussen Economie, Basiswiskunde 1 en Logica volgt.

Zoals het voorbeeld aangeeft, is het met het relationeel datamodel mogelijk grote hoeveelheden aan gegevens eenvoudig te bewaren. Deze gegevens kunnen ook aan elkaar gekoppeld worden door tabellen met elkaar te verbinden, zoals in het voorbeeld de tabel 'Deelname'.

De gegevens in de relationele database kunnen opgevraagd en aangepast worden met een programmeertaal, de gestandaardiseerde programmeertaal is SQL [2].

Het relationele datamodel ondersteunt enkel relaties, verzamelingen van tuples

Student	Id	Voornaam	Achternaam
	s0017203	Piet	Geelen
	s0118713	Sylvie	Hoessels

Cursus	Id	Naam	Moeilijkheidsgraad
	c001	Economie	4
	c002	Basis Wiskunde 1	5
	c003	Basis Wiskunde 2	3
	c004	Logica	4

Deelname	Student	Cursus
	s0017203	c003
	s0118713	c001
	s0118713	c002
	s0118713	c004

Figuur 1.1: Voorbeeld tabellen in relationele database.

van atomaire waarden. Complexe datastructuren dienen gedecomponeerd te worden in verschillende relaties. In dit voorbeeld willen we de relatie aantonen tussen de studenten en de cursussen die ze volgen. Omdat we de cursussen in het relationeel model niet als een object kunnen beheren, zijn we verplicht de relatie te decomponeren.

1.2 Geneste relationeel model

Verschillende applicaties worden echter gelimiteerd door de eenvoudigheid van het relationeel model. Deze applicaties willen complexe gegevens beheren zonder deze te moeten decomponeren. In de jaren '80 kwam er dan ook een andere ontwikkeling, de introductie van de geneste relaties. Het geneste relationeel model laat het willekeurig nesten van verzamelingen en tuples toe. Hierdoor is een meer natuurlijke presentatie van complexe data mogelijk. Men geeft dergelijke modellen ook de benaming object-gebaseerde of object-geöriënteerde modellen of nog anderen.

Om de objecten te kunnen beheren, hebben we ook een ondervragingstaal nodig voor de complex-object databases. De Nested Relational Calculus (NRC) [3, 4] is zo een gekende ondervragingstaal. Meer nog het is de theoretische fundering voor praktische complex-object ondervragingstalen, net zoals de relationele calculus en de relationele algebra (vormt het hart van SQL [2]) dat zijn voor het relationeel datamodel. NRC [3, 4] kan zelf gezien worden als het hart van OQL [5], een ondervragingstaal voor complex-object databases ontwikkeld door de 'Object Data Management Group' (ODMG). Bovendien inspireerde de NRC [3, 4] ook het ontwerp van verscheidene gegevensbanktalen voor semi-gestructureerde data zoals Quilt [6] waarop XQuery gebaseerd is.

Een andere ontwikkeling die zich voordeed was dat men object concepten ging to-

evoegen aan het relationele database systeem, dit resulteerde tot de object-relational database systemen[8]. Hierdoor kon de gebreuiker gebruik maken van zelf geschreven data types en functies.

1.3 Toepassingssectoren

De afzetmarkt voor complex-object database systemen bevindt zich in de bedrijfswereld, artificiële intelligentie, administratie automatisering, 'computer aided design' (CAD), bio-informatica en multimedia.[9] Er zijn nog andere sectoren, maar dit zijn de best gekende. CAD design objecten[11] of objecten gebruikt in administratie automatiserings systemen[12] zijn voorbeelden van complexe objecten.

Eén van de grootste sectoren op gebied van complexe objecten is de multimedia sector[10]. Hier heeft men bv. voor video nood aan multidimensionale array's. 'Video data maps' zijn een 3D compositie van een sequentie (1D array) van afbeeldingen (2D array's van pixels). Men moet deze array's direct kunnen aanspreken aan de hand van indexering en efficiënt door deze array's kunnen itereren. Het beheer en manipulatie van multimedia data is niet efficiënt in een relationele database. Er bestaan wel add-on's bij relationele databases, maar deze ondersteunen dan geen query optimalisatie. Vandaar dat in deze sector de nood aan complexe-object database systemen hoog is.

Toch zijn de object-complex database systemen nooit echt doorgebroken. Dit is te wijten aan de standaardisatie van relationele databases en de ondervragingstaal (SQL). Grote bedrijven, die hun applicaties op een commerciële relationele database hebben gebaseerd, zullen ook niet vlug overschakelen naar een complex-object database systeem alvorens ten volle overtuigd te zijn van de voordelen van een complex-object database systeem. Daar komt ook nog eens bij kijken dat de voordelen die ze halen groot moeten zijn om op te wegen tegen de kost van de omschakeling van het database systeem.

1.4 Overzicht thesis

Het doel van deze thesis is te onderzoeken hoe zulke complexe objecten en hun ondervraging, kunnen ondersteund worden in het relationeel model. Belangrijk hierbij is dat de decompositie dus automatisch gebeurt en niet meer de verantwoordelijkheid is van de gebruiker van de database. Het systeem, de compiler, krijgt als input een NRC expressie en geeft als output een bestand met een PSM programma in.De types van de complexe objecten worden getypeerd in Hoofdstuk 2. De complexe waarden worden uitgelegd in in Hoofdstuk 3. In Hoofdstuk 4 wordt de structuur gegeven hoe de complexe objecten, bewaard worden in de database. De syntax van de NRC expressies wordt verduidelijkt in Hoofdstuk 5, hier worden ook de nodige regels voor de welgedefiniëerdheid en de evaluatie van de expressies gegeven. Hoe de evaluatie van een expressie in ons systeem wordt gesimuleerd in de relationele database wordt uitgelegd in Hoofdstuk 7. Een belangrijke eigenschap van het systeem is dat niet enkel het eindresultaat van een geëvalueerde expressie wordt bewaard, maar ook de volledige "run", bestaande uit alle tussenresulaten van de deelexpressies. Hoe we complexe objecten inlezen in de database, wordt getoond in Hoofdstuk 8. Om de complexe data, die opgeslaan wordt volgens de structuur gegeven in Hoofdstuk 4, te beheren, worden de NRC expressies omgezet naar een PSM programma. In Hoofdstuk 6 geven we meer uitleg over SQL/PSM. Hoe de NRC expressies vertaald worden naar een PSM programma door de compiler, bespreken we in Hoofdstuk 9. De software architectuur van de compiler wordt gegeven in Hoofdstuk 10. Tot slot geef ik nog een kleine conclusie in Hoofdstuk 11.

Hoofdstuk 2

Complexe types

2.1 Syntax

Definitie 2.1 Stel $\mathcal{B}$, een onaftelbare verzameling van basistypes en $\mathcal{L}$, een onaftelbare verzameling van labels. Een complex type is één van volgende:

- Bottom type

  $\bot$

- Basistype

  $\in \mathcal{B}$

- Een eindige verzameling van complexe types

  $\{ \tau \}$

- Een tuple type

  $\langle l_1 : \tau_1, ..., l_n : \tau_n \rangle$

  alle labels l_i zijn verschillend

  positionering van $l_i : \tau_i$ is willekeurig

2.2 Toepassing

Als we in het voorbeeld van de inleiding in Figuur 1.1 de relatie niet willen decomponeren, dan hebben we nood aan complexe datastructuren, om de complexe objecten te kunnen beheren in een relationele databse.

```
Studenten = {Student}
Student = ⟨Voornaam : string , Achternaam : string , Cursn :
    Cursussen⟩
Cursussen = {Cursus}
Cursus = ⟨Naam : string , Moeilijkheidsgraad : int⟩
```

Listing 2.1: Een complex type.

Het is mogelijk dit in één geheel te schrijven:

```
Studenten = {⟨Voornaam : string , Achternaam : string ,
    Cursn : {⟨Naam : string , Moeilijkheidsgraad : int⟩})}
```

Listing 2.2: Verkorte notatie van het complex type.

Bij complexe types is het willekeurig nesten van verzamelingen en tuples toegestaan, dit is duidelijk zichtbaar in het voorbeeld. In Figuur 2.1 is het type in een boomstructuur weergegeven.

Hier gaat het dus eveneens om een verzameling studenten. Het object student is een tuple met drie velden. Deze velden zijn twee atomaire (string) waarden, nl. voornaam en achternaam. Het derde veld is een verzameling, waarvan de elementen van het type tuple, met twee velden, zijn. Dit type bestaat uit twee atomaire waarden, namelijk een naam en een moeilijkheidsgraad. Deze twee velden staan voor de cursus die de student volgt.

In tegenstelling met de relationele voorstelling hebben we geen id's meer nodig bij de complexe voorstelling om studenten te koppelen aan een cursus, zoals gebeurt in de 'Deelname' tabel. De verzameling cursussen die een student volgt, vallen nu rechtstreeks onder de student.

2.3 Waarden van types

Definitie 2.2 De verzameling van waarden van type τ, geschreven als $[\![\tau]\!]$, is gedefiniëerd als volgt:

- $[\![\bot]\!] = \emptyset$;
- $[\![b]\!]$, met b een basistype, is de verzameling basiswaarden van type b, dit is een niet lege deelverzameling van $\mathcal{A}$ bepaald door de applicatie;
- $[\![\{ \tau \}]\!]$ is de verzameling van alle eindige deelverzamelingen van $[\![\tau]\!]$;
- $[\![\langle l_1 : \tau_1, ..., l_n : \tau_n \rangle]\!]$ is de verzameling van alle tuples $\langle l_1 : \tau_1, ..., l_n : \tau_n \rangle$ met $v_i \in [\![\tau_i]\!]$ for $i = 1, ..., n$.

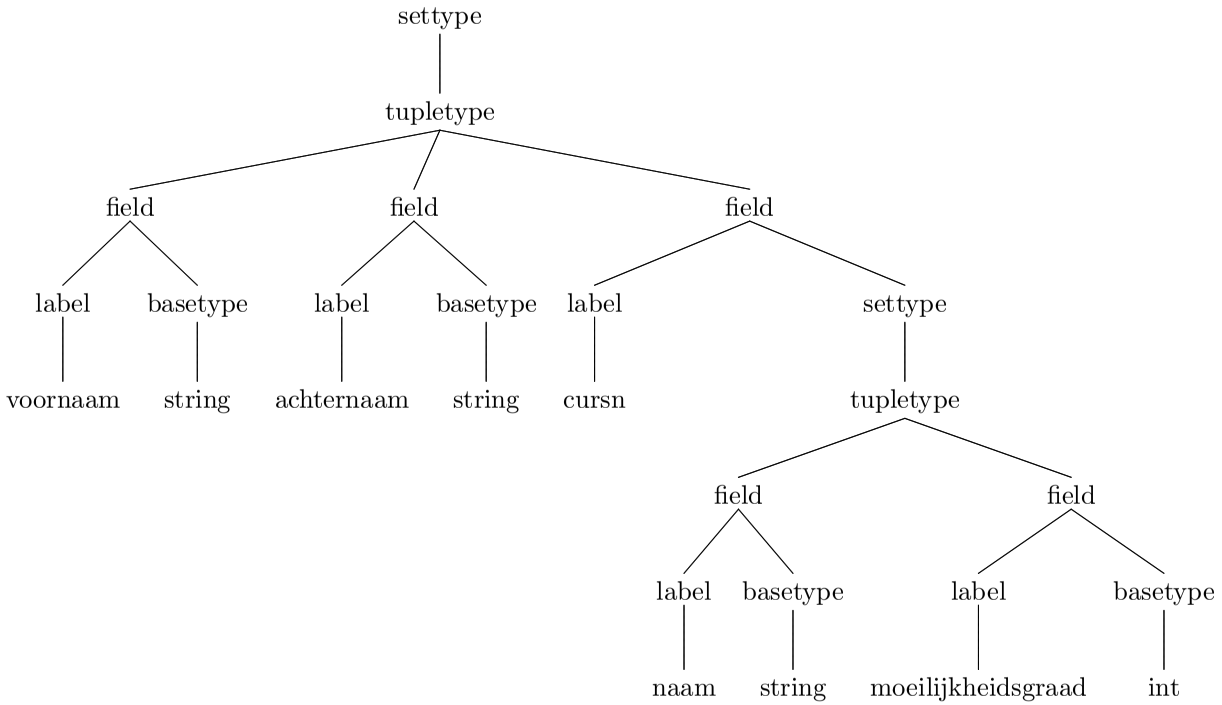
In volgend hoofdstuk wordt er dieper in gegaan op de complexe waarden.

