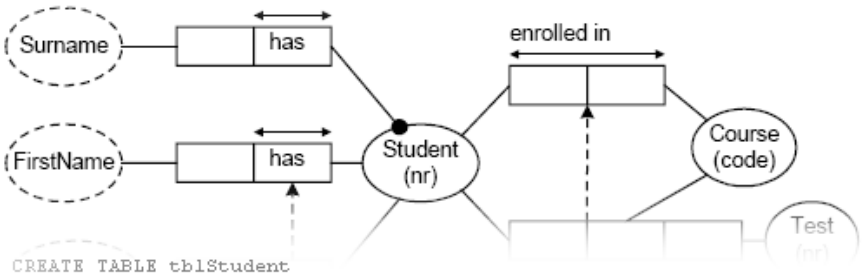
Conceptuele modellen versus SQL 2003

Scriptie versie 1.0




```
CREATE TABLE tblStudent
     (StudentNr CHAR(10) NOT NULL PRIMARY KEY,
     Surname CHAR(20) NOT NULL,
     FirstName CHAR(10) NULL,
     SecondName CHAR(10) NULL)

CREATE TABLE tblCourse
     (StudentNr CHAR(10) NOT NULL PRIMARY KEY REFERENCES tblStudent(Stud
     CourseCode CHAR(10) NOT NULL PRIMARY KEY)

CREATE TABLE tblTest
     (StudentNr CHAR(10) NOT NULL PRIMARY KEY REFERENCES tblStudent(Stud
     CourseCode CHAR(10) NOT NULL PRIMARY KEY REFERENCES tblCourse(Cours
     TestNr CHAR(10) NOT NULL PRIMARY KEY)
```

Naam: N.W.T.M. (Niek) Reulink
Afstudeerdocent: dr. P. (Patrick) van Bommel
Plaats, datum: Nijmegen, 16 juni 2006
Richting: Informatiekunde
Studentnummer: s0432784
E-mail: niekreulink@student.ru.nl
Versie: 1.0 (definitief)
Afstudeernummer: 26 IK

Samenvatting

De titel van dit onderzoek is *Conceptuele modellen versus SQL:2003*. Het woordje *versus* geeft al aan dat het hier om een vergelijking gaat. Conceptuele modellen – in deze scriptie zal hier de modelleertaal ORM voor gebruikt worden – zijn modellen die een bepaald domein (Universe of Discourse) beschrijven en die als basis dienen voor een (database-) systeem. SQL:2003 is de laatste standaardversie van SQL. SQL is een ANSI/ISO standaard databasetaal, waarmee gegevens opgevraagd en gemuteerd kunnen worden.

ORM en SQL zijn twee 'talen' die erg met elkaar verbonden lijken. Zo worden ORM modellen in de praktijk vaak gebruikt als basis voor een uiteindelijke databasestructuur. Omdat ORM en SQL in eerste oogopslag zoveel samenhang lijken te hebben, wordt er in dit onderzoek bekeken of dit inderdaad zo is. Het probleem waarop dit onderzoek gebaseerd is, is dat er zonder onderzoek te doen geen uitspraken gedaan kunnen worden over de precieze samenhang tussen SQL en ORM. ORM is een modelleertaal en SQL een implementeertaal. Het is bekend dat de mogelijkheden van ORM en SQL gedeeltelijk overlappen. Er zijn bepaalde zaken die in ORM wél uit te drukken zijn en in SQL niet, en vice versa, maar er zijn ook zaken die in beide talen uitgedrukt kunnen worden. Het is momenteel echter niet precies duidelijk in hoeverre ORM modellen te implementeren zijn in SQL.

De vergelijking tussen ORM en SQL is op een systematische manier gedaan (gedeeltelijk aan de hand van de methode die beschreven wordt in [OHFB92]). De vergelijking is op meerdere niveaus gedaan. Op metamodel niveau is de vergelijking wat algemener van aard, op het voorbeeldniveau worden er concrete praktijkvoorbeelden bij de vergelijking gegeven terwijl op het formele niveau een formele onderbouwing bij de vergelijking wordt gegeven.

Uit de vergelijking blijkt dat er veel meer overeenkomsten tussen ORM en SQL zijn, dan dat er verschillen zijn. Entiteittypen en labeltypen uit ORM zijn bijna één-op-één te vertalen naar tabellen en kolommen in een RDBMS (Relationeel DataBase Management Systeem). Ook zijn de meeste constraints uit ORM (zoals de uniqueness-, equality- en exclusion constraint) te vertalen naar SQL. Dit kan meestal niet één-op-één, maar middels omwegen kunnen de meeste constraints toch in SQL verwezenlijkt worden. Ook complexe modelleerconstructen zoals specialisatie en generalisatie komen zowel in ORM als SQL voor.

Uiteraard zijn er ook verschillen tussen twee talen, anders zou er geen onderscheid tussen de twee talen worden gemaakt. Er zijn echter een stuk minder verschillen dan overeenkomsten te vinden. De beide talen hebben in ieder geval een verschillend doel. ORM is een modelleertaal, dat gebruikt wordt om informatiesystemen te modelleren op het conceptuele niveau. Met SQL is het mogelijk om een database te creëren, de aangemaakte tabellen van een populatie te voorzien en deze populatie op te vragen en te muteren. Bepaalde ORM constructies zijn niet te vertalen naar SQL. Zo kent ORM wel multiple inheritance (een subtype kan eigenschappen overerven van meerdere supertypen), maar SQL niet. Een ander verschil is dat het belangrijkste bestanddeel van de SQL:2003 standaard, SQL/XML, niet terug komt in ORM. Er zijn wel ontwikkelingen op dit gebied gaande, maar op het moment van schrijven kan dit nog niet. SQL en ORM kennen ook een verschillende semantiek. De semantiek van ORM is een stuk 'zachter'. Hiermee worden de natuurlijke taal zinnen bedoeld waarmee feittypen worden beschreven. De SQL semantiek is veel strikter.

Nadat de overeenkomsten en verschillen op papier stonden (het metamodel- en het voorbeeldniveau), is de vergelijking naar het formele niveau verplaatst. In [OHFB92] worden er enkele functies gegeven die op metamodellen toegepast kunnen worden om deze te vergelijken. De functies zijn voorzien van een formele definitie. Deze functies zijn in dit onderzoek toegepast op de hier gebruikte metamodellen. De resultaten waren ook hier voornamelijk overeenkomsten tussen het metamodel van ORM en dat van SQL. De formele definities zijn, voor zover mogelijk, aangepast aan de ORM/SQL situatie.

Uiteindelijk was er genoeg informatie verzameld om de onderzoeksvraag te kunnen beantwoorden. Deze vraag luidde: "*Hoe kunnen conceptueel model en RDBMS dichter bij elkaar gebracht worden met gebruik van de standaard SQL:2003?"*. Het antwoord op deze vraag is dat dit kan door middel van de in deze scriptie gehanteerde vergelijkingsmethode. In deze scriptie is er een vergelijking gedaan tussen twee metamodellen. Het uiteindelijke doel was om deze twee metamodellen dichter bij elkaar te brengen. Hiermee wordt bedoeld de expressiviteit van enerzijds verschillende SQL standaarden en anderzijds Object Role Modeling (ORM) te inventariseren en vergelijken. Het was bij voorbaat al duidelijk dat er een samenhang bestaat tussen ORM en SQL. Dit onderzoek heeft getracht deze samenhang concreet te maken door de overeenkomsten en verschillen tussen deze twee talen te beschrijven.

De conclusie van dit onderzoek is dat beide talen inderdaad behoorlijk veel samenhang hebben. Heel veel ORM modellen zijn zonder moeite te vertalen naar SQL databaseschema's. Dit is een in de praktijk beproefde methodiek gebleken. Er zijn verschillende tools op de markt die de gebruiker ondersteunen bij de conversie van een ORM model naar een SQL relationele database. De samenhang tussen deze twee talen is een krachtig middel in het systeem ontwikkeltraject.

Het eigenlijke doel van de scriptie was om te kijken in hoeverre ORM en SQL dichter bij elkaar gebracht konden worden met gebruik van de standaard SQL:2003, zoals de hoofdvraag ook aangeeft. Het gebruik van de SQL:2003 standaard is achteraf minimaal gebleken. De vergelijking is gedaan door SQL als geheel (dat wil zeggen dat alle SQL standaarden in de vergelijking betrokken zijn) te vergelijken met ORM. De belangrijkste toevoeging die de SQL:2003 standaard doet ten opzichte van de eerder verschenen standaarden, is de introductie van SQL/XML. XML wordt in ORM echter in zijn geheel niet ondersteund. Hierdoor wordt de invloed van de SQL:2003 standaard beperkt tot enkele nieuwe datatypen en een nieuwe data mutatie methode.

Voorwoord

Deze scriptie is het resultaat van mijn onderzoek naar de overeenkomsten en de verschillen tussen de modelleertaal ORM (Object Role Modeling) en de databasetaal SQL (Structured Query Language). Dit onderzoek is gedaan in het kader van mijn studie Informatiekunde aan de Radboud Universiteit Nijmegen. Het afstudeeronderzoek vond plaats van februari tot juli 2006. Deze scriptie geeft een beschrijving van het onderzoek en de resultaten ervan.

Ik wil graag mijn afstudeerbegeleider, de heer Dr. P. van Bommel bedanken, voor zijn goede begeleiding en hulp bij de totstandkoming van deze scriptie. Verder bedank ik iedereen die een bijdrage aan het onderzoek en de scriptie geleverd hebben.

N.W.T.M. Reulink
Nijmegen, 16 juni 2006

Inhoudsopgave

1. Inleiding

De afstudeeropdracht zal uitgevoerd worden in opdracht van de Radboud Universiteit Nijmegen (RU), met als afstudeerdocent dhr. dr. P. (Patrick) van Bommel. Deze scriptie is het eindresultaat van een onderzoek met de titel “Conceptuele modellen versus SQL:2003”.

In hoofdstuk 2 zal de probleemstelling van het onderzoek worden beschreven. Dit hoofdstuk is de basis van het onderzoek en legt de doelstelling en de onderzoeksvragen vast.

Hoofdstuk 3 zal een korte introductie in SQL geven. Wat is het en hoe is het ontstaan?

De SQL standaard heeft in de loop der jaren een aantal versies doorlopen en veranderingen ondergaan. In hoofdstuk 4 zullen alle SQL standaarden kort worden beschreven.

In hoofdstuk 5 wordt als opmaat voor de hoofdstukken hierop volgend, een korte uitleg gegeven over twee soorten modellen: conceptuele modellen en metamodellen.

De metamodellen van SQL en ORM, die dienen als vergelijkingsmateriaal in dit onderzoek, worden beschreven in hoofdstuk 6. De modellen zijn te vinden in de appendices C en D.

In de hoofdstukken 7 en 8 wordt de daadwerkelijke vergelijking tussen de twee metamodellen uitgewerkt. Dit wordt gedaan middels concrete voorbeelden.

Hoofdstuk 9 bouwt hierop voort en biedt een formele onderbouwing voor de vergelijking.

De antwoorden op de onderzoeksvragen zullen gegeven worden in hoofdstuk 10.

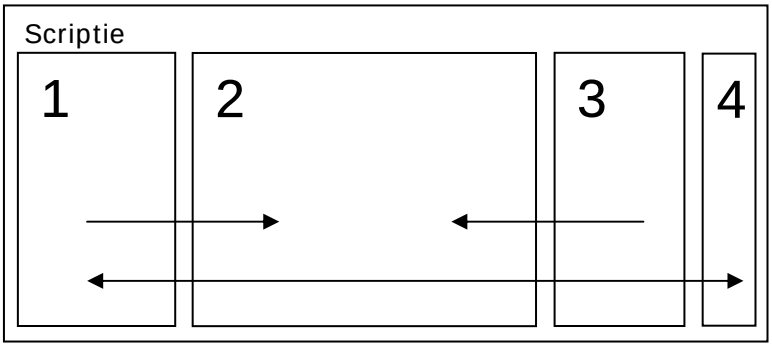
De scriptie wordt in hoofdstuk 11 afgesloten met een evaluatie, waarin onder meer beschreven wordt waarom deze scriptie van voldoende niveau is.

Verwijzingen in dit document die aangegeven zijn met vierkante haken [ ] zijn terug te vinden in de literatuurlijst.

De index van de gebruikte figuren is te vinden in de lijst van figuren achter in de scriptie.

1.1 Schematische opbouw scriptie

Hieronder is de opbouw van deze scriptie schematisch weergegeven. De verschillende delen zijn genummerd en worden hieronder verder uitgelegd. De pijlen in de figuur geven verbanden aan tussen de verschillende delen. Zo is er samenhang tussen deel 1 en 2, en tussen deel 3 en 2. Deel 4 doet uitspraken over de gehele scriptie.



Figuur 1: Schematische weergave opbouw scriptie

1) Inleiding
De scriptie begint met uitleg over SQL en ORM en een beschrijving van de gebruikte metamodellen. Deze informatie is de basis voor de rest van de scriptie.
2) Praktisch deel
Het grootste gedeelte van de scriptie geeft praktische voorbeelden bij de vergelijking tussen de twee metamodellen.
3) Theoretisch deel
Dit deel geeft een formele benadering bij de eerder gegeven praktische voorbeelden.
4) Conclusie
Ten slotte worden in dit gedeelte de onderzoeksvragen beantwoordt en wordt er een evaluatie van het afstudeerproject gegeven.

2. Probleemstelling

Ieder onderzoek wordt uitgevoerd aan de hand van een doelstelling en een onderzoeksvraag (opgedeeld in deelvragen), zo ook dit onderzoek. Hieronder worden de doelstelling, de onderzoeksvraag en de motivatie van dit onderzoek uiteengezet.

2.1 Probleemdefinitie

ORM en SQL zijn twee 'talen' die erg met elkaar verbonden lijken. Zo worden ORM modellen in de praktijk vaak gebruikt als basis voor een uiteindelijke databasestructuur. Er bestaan tools die een ORM model kunnen omzetten in SQL statements, waardoor direct een database aangemaakt kan worden gebaseerd op dat ORM model[1]. Omdat ORM en SQL in eerste oogopslag zoveel samenhang lijken te hebben, wordt er in dit onderzoek bekeken of dit inderdaad zo is. Stukje voor stukje worden delen van de metamodellen van beide talen met elkaar vergeleken, om zo uiteindelijk precies te weten te komen welke overeenkomsten en verschillen er zijn tussen ORM en SQL. Beide talen zijn ook in ontwikkeling waardoor ze misschien nog meer naar elkaar toe groeien.

Het probleem waarop dit onderzoek gebaseerd is, is dat er zonder onderzoek te doen geen uitspraken gedaan kunnen worden over de precieze samenhang tussen SQL en ORM. ORM is een modelleertaal en SQL een implementeertaal. Het is bekend dat de mogelijkheden van ORM en SQL gedeeltelijk overlappen. Er zijn bepaalde zaken die in ORM wél uit te drukken zijn en in SQL niet, en vice versa, maar er zijn ook zaken die in beide talen uitgedrukt kunnen worden. Het is momenteel echter niet precies duidelijk in hoeverre ORM modellen te implementeren zijn in SQL.

Dit onderzoek probeert een oplossing te geven voor dit probleem door de overeenkomsten en verschillen tussen deze twee talen uiteen te zetten. Een systematische aanpak waarbij de metamodellen worden vergeleken op theoretisch en praktisch niveau is hiervoor gehanteerd. De vergelijking zal niet tot in het kleinste detail gedaan worden, maar in voldoende mate om de onderzoeksvraag mee te kunnen beantwoorden en een goed beeld te geven van de overeenkomsten en verschillen tussen de beide talen.

2.2 Doelstelling

Het doel van dit onderzoek is om de expressiviteit van enerzijds verschillende SQL standaarden en anderzijds Object Role Modeling (ORM) te inventariseren en vergelijken. Dit zal op een systematische manier gedaan worden. In dit onderzoek wordt de vergelijking gedaan door het vergelijken van twee metamodellen. Eén van ORM en één van SQL. Het onderzoek kijkt op twee verschillende niveaus naar het probleem. Op een praktisch niveau worden concrete voorbeelden gegeven bij de vergelijking tussen de twee metamodellen. Op een technisch niveau wordt er getracht een formele benadering te geven bij de vergelijking. Hier zullen de metamodellen worden vergelijken aan de hand van verschillende formalismen.

[1] Een voorbeeld hiervan is het programma VisioModeler van Microsoft.

2.3 Onderzoeksvraag en deelvragen

De hoofdvraag van dit onderzoek luidt:

- Hoe kunnen conceptueel model en RDBMS dichter bij elkaar gebracht worden met gebruik van de standaard SQL:2003?

De hoofdvraag kan onderverdeeld worden in de volgende deelvragen:

- Wat is SQL?
- Welke SQL standaarden zijn er?
- Wat zijn de verschillen tussen deze standaarden?
- Wat is een conceptueel model?
- Wat is een metamodel?
- Welke overeenkomsten zijn er te vinden tussen de metamodellen van ORM en SQL?
- Welke verschillen zijn er te vinden tussen de metamodellen van ORM en SQL?

3. SQL algemeen

In dit hoofdstuk zal de geschiedenis van de ANSI/ISO standaardtaal voor een relationeel database management systeem (RDBMS) worden beschreven. Deze taal, genaamd SQL, wordt gebruikt om informatie uit relationele databases op te vragen en te muteren.

3.1 SQL geschiedenis

De geschiedenis van SQL begint bij dr. Edgar F. Codd, die in 1970 het relationele model uiteenzet in zijn artikel "A Relational Model of Data for Large Shared Data Banks" [Codd70]. In de jaren '70 ontwikkelt IBM een RDBMS, genaamd System R naar het gelijknamige project, dat gebaseerd was op het model van Codd. Het project moest aantonen dat de positieve gebruikersgemakken van het relationele model geïmplementeerd konden worden in een systeem dat voldeed aan alle eisen van een modern database management systeem. Een probleem dat in het System R-project moest worden opgelost, was dat er nog geen relationele databasetalen bestonden. Er moest er dus één worden ontwikkeld. Donald D. Chamberlin en Raymond F. Boyce ontwierpen de Structured English Query Language (SEQUEL) om data uit System R op te vragen en te muteren. Eind jaren '70 werd de naam gewijzigd in SQL, omdat SEQUEL al een bestaande merknaam bleek te zijn.

Chamberlin en Boyce publiceerden hun ideeën [ChBo74], waardoor de interesse in SQL groeide. IBM ontwikkelde een aantal commerciële producten waarin SQL geïmplementeerd was, gebaseerd op het prototype System R. In augustus 1979 bracht IBM System/38 op de markt, gevolgd door SQL/DS in 1981 en DB2 in 1983. System/38 was een minicomputer met daarop een operating system met geïntegreerd RDBMS. SQL/DS was IBM's eerste commerciële DBMS. DB2 ten slotte is IBM's RDBMS, dat tot op de dag van vandaag één van de grootste spelers op de RDBMS markt is. IBM was niet het enige bedrijf dat heil zag in SQL. Oracle Corporation (toen nog Relational Software, Inc.) was het eerste bedrijf dat een commercieel product gebaseerd op SQL op de markt bracht. Oracle V2 werd enkele weken voor IBM's System/38 gepresenteerd. De interesse in SQL bleef groeien en steeds meer bedrijven kwamen met hun eigen RDBMS.

Pas in de loop van de jaren '80 werd SQL gestandaardiseerd. De eerste standaarddefinitie van SQL werd in 1986 door ANSI[2] en in 1987 door ISO[3] geaccepteerd en heet SQL-86. Sindsdien hebben vele verschillende SQL standaarden het levenslicht gezien. Hierover meer in het volgende hoofdstuk.

3.2 Relationele model

Zoals hierboven ook al werd vermeld, is Codd de grondlegger van de relationele database. Zijn relationele model is van grote invloed geweest op de ontwikkeling van RDBMS'en en SQL. Vandaar dat er hier een aparte paragraaf aan het relationele model wordt besteed.

Het relationele model bestaat uit drie componenten [GaVa98]:

- Een structurele component: een verzameling tabellen,
- Een manipulatieve component dat bestaat uit een verzameling bewerkingen die uitgevoerd worden op tabellen en ook weer tabellen produceren,
- Een verzameling regels die de integriteit van de database moeten bewaken.

[2] American National Standards Institute
[3] International Organization for Standardization

De structurele component beschrijft de structuur van data. De data structuur bestaat volgens Codd uit twee componenten, namelijk entiteittypen en relaties tussen entiteittypen (Engels: relationship). Entiteittypen worden gerepresenteerd door relaties (Engels: relation). Een relatie is hier de wiskundige benaming voor een tabel. Een tabel wordt op zijn beurt weer gedefinieerd als een ongeordende verzameling van nul, één of meer tupels (rijen), waarbij elke tupel weer bestaat uit een ongeordende verzameling attributen (kolommen). Alle tupels bevatten dezelfde verzameling attributen.

De naamgeving in het relationele model is verwarrend. Hierboven is bijvoorbeeld het woord relatie ambigu. In de ene situatie betekent het een relatie tussen twee tabellen en in de andere situatie betekent het een tabel zelf. Dit is ook de reden dat er vaak wanneer er gesproken wordt over het relationele model, een andere terminologie wordt gebruikt.

- Tabel wordt gebruikt in plaats van Relatie (alleen wanneer er met relatie tabel wordt bedoeld)
- Rij wordt gebruikt in plaats van Tupel
- Kolom wordt gebruikt in plaats van Attribuut

De tweede component van het relationele model beschrijft een verzameling bewerkingen die uitgevoerd kunnen worden op tabellen. Codd beschreef oorspronkelijk acht relationele operatoren:

1. `SELECT` (oorspronkelijk `RESTRICT` genoemd)
   Selecteert een aantal rijen uit één of meerdere tabellen.
2. `PROJECT`
   Produceert een kopie van een bestaande tabel, waarin in kolommen kunnen worden weggelaten
3. `JOIN`
   Selecteert een aantal rijen uit meerdere tabellen, aan de hand van gemeenschappelijke kolomwaarden
4. `PRODUCT`
   Creëert een tabel, bestaande uit alle mogelijke combinaties van rijen van de twee doeltabellen
5. `UNION`
   Creëert een tabel, bestaande uit alle rijen die in één van de twee doeltabellen voorkomen
6. `INTERSECT`
   Creëert een tabel, bestaande uit alle rijen die in de twee doeltabellen voorkomen
7. `DIFFERENCE`
   Creëert een tabel, bestaande uit de rijen die wél in de ene doeltabel, maar niet in de andere doeltabel voorkomen
8. `DIVIDE`
   Creëert een tabel, bestaande uit de waarde(n) van kolom 1 uit tabel 1, waarbij alle waarden uit kolom 2 van tabel 1 ook voorkomen in tabel 2

De derde component van het relationele model bevat enkele integriteitsregels. Enkele voorbeelden hiervan zijn de volgende:

- Elke kolom mag maar één type waarden bevatten (bijvoorbeeld alleen telefoonnummers).
- Elke rij moet uniek geïdentificeerd worden door een `PRIMARY KEY` bestaande uit één of meer kolommen. Dit heeft tot gevolg dat een tabel geen dubbele rijen kan en mag bevatten.
- Een `PRIMARY KEY` mag geen `NULL` waarden bevatten.

4. SQL standaarden

In dit hoofdstuk zal een overzicht gegeven worden van alle SQL standaarden die in de loop der jaren zijn geaccepteerd door ANSI/ISO.

Sinds de eerste officiële SQL standaard, die in 1986 werd uitgegeven, is de SQL standaard verschillende malen herzien en aangevuld. In deze paragraaf zullen de verschillende ANSI/ISO SQL standaarden worden beschreven.

De SQL standaarden worden geproduceerd door werkgroepen van ISO en ANSI. Op het internationale toneel werkt de ISO werkgroep “ISO/IEC JTC 1/SC 32 Data Management and Interchange/WG 3 – Database Languages” aan de SQL standaard. Deze werkgroep bestaat uit delegaties uit negen landen, waaronder Nederland [JCC06].

De ANSI werkgroep genaamd “ANSI NCITS H2 Technical Committee on Database” [ANSIH2] richt zich alleen op de Verenigde Staten. Deze werkgroep bestaat uit vertegenwoordigers van alle grote RDBMS ontwikkelaars, aangevuld met enkele consultants en gebruikers.

De originele SQL standaard werd door ANSI geaccepteerd in 1986. Een jaar later werd SQL ook een ISO standaard. De originele SQL standaard is herzien en uitgebreid in 1989 en 1992. In onderstaande tabel wordt een overzicht gegeven van de ontwikkelingen op SQL gebied door de jaren heen.

Jaren '70	Relationele model DBMS prototypes (SEQUEL XRM) Eerste relationele DBMS
Jaren '80	ANSI SQL-86 standaard ISO SQL-87 standaard SQL-89 (uitbreiding) ANSI Embedded SQL
Jaren '90	SQL 92 standaard SQL/CLI (Call Language Interface) SQL/PSM (Persistent Stored Modules) SQL:1999 standaard
2003	SQL:2003 standaard

Tabel 1: De evolutie van SQL [CRBaBrP]

Bepaalde technieken evolueren sneller dan andere technieken. Dit is de reden dat de SQL standaard is opgesplitst in verschillende delen, die allemaal op een eigen snelheid ontwikkeld kunnen worden. Twee delen van de SQL standaard werden in de jaren ’90 voltooid, als aanvulling op de SQL-92 standaard. SQL/CLI (Call Language Interface) werd in 1995 afgerond en in 1996 gebeurde dit met SQL/PSM (Persistent Stored Modules). Herzieningen en toevoegingen op de eerste vijf delen van de SQL standaard werden voltooid in 1999 als onderdeel van SQL:1999 (zie appendix A).

Alle delen van de SQL standaard werden nogmaals herzien in 2003, wat leidde tot de SQL:2003 standaard. De volgende compleet nieuwe SQL standaard staat gepland voor 2007 of 2008. In de figuur op de volgende pagina wordt een overzicht gegeven van alle delen van de SQL standaard.

Part	Explanation
Part 1: SQL/Framework	Contains information common to all parts of the standard and describes the parts.
Part 2: SQL/Foundation	SQL Data definition and data manipulation syntax and semantics, including SQL embedded in non-object programming languages.
SQL/OLAP	Online Analytical Processing: Amendment describing functions and operations useful for analytical processing.
Part 3: SQL/CLI	Call Level Interface: Corresponds to ODBC.
Part 4: SQL/PSM	Persistent Stored Modules: Stored routines, external routines, and procedural language extensions to SQL.
Part 5: SQL/Bindings	Embedded SQL.
Part 6: SQL/Transaction	SQL specialization of the X-Open XA specification. Project has been canceled.
Part 7: SQL/Temporal	Extensions to SQL to deal with time-oriented data types. This project has been withdrawn.
Part 8: SQL/Objects - Extended Objects	SQL Object model. This part no longer exists.
Part 9: SQL/MED	Management of External Data: Adds syntax and definitions to SQL/Foundation to allow SQL access to non-SQL data sources (files).
Part 10: SQL/OLB	Object Language Bindings: Specifies the syntax and semantics of embedding SQL in Java™.
Part 11: SQL/Schemata	Information and Definition Schemas
Part 12: SQL/Replication	Replication facilities for SQL. The goal is to define syntax and semantics to allow definition of replication schemes and rules, including rules for resolution of update conflicts.
Part 13: SQL/JRT	Java Routines and Types: Routines using the Java™ Programming Language (Persistent Stored SQLJ)
Part 14: SQL/XML	SQL and XML

Tabel 2: Delen van de SQL standaard [JCC06]

4.1 Syntaxis en semantiek

In Tabel 2 hierboven worden een aantal keren de woorden syntax en semantics gebruikt. De syntaxis en semantiek van een taal zijn de bouwstenen van die taal.