2.4 DTD voor types

Types worden niet in de database bewaard. Het is overbodig de types op te slaan, omdat tijdens de uitvoering van de compiler op ieder moment de types van de objecten gekend zijn. Daar het bewaren van de types na de run niet van groot belang is, worden deze niet bewaard in de database. Voor de types zijn datastructuren ontwikkeld, die uitgelegd worden in Hoofdstuk 8. Deze worden voor sommige bewerkingen, bv. bij het vergelijken van twee types, omgezet in XML-formaat.

```
type      → bottom | settype | tupletype | basetype
bottom    → EMPTY
settype   → type
tupletype → field*
field     → label type
label     → #PCDATA
basetype  → #PCDATA
```

We gebruiken een eigen namespace 'cmplxtype' om duidelijk het verschil weer te geven tussen een element en '#PCDATA'. In Figuur 2.1 is een XML-string, die het type in Lisitng 2.2 voorstelt, gegeven in boomstructuur. Elke knoop bevindt zich in namespace 'cmplxtype'. Let op, de bladeren bevinden zich niet in namespace 'cmplxtype'.



Figuur 2.1: Het complex type in boomstructuur.

Hoofdstuk 3

Complexe waarde

3.1 Syntax

Definitie 3.1 Stel $\mathcal{A}$, een onaftelbare verzameling van basiswaarden en $\mathcal{L}$, een onaftelbare verzameling van labels. Een basiswaarde kan eender welke waarde zijn, die atomair is voor de applicatie, vb. een string, een getal, een aminozuur.
Een complexe waarde is één van volgende:

- Een atomair object

  SQL ingebouwde objecten zoals int, string, xml

  user-defined (xml-files)

  $\in \mathcal{A}$

- Een eindige verzameling van complexe waarden

  $\{ v_1, ..., v_n \}$

- Een tuple

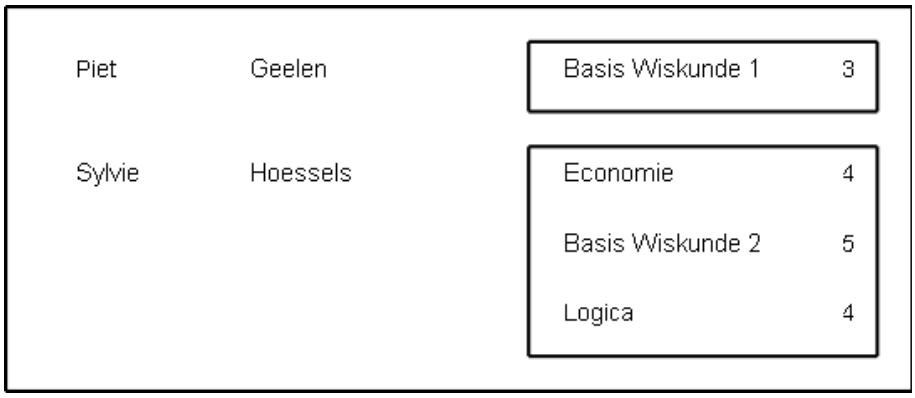
  $\langle l_1 : v_n, ..., l_k : v_n \rangle$

  alle labels l_i zijn verschillend

  positionering van $l_i : v_i$ is willekeurig

3.2 Toepassing

We kunnen nu dus complexe objecten rechtstreeks voorstellen. Een voorbeeld van een complex object van het type Studenten, uit Listing 2.2, die dezelfde informatie bevat als de relationele database uit Figuur 1.1, wordt getoond in Figuur 3.1.



Figuur 3.1: Een complex object.

Hoofdstuk 4

Complexe objecten in de relationele database

Zoals eerder aangehaald, worden de complexe objecten gedecomponeerd, zodat we ze kunnen bewaren in een relationele database. Elk complex object heeft een type. We zullen voor elk complex type, gedefiniëerd in definitie 2.1, apart bespreken hoe zij in de database bewaard worden.

4.1 Bottom type

In definitie 2.2 wordt aangegeven dat het enige object van type bottom de lege verzameling is, $[\![\perp]\!] = \emptyset$. De lege verzameling wordt niet in de database bewaard. De waarde van het id van de lege verzameling, $\emptyset$, is 0. Deze waarde is op voorhand bepaald en uniek.

4.2 Basistype

Bij basiswaarden is het type van het atomair object steeds gekend. Voor elk basistype dat in de complex-object database voorkomt, is er een verzameltabel. M.a.w. alle SQL [2] ingebouwde types, want user-defined types worden voorgesteld xml-formaat, zie definitie 2.2. De naam van een verzameltabel in de relationele database wordt als volgt gevormd, 'Pool_' + type. De structuur van zo'n tabel wordt gegeven in Figuur 4.2. Zo zullen bv. alle getallen in een tabel 'pool_int' terecht komen. Elke basiswaarde komt maximaal één maal voor in zijn pooltabel, elke basiswaarde is dus uniek in de database.

Pool_type	Id	Val
	int	int

Figuur 4.1: Relationele voorstelling verzameltabel, basistype.

4.3 Tuples

Een tuple bestaat volgens definitie 2.2 uit een gegeven aantal velden, $l_i : \tau_i$. Elk veld wordt in een aparte tabel bewaard. Zo'n tabel heeft als naam 'Att_' + l_i

(label). Figuur 4.2 geeft de structuur van de tabel weer. De records in deze tabel hebben een id-waarde voor het id van het tuple en een id-waarde van een andere basiswaarde, tuple of verzameling al naargelang het type τ_i van het attribuut met label l_i. Wanneer tabel Att_l_i nog niet bestaat in de relationele database, wordt deze aangemaakt.

Att_l_i	Id	Val
	int	int

Figuur 4.2: Relationele voorstelling attribuuttabel, tuple type.

4.4 Verzamelingen

De verzamelingen worden bijgehouden in één tabel, nl. 'Set'. De structuur van deze tabel vind je in Figuur 4.3. Het Id verwijst naar het id van de verzameling en El verwijst naar een element van de verzameling, hetzij een basiswaarde, tuple of verzameling. Als we volgende notatie gebruiken om een verzameling voor te stellen, $\{x|x \in Y\}$. Dan zal in de tabel Set voor elke $x \in Y$ een record aanwezig zijn met als Id het id van Y en als El het id van x.

Set	Id	El
	int	int

Figuur 4.3: Relationele voorstelling verzamelingen type.

4.5 Verantwoording relationele voorstelling

We hebben voor deze relationele voorstelling van objecten gekozen opdat we zo min mogelijk tabellen moeten construeren. We hebben nu maar één tabel waarin alle verzamelingen bewaard worden. De attribuuttabellen van een tuple hangen af van het label l_i, indien we dus twee tuple types, t_i en t_j, hebben en l_i en l_j zijn gelijk, dan bevinden de elementen bij deze twee labels van die twee tuples in dezelfde tabel, ook al is het type verschillend. Deze relationele voorstelling is mogelijk omdat we met verzameltabellen werken voor atomaire waarden. Hierdoor heeft een basiswaarde altijd een uniek id en zijn de attribuuttabellen en de verzamelingtabellen onafhankelijk van het type dat bij de attributen, respectievelijk de verzamelingen hoort.

4.6 Toepassing

Voorstelling van drie tuples $\langle a : 6, b :$'Jan'$\rangle$ (id:100), $\langle a : 7, b :$'Corneel'$\rangle$ (id:101) en $\langle a :$'Piet', $b : 10, c : true\rangle$ (id:208) en een verzameling (id:303) die tuples één en twee bevat.

De basiswaarden die voorkomen in het voorbeeld zijn, drie getallen, nl. 6,7 en 10, deze bevinden zich in de tabel Pool_int. Dan zijn er ook drie strings in de tabel Pool_string, nl. 'Jan', 'Piet' en 'Corneel'. Tot slot is er de tabel Pool_boolean die de waarden true en false bevat. Al deze waarden komen overeen met een uniek id.

Pool_int	Id	Val
	31	6
	32	7
	33	10

Pool_string	Id	Val
	77	'Jan'
	79	'Corneel'
	90	'Piet'

Pool_boolean	Id	Val
	34	true
	35	false

Figuur 4.4: Verzameltabellen voor atomaire waarden.

Het tuple $\langle a : 6, b :$'Jan'$\rangle$ wordt door twee records voorgesteld in de database, één in de tabel Att_a en één in de tabel Att_b. Het id van het tuple is 100, deze is ook terug te vinden in de tabel Att_a waarbij de waarde 31 voorkomt, dit is het id van het getal 6 in de verzameltabel voor getallen. In tabel Att_b is er eveneens een record met id 100 en waarde 77, het id van de string 'Jan'. Idem voor de tuples $\langle a : 7, b :$'Corneel'$\rangle$ met als id 101 en $\langle a :$'Piet', $b : 10, c : true\rangle$ met als id 208. Dit laatste tuple heeft ook nog een attribuut c en heeft dus ook een record in de tabel Att_c waarbij haar id, 208, gekoppeld wordt aan een waarde 34, het id van true in de Pool_boolean tabel.

Om verzamelingen uit te drukken in de database hebben we een tabel Set met als velden Id, de verzameling en El, de elementen die tot die verzameling behoren. In dit geval hebben we een verzameling met id 303, die twee elementen bevat, de tuples met id 100 en id 101.

Att_a	Id	Val
	100	31
	101	32
	208	61

Att_b	Id	Val
	100	77
	101	79
	208	33

Att_c	Id	Val
	208	34

Figuur 4.5: Atribuuttabellen voor tuples.

Set	Id	Val
	303	100
	303	101

Figuur 4.6: Tabel voor verzamelingen.

Hoofdstuk 5

NRC expressies

Alvorens de syntax te geven, leggen we enkele termen uit en bespreken kort de mogelijke NRC expressies.

Database schema: is een verzameling variabelen, waarbij elke variabele gedeclareerd is als zijnde van een bepaald type.

Database: over een gegeven schema, is een verzameling complexe objecten.

Input schema van een uitdrukking: Elke NRC expressie e heeft een input schema. Dit schema geeft het aantal variabelen als input voor de expressie e aan en van welk type ze zijn. We kunnen een expressie e evalueren op eender welke database D over haar input schema. Het resultaat van de evaluatie van expressie e over het input schema is een complex object.

Constante: de expressie kan gewoon een constante zijn. Deze expressie geeft dan gewoon de waarde van de constante terug.

Variabele: wanneer de expressie een variabele is (uit het input schema), kopiëert de expressie de input, die overeenkomt met de variabele, naar de output.

De lege verzameling: We kunnen de uitdrukking $\emptyset$ schrijven.

Singleton constructie: Het construeren van een singleton doen we door het schrijven van $\{e\}$.

Unie: Als x en y van hetzelfde (verzameling)type zijn of als we de 'join', zie Definitie 5.1, kunnen bepalen, dan kunnen we de unie nemen, $x \cup y$.

Opmerking: Wanneer we de unie moeten nemen van verzamelingen met elementen die van het type tuple zijn, kan het zijn dat deze tuples niet dezelfde labels hebben. Indien dit het geval is, zijn de elementen van de resulterende verzameling van het type met de gemeenschappelijke velden. Dit wordt theoretisch gestaafd in Definitie 5.1.