De syntaxis van een taal is de verzameling van regels die specificeren hoe zinnen uit letters, cijfers en andere tekens worden samengesteld. Natuurlijke talen, zoals het Nederlands, onderscheiden zich van formele talen, zoals wiskunde en programmeertalen, doordat ze veel minder strikt zijn. Natuurlijke talen bezitten de eigenschappen van dubbelzinnigheid, overbodigheid, en letterlijkheid. In de meeste natuurlijke talen zijn er veel woorden die meerdere betekenissen hebben. Formele talen hebben deze drie eigenschappen pertinent niet, en de regels over welke symbolen geldig zijn, hoe met symbolen omgegaan moet worden en hoe de structuur van zinnen moet zijn, zijn zeer strikt. Het is voor een formele taal, waaronder ook SQL valt, vaak wel duidelijk hoe de syntaxis eruit moet zien. Het is namelijk bekend wat de taal moet kunnen en er hoeft alleen nog een notatie bedacht te worden. In dit geval de SQL statements als CREATE TABLE, SELECT etcetera.

De betekenis van een zin in een taal wordt ook wel de semantiek van de taal genoemd. Ook dit is een noodzakelijk onderdeel van de taal. De betekenis van de zin kan pas blijken als we alle woorden van de zin hebben gelezen of gehoord, en de structuur van de zin tot ons is doorgedrongen. Programmeertalen zijn net als wiskunde formele talen. Semantische regels in programmeertalen bepalen net als in een natuurlijke taal de 'betekenis' van een syntactisch correct programma. Deze betekenis is echter een stuk lastiger te definiëren dan de syntaxis van een taal. De betekenissen van woorden, termen of keywords in een taal als SQL, moeten eenduidig vastliggen. Dit is vaak lastig. Er zijn veel discussies gevoerd over bijvoorbeeld de NULL waarden in SQL. Is NULL nou het cijfer 0, een spatie, ‘onbekend’ of een leegte? Hier moeten dus duidelijke afspraken over gemaakt worden.

Eén van de manieren om de semantiek van een programmeertaal te beschrijven is het geven van een beschrijving van elke geldige constructie in de taal. Daarmee introduceren we een nadeel doordat de dubbelzinnigheid die elke natuurlijke taal bezit, behouden blijft. Het voordeel is dat er een redelijk intuïtief beeld van de taal ontstaat waarmee men makkelijker omgaat dan met een beschrijving in een streng formele notatie [SynSem].

In de SQL:2003 standaard [MEL03], wordt de semantiek beschreven door een beschrijving van elke constructie te geven. Hier worden voorbeelden bij gegeven. Er worden ook waarheidstabellen gebruikt om bijvoorbeeld de betekenissen van de keywords AND, OR, IS en NULL te geven. Hier is dan precies in te zien in welke situatie een keyword TRUE en in welke situatie een keyword FALSE oplevert.

4.2 ANSI SQL-86 / ISO SQL-87

De allereerste SQL standaard, ook wel SQL1[4] genoemd, werd in 1986 door ANSI en in 1987 door ISO geaccepteerd. In het vervolg zal deze standaard als SQL-86 worden aangeduid. Deze standaard bevatte de basisconcepten achter SQL, die als bouwstenen dienden voor alle standaarden die volgden. Hieronder zullen deze bouwstenen kort besproken worden. SQL-86 was een redelijk eenvoudige standaard, en deze werd ook direct door bedrijven in hun DBMS geïmplementeerd.

SQL is opgedeeld in drie gebieden, Data Definition Language (DDL), Data Manipulation Language (DML) en Data Control Language (DCL).

Met DCL kunnen rechten aan gebruikers worden toegekend. Dit zijn rechten voor het bekijken, verwijderen en veranderen van gegevens in een database. Door middel van DCL kan bijvoorbeeld worden geregeld dat Jan de gegevens in Studenten wel mag bekijken, maar ze niet mag aanpassen.

```
GRANT SELECT
ON Studenten
TO Jan
```

Met bovenstaand statement wordt vastgelegd dat Jan de tabel Studenten mag bekijken. In SQL heeft niemand rechten, tenzij anders is aangegeven (door middel van GRANT statements). Omdat in dit GRANT statement alleen wordt gesproken over SELECT, betekent dit dat alle andere statements door Jan niet uitgevoerd mogen worden.

DDL is het gebied binnen SQL waarmee de structuur van de database gedefinieerd kan worden. Dit gebeurt door middel van een CREATE statement.

```
CREATE TABLE Studenten (
   studentID      integer     NOT NULL     PRIMARY KEY,
   voornaam       char(20)    NOT NULL,
   achternaam     char(50)    NOT NULL,
   woonplaats     char(50)    NOT NULL,
   uitgeschreven  boolean     NOT NULL,
   sportkaartnr   integer     NULL )
```

Bovenstaand CREATE statement definieert een tabel genaamd Studenten waarin informatie over studenten opgeslagen kan worden. Een student wordt uniek geïdentificeerd met een studentID. Deze kolom is dan ook ingesteld als PRIMARY KEY, ook wel identificerende sleutel genoemd. Alle kolommen dienen verplicht ingevuld te worden, behalve de kolom sportkaartnr, aangezien niet iedere student in het bezit is van een sportkaart. Vandaar dat hier NULL staat, wat aangeeft dat er in dit veld niet verplicht iets ingevuld hoeft te worden.

[4] In de geraadpleegde bronnen is er verwarring over de term SQL1. Sommige bronnen melden dat dit SQL-86 is, andere bronnen melden dat dit SQL-89 is. Aangezien SQL-89 een kleine uitbreiding op SQL-86 is, zal de term SQL1 hier synoniem zijn voor SQL-86.

DDL is niet alleen in staat om tabellen te creëren, maar tabellen kunnen ook verwijderd worden. Dit wordt gedaan met een DROP statement.

```
DROP TABLE Studenten
```

DML is het gebied binnen SQL dat de gebruiker de mogelijkheid biedt om de gegevens in de tabellen te raadplegen of te manipuleren. Het meest gebruikte SQL statement is het SELECT statement. Hiermee kunnen door middel van queries specifieke gegevens (records) uit een database opgevraagd worden.

```
SELECT  voornaam, achternaam
FROM    Studenten
WHERE   woonplaats = 'Nijmegen'
```

Met deze query wordt van alle studenten die in Nijmegen wonen, de voor- en achternaam getoond.

De volgende mogelijkheid die DML biedt, is om gegevens aan tabellen toe te voegen. Dit wordt gedaan met een INSERT statement.

```
INSERT INTO Studenten
VALUES (0123456, 'Klaas', 'Jansen', 'FALSE', 'Nijmegen')
```

Met deze query wordt de student Klaas Jansen aan de tabel Studenten toegevoegd. Deze student heeft geen sportkaart dus het sportkaartnr hoeft niet aan de tabel toegevoegd te worden.

DML biedt ook de mogelijkheid om gegevens in tabellen aan te passen. Hier is het UPDATE statement voor in het leven geroepen.

```
UPDATE Studenten
SET woonplaats = 'Amsterdam'
WHERE achternaam = 'Jansen' AND voornaam = 'Klaas'
```

De query hierboven zorgt ervoor dat de woonplaats van de student Klaas Jansen wordt veranderd van Nijmegen naar Amsterdam. Met dit statement wordt één enkele rij in de tabel Studenten aangepast. Het is in SQL echter ook mogelijk om meerdere rijen in één keer aan te passen. Dit kan bijvoorbeeld met het volgende statement:

```
UPDATE Studenten
SET uitgeschreven = TRUE
WHERE studentID < 999
```

Met dit statement wordt van alle studenten die een studentID hebben dat minder is dan 999, de boolean waarde van uitgeschreven op TRUE gezet.

Ten slotte kunnen door middel van een DELETE statement gegevens uit een tabel verwijderd worden.

```
DELETE FROM Studenten
WHERE achternaam = 'Jansen' AND voornaam = 'Klaas'
```

Hiermee wordt de student Klaas Jansen uit de tabel Studenten verwijderd. In dit voorbeeld wordt alleen Klaas Jansen verwijderd uit de tabel. Het is ook mogelijk om meerdere rijen in één keer te verwijderen.

```
DELETE FROM Studenten
WHERE studentID < 999
```

Met dit statement worden alle studenten met een studentID lager dan 999 verwijderd.

Naast bovenstaande statements om tabellen te creëren en de gegevens daarbinnen te manipuleren en op te vragen, beschreef de SQL-86 standaard ook de mogelijkheid om views te gebruiken. SQL kent twee soorten tabellen: echte tabellen, ook wel basistabellen genoemd, en afgeleide tabellen, ook wel views genoemd. Basistabellen worden met CREATE TABLE instructies gecreëerd en dit is de enige tabelvorm waarin gegevens kunnen worden opgeslagen. Een view bevat zelf geen rijen, maar kan gezien worden als een voorschrift of formule om bepaalde gegevens uit de basistabellen in een 'virtuele' tabel samen te voegen. Een view is virtueel omdat de inhoud van een view alleen bestaat als de view in een instructie gebruikt wordt. Een view neemt dus ook geen opslagruimte in beslag [Lans01].

```
CREATE VIEW StudentVak AS
SELECT  Student.voornaam, Student.achternaam, Vakken.keuzevak
FROM    Studenten, Vakken
WHERE   Student.studentID = Vakken.studentID
```

Bovenstaande query creëert een view die de voor- en achternaam van alle studenten laat zien uit de tabel Studenten plus het keuzevak van elke student dat uit de tabel Vakken wordt gehaald. De WHERE clausule zorgt ervoor dat er een relatie wordt gelegd tussen de twee tabellen op de kolom studentID. Hierdoor kan het keuzevak aan de juiste studentnaam worden gekoppeld.

Deze eerste SQL standaard bood ten slotte de mogelijkheid om enkele integriteitsregels toe te voegen aan een database. Voorbeelden hiervan zijn PRIMARY KEY en NOT NULL in de queries hierboven.

4.3 SQL-89

SQL-86 was de eerste stap in de richting van een standaard relationele database query taal. De basis was gelegd maar al snel werd duidelijk dat SQL verre van compleet was. De belangrijkste kritiek op SQL-86 was dat referentiële integriteit niet werd ondersteund.

De SQL-89 standaard was een update van de SQL-86 standaard. Er was geen sprake van een compleet nieuwe standaard. In SQL-89 werd referentiële integriteit toegevoegd aan de SQL-86 concepten.

Referentiële integriteit is een database concept dat er voor zorgt dat relaties tussen tabellen in een database consistent zijn en blijven. In de voorbeeldtabel hierboven was de kolom `studentID` de `PRIMARY KEY`. In tabellen kunnen ook `FOREIGN KEY`s of verwijzende sleutels bestaan. Dit zijn kolommen die verwijzen naar een bijbehorende kolom in een andere tabel. Een `FOREIGN KEY` wordt gebruikt om relaties op te zetten, en is altijd een key in één tabel in de relatie. In Figuur 2 is `begeleid_door` een `FOREIGN KEY`, die waarden bevat uit de kolom `DocentID` uit de tabel `Docent`, dat `PRIMARY KEY` is. Als een tabel een `FOREIGN KEY` naar een andere tabel heeft, zorgt referentiële integriteit ervoor dat je geen gegevens aan de tabel met de verwijzende sleutel mag toevoegen, tenzij er bijbehorende gegevens staan in de tabel waarnaar verwezen wordt.

Twee technieken die bij referentiële integriteit horen zijn ‘cascading update’ en ‘cascading delete’. Deze technieken zorgen ervoor dat wanneer er wijzigingen zijn in de verwijzende tabel, deze wijzigingen ook worden doorgevoerd in de primaire tabel. Om het een en ander te verduidelijken volgt hieronder een voorbeeld.

In deze voorbeeldsituatie zijn er twee tabellen: `Student` en `Docent`. In de tabel `Student` is de kolom `begeleid_door` een `FOREIGN KEY`. Deze verwijst naar de gegevens van de docent die deze student begeleid in de tabel `Docent`.

Figuur 2: Referentiële integriteit

Referentiële integriteit heeft in deze situatie onderstaande gevolgen:

1. Er mag geen record aan de tabel `Student` worden toegevoegd, tenzij de kolom `begeleid_door` verwijst naar een bestaand record in de tabel `Docent`.
2. Als de `PRIMARY KEY` in de tabel `Docent` verandert, moeten alle bijbehorende records in de tabel `Student` aangepast worden middels een cascading update.
3. Als een record in de tabel `Docent` wordt verwijderd, moeten de records uit de tabel `Student` die naar deze docent verwijzen, ook worden verwijderd middels een cascading delete.

Deze regels bevorderen een 'schone' database. Immers, wanneer er docenten weggaan, kunnen zij ook geen studenten meer begeleiden. Cascading update en delete zijn geen mechanismen die standaard actief zijn. Ze moeten handmatig geactiveerd worden. Het zijn ook gevaarlijke mechanismen, waar zorgvuldig mee omgegaan dient te worden. Het kan bijvoorbeeld zo zijn, dat na het verwijderen van één docent, ook de halve studenten tabel wordt verwijderd. In de praktijk krijgt een gebruiker dan wel een waarschuwing te zien, zoals in Figuur 3 is weergegeven.

Figuur 3: Foutmelding cascading delete

Deze waarschuwing verscheen op het scherm toen er geprobeerd werd één docent te verwijderen, die drie studenten begeleidde. Het systeem wil nu dus ook deze drie studenten verwijderen, aangezien cascading delete (Nederlands: trapsgewijs verwijderen) actief was.

4.4 SQL-92

De SQL-92 (SQL2) standaard was een enorme uitbreiding ten opzichte van zijn voorgangers. Het aantal bladzijden waaruit de standaard bestaat is verdubbeld naar 1120 bladzijden om alle nieuwe mogelijkheden die SQL-92 biedt, te beschrijven [JCC06].

In deze paragraaf zullen enkele van deze nieuwe mogelijkheden worden beschreven. Voor een compleet overzicht van de mogelijkheden van SQL-92 wordt u verwezen naar [Mimer].

De eerste nieuwe mogelijkheid in SQL-92 is Dynamic SQL. Dynamic SQL is SQL waarbij statements automatisch gegenereerd en uitgevoerd kunnen worden. Dit kan handig zijn wanneer een bepaalde query meerdere malen uitgevoerd moet worden. Het kan bijvoorbeeld zo zijn dat er een studentenadministratiesysteem bestaat, waarmee gegevens van studenten opgevraagd kunnen worden. De gegevens die opgevraagd moeten worden zijn dan steeds hetzelfde (bijvoorbeeld naam, adres, studie), maar de student is steeds verschillend. De gebruiker van het systeem kan bijvoorbeeld uit een lijst een student kiezen en met een druk op de knop de bijbehorende gegevens opvragen. De query die daarvoor nodig is, wordt door middel van Dynamic SQL automatisch gegenereerd, afhankelijk van de gekozen student.