Voorbeeld:

$$\{\langle a : 6, b : 5, c : \text{'Jan'}\rangle\} \cup \{\langle a : 6, c : \text{'Bert'}, d : \text{'Atuo'}\rangle\} \text{ geeft}$$
$$\{\langle a : 6, c : \text{'Jan'}\rangle, \langle a : 6, c : \text{'Bert'}\rangle\}$$

Ontnestelen: Stel x is van het type int. x is dus een verzameling van verzamelingen van id's. $\bigcup x$ neemt dan de unie van alle verzamelingen van x. Als resultaat krijgen we dus een verzameling van alle id's.

Tuple constructie: Volgende expressie construeert uit haar twee input variabelen a en b een tupel $\langle id : a, naam : b\rangle$.

Selectie velden van tupel: Stel x, een input variabele, van het type $\langle id : int, naam : string\rangle$. De expressie $x.naam$ geeft dan de naam.

For constructie: Stel y van het type $\{\langle a : int, b : string\rangle\}$. Volgende expressie is een ingewikkelde manier om gewoon y te schrijven.

$$\text{for } x \text{ in } y \text{ return } x$$

De betekenis van de for-loop is deze van de verzamelingenconstructie $\{x|x \in y\}$. Als resultaat van voorgaande expressie krijgen we dus een verzameling waarin alle elementen van de input y opgesomd zijn. Let op, x is hier geen vrije variabele in de expressie. De variabele x is hier geen input, maar dient om door de elementen van de input y te lopen. We noemen zo'n variabele een gebonden variabele. Het type van deze variabelen kunnen we uit de input y halen. Vermits y van het type $\{\langle a : int, b : string\rangle\}$ is, is x van het type $\langle a : int, b : string\rangle$. Volgende expressie selecteert uit alle elementen van de input verzameling y, enkel de waarde die overeenkomt met label a.

$$\text{for } x \text{ in } y \text{ return } x.a$$

Het resultaat van deze expressie is een verzameling van integers, want het veld met label a heeft als type int. Deze expressie is het equivalent van de projectie $\pi(y)$ uit de relationele algebra. In de deelexpressie $x.a$ is x wel een vrije variabele, x wordt gebonden in de omvattende for-loop.

If-then-else: Deze constructie heeft een voor de hand liggende betekenis. De if-test kan de gelijkheid of de leegheid zijn.

Gelijkheid: Indien de twee resultaten van de twee deelexpressies, in de expressie $e = e$, gelijk zijn, geeft deze expressie als resultaat true en anders false.

Leegheid: Indien het resultaat van de deelexpressie, die een verzamelingtype heeft, leeg is, geeft deze als resultaat true en anders false.

Let constructie: In deze constructie

$$\text{let } x = e_1 \text{ in } e_2$$

is x een variabele die voorkomt in deelexpressie e_2. De variabele x krijgt de waarde van het resultaat van deelexpressie e_1. In de let-expressie behoort x niet tot de input, x is een gebonden variabele.

Functie: Hier wordt een functie f losgelaten op het resultaat van de deelexpressie. De functie f is een UDF, met een vast input type en vast output type.

5.1 Compatibele types

Definitie 5.1 Voor twee types τ en σ, definiëren we wanneer ze *compatibel* zijn, en indien zo, definiëren we hun *join*, dat opnieuw een type is, aangeduid door $\tau \vee \sigma$:

- twee basistypes $\mathbf{b_1}$ en $\mathbf{b_2}$ zijn compatibel indien ze een *least upper boud* bezitten, we schrijven dit als volgt $<:$, dit is een basistype $\mathbf{b_1} \vee \mathbf{b_2}$ waarvoor

$\mathbf{b_1} <: \mathbf{b_1} \vee \mathbf{b_2}$ en $\mathbf{b_2} <: \mathbf{b_1} \vee \mathbf{b_2}$ en $\mathbf{b_1} \vee \mathbf{b_2} <: \mathbf{b}$ voor een ander basis type $\mathbf{b}$ zodat $\mathbf{b} <: \mathbf{b_1}$ en $\mathbf{b} <: \mathbf{b_2}$;

- twee verzamelingen types $\{\, \tau \,\}$ en $\{\, \sigma \,\}$ zijn compatibel als en slecht als τ en σ compatibel zijn, en in dit geval zeggen we $\{\, \tau \,\} \vee \{\, \sigma \,\} = \{\, \tau \vee \sigma \,\}$;
- twee tuple types $\langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle$ en $\langle\, l_1 : \sigma_1, \ldots, l_m : \sigma_m \,\rangle$ met $n \geq m$ zijn compatibel als en slecht als τ_i en σ_i compatibel zijn voor $i = 1, \ldots, m$, en in dit geval zeggen we $\langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle \vee \langle\, l_1 : \sigma_1, \ldots, l_m : \sigma_m \,\rangle = \langle\, l_1 : \tau_1 \vee \sigma_1, \ldots, l_m : \tau_m \vee \sigma_m \,\rangle$.

5.2 Externe functies

Stel een eindige verzameling $\mathcal{E}$ van namen die externe functies aanduiden. Elke $f \in \mathcal{E}$ is een afbeelding van $\tau_{in} \to \tau_{out}$, waarbij τ_{in} en τ_{out} complexe types zijn. We duiden dit aan door $f : \tau_{in} \to \tau_{out}$. Externe functies zijn 'User Defined Functions' (UDF).

5.3 Syntax

Definitie 5.2 Stel een onaftelbare verzameling $\mathcal{X}$ van variabelen. Een *Nested Relational Calculus expressie* e is gedefiniëerd door volgende grammatica, met $\mathsf{a_b} \in [\![\, \mathbf{b} \,]\!]$, $x \in \mathcal{X}$, $f \in \mathcal{E}$ en $l_1, \ldots, l_n \in \mathcal{L}$.

$$
\begin{array}{rcl}
e & ::= & \mathsf{a_b} \mid x \\
& \mid & \emptyset \mid \{\, e \,\} \mid e \cup e \mid \bigcup e \\
& \mid & \langle\, l_1 : e, \ldots, l_n : e \,\rangle \mid e.l \\
& \mid & \mathsf{for}\ x\ \mathsf{in}\ e\ \mathsf{return}\ e \\
& \mid & e = e \mid e = \emptyset \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& \mid & f(e)
\end{array}
$$

In een tuple expressie $\langle\, l_1 : e_1, \ldots, l_n : e_n \,\rangle$ zijn alle labels l_i verschillend.

Alle NRC expressies voldoen aan voorgaande syntax. Een voorbeeld van een expressie, is de volgende:

```
⋃for y in input return
    if y.voornaam = 'Sylvie' then
        if y.achternaam = 'Hoessels' then
            ⋃for x in y.cursn return
                if x.moeilijkheidsgraad = 4 then
                    {x.naam}
                else
                    ∅
        else
            ∅
    else
        ∅
```

Listing 5.1: Een voorbeeld van een NRC expressie.

De expressie in Listing 5.1 voldoet aan de syntax gegeven in Definitie 5.2. Deze expressie geeft de namen van de cursussen terug, die gevolgd worden door Sylvie Hoessels en die een moeilijkheidsgraad hebben die gelijk is aan 4.

5.4 Vrije variabelen

Definitie 5.3 De *verzameling van vrij variabelen* van een NRC expressie e wordt genoteerd als $FV(e)$ en is gedefiniëerd als volgt.

- $FV(\mathtt{a_b}) = \emptyset$
- $FV(x) = \{ x \}$
- $FV(\emptyset) = \emptyset$
- $FV(\{ e \}) = FV(e)$
- $FV(e_1 \cup e_2) = FV(e_1) \cup FV(e_2)$
- $FV(\bigcup e) = FV(e)$
- $FV(\langle l_1 : e_1, \ldots, l_n : e_n \rangle) = \bigcup_{i=1}^{n} FV(e_i)$
- $FV(e.l) = FV(e)$
- $FV(\mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2) = FV(e_1) \cup (FV(e_2) \setminus \{ x \})$
- $FV(e_1 = e_2) = FV(e_1) \cup FV(e_2)$
- $FV(e = \emptyset) = FV(e)$
- $FV(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) = FV(e_1) \cup FV(e_2) \cup FV(e_3)$
- $FV(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = FV(e_1) \cup (FV(e_2) \setminus \{ x \})$
- $FV(f(e)) = FV(e)$

Opmerking $FV(e) \subseteq \mathcal{X}$.

5.5 Welgetypeerdheid

Definitie 5.4 Een *type toekenning* is een partiële afbeelding $\mathcal{T}$ van $\mathcal{X}$ naar de verzameling van complexe types. Voor elke expressie e en elke type toekenning $\mathcal{T}$, die gedefiniëerd is over $FV(e)$, kunnen we nagaan of e *welgetypeerd* is onder $\mathcal{T}$, en als dit zo is, kunnen we het mogelijke eindtype τ. We noteren dit als volgt $\mathcal{T} \vdash e : \tau$ bepalen. De definitie is gegeven door de regels in Figuur 5.1[1]. Wanneer $\mathcal{T} \vdash e : \tau$ niet kan worden afgeleid aan de hand van deze regels voor een gegeven $\mathcal{T}$ en e, is e niet welgetypeerd onder $\mathcal{T}$. De regels voor for-expressie en de let-expressie gebruiken volgende notatie. Zij $\mathcal{T}$ een type toekenning, $x \in \mathcal{X}$ en σ een complex type. Dan stelt $\mathcal{T}[x \mapsto \sigma]$ de type toekenning voor die gelijk is aan $\mathcal{T}$, behalve voor de variabele x die van het type σ is.

Neem NRC expressie e uit Listing 5.1. Zij $\mathcal{T} : \mathit{input} \rightarrow \{\langle$Voornaam : string, Achternaam : string, Cursn : $\{\langle$Naam : string, Moeilijkheidsgraad : int$\rangle\}\rangle\}$, dan is e welgedefiniëerd onder $\mathcal{T}$.

[1] De regels worden gelezen als volgt: boven de streep staan de voorwaarden en onder de streep de conclusie indien er voldaan wordt aan de voorwaarden.

$$\frac{}{\mathcal{T} \vdash \mathsf{a_b} : \mathbf{b}} \qquad \frac{}{\mathcal{T} \vdash x : \mathcal{T}(x)} \qquad \frac{\mathcal{T} \vdash e : \tau \text{ and } \tau <: \sigma}{\mathcal{T} \vdash e : \sigma} \qquad \frac{}{\mathcal{T} \vdash \emptyset : \{\, \bot \,\}}$$

$$\frac{\mathcal{T} \vdash e : \tau}{\mathcal{T} \vdash \{\, e \,\} : \{\, \tau \,\}} \qquad \frac{\mathcal{T} \vdash e_1 : \{\, \tau_1 \,\} \text{ and } \mathcal{T} \vdash e_2 : \{\, \tau_2 \,\} \text{ and } \tau_1 \vee \tau_2 \text{ exists}}{\mathcal{T} \vdash e_1 \cup e_2 : \{\, \tau_1 \vee \tau_2 \,\}}$$

$$\frac{\mathcal{T} \vdash e : \{\, \{\, \tau \,\} \,\}}{\mathcal{T} \vdash \bigcup e : \{\, \tau \,\}} \qquad \frac{\mathcal{T} \vdash e_i : \tau_i \text{ for } i = 1, \ldots, n}{\mathcal{T} \vdash \langle\, l_1 : e_1, \ldots, l_n : e_n \,\rangle : \langle\, l_1 : \tau_1, \ldots, l_n : \tau_n \,\rangle}$$

$$\frac{\mathcal{T} \vdash e : \langle\, \ldots, l : \tau, \ldots \,\rangle}{\mathcal{T} \vdash e.l : \tau} \qquad \frac{\mathcal{T} \vdash e_1 : \{\, \tau_1 \,\} \text{ and } \mathcal{T}[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\mathcal{T} \vdash \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 : \{\, \tau_2 \,\}}$$

$$\frac{\mathcal{T} \vdash e_1 : \tau_1 \text{ and } \mathcal{T}[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\mathcal{T} \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\mathcal{T} \vdash e_1 : \tau \text{ and } \mathcal{T} \vdash e_2 : \tau}{\mathcal{T} \vdash e_1 = e_2 : \mathbf{Boolean}}$$

$$\frac{\mathcal{T} \vdash e : \{\, \tau \,\}}{\mathcal{T} \vdash e = \emptyset : \mathbf{Boolean}}$$

$$\frac{\mathcal{T} \vdash e_1 : \mathbf{Boolean} \text{ and } \mathcal{T} \vdash e_2 : \tau_1 \text{ and } \mathcal{T} \vdash e_3 : \tau_2 \text{ and } \tau_1 \vee \tau_2 \text{ exists}}{\mathcal{T} \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau_1 \vee \tau_2}$$

$$\frac{\mathcal{T} \vdash e : \tau \text{ and } f \in \mathcal{E} \text{ with } f : \tau \to \sigma}{\mathcal{T} \vdash f(e) : \sigma}$$