In SQL-92 is het `JOIN` keyword geïntroduceerd. Een `JOIN` wordt gebruikt om gegevens uit meerdere tabellen te halen. Dit was in eerdere versies van SQL ook mogelijk, zoals in onderstaand statement gebeurt.

```
SELECT Student.voornaam, Docent.voornaam
FROM Student, Docent
WHERE Student.begeleid_door = Docent.docentID
```

In SQL-92 zijn verschillende soorten joins geïntroduceerd: `INNER JOIN`, `LEFT OUTER JOIN`, `RIGHT OUTER JOIN` en `FULL OUTER JOIN`. Inner joins laten alleen die rijen zien, waarbij de `JOIN` conditie waar is. De `LEFT OUTER JOIN` laat alle gegevens uit de tabel links van de `JOIN` zien, en alleen die gegevens uit de tabel rechts van de `JOIN` waarbij de

JOIN conditie waar is. De RIGHT OUTER JOIN is juist het tegenovergestelde. Deze laat alle gegevens uit de tabel rechts van de JOIN zien, en alleen die gegevens uit de tabel links van de JOIN waarbij de JOIN conditie waar is. De FULL OUTER JOIN ten slotte laat alle gegevens uit beide tabellen zien.

Een variant op het JOIN statement is het UNION statement. Met dit statement worden gegevens uit meerdere tabellen gehaald. Deze gegevens moeten echter wel van hetzelfde type zijn (tekst, numeriek, datum). De gegevens worden eigenlijk onder elkaar geplakt.

```
SELECT  voornaam, achternaam
FROM    Studenten
UNION
SELECT  voornaam, achternaam
FROM    Docenten
```

Met deze query worden alle voor- en achternamen van zowel studenten als docenten gegeven.

Sinds SQL-92 is er een constraint genaamd de CHECK constraint. Deze constraint (beperkingsregel) wordt gebruikt in het CREATE TABLE statement om de mogelijke waarden die in een kolom ingevoerd kunnen worden, te beperken.

```
CREATE TABLE Studenten (
   studentID       integer     NOT NULL     PRIMARY KEY,
   voornaam        char(20)    NOT NULL,
   achternaam      char(50)    NOT NULL,
   woonplaats      char(50)    NOT NULL,
   leeftijd        integer     NOT NULL,
   sportkaartnr    integer     NULL,
   CONSTRAINT      volwassen   CHECK (leeftijd >= 18) )
```

In bovenstaand CREATE TABLE statement, is een CHECK constraint opgenomen, die ervoor zorgt dat de waarden in de kolom leeftijd groter of gelijk aan 18 moeten zijn.

SQL-92 kent een aantal nieuwe datatypen. Er zijn drie tijdgerelateerde datatypen bedacht. DATE bevat dag, maand en jaar. TIME bevat uren, minuten en seconden. TIMESTAMP combineert DATE en TIME en voegt daar nog nanoseconden aan toe. Het BIT datatype is een 0 of een 1.

Als toevoeging op SQL-92 werd in 1995 SQL Call-Level Interface (CLI) voltooid. Call-Level Interface wordt gebruikt voor het op een dynamische manier verkrijgen van toegang tot verschillende databases. Vaak gebeurt dit in een client/server omgeving, waarbij de applicatie op de client-computer en de database op de server staat. CLI zorgt er in een dergelijke situatie voor, dat opdrachten die gegeven worden (zoals SELECT, INSERT, etcetera) worden doorgestuurd naar de database op de server. Het standaard stappenplan bij het gebruik van een Call-Level Interface is de volgende [CLI01]:

1. De applicatie roept een CLI functie aan om toegang te krijgen tot de database
2. De applicatie bouwt een SQL statement op, en plaatst deze in een buffer. Hierna roept de applicatie één of meerdere CLI functies aan om het statement naar de database te versturen en deze uit te voeren
3. Als het statement een SELECT statement was, roept de applicatie een CLI functie aan om de resultaten van het SELECT statement in een buffer te plaatsen. Het is gebruikelijk dat er één rij of kolom per keer naar de applicatie wordt verzonden. Bij een ander statement hoeven er geen resultaten naar de applicatie te worden teruggestuurd
4. De applicatie roept een CLI functie aan om de connectie met de database te verbreken

Ten slotte werd in 1996 nog een toevoeging op de SQL-92 standaard voltooid. Dit was SQL Persistent Stored Module (PSM). PSM is een taal om stored procedures en triggers mee te schrijven. Meer hierover in de paragraaf over de SQL:1999 standaard.

4.5 SQL:1999

Na de SQL-92 standaard lagen de ontwikkelingen op databasegebied niet stil. In 1995 en 1996 werden er al toevoegingen voltooid voor de SQL-92 standaard. SQL/CLI en SQL/PSM werden ook toegevoegd aan de nieuwe SQL:1999 standaard, alias SQL3. Oorspronkelijk was deze standaard gepland voor 1996 maar de ontwikkeling van objectgeoriënteerd SQL liep vertraging op.

In SQL:1999 worden weer een aantal nieuwe datatypen geïntroduceerd. Het type BOOLEAN kan TRUE of FALSE zijn. Het datatype ARRAY is een complex datatype. Met het ARRAY type kunnen rijen van waarden direct in een tabel worden geplaatst [White99].

```
CREATE TABLE Artikelen (
   ArtikelID     INTEGER, PRIMARY KEY,
   Auteurs       VARCHAR(15) ARRAY[5],
   Titel         VARCHAR(100) )
```

Dit CREATE TABLE statement creëert een tabel met daarin een kolom Auteurs. Aangezien artikelen vaak door meerdere auteurs worden geschreven, kan deze kolom tot vijf auteurs per artikel opslaan (ARRAY[5]). Iedere auteursnaam kan uit maximaal 15 tekens bestaan (VARCHAR(15)). Met onderstaand SELECT statement kan er uit een waarde uit een ARRAY geselecteerd worden.

```
SELECT  ArtikelID, Auteurs[1] AS Auteurnaam
FROM    Artikelen
```

SQL:1999 introduceerde ook triggers en stored procedures, onderdelen van objectgeoriënteerd SQL en SQL/PSM.

Een trigger is een stuk code dat door het DBMS geactiveerd wordt indien een bepaalde operatie op de database wordt uitgevoerd, en alleen dan als een bepaalde conditie waar is. Hieronder volgt een voorbeeld van een trigger.

```
CREATE TRIGGER start_eind_onderzoek
   BEFORE INSERT, UPDATE(startdatum, einddatum) OF Onderzoek
     FOR EACH ROW
   WHEN (startdatum >= einddatum)
   BEGIN
     ROLLBACK WORK;
   END;
```

Deze trigger checkt vóór een INSERT of een UPDATE statement op de tabel Onderzoek wordt uitgevoerd, of de startdatum van het onderzoek groter is dan de einddatum. Als dit zo is, wordt de INSERT of UPDATE actie niet uitgevoerd. In alle andere gevallen (als de startdatum dus kleiner is dan de einddatum, zoals wenselijk is) wordt de INSERT of UPDATE actie wel uitgevoerd. ROLLBACK WORK is het statement dat gebruikt wordt om ervoor te zorgen dat de uitvoering van de INSERT of UPDATE actie wordt teruggedraaid en de verkeerde gegevens dus niet in de tabel terecht komen.

Een stored procedure is een stuk code dat geactiveerd wordt door deze aan te roepen vanuit een programma, een trigger of een andere stored procedure. Zowel een trigger als een stored procedure zijn stukken code die opgeslagen liggen in de database. Het belangrijke verschil tussen een trigger en een stored procedure is de manier van

aanroepen. Triggers kunnen namelijk niet direct door een programma of een andere stored procedure worden aangeroepen. Stored procedures kunnen wel op die manier aangeroepen worden. Hieronder volgt een voorbeeld van een stored procedure.

```
CREATE PROCEDURE onderzoek_verwijderen
   (ID_VAR IN INTEGER) AS
DECLARE
   VAR afgerond_VAR BOOLEAN;
BEGIN
   SELECT afgerond
   INTO afgerond_VAR
   FROM Onderzoek
   WHERE onderzoeksID = ID_VAR;

   IF   afgerond_VAR = TRUE THEN
     DELETE FROM Onderzoek
     WHERE onderzoeksID = ID_VAR;
   ENDIF;
END
```

Deze stored procedure verwijdert een onderzoek uit de tabel Onderzoek. Dit mag echter alleen als het onderzoek al is afgerond. ID_VAR is een input variabele die gevuld wordt met een onderzoeksID. Vervolgens wordt het specifieke onderzoek uit de tabel geselecteerd. Ten slotte wordt er gekeken of dat project afgerond is. Als dat zo is wordt het project verwijderd uit de tabel. Als het niet zo is wordt er niets uitgevoerd.

Er bestaan drie manieren om resultaten van stored procedures door te geven aan de omgeving [SQLTeam]. De eerste manier is door gebruik te maken van OUTPUT variabelen.

```
CREATE PROCEDURE aantalStudenten ( @Woonplaats char(50), @WPCount int OUTPUT )
AS
SELECT @WPCount = Count(*)
FROM Studenten
WHERE Woonplaats = @Woonplaats
go
```

De variabele @WPCount wordt gebruikt om het resultaat van deze stored procedure terug te geven aan het aanroepende programma. Deze stored procedure telt het aantal studenten uit een bepaalde woonplaats. Deze woonplaats wordt door het programma in de variabele @Woonplaats geplaatst. De stored procedure wordt als volgt aangeroepen:

```
DECLARE @CountStudenten INT
EXEC aantalStudenten
  @Woonplaats = 'Nijmegen',
  @WPCount = @CountStudenten OUTPUT
SELECT CountStudenten = @CountStudenten
```

Het EXEC statement wordt gebruikt om een stored procedure aan te roepen. Hierbinnen worden de variabelen gevuld. Dit voorbeeld zal het aantal studenten tellen dat in Nijmegen woont.

De tweede manier om resultaten van stored procedures door te geven aan de omgeving is door gebruik te maken van tijdelijke tabellen.

```
CREATE TABLE #Studenten (StudentID INT)

INSERT #Studenten
EXEC aantalStudenten2 @Woonplaats = 'Nijmegen'

SELECT *
FROM #Studenten

DROP TABLE #Studenten
```

Deze methode maakt een tijdelijke tabel aan, vult deze met de resultaten van de aangeroepen stored procedure, geeft de resultaten door aan de omgeving en verwijdert ten slotte de tijdelijke tabel weer.

De bijbehorende stored procedure ziet er als volgt uit:

```
CREATE PROCEDURE aantalStudenten2 ( @Woonplaats char(50) )
AS
SELECT StudentID
FROM Studenten
WHERE Woonplaats = @Woonplaats
go
```

De laatste manier om resultaten door te geven is ook de meest gelimiteerde manier.

```
CREATE PROC TestReturn (@InvoerParam INT)
AS
RETURN @InvoerParam
go
```

Het enige dat deze stored procedure doet, is de invoer variabele terug geven aan het aanroepende programma. Het script om deze stored procedure aan te roepen is de volgende:

```
Declare @UitvoerParam INT
EXEC @UitvoerParam = TestReturn 3
Select UitvoerParam =@UitvoerParam
```

Deze aanroep levert simpelweg het getal 3 op. Dit soort stored procedures worden meestal gebruikt om een programma te testen op fouten. Wanneer een fout wordt gevonden, wordt de foutcode teruggegeven aan het aanroepende programma.

SQL:1999 was de eerste SQL standaard die User-Defined Types (UDT's) ondersteunde. UDT's zijn, zoals de naam al doet vermoeden, datatypen die door de gebruiker geïntroduceerd kunnen worden. De gebruiker wordt dus in staat gesteld om de standaard verzameling datatypen uit te breiden met zelfverzonnen datatypen, gebaseerd op bestaande datatypen. Zo kan de gebruiker bijvoorbeeld een datatype `Geslacht` introduceren, gebaseerd op het datatype `CHAR`, zoals in het volgende voorbeeld is gedaan.

```
CREATE TYPE Geslacht AS CHAR(1)
```

Hier wordt het nieuwe type `Geslacht` gerepresenteerd als een `CHAR` met de lengte van één teken. In de praktijk zou het namelijk wenselijk kunnen zijn dat er in een kolom van het type `Geslacht` alleen M of V ingevuld kan worden. Dit zou middels een `CHECK` constraint in het `CREATE TABLE` statement afgedwongen kunnen worden.

```
CREATE TABLE Persoon (
   persoonID      integer     NOT NULL     PRIMARY KEY,
   voornaam       char(20)    NOT NULL,
   achternaam     char(50)    NOT NULL,
   sexe          geslacht    NOT NULL,
   CONSTRAINT     geslacht       CHECK (sexe IN ('M',
```

Er zijn twee verschillende soorten User-Defined Types; distinct types, waarvan hierboven een voorbeeld staat, en structured types, waarvan nu een voorbeeld volgt [EiMe99].

Structured User-Defined Types zijn een SQL methode om subtypering aan te brengen in een database. Dit zal verduidelijkt worden aan de hand van een voorbeeld.

```
CREATE TYPE emp_type
UNDER person_type
AS ( EMP_ID      INTEGER,
     SALARY      REAL )
INSTANTIABLE
NOT FINAL
```

In dit voorbeeld wordt er een UDT genaamd `emp_type` gecreëerd, onder het UDT `person_type`. `emp_type` is een collectie met twee velden; `emp_id` en `salary`. `Person_type` zou ook een collectie van velden kunnen zijn (bijvoorbeeld naam en adresvelden). Die velden zouden dan ook toebehoren aan alle velden van het type `emp_type`. Iedereen is bijvoorbeeld van het type `person_type`, maar alleen mensen met een baan behoren óók tot de groep `emp_type`. Van iemand met een baan wordt dan opgeslagen wat zijn naam en adres zijn, maar ook zijn personeelsnummer en salaris. Dit komt dus overeen met de subtypering uit ORM, waarin subtypen alle velden erven van het supertype, maar ook eigen velden hebben die specifiek toebehoren tot dat subtype.

`INSTANTIABLE` in het bovenstaande voorbeeld geeft aan dat er waarden van dit specifieke type gecreëerd kunnen worden. `NOT FINAL` geeft aan dat het toegestaan is dat er nog subtypen worden toegevoegd onder `emp_type`.

4.6 SQL:2003

SQL:2003 is de meest recente SQL standaard. Zoals al eerder is aangegeven, staat een nieuwe SQL standaard gepland voor 2007 of 2008. De 2003 standaard is een belangrijk ingrediënt voor dit onderzoek. In deze paragraaf zal, net als bij de andere standaarden is gebeurd, een korte weergave worden gegeven van de nieuwe onderdelen van de standaard ten opzichte van de eerdere versies. Omdat deze standaard een belangrijk onderdeel van dit onderzoek is, zal verderop in deze scriptie dieper op de standaard worden ingegaan.

Het belangrijkste nieuwe bestanddeel van SQL:2003 is SQL/XML. XML (eXtensible Markup Language) is een taal voor de representatie van gestructureerde gegevens in de vorm van platte tekst.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<course name="SQL1">
 <student>
  <voornaam>Jan</voornaam>
  <achternaam>Pietersen</achternaam>
 </student>
 <student>
  <voornaam>Klaas</voornaam>
  <achternaam>Jansen</achternaam>
 </student>
</course>
```

Hierboven staat een voorbeeld van een XML bestand. Hierin is de gegevensstructuur aangegeven van een cursus. Van de cursus met de naam “SQL1” wordt opgeslagen welke studenten die cursus volgen. Van elke student wordt de voor- en achternaam opgeslagen. XML heeft dus een duidelijk verband met SQL. Beiden kunnen worden gebruikt om structuur in gegevens aan te brengen. Een overzicht van de overeenkomsten en verschillen tussen SQL en XML is te vinden in Tabel 3. SQL/XML is in het leven geroepen om steun te bieden bij het gebruiken van XML in de context van een SQL database systeem. SQL/XML maakt het mogelijk om XML documenten op te slaan in een database. Vervolgens kunnen deze XML documenten door middel van queries opgevraagd en gewijzigd worden. Ook kunnen uit tabellen van een database, XML bestanden worden gegenereerd. De tabel zoals gedefinieerd in het CREATE TABLE statement op bladzijde 15 zou er in SQL/XML als volgt uit zien:

```
<Studenten>
<row>
<studentID>0000001</studentID>
<voornaam>Jan</voornaam>
<achternaam>Pietersen</achternaam>
<woonplaats>Enschede</woonplaats>
<sportkaartnr xsi:nil="true" />
</row>
<row>
<studentID>0000002</studentID>
<voornaam>Klaas</voornaam>
<achternaam>Jansen</achternaam>
<woonplaats>Nijmegen</woonplaats>
<sportkaartnr>12345</sportkaartnr>
</row>
.
.
</Studenten>
```

In het voorbeeld hierboven worden een `CREATE TABLE` statement en een `INSERT` statement samengevoegd. Niet alleen de structuur wordt vastgelegd, maar ook wordt er direct een vulling aan de verschillende velden toegekend.

SQL:2003 introduceert drie nieuwe datatypen: BIGINT, MULTISET en XML. Het type BIGINT is hetzelfde als een INTEGER, maar dan met een grotere precisie. Het datatype INTEGER heeft een precisie van 32 bit en de BIGINT heeft een precisie van 64 bit. Het nieuwe datatype MULTISET is afgeleid van het type ARRAY. Waar een ARRAY altijd een maximumgrootte heeft, heeft een MULTISET dit niet. Een ARRAY is ook een geordende rij, terwijl een MULTISET ongeordend is. Het laatste nieuwe datatype is XML. Dit datatype kan gebruikt worden om XML bestanden in op te slaan.

RDBMS (SQL)	XML
Wordt gebruikt voor het opslaan van grote hoeveelheden data en zorgt voor efficiënte toegang en consistentie	Wordt gebruikt als een format voor het structureren en uitwisselen van hypertext documenten
Structuur wordt bepaald aan de hand van een onderliggend database schema	Structuur wordt bepaald aan de hand van een (incompleet) onderliggend schema
Er bestaat één schema voor de hele database	Ieder XML document bevat delen van het onderliggende schema
SQL kent relaties en attributen	XML kent elementtypen en attributen
Domeinen bestaan alleen op attribuutniveau	Domeinen bestaan voor zowel elementtypen als attributen
Een waarde hoort bij één attribuut	Een waarde kan bij meerdere attributen horen
Binnen het relationele schema is de naam van een relatie uniek	Binnen het schema is de naam van een elementtype uniek, tenzij er `NAMESPACES` worden gebruikt, dan mogen de namen hetzelfde zijn
De naam van een attribuut is uniek binnen een relatie	De naam van een attribuut is uniek binnen een elementtypen
`NULL` waarden bestaan alleen op attribuut niveau	`NULL` values bestaan op attribuut- en elementtype niveau
In een RDBMS kunnen 'default values' worden aangegeven	In XML kunnen 'default values' worden aangegeven
Unieke identificatie wordt geregeld door middel van een `PRIMARY KEY` over één of meerdere kolommen	Unieke identificatie wordt geregeld door middel van een attribuut type `ID` over één attribuut
`PRIMARY KEY` is niet verplicht	Attribuut type `ID` is niet verplicht
Relaties worden gelegd door middel van `FOREIGN KEYS`	Relaties tussen elementtypen worden gelegd door middel van het attribuut type `IDREF`
Relaties en tupels hebben geen expliciete volgorde	Elementtypen hebben een expliciete volgorde

Tabel 3: RDBMS (SQL) vs XML [KKR01]

In voorgaande SQL standaarden werden er drie manieren geïntroduceerd om data in tabellen te muteren: INSERT, UPDATE en DELETE. SQL:2003 introduceert een vierde manier: MERGE. MERGE is een combinatie van INSERT en UPDATE.

Table 1 — INVENTORY table

PARTNUM	DESCRIPTION	QUANTITY
1	Cool Part	10
2	Another Cool Part	15
3	Really Cool Part	20

Table 2 — SHIPMENT table

PARTNUM	DESCRIPTION	QUANTITY
2	Another Cool Part	5
4	Yet Another Cool Part	15
1	Cool Part	10

Table 3 — INVENTORY table after MERGE

PARTNUM	DESCRIPTION	QUANTITY
1	Cool Part	20
2	Another Cool Part	20
3	Really Cool Part	20
4	Yet Another Cool Part	15

Figuur 4: MERGE **voorbeeld [EMeKMiZ]**

Zoals in de voorbeeldtabellen in Figuur 4 te zien is, komen de eerste en de derde rij uit de SHIPMENT tabel ook voor in de INVENTORY tabel. Deze twee rijen zullen door het MERGE statement geüpdate worden en de derde rij zal toegevoegd worden.

Het MERGE statement dat nodig is om het resultaat uit Figuur 4 te bewerkstelligen is het volgende:

```
MERGE  INTO INVENTORY AS INV
USING  (SELECT PARTNUM, DESCRIPTION, QUANTITY FROM SHIPMENT)AS SH
ON     (INV.PARTNUM = SH.PARTNUM)
WHEN MATCHED THEN UPDATE
SET    QUANTITY = INV.QUANTITY + SH.QUANTITY
WHEN NOT MATCHED THEN
INSERT (PARTNUM, DESCRIPTION, QUANTITY)
VALUES (SH.PARTNUM, SH.DESCRIPTION, SH.QUANTITY)
```

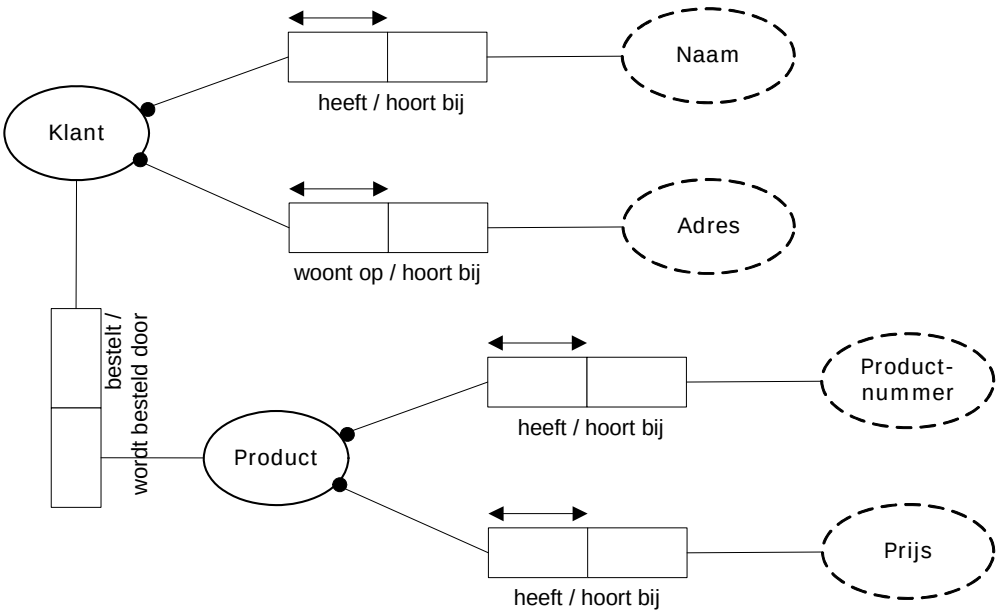
SQL:2003 wordt door geen enkel RDBMS volledig ondersteund. In plaats hiervan ondersteunen de verschillende bedrijven een deel van de standaard plus eigen functionaliteiten. Desalniettemin is het belang van een standaard groot. Doordat elk RDBMS aan een groot deel van de standaard voldoet, kunnen verschillende systemen prima met elkaar communiceren. Het versturen van gegevens naar een ander systeem behoort daardoor tot de mogelijkheden. Standaarden zijn voor de consument ook van belang. Doordat een SQL standaard wordt geaccepteerd door organisaties als het ISO, die geen commercieel belang bij SQL hebben, en de standaard wordt ontwikkeld door werkgroepen vol met experts van veel verschillende organisaties, wordt een standaard een kwaliteitskarakteristiek. Als een bedrijf een RDBMS ontwikkeld met daarin SQL dat voldoet aan de ANSI/ISO standaard, kan de klant ervan uitgaan dat het product op SQL gebied kwalitatief goed is. De kwaliteit van een product wordt ook vergroot door de concurrentie tussen verschillende RDBMS ontwikkelaars. Aangezien een consument eerder een RDBMS aanschaft dat aan een standaard voldoet, zullen RDBMS ontwikkelaars moeten zorgen dat hun product ook aan de standaard voldoet. Zo kan een consument een keuze maken tussen een aantal kwalitatief goede producten, en wordt de keuze uiteindelijk gemaakt aan de hand van de eisen die de consument aan de functionaliteiten van het product stelt.

5. Modellen

Omdat er in dit onderzoek een belangrijke rol is weggelegd voor verschillende soorten modellen, zullen deze modellen in dit hoofdstuk kort worden besproken.

5.1 Conceptuele modellen

Bij het ontwerpen van een database, worden conceptuele modellen vaak gebruikt om de database structuur te bepalen. In een systeem ontwikkeltraject spelen de domeinexpert (meestal de klant) en de databaseontwerper belangrijke rollen. De domeinexpert weet alles over het domein (Universe of Discourse) waarvoor het systeem gebouwd dient te worden. De databaseontwerper zal door middel van interviews alle belangrijke informatie over het domein moeten bemachtigen om zo relaties tussen onderdelen uit het domein te kunnen leggen. Als alle informatie is bemachtigd, kan de databaseontwerper een conceptueel schema creëren. Uit dit conceptuele schema zal later in het systeem ontwikkeltraject de databasestructuur (tabellen, kolommen, relaties en constraints) worden gegenereerd. Een belangrijk aspect bij het gebruik van conceptuele modellen is dat de modellen vanzelfsprekend moeten zijn. Ze moeten voor de domeinexpert (een leek op het gebied van modelleren) zonder al te veel moeite te begrijpen zijn. De domeinexpert moet namelijk in staat zijn fouten in het conceptuele model op te sporen en zo het model te verbeteren tot het model aan de wensen van de domeinexpert voldoet. In Figuur 5 staat een voorbeeld van een conceptueel (ORM) model.



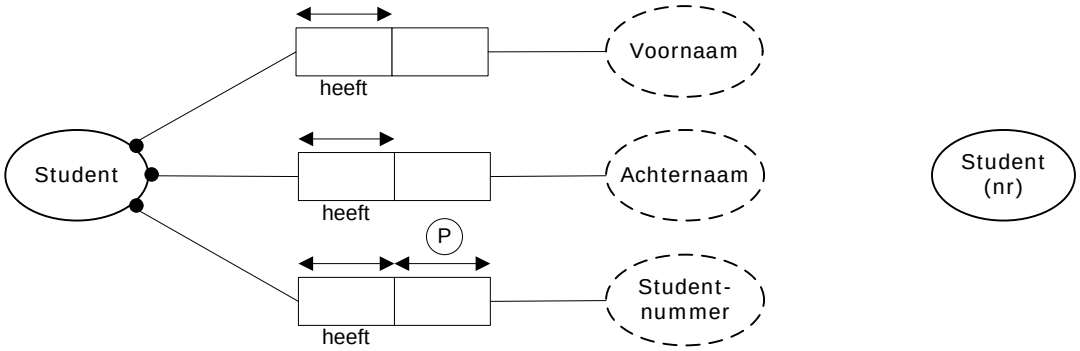
Figuur 5: Voorbeeld conceptueel model van een bestellingsysteem [ModMet]

Het bestellingsysteem draait om klanten die producten bestellen. Een klant heeft een naam en een adres, zoals te zien is aan de twee feittypen tussen het objecttype klant (de gesloten cirkel) en de labeltypen naam en adres (de onderbroken cirkels). De labeltypen zijn attributen van een objecttype.

De dubbele pijlen boven een rol (uniqueness constraint) houden in dat de inhoud van die rol uniek moet zijn. Dit heeft als gevolg dat in Figuur 4 een klant maar één naam kan hebben en maar op één adres kan wonen. Iedere klant mag echter maar één keer in die rollen voorkomen. Hetzelfde geldt voor een product, dat maar één productnummer en één prijs mag hebben.

De dichte bolletjes bij de objecttypen (total role constraint) ten slotte, betekenen dat de volledige inhoud van het objecttype aan de rol moet deelnemen. Iedere klant die aanwezig is in het objecttype klant moet dus een naam en een adres hebben. Hetzelfde geldt voor de producten: alle producten hebben een productnummer en een prijs. In combinatie met de uniqueness constraints betekent het dus, dat elke klant maar één naam en één adres mag hebben en ieder product maar één productnummer en één prijs mag hebben.

In ORM modellen wordt soms een 'primary uniqueness constraint' gebruikt, zoals in Figuur 6 wordt weergegeven door middel van een omcirkelde 'P'.



Figuur 6: ORM primary uniqueness constraint

Als een objecttype meerdere labeltypen heeft, geeft de P aan dat het objecttype door middel van dat labeltype geïdentificeerd wordt. In dit geval staat de P bij het labeltype studentnummer en betekent dat hetzelfde als het opnemen van nr in het objecttype zoals aan de rechterkant is weergegeven.

Voor een verdere uitleg van de gebruikte symbolen in dit model wordt u verwezen naar Appendix B.

5.2 Metamodellen

Een metamodel is een model van een model. Het beschrijft de structuur en eigenschappen van een modelleertaal. In een metamodel van bijvoorbeeld ORM wordt beschreven waar ORM modellen aan moeten voldoen. De metamodellen van ORM en SQL zullen in de volgende hoofdstukken worden beschreven en zijn te vinden in Appendices C en D.

6. Metamodellen

De titel van deze scriptie is “Conceptuele modellen versus SQL 2003”. Zoals deze titel al doet vermoeden zal de SQL:2003 standaard naast conceptuele modellen worden gelegd om zo overeenkomsten en verschillen op te sporen. Dit alles wordt gedaan om te proberen de SQL standaard en conceptuele modellen dichter tot elkaar te brengen.

In dit hoofdstuk zal een begin gemaakt worden met de vergelijking. Dit zal gedaan worden door de metamodellen van SQL en ORM (Object Role Modeling) te vergelijken. De keuze voor een vergelijking op metamodel niveau is gemaakt omdat op deze manier op hoog niveau de structuren van beide modellen vergeleken kunnen worden.

Allereerst zullen nu de twee metamodellen kort beschreven worden en in de volgende hoofdstukken zal de vergelijking uitgevoerd worden.

6.1 Object Role Modeling (ORM) metamodel

ORM is een methode voor het modelleren van informatiesystemen op het conceptuele niveau [Halp98]. In Europa wordt deze methode ook vaak NIAM (Natural Language Information Analysis Method) genoemd. Object Role Modeling dankt zijn naam aan het feit dat het de wereld ziet als een verzameling objecten (entiteiten of waarden) die bepaalde rollen spelen (onderdelen van relaties). Voor een legenda van de gebruikte symbolen in ORM, wordt u verwezen naar Appendix B.

Dr. Terry Halpin, een belangrijke figuur in de ontwikkeling van ORM, doet in [Halp00] een suggestie voor een metamodel van ORM. Dit metamodel zal in deze scriptie gebruikt worden als vergelijkingsmateriaal met het SQL metamodel. Halpin stelt een metamodel voor dat bestaat uit drie onderdelen. Het eerste metamodel beschrijft de objecttypen en relaties. Het tweede metamodel beschrijft objecttypen, predikaten en rollen en het derde metamodel beschrijft de ORM constraints. De ORM metamodellen zijn te vinden in Appendix D.

6.1.1 Beschrijving ORM metamodellen

Het ORM metamodel is opgesplitst in drie submodellen. Hieronder volgt een beschrijving van elk van deze submodellen [Halp00].

Labeltypen en objecttypen worden soms ook lexicale of niet-lexicale object typen genoemd. Waarden zoals nummers, zijn constanten met een standaardnotatie. Objecten zoals personen of landen hebben een referentieschema nodig om in natuurlijke taal geïdentificeerd te worden door middel van bepaalde waarden (bijvoorbeeld Student met studentnummer 12345). Dit kan gedaan worden door een object te relateren aan één waarde. Dit wordt dan weergegeven door deze waarde tussen haakjes in het objecttype te plaatsen, zoals in Figuur 37 gedaan is met het objecttype ObjectType.

In ORM Metamodel 1 wordt Relationship gebruikt om een ‘relationship type’ aan te geven. Een relationship type bestaat uit de objecttypen en een predicaat (bijvoorbeeld Student heeft Naam).

Een objecttype is ‘independent’ (onafhankelijk) als instanties van dit objecttype kunnen bestaan zonder onderdeel te zijn van een feittype.

In ORM Metamodel 2 worden objecttypen, predikaten en rollen beschreven. Elk objecttype heeft precies één naam, dat uniek moet zijn binnen het schema. Een rol is een deel van een relatie. Elke rol wordt geïdentificeerd door middel van een nummer. Een rol heeft ook een positie. In een binair feittype (dat dus bestaat uit twee rollen), kan de positie één of twee zijn.

Elke relatie wordt geïdentificeerd met een nummer. Elke relatie heeft ook een 'relationship reading', de manier waarop de relatie gelezen dient te worden. Deze readings staan vaak boven of onder een feittype, zoals in Figuur 38 gedaan is bij het feittype tussen ObjectType en Role. Hier staat plays/is played by. Van links naar rechts staat er dan 'ObjectType plays Role' en van rechts naar links 'Role is played by ObjectType'.

Sommige versies van ORM ondersteunen alleen binaire relaties. Dit maakt de ORM modellen minder complex en wordt het gemakkelijker om ORM tools te maken. Aan de andere kant is het voor domeinexperts moeilijker om modellen te maken en te valideren, aangezien veel feittypen die oorspronkelijk unaire feittypen (één rol), ternaire feittypen (drie rollen) of nog grotere feittypen waren, omgezet moeten worden naar binaire feittypen. De resultaten zijn vaak minder natuurlijk als de oorspronkelijke situatie.

Onder ORM Metamodel 2 staan drie regels (R1 tot en met R3). Binnen een relatie moeten de rolnamen uniek zijn. Over een heel schema mogen rollen wel dezelfde naam hebben. Rollen in dezelfde relatie worden 'coroles' genoemd. Een 'far role' van een objecttype is de rol die het verst van het objecttype afstaat. In een binair feittype is dat de twee rol vanaf het objecttype gelezen. Bij een ternair feittype zijn het de twee verste rollen. Regel 1 geeft aan dat de rolnamen van de far roles uniek moeten zijn. Een voorbeeld van een rolnaam is te zien in Figuur 38. Hier staat het woord 'player' tussen blokhaken onder de eerste rol van het feittype tussen ObjectType en Role. Rolnamen worden wegens ruimtegebrek vaak niet weergegeven in een model. Regel 2 zegt dat elke relatie ten minste één 'reading' moet hebben. Regel 3 ten slotte, zegt dat rollen die dezelfde PredicateText hebben (onderdeel van een reading), ook hetzelfde moeten zijn. Met andere woorden: twee rollen in dezelfde relatie mogen niet dezelfde PredicateText hebben.

Het laatste ORM metamodel beschrijft de ORM constraints. De legenda onder het metamodel laat alle soorten constraints zien en het metamodel beschrijft in welke situaties een constraint wordt gebruikt.

Onder de metamodellen staan zogenaamde subtype definiërende regels. Deze regels geven aan in welke situatie de populatie van een objecttype tot een subtype van dat objecttype behoort. De subtype definiërende regels zijn opgeschreven in de formele notatie van de taal FORML (Formal Object Role Modeling Language). Deze taal wordt gebruikt in het programma VisioModeler [Visio97].

6.2 SQL metamodel

Als tegenhanger van het ORM metamodel in de vergelijking, is er een SQL metamodel nodig. Eigenlijk zou deel 11 van de SQL:2003 standaard beschouwd kunnen worden als het metamodel van SQL. Dit deel, SQL/Schemata, telt meer dan 300 pagina's tekst en beschrijft alle elementen van SQL. Omdat dit deel zo uitgebreid en in tekstvorm is, is een overzicht van alle elementen in modelvorm gewenst. In [CRBaBrP] geven Calero, Ruiz, Baroni, Brito e Abreu en Piattini een metamodel voor SQL in modelvorm. Dit model wordt onderverdeeld in vijf delen. Het eerste metamodel beschrijft de SQL datatypen. Het tweede metamodel geeft een algemene weergave van de SQL schema objecten. Metamodel drie beschrijft het schema object tabel. Metamodel vier beschrijft het schema object constraints en het vijfde metamodel beschrijft het schema object kolom. Deze metamodellen zijn te vinden in Appendix E.

6.2.1 Beschrijving SQL metamodellen

Het SQL metamodel is opgesplitst in vijf submodellen. Hieronder volgt een beschrijving van elk van deze submodellen [CRBaBrP].

SQL Metamodel 1 laat drie verschillende datatypen zien: Constructed, Predefined en UserDefined Types (UDT's). De Constructed datatypen kunnen Composite Types (samengestelde typen) en Reference Types (refererende typen) zijn. De Composite Types kunnen vervolgens weer opgedeeld worden in Collection Types (Arrays en Multisets) die bestaan uit elementen en Row Types die bestaan uit velden. In collecties hebben alle elementen hetzelfde datatype, daarom wordt in dit model het datatype gedefinieerd voor de collectie, en niet voor het element. Bij rijen is dit juist het tegenovergestelde. De velden waaruit een rij bestaat mogen verschillende datatypen hebben, vandaar dat de datatypen gedefinieerd worden voor een veld en niet voor een rij. UDT's zijn typen die door de gebruiker zelf aangemaakt kunnen worden, naast de standaard datatypen van SQL. UDT's worden onderverdeeld in Structured Types en Distinct Types. Distinct Types hebben een voorgedefinieerd datatype. Structured Types worden samengesteld uit één of meer attributen en optionele methoden. Elk attribuut heeft één datatype.

SQL Metamodel 2 beschrijft vier verschillende Schema Objecten: tabellen, constraints, domeinen en UDT's. Tabellen worden samengesteld uit kolommen, die elk een datatype hebben. Een kolom kan ook gedefinieerd worden door een domein. Ook elk domein heeft een datatype. Een Catalog is een verzameling schema's. Een SQL omgeving kan meerdere catalogs bevatten die elk weer uit verschillende schema's kunnen bestaan. Gedetailleerde informatie over tabellen, constraints en kolommen staan in SQL Metamodel 3 tot en met 5.

SQL Metamodel 3 laat de hiërarchieën binnen tabellen zien. Aan de ene kant kan een tabel een tijdelijke tabel, een afgeleide tabel (inclusief views) of een basistabel zijn. Aan de andere kant zijn er Typed en Not Typed tabellen. Een Typed Table is een specialisatie van een basistabel of een view, maar niet allebei tegelijk. Een typed table is van een bepaald StructuredType. Dit type is een UDT.

SQL Metamodel 4 laat zien dat constraints binnen SQL onder te verdelen zijn in Assertions (bepaalde eisen waar de data aan moet voldoen), Table Constraints (constraints die gelden voor een tabel) en Domain Constraints (constraints die gelden voor een domein). Een tabel constraint kan een Check Constraint, een Unique Constraint (hieronder vallen ook de Primary Keys) of een Referential Constraint zijn. Een Unique Constraint geldt voor één of meerdere kolommen terwijl een Referential Constraint voor één kolom geldt. Een Base Table heeft één of meer Candidate Keys en elk van deze kandidaatsleutels heeft een Unique Constraint.

Het laatste SQL Metamodel, nummer 5, laat de details betreffende kolommen zien. Er wordt onderscheid gemaakt tussen drie soorten kolommen: een Identity Column als het de kolom is die wordt gebruikt als identificerende kolom, een Generated Column als de waarden binnen die kolom uit andere kolommen worden gehaald en een Unique Column als er op die kolom een Unique Constraint van toepassing is. Een kolom bestaat uit een aantal eigenschappen. Een self-referencing column is een unieke kolom in een Typed Table.

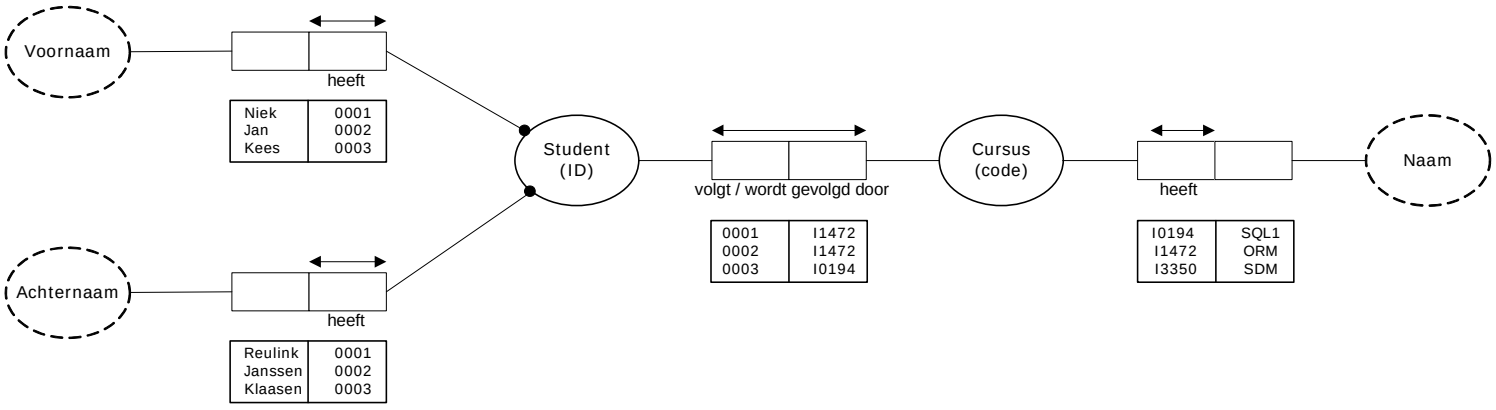
7. Overeenkomsten metamodellen

In dit hoofdstuk worden de overeenkomsten tussen de twee metamodellen stuk voor stuk behandeld. De verschillen tussen de metamodellen worden in het volgende hoofdstuk behandeld. De metamodellen zullen niet als geheel met elkaar worden vergeleken, maar er zullen steeds kleinere delen van de metamodellen met elkaar worden vergeleken.

7.1 Relaties en datastructuren

In deze paragraaf zullen de relaties en datastructuren uit ORM bekeken worden en zal er een vergelijking gemaakt worden met de relaties en datastructuren uit SQL.

Veel modelleertalen leggen relaties tussen entiteiten. Een ORM model presenteert de rollen die entiteiten en attributen spelen in de organisatiestructuur. In het ORM schema in Figuur 7 is goed te zien dat de feittypen in ORM gebruikt worden om de relaties tussen objecttypen en labeltypen te leggen.



Figuur 7: ORM schema (relaties en tabellen)

De datastructuren die we kennen uit SQL, zien we dus terug in ORM. De tabellen die we aanmaken door middel van SQL statements en waar we onze gegevens in bewaren in het RDBMS, zijn te vergelijken met de feittypen in ORM (Relations in het Metamodel). Deze feittypen kunnen namelijk gevuld worden met een voorbeeldpopulatie, zie Figuur 7 hierboven, waardoor er eigenlijk al een tabel ontstaat.

Als we het ORM schema hierboven zouden vertalen naar tabellen in een RDBMS, zou er een tabel `Student` bestaan, met de kolommen `ID`, `Voornaam` en `Achternaam`. Een student wordt geïdentificeerd met een `ID`, zeg maar een studentnummer. Dit zou de `PRIMARY KEY` worden. Zoals in SQL Metamodel 4 te zien is, kunnen er constraints op tabellen liggen en is een `PRIMARY KEY` daar een voorbeeld van. Er zou ook een tabel `Cursus` bestaan, met de kolommen `Code` en `Naam`. In deze tabel zou `Code` de `PRIMARY KEY` worden, aangezien een cursus hiermee wordt geïdentificeerd. Er zou ook nog een extra tabel aangemaakt worden met een naam als `Student_volgt_Cursus`, om de relatie tussen een `Student` en een `Cursus` te kunnen leggen. Dit is nodig omdat de tabellen `Student` en `Cursus` geen gemeenschappelijke kolommen hebben waardoor er geen directe relatie mogelijk is.

Vertaald naar SQL zouden de tabellen met de volgende CREATE TABLE statements gegenereerd kunnen worden:

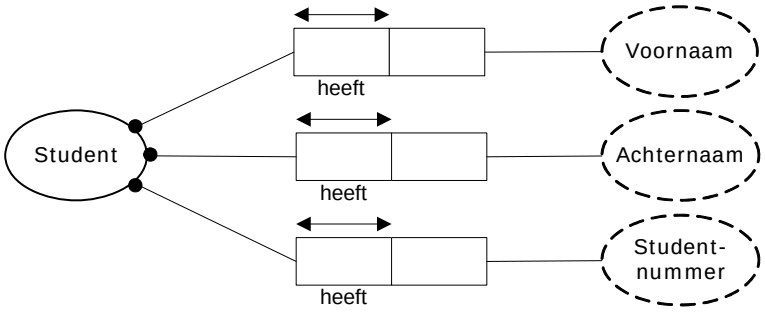
```
CREATE TABLE Student (
   ID            CHAR(10)        NOT NULL     PRIMARY KEY,
   Voornaam      CHAR(20)        NOT NULL,
   Achternaam    CHAR(50)        NOT NULL )