Figuur 5.1: Welgetypeerdheid van NRC expressions

$$\frac{\mathcal{T} \vdash e_1 : \{\, \{\, \tau_1 \,\} \,\}}{\mathcal{T} \vdash \bigcup e : \{\, \tau_1 \,\}}$$

Om het type τ_1 en dus ook het resulterende type te bepalen, moeten we e_1 evalueren onder $\mathcal{T}$.

$$\frac{}{\mathcal{T} \vdash x : \mathcal{T}(x)} \qquad \frac{\mathcal{T} \vdash e_2 : \{\, \tau_2 \,\} \text{ and } \mathcal{T}[x \mapsto \tau_2] \vdash e_3 : \tau_3}{\mathcal{T} \vdash \mathsf{for}\ x\ \mathsf{in}\ e_2\ \mathsf{return}\ e_3 : \{\, \tau_3 \,\}}$$

De expressie e_2 is de var *input* en $\mathcal{T}(input) = \{\langle$Voornaam, Achternaam , Cursn : $\{\langle$Naam, Moeilijkheidsgraad$\rangle\}\rangle\}$, hieruit volgt $\tau_2 = \langle$Voornaam, Achternaam , Cursn : $\{\langle$Naam, Moeilijkheidsgraad$\rangle\}\rangle$. We evalueren e_3 nu onder $\mathcal{T}$, waarbij $\mathcal{T}(x) = \tau_2$.

$$\frac{\mathcal{T} \vdash e_4 : \mathbf{Boolean} \text{ and } \mathcal{T} \vdash e_5 : \tau_4 \text{ and } \mathcal{T} \vdash e_6 : \tau_5 \text{ and } \tau_4 \vee \tau_5 \text{ exists}}{\mathcal{T} \vdash \mathsf{if}\ e_4\ \mathsf{then}\ e_5\ \mathsf{else}\ e_6 : \tau_4 \vee \tau_5}$$

$$\frac{\mathcal{T} \vdash e_7 : \tau_6 \text{ and } \mathcal{T} \vdash e_8 : \tau_7}{\mathcal{T} \vdash e_7 = e_8 : \mathbf{Boolean}} \qquad \frac{\mathcal{T} \vdash e : \langle\, \ldots, l : \tau_8, \ldots \,\rangle}{\mathcal{T} \vdash e.l : \tau_8} \qquad \frac{}{\mathcal{T} \vdash \mathsf{a_b} : \mathbf{b}}$$

De expressie e_4 zou van het type boolean moeten zijn, dit klopt want e_4 is een conditie met output type boolean, want $\tau_6 = \tau_7$, omdat $x.voornaam$ van het type string is, x is nl. van het type τ_1, en 'Sylvie' is een string constante. De expressie e_6 is van het type $\bot$, om het type van de expressie e_5 te weten moeten we e_5 evalueren onder $\mathcal{T}$. De expressie e_5 is opnieuw een if-expressie, voor de conditie hiervan geldt dezelfde procedure als net beschreven, omdat $x.achternaam$ en 'Hoessels' wederom hetzelfde type hebben, nl. string. De evaluatie van de then-expressie onder $\mathcal{T}$ geeft als resulterend type $\{string\}$, dit kan je zelf nagaan aan de hand van de definitie. De else-expressie is van het type $\bot$. $\{string\} \vee \bot = \{string\}$. Het resulterende type van de if-expressie is dus $\{string\}$. Dan is $\tau_4 \vee \tau_5 = \tau_3$, dus $\{string\} \vee \bot = \{string\}$. Dus het resulterende type van de for-expressie e_1 is $\{\{string\}\}$. Het resulterende type van de hele expressie is dan $\{string\}$.

We mogen nu concluderen dat de expressie welgetypeerd is en het mogelijk resulterende eindtype is $\{string\}$.

5.6 Resulterende waarde

Definitie 5.5 Zij $\mathcal{T}$ een type toekenning en e een NRC expressie. We definiëren een *waarde toekenning* voor $\mathcal{T}$ als een afbeelding Γ op de verzameling van de vrije variabelen $FV(e)$ van e die aan elke variabele $x \in FV(e)$ een complexe waarde van type $\mathcal{T}(x)$ toekent. Zij e een welgetypeerde NRC expressie onder $\mathcal{T}$ en Γ een waarde toekenning op $FV(e)$. Dan kunnen we de *resulterende eindwaarde* v van expressie e specifiëren onder Γ. We noteren dit als volgt $\Gamma \models e \Rightarrow v$. De definitie is gegeven door de regels in Figuur 5.2[2]. De regels voor de for-expressie en de let-expressie gebruiken de volgende notatie. Zij Γ een waarde toekenning, $x \in \mathcal{X}$ en v een complexe waarde. Dan stelt $\Gamma[x \mapsto v]$ de waarde toekenning voor die gelijk is aan Γ, behalve voor de variabele x die de waarde v krijgt toegekend.

We nemen als type toekening, $\mathcal{T}$ uit vorige sectie, en als waarde toekenning vA, de verzameling gegeven in Figuur 3.1. Dan krijgen we als resulterende waarde de verzameling $\{'Economie', 'Logica'\}$.

5.7 DTD voor expressies

We hebben een DTD opgesteld voor expressies om een expressie e op een ontleedde wijze voor te stellen. De parser krijgt de NRC expressie e binnen en zet deze om in een *XML abstract syntax tree*. In Figuur 5.4 is een dergelijke vereenvoudigde XML abstract syntax tree gegeven voor de binnenste for-loop in de expressie uit Listing 5.1. Nu we deze boom als resultaat van de parser hebben, is onze expressie e correct ontleed, want het XML-formaat voldeed aan de opgelegde structuur door de DTD, die gegeven is in Figuur 5.3.

[2]De regels worden gelezen als volgt: boven de streep staan de voorwaarden en onder de streep de conclusie indien er voldaan wordt aan de voorwaarden.

$$\frac{}{\Gamma \models \mathtt{a_b} \Rightarrow \mathtt{a_b}} \qquad \frac{}{\Gamma \models x \Rightarrow \Gamma(x)} \qquad \frac{}{\Gamma \models \emptyset \Rightarrow \emptyset} \qquad \frac{\Gamma \models e \Rightarrow v}{\Gamma \models \{\, e \,\} \Rightarrow \{\, v \,\}}$$

$$\frac{\Gamma \models e_1 \Rightarrow v_1 \text{ and } \Gamma \models e_2 \Rightarrow v_2}{\Gamma \models e_1 \cup e_2 \Rightarrow v_1 \cup v_2} \qquad \frac{\Gamma \models e \Rightarrow \{\, v_1, \ldots, v_n \,\}}{\Gamma \models \bigcup e \Rightarrow v_1 \cup \cdots \cup v_n}$$

$$\frac{\Gamma \models e_i \Rightarrow v_i \text{ for } i = 1, \ldots, n}{\Gamma \models \langle\, l_1 : e_1, \ldots, l_n : e_n \,\rangle \Rightarrow \langle\, l_1 : v_1, \ldots, l_n : v_n \,\rangle} \qquad \frac{\Gamma \models e \Rightarrow \langle \ldots, l : v, \ldots \rangle}{\Gamma \models e.l \Rightarrow v}$$

$$\frac{\Gamma \models e_1 \Rightarrow \{\, v_1, \ldots, v_n \,\} \text{ and } \forall i = 1, \ldots, n : \Gamma[x \mapsto v_i] \models e_2 \Rightarrow w_i}{\Gamma \models \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \Rightarrow \{\, w_1, \ldots, w_n \,\}}$$

$$\frac{\Gamma \models e_1 \Rightarrow v_1 \text{ and } \Gamma[x \mapsto v_1] \models e_2 \Rightarrow v_2}{\Gamma \models \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Rightarrow v_2}$$

$$\frac{\Gamma \models e_1 \Rightarrow v_1 \text{ and } \Gamma \models e_2 \Rightarrow v_2 \text{ and } v_1 = v_2}{\Gamma \models e_1 = e_2 \Rightarrow \mathtt{true}}$$

$$\frac{\Gamma \models e_1 \Rightarrow v_1 \text{ and } \Gamma \models e_2 \Rightarrow v_2 \text{ and } v_1 \neq v_2}{\Gamma \models e_1 = e_2 \Rightarrow \mathtt{false}} \qquad \frac{\Gamma \models e \Rightarrow v \text{ and } v = \emptyset}{\Gamma \models e = \emptyset \Rightarrow \mathtt{true}}$$

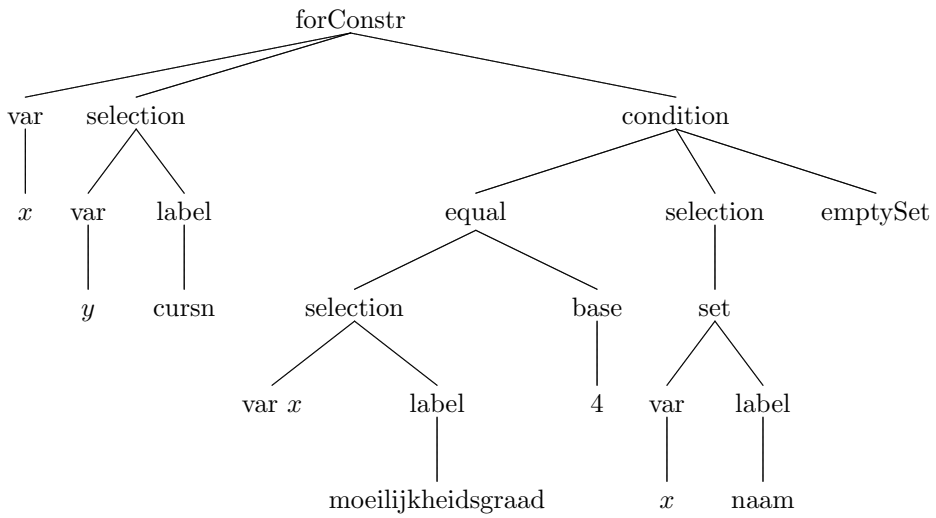
$$\frac{\Gamma \models e \Rightarrow v \text{ and } v \neq \emptyset}{\Gamma \models e = \emptyset \Rightarrow \mathtt{false}} \qquad \frac{\Gamma \models e_1 \Rightarrow v_1 \text{ and } \Gamma \models e_2 \Rightarrow v_2 \text{ and } v_1 = \mathtt{true}}{\Gamma \models \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow v_2}$$

$$\frac{\Gamma \models e_1 \Rightarrow v_1 \text{ and } \Gamma \models e_3 \Rightarrow v_3 \text{ and } v_1 = \mathtt{false}}{\Gamma \models \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Rightarrow v_3} \qquad \frac{\Gamma \models e \Rightarrow v \text{ and } f \in \mathcal{E}}{\Gamma \models f(e) \Rightarrow [\![f]\!](v)}$$

Figuur 5.2: Evaluatie van NRC expressies

```
expressie       → baseVal | var | emptySet | set | union
    | bigUnion | tuple | selection | forConstr | equal |
    condition | letConstr | funcCall
baseVal         → #PCDATA
var             → #PCDATA
emptySet        → EMPTY
set             → lbrace expressie rbrace
lbrace          → EMPTY
rbrace          → EMPTY
union           → expressie cup expressie
cup             → EMPTY
bigUnion        → bigCup EMPTY
bigCup          → EMPTY
tuple           → langle (label sep expressie)+ rangle
langle          → EMPTY
rangle          → EMPTY
label           → #PCDATA
sep             → EMPTY
selection       → expressie dot label
dot             → EMPTY
forConstr       → for var in expressie return expressie
for             → EMPTY
in              → EMPTY
return          → EMPTY
equal           → expressie equals (expressie | emptySet
   )
equals          → EMPTY
condition       → if expressie then expressie else
   expressie
if              → EMPTY
then            → EMPTY
else            → EMPTY
letConstr       → let var equals expressie in expressie
let             → EMPTY
funcCall        → function lbracket expressie rbracket
function        → #PCDATA
lbracket        → EMPTY
rbracket        → EMPTY
```

Figuur 5.3: DTD voor expressies.