CREATE TABLE Cursus (
   Code          CHAR(10)        NOT NULL     PRIMARY KEY,
   Naam          CHAR(50)        NOT NULL )

CREATE TABLE Student_volgt_Cursus (
   StudentID     CHAR(10)        NOT NULL     PRIMARY KEY,
   Cursuscode    CHAR(10)        NOT NULL     PRIMARY KEY)
```

Dit zou er in de grafische weergave van Microsoft Access als volgt uit zien:

Figuur 8: Relaties in grafische weergave

Het feittype tussen Student en Cursus, legt de relatie tussen de tabellen Student en Cursus, terwijl de andere feittypen in dit model worden gebruikt om bepaalde attributen aan een Student of Cursus toe te kennen. In dit model is dus eigenlijk alleen het middelste feittype te vergelijken met een relatie tussen twee tabellen in een RDBMS. Het is dus niet zo dat een feittype altijd één-op-één te vergelijken is met een relatie in een RDBMS.

Het ORM model in Figuur 9 geeft aan dat een Student een Voornaam, een Achternaam en een studentnummer heeft. Dit zijn in deze situatie labeltypen. Een labeltype uit ORM is ook te vertalen naar een kolom van een tabel in SQL. In Figuur 9 is Student de tabel en ieder labeltype dat aan het objecttype Student is gekoppeld, is een kolom in de tabel Student. Hieruit volgt dus automatisch ook, dat een objecttype in ORM het uitgangspunt is voor een tabel in SQL. Het objecttype zal namelijk samen met alle bijbehorende labeltypen een tabel gaan vormen.



Figuur 9: Objecttype met labeltypen

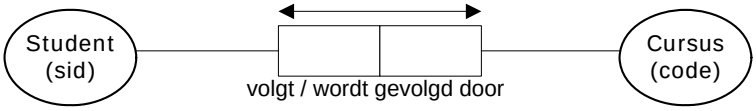
De rijen van een tabel in SQL ten slotte, zijn te vergelijken met de tupels van een feittype. De populatie van een feittype bestaat uit een aantal tupels die dus de rijen van dat feittype voorstellen.

7.2 Constraints

Een andere overeenkomst tussen de metamodellen van ORM en SQL is dat ze beiden gebruik maken van constraints om te zorgen dat de data in de uiteindelijke database 'schoon' blijft. Hieronder zullen de constraints van ORM en SQL met elkaar vergeleken worden.

7.2.1 Uniqueness constraint

Eén van die constraints, is de uniqueness constraint. Een voorbeeld daarvan in ORM staat in Figuur 10. De uniqueness constraint is ook terug te vinden in ORM Metamodel 3 en SQL Metamodel 4.

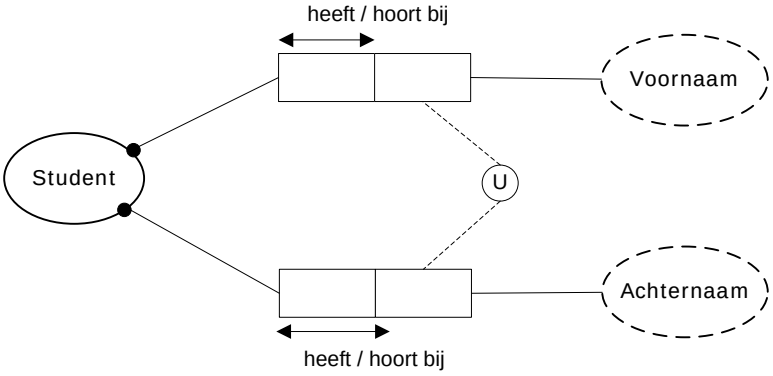


Figuur 10: ORM uniqueness constraint

Bovenstaande uniqueness constraint geeft aan dat de combinatie van student en cursus maar één keer voor mag komen (de combinaties van waarden moeten dus uniek zijn). Een student mag dus wel meerdere cursussen volgen en een cursus mag ook door meerdere studenten gevolgd worden, maar een student mag niet twee keer dezelfde cursus volgen.

SQL bevat ook keywords die een uniqueness constraint als hierboven kunnen verwezenlijken. Allereerst is er de PRIMARY KEY. Als er een PRIMARY KEY op een kolom wordt aangebracht, betekent dit automatisch dat alle waarden in die kolom uniek moeten zijn. Het is ook mogelijk om, zoals in Figuur 10 hierboven, een uniqueness constraint over meerdere kolommen te plaatsen door de PRIMARY KEY ook over meerdere kolommen te plaatsen. De CANDIDATE KEYS uit SQL zijn kolommen binnen een tabel waarvan de waarden altijd uniek zijn. Dit wordt in SQL Metamodel 4 ook gevisualiseerd door de CANDIDATE KEYS en de PRIMARY KEYS te koppelen aan de UNIQUE CONSTRAINT. Binnen SQL bestaan ook nog zogenaamde ALTERNATE KEYS. Het verschil tussen CANDIDATE KEYS en ALTERNATE KEYS heeft alles te maken met de keuze van de PRIMARY KEY(s). Als er binnen een tabel nog geen PRIMARY KEY is aangewezen, zijn alle unieke kolommen CANDIDATE KEYS. Ze zijn immers uniek en dat maakt ze kandidaat om als PRIMARY KEY dienst te doen. Als de PRIMARY KEY is gekozen, worden alle CANDIDATE KEYS die niet zijn gepromoveerd tot PRIMARY KEY, ALTERNATE KEY genoemd. De ALTERNATE KEYS zijn niet opgenomen in het SQL Metamodel, om de eenvoudige reden dat een ALTERNATE KEY en een CANDIDATE KEY hetzelfde zijn. Ze verschillen alleen in naam.

Naast de uniqueness constraint zoals weergegeven in Figuur 10, kent ORM ook externe uniqueness constraints. Deze constraints worden aangebracht tussen twee feittypen, zoals te zien is in Figuur 11.



Figuur 11: ORM externe uniqueness constraint

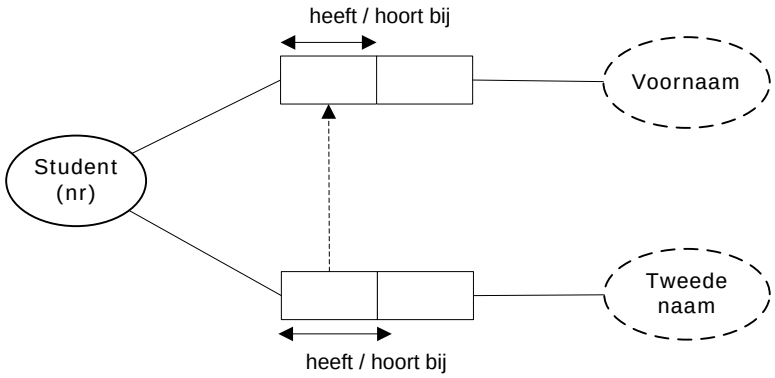
Deze uniqueness constraint zorgt ervoor dat de combinatie van voor- en achternaam per student uniek is. In SQL is dit ook net als bij de uniqueness constraint over meerdere rollen, te verwezenlijken door een `PRIMARY KEY` over meerdere kolommen aan te brengen. In deze situatie zou er een tabel `Student` zijn, met de kolommen `Voornaam` en `Achternaam`, waarbij er een `PRIMARY KEY` over beide kolommen ligt.

In SQL kunnen ook indexen op tabellen aangemaakt worden. Deze indexen zijn bedoeld om het zoeken in tabellen te versnellen en ze worden dan ook vaak toegevoegd aan tabellen waar veelvuldig queries op uitgevoerd moeten worden. In deze indexen kan ook aangegeven worden dat gegevens in een kolom uniek moeten zijn. Dit kan door middel van het volgende SQL statement.

```
CREATE UNIQUE INDEX ind_studenten
ON Studenten (StudentID)
```

7.2.2 Subset constraint

Een subset constraint wordt gebruikt om aan te geven dat de populatie van een bepaalde rol een deelverzameling is van de populatie van een andere rol. Een voorbeeld hiervan is te zien in Figuur 12.



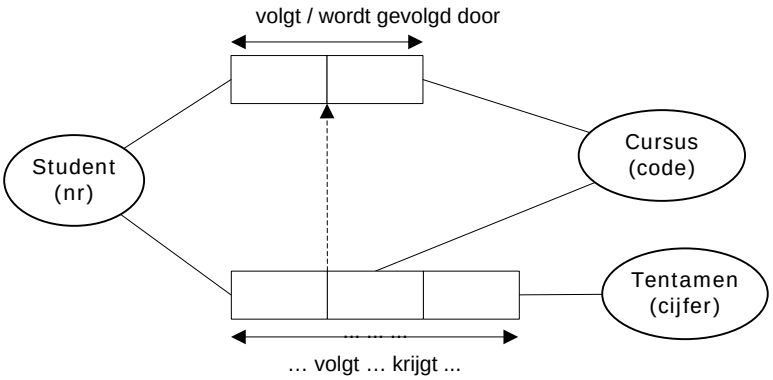
Figuur 12: ORM subset constraint

In deze figuur is de subset constraint gevisualiseerd door de gestippelde lijn tussen de eerste rollen van beide feittypen. De rol aan de kant van de pijlpunt is de superset. In deze specifieke situatie wil dit zeggen dat een student alleen een tweede naam kan hebben als hij ook een voornaam (eerste naam) heeft. Het aantal studenten met een tweede naam is dus maximaal evenveel als het aantal studenten met een voornaam, maar zal in de meeste gevallen minder zijn. De studenten die een rol spelen in het onderste feittype spelen ook een rol in het bovenste feittype. Andersom geldt dit niet. Iemand met een voornaam hoeft niet persé een tweede naam te hebben.

Als bovenstaand ORM model vertaald zou worden naar SQL zou er een tabel `Student` ontstaan met daarin de kolommen `Voornaam` en `Tweede_Naam`. Beide kolommen zijn niet verplicht, aangezien er geen totale rol constraints (zwarte bolletjes) aanwezig zijn.

```
CREATE TABLE Student (
   nr            integer     NOT NULL,        PRIMARY KEY,
   voornaam      char(20)    NULL,
   tweede_naam   char(50)    NULL,
   CONSTRAINT    isnull      CHECK (
   voornaam IS NULL AND tweede_naam IS NULL)
   OR (voornaam IS NOT NULL AND tweede_naam IS NOT NULL)
   OR (voornaam IS NOT NULL AND tweede_naam IS NULL) )
```

Dit `CREATE TABLE` statement bevat een uitgebreide `CHECK CONSTRAINT` die ervoor moet zorgen dat ORM subset constraint uit Figuur 12 geïmplementeerd wordt in de database. Er zijn vier mogelijke situaties met betrekking tot de `NULL` waarden in deze tabel. De enige die daarvan niet mag is de situatie waarbij `voornaam NULL` is en `tweede_naam NOT NULL`. Dit zou namelijk betekenen dat een student geen voornaam heeft maar wel een tweede naam. De `CHECK CONSTRAINT` controleert de gegevens alvorens ze ingevoerd worden in de tabel. De drie situaties die wél mogen staan vermeld in de constraint. Als de invoer niet aan één van deze drie situaties voldoet, wordt de invoer afgekeurd. De `CHECK CONSTRAINT` staat vermeld in SQL Metamodel 4.



Figuur 13: ORM subset constraint (complex)

Figuur 13 laat een complexe variant van de subset constraint zien, waarbij de constraint geldt voor een combinatie van rollen. Een student volgt een cursus en maakt hier een tentamen voor. De subset constraint wordt nu gebruikt om aan te geven dat een student alleen een tentamen voor een cursus mag maken als hij de cursus ook gevolgd heeft. De combinatie student-cursus in het ternaire feittype moet dus een subset zijn van de combinatie student-cursus in het binaire feittype.

Deze complexe situatie is toch eenvoudig te implementeren in SQL. Het kan zelfs allemaal in één tabel worden geplaatst. Het SQL commando om deze tabel te creëren is de volgende:

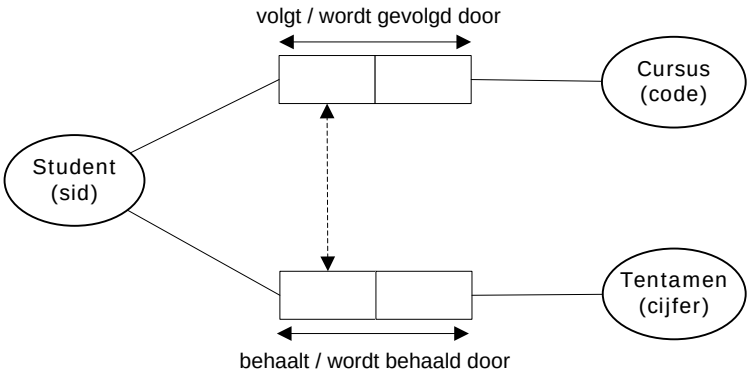
```
CREATE TABLE Student (
   nr            integer     NOT NULL
   cursuscode    char(20)    NOT NULL,
   cijfer        char(50)    NULL,
   PRIMARY KEY (nr, cursuscode)
```

Er wordt in deze tabel een primaire sleutel over de kolommen nr en cursuscode gelegd, zodat de combinatie van deze twee kolommen uniek moet zijn. Dit heeft tot gevolg dat de unieke combinatie van alle drie de kolommen, zoals het ternaire feittype in Figuur 13 voorschrijft, ook automatisch afgedwongen wordt. In de kolom tentamencode mogen wel dezelfde waarden voorkomen, maar omdat de combinatie nr-cursuscode maar eenmalig mag voorkomen, kan een student nooit voor hetzelfde vak meerdere cijfers halen.

De subset constraint wordt ook automatisch afgedwongen. De kolom cijfer mag NULL waarden bevatten, zodat bij het begin van een cursus alle deelnemende studenten al aan de tabel toegevoegd kunnen worden. De tentamencijfers zijn dan nog niet bekend dus kan die kolom leeg blijven. Op het moment dat de tentamencijfers bekend zijn, kunnen door middel van een UPDATE statement alle tentamencijfers aan de tabel worden toegevoegd. Het kan in deze situatie nooit voorkomen dat er een student aan de tabel wordt toegevoegd die wél een cijfer heeft, maar waarbij de kolom cursuscode leeg is. De subset constraint geldt dus altijd.

7.2.3 Equality constraint

De equality constraint geeft aan dat de populaties van twee of meerdere rollen gelijk moeten zijn. In ORM ziet dit er als volgt uit:



Figuur 14: ORM equality constraint

Bovenstaand ORM model is een uitbreiding op het model in paragraaf 7.2.1. De toegevoegde equality constraint (de onderbroken dubbele pijl) geeft aan dat alle studenten die meedoen aan een cursus ook mee moeten doen aan het tentamen. Uiteraard volgt hier uit dat studenten die deelnemen aan het tentamen, ook de cursus hebben gevolgd. De populaties van beide rollen zijn dan namelijk gelijk.

SQL kent geen equality constraint zoals in ORM gebruikt kan worden. Daarom is de equality constraint ook niet opgenomen in het SQL Metamodel. Dit betekent echter niet dat in SQL een equality constraint niet verwezenlijkt kan worden. Een equality constraint is namelijk hetzelfde als twee subset constraints in tegenovergestelde richting. In SQL kan dit door middel van een `INNER JOIN` tussen twee tabellen gecreëerd worden. De `INNER JOIN` wordt ook vaak `EQUI-JOIN` genoemd, omdat hij gebruikt wordt om gegevens te selecteren uit meerdere tabellen op basis van gemeenschappelijke waarden.

Stel dat bovenstaande figuur resulteert in twee tabellen. Eén met alle studenten met hun vakken, en één van alle studenten met hun cijfers. Een INNER JOIN kan nu gebruikt worden om de cijfers aan de vakken te koppelen.

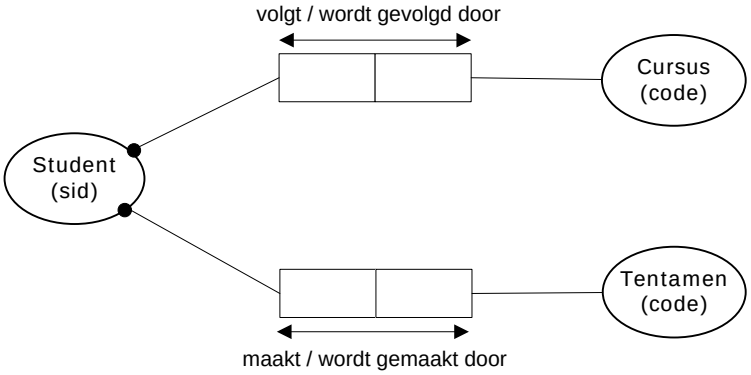
```
SELECT  Tentamen.cijfer, Cursus.Code
FROM    Tentamen, Cursus
WHERE   Cursus.sid = Tentamen.sid
```

Dit SQL statement zal alle combinaties van vakken en cijfers laten zien.

In de volgende paragraaf zal er nog een manier om een equality constraint te verwezenlijken in SQL worden besproken.

7.2.4 Total role / mandatory constraint

De total role of mandatory constraint geeft aan dat de gehele populatie van een objecttype een bepaalde rol moet spelen.



Figuur 15: ORM total role constraint

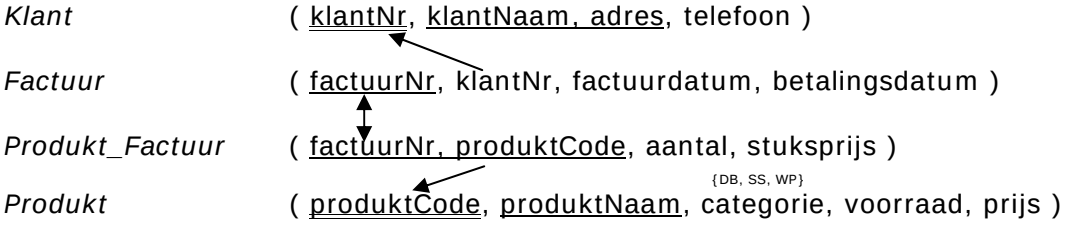
De total role constraint wordt in ORM aangegeven door een zwart bolletje op de betreffende rol te plaatsen. Figuur 15 zegt dus dat alle studenten minimaal één cursus moeten volgen en dat alle studenten minimaal één tentamen moeten maken. De twee total role constraints bereiken hetzelfde als de equality constraint. Daarom is die laatste in bovenstaande figuur weggelaten.

In SQL zien we de total role constraint gedeeltelijk terug in de vorm van het `NOT NULL` statement. Alle kolommen die als constraint `NOT NULL` hebben meegekregen in SQL, moeten verplicht gevuld zijn. De total role constraint is ook ingebakken in een `PRIMARY KEY`, want naast het feit dat een `PRIMARY KEY` een unieke waarde moet zijn, moet dit veld ook verplicht gevuld zijn. `NOT NULL` dekt echter de total role constraint niet helemaal af, omdat de total role constraint ook verplicht dat de gehele populatie van een objecttype meespeelt in de rol. In SQL is dit vrij lastig te verwezenlijken.

Zoals hierboven al is aangegeven, komen de twee total role constraints uit Figuur 15 overeen met een equality constraint. Het doel van de total role constraint in deze situatie is ervoor te zorgen dat de populatie van een kolom in de ene tabel gelijk is aan de corresponderende kolom in de andere tabel. Hieronder volgt een SQL script dat een dergelijke situatie creëert [Halp01].

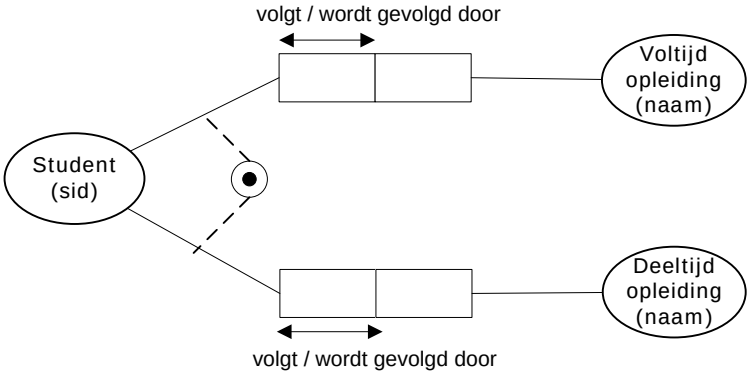
```
CREATE TABLE Klant (
 klantNr         smallint         NOT NULL     PRIMARY KEY,
 klantNaam       varchar(20)      NOT NULL,
 adres           varchar(40)      NOT NULL,
 telefoon        varchar(10),
 UNIQUE(klantNaam, adres) )

CREATE TABLE Produkt (
 produktCode     char(4)          NOT NULL     PRIMARY KEY,
 produktNaam     varchar(20)      NOT NULL     UNIQUE,
 categorie       char(2)          NOT NULL,
 voorraad        smallint         NOT NULL,
 prijs           decimal(6,2)     NOT NULL,
 CHECK           (categorie IN ('DB', 'SS', 'WP') )


CREATE TABLE Factuur (
 factuurNr       smallint         NOT NULL     PRIMARY KEY,
 klantNr         smallint         NOT NULL     REFERENCES Klant,
 factuurdatum    date             NOT NULL,
 betalingsdatum  date             NULL )

CREATE TABLE Produkt_Factuur (
 factuurNr       smallint         NOT NULL     REFERENCES Factuur,
 produktCode     char(4)          NOT NULL     REFERENCES Produkt,
 aantal          smallint         NOT NULL,
 stuksprijs      decimal(6,2)     NOT NULL,
 PRIMARY KEY (factuurNr, produktCode ) )

CREATE ASSERTION elke_factuur_heeft_produkt
CHECK ( NOT EXISTS
           (SELECT * FROM Factuur
            WHERE factuurNr NOT IN
                  (SELECT factuurNr FROM Produkt_Factuur)) )
```

In verkorte, schematische weergave is dit script als volgt weer te geven:



De velden die dubbel onderstreept zijn, zijn de primaire sleutels van de tabellen. De velden die enkel onderstreept zijn, moeten uniek zijn. Wanneer meerdere velden enkel onderstreept zijn, moet de combinatie van deze velden uniek zijn. We richten ons nu vooral op die dubbele pijl tussen `Factuur.factuurNr` en `Produkt_Factuur.factuurNr`. Dit is namelijk de equality constraint die nodig is om de total role constraint te kunnen verwezenlijken. De populatie van `Factuur.factuurNr` en `Produkt_Factuur.factuurNr` moeten gelijk zijn. Dit dwingt namelijk af dat er op iedere factuur minstens één besteld artikel staat. Dit is ook gelijk aan een total role constraint. De `CHECK` constraint die onderaan het script staat, zorgt er voor dat er geen facturen mogen bestaan zonder een besteld product erop.

Een variant van de total role constraint, is de externe total role constraint. Deze is weergegeven in Figuur 16.



Figuur 16: ORM externe total role constraint

In deze situatie volgen alle studenten óf een voltijd opleiding óf een deeltijd opleiding óf allebei. De bijbehorende tabel zal bestaan uit de kolommen `sid`, `voltijd opleiding` en `deeltijd opleiding`. Om aan de externe total role constraint te voldoen, moet minstens één van de twee opleidingsvelden ingevuld worden.

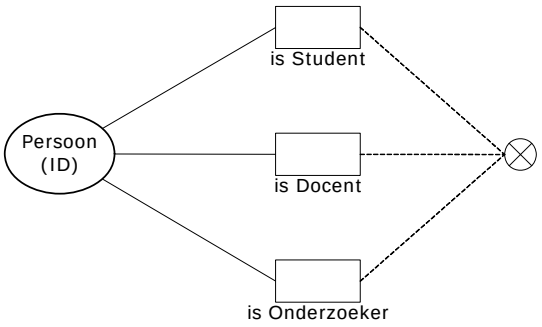
Hier zal op gecontroleerd moeten worden tijdens het invoeren van gegevens in de tabel. Zoals al vaker is aangegeven, kan dit gedaan worden door een `CHECK` constraint in het `CREATE TABLE` statement te plaatsen. Het `CREATE TABLE` statement zal er als volgt uit zien:

```
CREATE TABLE Student (
   sid                integer     NOT NULL,  PRIMARY KEY,
   voltijd_opleiding  char(20)    NULL,
   deeltijd_opleiding char(50)    NULL,
   CONSTRAINT opleiding           CHECK
   (voltijd_opleiding IS NULL AND deeltijd_opleiding IS NOT NULL)
   OR (voltijd_opleiding IS NOT NULL AND deeltijd_opleiding IS NULL)
   OR (voltijd_opleiding IS NOT NULL AND deeltijd_opleiding IS NOT NULL))
```

De kolommen waarin de opleidingsgegevens staan zijn beide `NULL` kolommen. Dit is noodzakelijk aangezien één van de twee kolommen leeg moet kunnen blijven. De `CHECK` constraint controleert dat wanneer de ene kolom ingevuld wordt, de andere kolom leeg blijft. Als er niet aan deze voorwaarde wordt voldaan, worden de waarden niet toegevoegd aan de tabel. Er is in deze situatie maar één situatie die