Figuur 5.4: XML abstract syntax tree.

Hoofdstuk 6

SQL/PSM

PSM (Persistent Stored Modules)[1] is een krachtige database-geöriënteerde programmeertaal, die SQL[2] uitbreidt met procedurale eigenschappen. PSM voegt selectieve, bv. if..then..else, en iteratieve, bv. for-loops, structuren toe aan SQL. PSM is gebaseerd op de ISO[2] standaarden.

De *stored procedures* worden in gecompileerde vorm bewaard in de database en resulteren daardoor in een snellere uitvoering dan de uitvoering van SQL code vanuit een applicatie. Een ander voordeel is dat stored procedures herbruikt kunnen worden of door meerdere applicaties gebruikt kunnen worden, waardoor tijd wordt bespaard. Hierdoor moeten applicaties niet heruitgevonden, herschreven of 'gedebugged' worden. De code in een applicatie wordt ook veel overzichtelijker. Onderhoudskosten dalen, omdat als een stored procedure voor één applicatie verbeterd wordt, is deze eveneens verbeterd voor alle applicaties die gebruik maken van deze stored procedure.

De vertaling van een NRC expressie e is een PSM programma p, de wijze waarop wordt toegelicht in Hoofdstuk 9. Het PSM programma p ,dat de vertaling van de hoofdexpressie e voorstelt, wordt losgelaten op de relationele database en simuleert de uitvoering van de NRC expressie e op de complex-object database.

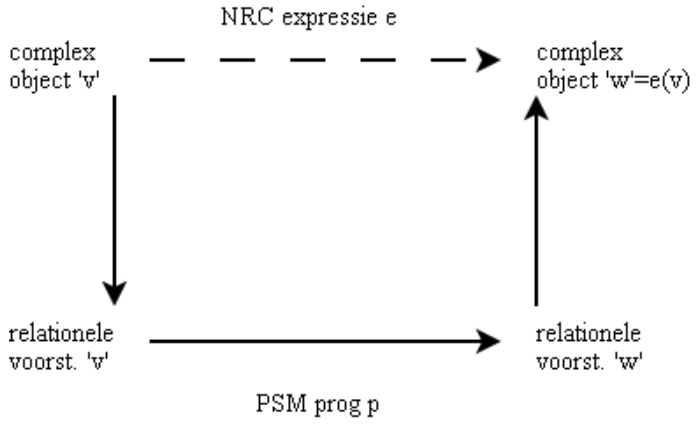
[1] PSM wordt ook ondersteund door DB2[7]
[2] International Organization for Standardization

Hoofdstuk 7

Het evalueren van NRC expressies

Het ondervragen van de complex-object database doen we aan de hand van NRC expressies. Iedere NRC expressie e wordt omgezet in een PSM programma p, dat uitgevoerd wordt onder een bepaalde input waarde toekenning. Uiteindelijk zal de run van het PSM programma p het resultaat geven van de evaluatie van de NRC expressie e over de bepaalde input waarde toekenning. Dit principe wordt weergegeven in Figuur 7.1.

De expressie e wordt geëvalueerd onder een *input waarde toekenning*, die de input afbeeldt op v. Het object v heeft een relationele voorstelling in het systeem, zoals in Hoofdstuk 4. De expressie e zelf, wordt vertaald naar een PSM programma p, volgens de principes in Hoofdstuk 9. Het object v zit nu onder een relationele voorstelling in de database en we hebben het vertaalde PSM programma p. Het PSM programma p wordt uitgevoerd op het object v. De evaluatie hiervan geeft als resultaat een object w, dat relationeel voorgesteld is in de database, $w = e(v)$.



Figuur 7.1: Schema evaluatie NRC expressies.

Het opvolgen van de run van de NRC uitdrukking over de input gebeurt door drie tabellen. We bespreken nu deze tabellen.

7.1 Expressies

De tabel voor de expressies, *Expressions*, wordt beschreven in Figuur 7.2:

Expressions	E_Id	Parent_Id	Expr_String
	int	int	xmlvarchar

Figuur 7.2: Voorstelling tabel Expressions.

Elke expressie wordt bewaard in deze tabel. Tijdens de uitvoering van het systeem wordt het id van zijn XML representatie bijgehouden in een variabele, we weten dus op elk moment in de run met welke expressie we bezig zijn. De expressie zelf wordt omgezet in een XML-formaat, zie Hoofdstuk 5. Een expressie heeft ook een parent-id zodat we weten van welke expressie de expressie een deelexpressie is. Indien de expressie de hoofdexpressie is zal deze als parent-id het NULL-element bevatten.

7.2 Waarde toekenning

De toekenningstabel, *V_Assignments*, is van de structuur gegeven in Figuur 7.3:

V_Assignment	Ass_Id	Parent_Id	Var	Val
	int	int	string	int

Figuur 7.3: Voorstelling tabel V_Assignment.

Elke waarde toekenning aan een variabele wordt in deze tabel bewaard. Een waarde toekenning is een toekenning van een id (Val), zoals in Hoofdstuk 4 aan een variabele. We starten met de toekenning van waarden, dit is wederom een id van een tuple, verzameling of basiswaarde, aan de input variabelen. Zij τ deze toekenning, voor elke nieuwe toekenning τ_1 zal een nieuw record met uniek id, in deze tabel, aangemaakt worden, waarbij τ de parent-id is. De toekenning τ_1 is een uitbreiding van de toekenning t. Voor elke uitbreiding van een toekenning vullen we als parent-id de huidige toekenning τ_i, toekenning van expressie die we aan het evalueren zijn, in, m.a.w. elk Ass_id is een uitbreiding van parent-id. Een variabele die tot de input behoort, heeft als parent-id het NULL-element. Elke expressie e_i heeft een toekenning τ_i op e_i. Om een waarde van een variabele x in expressie e_i op te zoeken, zoeken we de variabele x op bij de toekenning τ_i, indien de waarde niet gevonden wordt, wordt er verder gezocht via de parent-id's tot we een waarde voor de variabele x in expressie e_i vinden. We vinden altijd een waarde, want anders is de expressie e niet welgetypeerd.

7.3 Resultaten evaluatie

Om de expressies te evalueren over de waarde toekenningen is er de tabel *Eval*. De voorstelling van deze tabel wordt gegeven in Figuur 7.4.

Een expressie wordt geëvalueerd over een waarde toekenning. Deze evaluatie levert een waarde als resultaat, het id, zoals in Hoofdstuk 4, van deze waarde wordt bewaard in het veldje Val. De waarde die bij het id hoort van het record in de tabel met als expressie de hoofdexpressie, is het resultaat van de NRC uitdrukking e onder

Eval	E_Id	Ass_Id	Val
	int	int	int

Figuur 7.4: Voorstelling tabel Eval.

de waarde toekenning $\mathcal{T}$. Merk op dat alle tussenresultaten van de deelexpressies ook bewaard worden in de database.

Hoofdstuk 8

Inlezen van complexe objecten in de database

In Hoofdstuk 7 hebben we gemerkt dat de evaluatie van een NRC expressie ook mogelijk nieuwe complexe objecten introduceert. Doordat we aan de hand van NRC expressie dus nieuwe complexe objecten kunnen introduceren in de complex-object database, hebben we geen nood aan een volledig nieuwe syntax om complexe objecten in te lezen. We kunnen dus de syntax gebruiken, die gegeven is in definitie 5.2. We worden wel gelimiteerd door de singleton constructie, $\{\, e \,\}$. Als we deze constructie uitbreiden tot een verzameling constructie $\{\, e_1, ..., e_n \,\}$, kunnen we een grote verzameling, van basiswaarden, tuples of verzamelingen, ineens aanmaken in de complex-object database, zonder gebruik te moeten maken van '$\cup$'. Deze uitbreiding van de syntax vergemakkelijkt het aanmaken van complexe objecten in de complex-object database dus behoorlijk. De uitgebreide syntax wordt gegeven in Definitie 8.1.

8.1 Uitgebreide syntax

Definitie 8.1 Stel een onaftelbare verzameling $\mathcal{X}$ van variabelen. Een *Nested Relational Calculus expressie* e is gedefiniëerd door volgende grammatica, met $\mathsf{a}_\mathbf{b} \in [\![\, \mathbf{b} \,]\!]$, $x \in \mathcal{X}$, $f \in \mathcal{E}$ en $l_1, \ldots, l_n \in \mathcal{L}$.

$$
\begin{array}{rcl}
e & ::= & \mathsf{a}_\mathbf{b} \mid x \\
& \mid & \emptyset \mid \{\, e_1, ..., e_n \,\} \mid e \cup e \mid \bigcup e \\
& \mid & \langle\, l_1 : e, \ldots, l_n : e \,\rangle \mid e.l \\
& \mid & \mathsf{for}\ x\ \mathsf{in}\ e\ \mathsf{return}\ e \\
& \mid & e = e \mid e = \emptyset \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& \mid & f(e)
\end{array}
$$

In een tuple expressie $\langle\, l_1 : e_1, \ldots, l_n : e_n \,\rangle$ zijn alle labels l_i verschillend.

$$FV(\{\, e_1, ..., e_n \,\}) = \bigcup_{i=1}^{n} FV(e_i)$$

$$\frac{\mathcal{T} \vdash e_1 : \tau \ ... \ \mathcal{T} \vdash e_n : \tau}{\mathcal{T} \vdash \{\, e_1, ..., e_n \,\} : \{\, \tau \,\}}$$

$$\frac{\Gamma \models e_1 \Rightarrow v_1 \ ... \ \Gamma \models e_n \Rightarrow v_n}{\Gamma \models \{\, e_1, ..., e_n \,\} \Rightarrow \{\, v_1, ..., v_n \,\}}$$

Figuur 8.1: Aangepaste regels voor uitgebreide syntax.

Let op: De regels voor singleton constructie in definities 5.3, 5.4 en 5.5 moeten dan ook aangepast worden. Deze aanpassingen zijn weergegeven in Figuur 8.1.

Hoofdstuk 9

Vertaling van NRC naar SQL

Voor elk soort expressie dat kan voorkomen volgens de syntax wordt er een PSM programma geproduceerd. Bij het parsen van de NRC expressie, wordt er een *XML abstract syntax boom* opgebouwd, zie Hoofdstuk 5. Om het PSM programma te bekomen, starten we bij de hoofdexpressie en doorlopen de boom depth first. Elke knoop, soort expressie, heeft een E_Id, een verwijzing naar de expressie in de tabel Expressions, en een Assign Id, een verwijzing naar een waarde toekenning τ in de tabel V_Assignments. Indien het soort expressie uit een aantal deelexpressies bestaat, worden deze ook bijgehouden in een variabele subE_Id. De functie, Eval(int assign_Id), construeert het PSM programma, deze heeft als parameter de waarde toekenning τ van de expressie en wordt recursief opgeroepen op de deelexpressie(s).

9.1 Sequentie objecten

Om unieke id's te genereren maken we gebruik van *sequence objects* [14]. Dit is een gescheiden structuur in DB2 die sequentiële nummers genereert. Een sequentie object wordt gecreëerd door het 'CREATE SEQUENCE' statement.

Wanneer een sequentie object is aangemaakt, kan het gebruikt worden door meerdere applicaties. Sequentie objecten zijn ideaal om sequentiële, unieke, numerieke waarden te creëren. Omdat dit een gescheiden structuur is, wacht DB2 niet op een transactie om te 'committen' alvorens toe te staan het sequentie object te incrementeren.

In het programma worden drie sequentie objecten gebruikt.

```
CREATE SEQUENCE OBJECTS
AS INTEGER
START WITH 1
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
NO CYCLE
NO CACHE
ORDER
```

Dit sequentiëel object wordt gebruikt om unieke id's te genereren voor de objecten in de database. Op deze manier zijn we zeker dat we altijd een uniek id hebben voor elk object. We beginnen vanaf waarde 1, omdat 0 het id is voor de lege verzameling.

```
INSERT INTO Pool_Int VALUES (NEXTVAL FOR OBJECTS, 1);
```

Het statement 'NEXTVAL FOR OBJECTS' wordt erkend als een sequentie expressie, DB2 zal het genoemde sequentie object gebruiken om een volgende, unieke waarde te genereren. Het statement 'PREVVAL FOR OBJECTS' geeft de laatst gegenereerde waarde voor het genoemde sequentie object terug.

```
UPDATE Pool_Int VALUES SET Val = 2 WHERE Id = PREVVAL
    FOR OBJECTS;
```

We hebben nog een object voor waarde toekenning in de tabel V_Assignments en voor expressies in de tabel Expression.

```
CREATE SEQUENCE ASSIGNMENTS
AS INTEGER
START WITH 1
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
NO CYCLE
NO CACHE
ORDER

CREATE SEQUENCE EXPRESSIONS
AS INTEGER
START WITH 1
INCREMENT BY 1
NO MINVALUE
NO MAXVALUE
NO CYCLE
NO CACHE
ORDER
```

Een sequentie object verwijderen doen we op de volgende manier.

```
DROP SEQUENCE OBJECTS RESTRICT
```

De parameter RESTRICT is niet optioneel, deze dient om te voorkomen dat het object verwijderd wordt, als er een tabel naar refereert.

Opmerking: Als je 'NEXTVAL FOR' meer dan één keer gebruikt in hetzelfde SQL statement, dan geeft DB2 dezelfde waarde terug voor elke 'NEXTVAL FOR' in dat statement.

9.2 Vertaling

9.2.1 $e ::= a_b$

Op dit niveau is geweten van welk basistype, b, de expressie is. Dus kunnen we in de juiste pooltabel zien welk Id hoort bij de waarde a_b. Indien de waarde nog niet aanwezig is in de tabel, wordt deze eerst toegevoegd. We creëren een uniek id, waarmee de constante waarde wordt opgeslaan in de juiste tabel. Om unieke id's te genereren maken we gebruik van het sequentie object 'objects'. Het PSM programma dat deze expressie zal opleveren, is:

INPUT:

- int v_E_Id : Id van de expressie.
- int v_Ass_Id : de waarde toekenning waaronder de expressie geëvalueerd wordt.

- b a_b : b is het basistype van de expressie.

```
SET Id = SELECT Id FROM Pool_b WHERE Val = a_b;
IF (Id IS NULL) THEN
        SET Id = NEXTVAL FOR OBJECTS;
        INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, ('
            SELECT Id FROM Pool_b WHERE Val = a_b '));
END IF;
```

met b in pool_**b** het basis type b en a_b de basiswaarde.

9.2.2 $e ::= x$

Elke expressie e heeft een Ass_id waaronder het geëvalueerd wordt. Nu wordt dus gezocht of de variabele x een waarde heeft voor deze toekenning, indien niet, zoeken we verder in de uitbreidingen van deze waarde toekenning. Dit principe wordt uitgelegd in Hoofdstuk 7.

INPUT:

- int v_E_Id : Id van de expressie.
- int v_Ass_Id : de waarde toekenning waaronder de expressie geëvalueerd wordt.
- string x : x is de naam van de variabele.

```
DECLARE v_result INTEGER;
DECLARE v_parent INTEGER;

SET v_parent = v_Ass_Id;
WHILE (v_result IS NULL AND NOT(v_parent IS NULL)) DO
        SET v_result = (SELECT Val FROM V_Assignments
            WHERE Ass_Id = v_parent AND Var = x);
        SET v_parent = (SELECT Parent_Id FROM
            V_Assignments WHERE Ass_Id = v_parent);
END WHILE;

IF NOT(v_result IS NULL) THEN
        INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id,
            v_result);
END IF;
```

9.2.3 $e ::= \emptyset$

De lege verzameling wordt niet opgeslaan in de database. Het id van de lege verzameling is 0. Indien dus een expressie de lege verzameling is, zal in de Eval tabel de waarde 0 bewaard worden voor deze expressie onder zijn waarde toekenning.

INPUT:

- int v_E_Id : Id van de expressie.
- int v_Ass_Id : de waarde toekenning waaronder de expressie geëvalueerd wordt.

```
INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, 0);
```

9.2.4 $e ::= \{e\}$

Het id van de waarde van de deelexpressie wordt uit de eval tabel opgehaald, met als id voor de waarde toekenning v_Ass_Id. Dit Id wordt dan een element van de verzameling.

INPUT:

- int v_E_Id : Id van de expressie.
- int v_Ass_Id : de waarde toekenning waaronder de expressie geëvalueerd wordt.
- int v_SubE_Id : dit is het id van de deelexpressie.

```
DECLARE v_newId INTEGER;

SET v_newId = NEXTVAL FOR OBJECTS;
INSERT INTO Set VALUES (v_newId, (SELECT Val FROM Eval
    WHERE E_id = v_subE_Id));

INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, v_newId);
```

Indien we de uitgebreide syntax gebruiken, zodat we verzamelingen van meerdere elementen ineens kunnen voorstellen, moeten we de regel waar we waarden in de verzameltabel steken, meermaals oproepen, nl. voor elke deelexpressie e_n.

9.2.5 $e ::= e_1 \cup e_2$

De deelexpressies e_1 en e_2 zijn van het type verzameling. In het systeem wordt eerst gecontroleerd of de join $\tau_1 \vee \tau_2$ bestaat. Indien deze join bestaat en verschillend is van τ_1 en τ_2, worden de complexe waarden aangepast zodat ze voldoen aan dit type. Nu voegen we de bekomen waarden van deelexpressie e_1 en deelexpressie e_2 samen in een nieuwe verzameling met een nieuw, uniek Id. De duplicaten worden er ook uitgefilterd.

INPUT:

- int v_E_Id : Id van de expressie.
- int v_Ass_Id : de waarde toekenning waaronder de expressie geëvalueerd wordt.
- int v_SubE1_Id : dit is het id van de deelexpressie e_1.
- int v_SubE2_Id : dit is het id van de deelexpressie e_2.

```
DECLARE v_newId INTEGER;
DECLARE newEl INTEGER;

SET v_newId = NEXTVAL FOR OBJECTS;
FOR Element AS SELECT El FROM Set WHERE Id IN (SELECT
    Val FROM Eval WHERE E_Id = v_subE1_Id OR E_Id =
    v_subE2_Id)
        DO
                SET newEl = ADJUSTJOIN(El);
                IF NOTCONTAINED(newEl, v_newId) = 1 THEN
```

```
                    INSERT INTO Set VALUES (v_newId,
                        El);
                END IF;
END FOR;

INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, v_newId);
```

De functie 'ADJUSTJOIN' slaat de overeenkomstige waarden met het join type in de database op. Verder in deze sectie wordt de constructie van de functie NOT-CONTAINED verklaard.

9.2.6 $e ::= \bigcup e$

De deelexpressie is van het type verzameling, waarbij het type van de elementen van deze verzameling eveneens van het type verzameling zijn. De elementen van deze laatste verzamelingen gaan we dan samen voegen tot één verzameling met een nieuw, uniek Id. De duplicaten worden hier eveneens uitgefilterd.

```
DECLARE v_newId INTEGER;

SET v_newId = NEXTVAL FOR OBJECTS;
FOR Element AS SELECT s2.El FROM Set s1 JOIN Set s2 ON
    s1.El = s2.Id WHERE s1.id IN (SELECT Val FROM Eval
    WHERE E_Id = v_subE_Id);
        DO
                IF NOTCONTAINED(El, v_newId) = 1 THEN
                        INSERT INTO Set VALUES (v_newId,
                            El);
                END IF;
END FOR;

INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, v_newId);
```

9.2.7 $e ::= \langle l_1 : e_1, ..., l_n : e_n \rangle$

Alle labels en het type van alle deelexpressies zijn gekend dus kunnen we het type bepalen van het tuple. We itereren door velden en construeren voor ieder label een tabel 'Att_'+l_i, indien deze nog niet bestaat. Of een tabel al bestaat, kunnen we aan *SYSCAT* [13] vragen. Het Id van de waarde die overeenkomt met de expressie e_n in de Eval tabel wordt gekoppeld aan het tuple in de juiste attribuuttabel. Volgende SQL statements gaan we dan ook herhaaldelijk moeten uitvoeren, één keer voor ieder veld van het tuple.
Opmerking: we kunnen de tabelnaam niet variabel meegeven aan een procedure, maar in het systeem weten we in welke tabel we moeten zijn, dus kunnen we de query correct samenstellen. Hier maken we de tabelnaam even variabel, om het principe duidelijk te maken. Deze variabele noemen we 'tablename'. Bij het itereren door de velden van het tuple, krijgen we niet alleen de labels l_i, maar ook de Id's van de deelexpressies. Deze zijn eveneens variabel en worden voorgesteld door subE_Id.

Bij de test voor definer wordt de string 's0118713' gebruikt, dit omdat dit de login is in de database die gebruikt wordt.

```
IF ((SELECT * FROM SYSCAT.TABLES WHERE DEFINER='s0118713
    ' AND TYPE='T' AND TABNAME=tablename) IS NULL) THEN
        CREATE TABLE tablename(Id int, Val int);
END IF;
```

```
SET Id = NEXTVAL FOR OBJECTS;
INSERT INTO tablename VALUES(Id, (SELECT Val FROM Eval
    WHERE E_Id = subE_Id AND Ass_Id = v_Ass_Id));
INSERT INTO Eval VALUES(v_E_Id, v_Ass_Id, Id);
```

9.2.8 $e ::= e.l$

De deelexpressie is van het type tuple, we zoeken dus het Id van de waarde van het veld dat bij label l hoort.

```
INSERT INTO Eval VALUES(v_E_Id, v_Ass_Id, (SELECT Val
    FROM Att_l WHERE Id = IN (SELECT Id FROM Eval WHERE
    E_Id = subE_Id AND Ass_Id)));
```

met Att_l de attribuut die bij label l hoort.

9.2.9 $e ::=$ **for** x **in** e_1 **return** e_2

Het type van e_1 en dus ook van x is gekend. Ook het output type van e_2 is gekend. Voor elk element van de verzameling, bekomen door evaluatie van expressie e_1, wordt er een nieuw record toegevoegd met als Parent_Id het Assign_Id van de for-expressie en als Var x aan de assignment tabel. Voor deze records wordt een nieuw, uniek assignment id gegenereert. Het sequentie object voor assignments is 'Assignments'. Expressie e_2 wordt voor iedere aangemaakte waarde toekenning uitgevoerd en het resultaat wordt bewaard in de Eval tabel.

deelexpressie e_1 is geëvalueerd en waarden zijn dus bekend in de Eval tabel. Dus nu waarde toekeningen aanmaken voor variabele x.

```
DECLARE v_newAssId INTEGER;

FOR Element AS SELECT El FROM Set WHERE Id IN (SELECT
    Val FROM Eval WHERE E_Id = v_subE1_Id)
        DO
                SET v_newAssId = NEXTVAL FOR ASSIGNMENTS
                    ;
                INSERT INTO V_Assignments VALUES (
                    v_newAssId, v_Ass_Id, v_Var, El);
END FOR;
```

Nu wordt de deelexpressie e_2 geëvalueerd voor iedere aangemaakte waarde toekening. We moeten deze resultaten nog bewaren. Het resultaat van een for-loop is van het type Set, dus de resultaten worden bewaard in de tabel Set.

```
DECLARE v_newId INTEGER;

v_newId = NEXTVAL FOR OBJECTS;

FOR Element AS SELECT Val FROM Eval WHERE Ass_Id IN (
    SELECT Ass_Id FROM V_Assignments WHERE Var = v_Var
    AND Parent_Id = v_Ass_Id) AND E_Id = v_subE2_Id
        DO
                INSERT INTO Set VALUES (v_newId, Val);
        END FOR;

INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, v_newId);
```

9.2.10 $e ::= e_1 = e_2$

De types van beide expressies zijn gekend. Als eerste conditie moeten de twee verzamelingen hetzelfde type hebben, anders kunnen ze nooit dezelfde objecten bevatten. Dit wordt op voorhand getest in de java code. Nu moeten we nog controleren of de twee expressies, e_1 en e_2, dezelfde waarde teruggeven. Let op, het wil niet zeggen dat als de Id's niet gelijk zijn, dat de objecten niet hetzelfde kunnen zijn. Dit is enkel het geval bij basiswaarden, omdat een waarde maar één keer kan voorkomen in de database. Bij tuples en verzamelingen worden telkens nieuwe id's gegenereert, dus gelijke objecten kunnen verschillende id's hebben. We moeten de gelijkheid dus recursief gaan controleren. Omdat we tijdens de uitvoering van het PSM programma niet weten met welke types we te maken hebben, gaan we stored procedures moeten schrijven tijdens de uitvoering van ons systeem, in het java programma. Het boolean type wordt niet ondersteund door DB2, deze stored procedures hebben als return type een smallint, waarde 1 indien true, waarde 0 indien false.

Basiswaarden

Hier kunnen we gewoon testen of de Id's gelijk zijn. Dit wordt opgenomen in de stored procedures die gegenereert worden bij tuples en verzamelingen.

Tuples

De SQL query om te testen of de twee attributen van twee tuples gelijk zijn, is als volgt:

```
EXISTS(SELECT * FROM Att_a1 A1, Att_a1 A2 WHERE A1.Id =
    id1 AND A2.Id = id2 AND EQUAL_T(A1.Val, A2.Val) = 1)
```

met a_1 de waarde van het label van de attributen en id1 en id2 de id's die bij dit label horen. De id's id1 en id2 zijn de input variabelen voor de omvattende procedure, die aangemaakt wordt. Functie 'EQUAL_T' is opnieuw een procedure afhankelijk van het type van het attribuut. Aan deze procedure worden dan de id's van de twee waarden, die bij de twee attributen horen, meegegeven.

Voorbeeld:

De gelijkheid wordt getest tussen de volgende tuples, met als type $\langle a : int, b : \{int\}\rangle$.

```
CREATE FUNCTION equal_tuple(Id1 int, Id2 int) RETURNS
    SMALLINT
 BEGIN ATOMIC
        IF EXISTS (SELECT * FROM Att_a a1, Att_a a2
            WHERE a1.Id = Id1 AND a2.Id = Id2 AND a1.val
            = a2.val)
                AND EXISTS (SELECT * FROM Att_b b1,
                    Att_b b2 WHERE b1.Id = Id1 AND b2.Id
                    = Id2 AND EQUAL_SET(b1.val = b2.val)
                    = 1)  THEN
                RETURN 1;
        ELSE
                RETURN 0;
        END IF;
 END
```

met Id1 en Id2 de waarden voor de twee tuples en EQUAL_SET een functie om de gelijkheid van twee verzamelingen van type $\{int\}$ te testen.

Verzamelingen

Voor het verzamelingtype worden er twee procedures aangemaakt. Eén procedure zal zoeken achter een tegenvoorbeeld. Dit wil zeggen, een object dat een element is van de eerste verzameling, maar geen element is van de tweede verzameling. Verzameling met Id1 moet worden vergeleken met verzameling met Id2 en omgekeerd, omdat we anders enkel zouden testen of de één een deelverzameling is van de ander. Deze testen worden opgeroepen in de tweede procedure.

Voorbeeld:

Stel dat de twee verzamelingen van type int zijn, dan kunnen we verder bouwen op het voorbeeld van gelijkheid bij tuples. Eerst geven we de procedure notcontained en vervolgens de procedure EQUAL_SET.

```
CREATE FUNCTION notcontained(Id1 int, Id2 int) RETURNS
   SMALLINT
BEGIN ATOMIC
        IF EXISTS(SELECT * FROM Set s1 WHERE s1.Id = Id1
             AND NOT EXISTS (SELECT * FROM Set s2 WHERE
            s2.Id = Id2 AND s1.El = s2.El)) THEN
                RETURN 1;
        ELSE
                RETURN 0;
        END IF;
END

CREATE FUNCTION equal_set(Id1 int, Id2 int) RETURNS
   SMALLINT
BEGIN ATOMIC
        IF ((not(notcontained(Id1, Id2)) = 1) AND (not(
            notcontained(Id2, Id1)) = 1)) THEN
                RETURN 1;
        ELSE
                RETURN 0;
        END IF;
END
```

Let op, we mogen in de notcontained procedure rechtstreeks testen of s1.El = s2.El, omdat dit id's zijn die refereren naar basiswaarden en elke basiswaarde komt maar één maal voor in de database.

In de procedure die gemaakt werd voor de buitenste test komt er een regel bij die het Id van 'true' of 'false' in de Eval tabel opslaat.

In het systeem moeten we unieke namen kiezen voor de stored procedures, omdat we bv. misschien eerst de gelijkheid van twee verzamelingen van een bepaald type moeten testen en erna nog eens de gelijkheid van twee verzamelingen, maar met een ander type. De procedures zijn gebonden aan het type en zijn dus verschillend.

9.2.11 $e ::= e = \emptyset$

Hier gaan we het Id van de waarde van expressie e uit de Eval tabel opvragen. Indien dit Id 0 is, is de test waar anders is ze niet waar.

```
IF (SELECT El FROM Set WHERE Id IN (SELECT Val FROM Eval
    WHERE E_Id = v_subE_Id AND Ass_Id = v_Ass_Id)) = 0
   THEN
```

```
        INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, (
            SELECT Id FROM Pool_Boolean WHERE Val = 'TRUE
            '));
ELSE
        INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, (
            SELECT Id FROM Pool_Boolean WHERE Val = '
            FALSE'));
END IF;
```

9.2.12 $e ::=$ **if** e_1 **then** e_2 **else** e_3

Indien e_1 waar is, krijgt de expressie de waarde van de geëvalueerde expressie e_2, anders krijgt ze de waarde van de geëvalueerde expressie e_3. Deze waarden worden wel aangepast naar hun join type.

```
IF (SELECT Val FROM Eval WHERE E_Id = v_subE1_Id AND
    Ass_Id = v_Ass_Id) = (SELECT Id FROM Pool_Boolean
    WHERE Val = 'TRUE') THEN
        INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id,
            ADJUSTJOIN(SELECT Val FROM Eval WHERE E_Id =
            v_subE2_Id AND Ass_Id = v_Ass_Id));
ELSE
        INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id,
            ADJUSTJOIN(SELECT Val FROM Eval WHERE E_Id =
            v_subE3_Id AND Ass_Id = v_Ass_Id));
END IF;
```

hier geeft de functie ADJUSTJOIN het id terug van de aangepaste waarde.

9.2.13 $e ::=$ **let** $x = e_1$ **in** e_2

Als eerste zal er een record toegevoegd worden aan de tabel V_Assignment waarbij de variabele x de waarde van de geëvalueerde expressie e_1 toegekend krijgt.

```
SET v_newAssId = NEXTVAL FOR ASSIGNMENTS;
INSERT INTO V_Assignments VALUES (v_New_Ass_Id, v_Ass_Id
    , v_Var, (SELECT Val FROM Eval WHERE E_Id = subE1_Id
    AND Ass_Id = v_Ass_Id));
```

Nu evalueren we expressie e_2 over de nieuwe, uitgebreidde waarde toekenning, die variabele x afbeeldt op de resultaten van expressie $e1$. Het resultaat van deze evaluatie is ook het resultaat van de evaluatie van deze expressie.

```
NSERT INTO Eval VALUES (v_E_Id , v_Ass_Id , (SELECT Val
    FROM Eval WHERE E_Id = v_subE2_Id AND Ass_Id =
    v_New_Ass_Id)) ;
```

9.2.14 $e ::= f(e)$

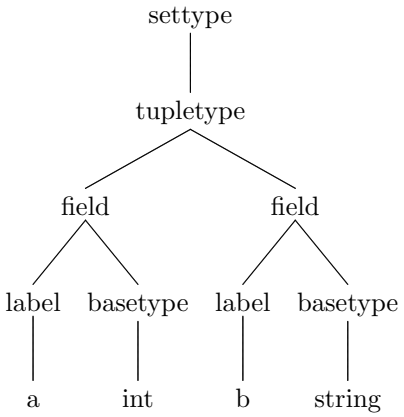
Een functie f is een UDF en krijgt als input een XMLVarChar en geeft als output opnieuw een XMLVarChar. Deze XMLVarChar's hebben we nodig om een object voor te stellen, want de functies kunnen niet om met de interne structuur van de database. Hiervoor hebben we twee functies nodig, één om een object uit de database om te zetten naar een XMLVarChar, 'encapsulate' en één om een XMLVarChar om te zetten naar een object in de database, 'decapsulate'. De functie encapsulate heeft als input een XMLVarChar, met het type van van het object, en

een value-Id. De XMLVarChar voldoet aan de DTD voor types, die gegeven is in Hoofdstuk 2. De value-Id is een id dat verwijst naar de waarde bekomen door de evaluatie van expressie e. De output van encapsulate voldoet aan volgende DTD.

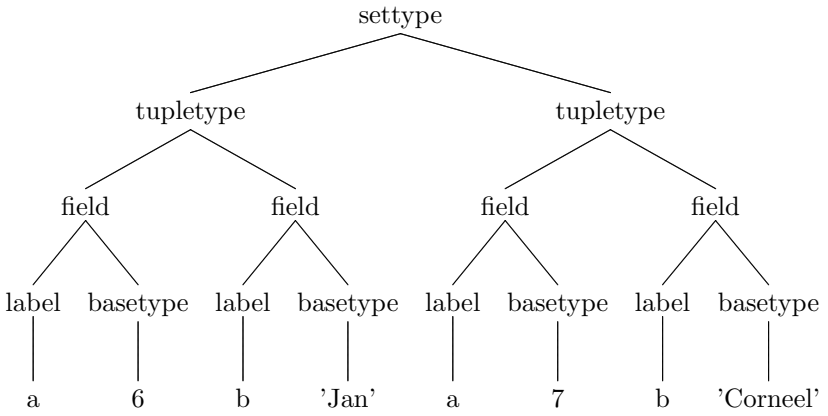
```
val     →     set | tuple | base
set     →     val*
tuple   →     field*
field   →     label val
label   →     #PCDATA
base    →     #PCDATA
```

Voorbeeld:

Stel we geven het id 303, uit Hoofdstuk 3, mee als value-Id aan encapsulate. De XMLVarChar, die als type wordt meegegeven aan encapsulate, wordt getoond in boomstructuur in Figuur 9.1. De output van encapsulate op id 303 is gegeven in Figuur 9.2.



Figuur 9.1: Het complex type dat meegegeven wordt aan encapsulate in boomstructuur.



Figuur 9.2: Resultaat van encapsulate in boomstructuur.

De functie decapsulate heeft als input twee XMLVarChar's. Eén die het type van het object bevat en een ander die het object zelf bevat. De voorstelling van de

output van functie f is een XMLVarChar, deze moet nu omgezet worden naar een object in de database.

Voorbeeld:

Stel dat de functie f de XMLVarChar uit Figuur 9.2 output levert. We weten op elk moment met welk type we bezig zijn, want dit is gegeven, hierdoor weten we welke updates we moeten doorvoeren in de database. Hier starten we met een verzameling, deze bestaat uit tuples. Elk tuple bestaat uit een integer bij label a en een string bij label b. Het eerste tuple heeft voor het attribuut a de waarde 6. De waarde 6 wordt dan opgezocht in de tabel Pool_int, indien deze nog niet aanwezig is, wordt deze toegevoegd. Beiden leveren een id, dit id wordt met een uniek id voor het tuple in de tabel Att_a bewaard. De waarde 'Piet' wordt opgezocht in de tabel Pool_string en ook toegevoegd indien deze niet gevonden wordt. Dit doen we ook voor het tweede tupel. De id's van de twee tuples zijn gekend en uniek. Deze worden dan toegevoegd aan de tabel Set samen met een nieuw, uniek id voor de verzameling. Dit laatste id is dan het id dat bewaard zal worden in de Eval tabel als resultaat van de expressie $f(e)$.

Eerst gaan we de waarde ophalen van de evaluatie van expressie e in de Eval tabel.

```
id = 'SELECT Val FROM Eval WHERE E_Id = subE_Id AND
    Ass_Id = v_Ass_Id ';
```

Op dit ogenblik hebben we een value-Id, maar hier kan de functie f niet mee overweg. Eerst gebeurt de omschakeling van vid naar een XMLVarChar en omgekeerd.

```
XMLvarchar  = encapsulate(type, id);
XMLvarchar2 = f(XMLvarchar);
id2         = decapsulate(outputtype, XMLvarchar2);
```

Als laatste moeten we dan de bekomen waarde, vid2, in de database opslaan in de tabel Eval.

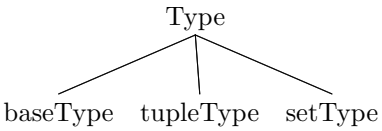
```
INSERT INTO Eval VALUES (v_E_Id, v_Ass_Id, id2);
```

De functies encapsulate en decapsulate zijn UDF's en worden in SQLJ geschreven. Deze functies worden aangeroepen vanuit het PSM bestand, dat gegenereerd werd door de compiler.

Hoofdstuk 10

Software architectuur van de compiler

In dit hoofdstuk som ik de constructors van de type datastructuren en die van de NRC expressies op en geef ik een kort overzicht van deze datastructuren. Dit omdat deze informatie me nuttig lijkt voor een collega-programmeur, die verder zal werken aan dit project.



Type is een interface klasse zonder variabelen. De klasse basetype heeft een String variabele '_base' waarin het soort basistype opgeslaan wordt. Een object van de klasse tupleType heeft een TreeSet van tupleFields '_fields', deze hebben twee variabelen '_label', van het type String en '_type' van het type Type. De tupleFields bevinden zich in een TreeSet, zodat we een orde kunnen leggen op de labels. Hierdoor zal, bij het omzetten van de types naar XMLString een orde liggen op de labels en kunnen we deze XMLStrings makkelijk vergelijken. SetType heeft een variabele '_subType' van het type Type, die het type voorstelt van de elementen van de verzameling.

Hieronder worden de constructors van de vernoemde klassen opgesomd.

- public BaseType(String b)
- public TupleType(TreeSet¡TupleField¿ tf)
  public TupleField(String l, Type f)
- public setType(Type t)

Voor de NRC expressies hebben we ook een interface klasse nl. Expression. Deze klasse heeft drie variabelen, twee int's '_eId' en '_aId' en een Type '_OType'. '_eId' is een integer die de waarde bevat van de expressie in de Expressions tabel in de database en '_aId' is een integer die de waarde heeft van een waarde toekenning in de tabel V_Assignments. '_OType' is het output type van de expressie. Hieronder worden de constructors van de soorten NRC expressies opgesomd. De interpretatie hiervan lijkt me triviaal. Voor elke constante expressie moeten we een aparte klasse aanmaken, afgeleid van ConstantExpr, omdat de where condities in een expressie verschillen. Bv. constante string moeten enkele quotes rond, bij integer niet.De variabele 'se' staat voor 'subExpressie', dit is een id naar de tabel Expressions. De

variabele 'v' staat voor een variabele, deze is van type VarExpr, waar een String var, de naam van de variabele, moet meegegeven worden.

- public ConstantExpr(int e, Type t, String c)
  public ConstantExprInteger(int e, Type t, String c)
  public ConstantExprString(int e, Type t, String c)
  ...
- public VarExpr(int e, Type t, String var)
- public EmptyExpr(int e, Type t)
- public SetExpr(int e, Expression se, Type t)
- public UnionExpr(int e, Expression se1, Expression se2, Type t)
- public BigUnionExpr(int e, Expression se, Type t)
- public TupleExpr(int e, HashSet¡TupleFieldValue¿ f, Type t)
  public TupleField(String l, Expression e)
- public LabelExpr(int e, Expression se, String l, Type t)
- public ForExpr(int e, Expression se1, Expression se2, VarExpr v, Type t)
- public EqualityExpr(int e, Expression se1, Expression se2, Type t)
- public EmptyEqExpr(int e, Expression se, Type t)
- public IfExpr(int e, Expression se1, Expression se2, Expression se3, Type t)
- public LetExpr(int e, Expression se1, Expression se2, VarExpr v, Type t)
- public FunctionExpr(int e, Expression se, Type t)

Hoofdstuk 11

Conclusie

Complex-object databases zijn een verademing voor applicatie ontwikkelaars. Ze bieden nieuwe mogelijkheden waar velen al jaren naar uitkijken sinds hun gevechten met relationele databases. Helaas, voor de ontwikkelaars van complex-object database systemen en hun ondervragingstalen worden zij vaak op de achtergrond gezet door de gestandaardiseerde relationele database systemen en hun ondervragingstalen (SQL[2]). De voordelen van complex-object database systemen voor business toepassingen moeten veel te groot zijn ten opzichte van de relationele database systemen, vooraleer een bedrijf de optie overweegt om over te schakelen.

De interesse voor dit thesis onderwerp, werd gelegd in de cursus *Geavanceerde database toepassingen*. In deze cursus werd de Nested Relationele Calculus (NRC[3, 4]) kort aangehaald. Dit bleek ook geen verkeerde keuze te zijn, daar het onderwerp me gedurende ettelijke maanden heeft weten te boeien. Desondanks was het niet altijd even gemakkelijk de foutmeldingen van DB2 te ontcijferen. Ook was het niet altijd even gemakkelijk duidelijke informatie te vinden over bepaalde aspecten van DB2 en complex-object databases. Informatie over complex-object databases bleek soms zelfs tegenstrijdig te zijn. Dit heeft me enerverende momenten en demotivatie bezorgd. Gelukkig werden deze momenten gecompenseerd met de gedrevenheid van promotor Jan Van den Bussche en kon ik met vragen over DB2 steeds terecht bij Dirk Leinders, waarvoor dank.

Ik hoop dat deze tekst u een duidelijk beeld heeft gegeven over complex object databases en de ondervragingstaal NRC[3, 4]. De studie heeft een mogelijke omzetting van complexe objecten naar relaties in een relationeel model geïntroduceerd. Hierdoor hadden we een systeem nodig die NRC expressie kon vertalen naar een PSM programma, dit onderwerp werd verder behandeld in Hoofdstuk 9. Daar dit systeem zeker nog niet compleet is, heb ik een hoofdstuk ingelast met de software architectuur van het systeem, met als doel het uitbreiden van het systeem, door collega-programmeurs, te vergemakelijken.

Bibliografie

[1] E.F. Ted Codd:, "A Relational Model of Data for Large Shared Data Banks" 13(6), 377-387 , 1970 http://www.acm.org/classics/nov95/toc.html

[2] Jim Melton and Alan R. Simon. SQL 1999: "Understanding Relational Language Components." Morgan Kaufmann, 2002.

[3] Peter Buneman, Shamim A. Naqvi, Val Tannen, and LimsoonWong. "Principles of programming with complex objects and collection types." Theoretical Computer Science, 149(1):348, 1995.

[4] Limsoon Wong. "Querying nested collections." PhD thesis, University of Pennsylvania, 1994.

[5] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, , and Fernando Velez, editors. "The Object Data Standard: ODMG 3.0." Morgan Kaufmann, 2000.

[6] Don Chamberlin, Jonathan Robie, and Daniela Florescu. "Quilt: An XML query language for heterogeneous data sources." In The World Wide Web and Databases: WebDB 2000. Selected Papers, volume 1997 of Lecture Notes in Computer Science, pages 125. Springer-Verlag, 2001.

[7] Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM) INCITS/ISO/IEC 9075-4-2003

[8] Stonebraker, Michael with Moore, Dorothy. Object-Relational DBMSs: The Next Great Wave. Morgan Kaufmann Publishers, 1996. ISBN 1-55860-397-2.

[9] Alex van Ballegooij, Arjen P. de Vries and Martin Kersten. RAM: Array Processing over a Relational DBMS, 2001.

[10] Patrick Valduriez, Setrag Khoshajian and George Copeland. Implementation Techniques of Complex Objects.

[11] Batory D.S., Kim W., Modeling Concepts for VLSI CAD Objects ACM Trans. on Database Systems, vol. 10, no. 3, September 1985.

[12] Adiba M., Nguyen G.T., Handling Constraints and MetaData on Generalized Data Management Systems Int. Workshop on Expert Database Systems, Kiowah Island, South Carolina, October 1984.

[13] Catalog views, http://publib.boulder.ibm.com/infocenter/db2luw/v8/index.jsp?topic=/com.ibm.db2.udb.doc/admin/r0001063.htm

[14] Sequence objects, http://www-128.ibm.com/developerworks/db2/library/techarticle/0205pilaka/0205pilaka2.html

Auteursrechterlijke overeenkomst

Opdat de Universiteit Hasselt uw eindverhandeling wereldwijd kan reproduceren, vertalen en distribueren is uw akkoord voor deze overeenkomst noodzakelijk. Gelieve de tijd te nemen om deze overeenkomst door te nemen en uw akkoord te verlenen.

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
De compilatie van NRC expressies naar SQL
Richting: **Licentiaat in de informatica** Jaar: **2006**
in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Deze toekenning van het auteursrecht aan de Universiteit Hasselt houdt in dat ik/wij als auteur de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij kan reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

U bevestigt dat de eindverhandeling uw origineel werk is, en dat u het recht heeft om de rechten te verlenen die in deze overeenkomst worden beschreven. U verklaart tevens dat de eindverhandeling, naar uw weten, het auteursrecht van anderen niet overtreedt.

U verklaart tevens dat u voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen hebt verkregen zodat u deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal u als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze licentie

Ik ga akkoord,

Jan STULENS

Datum: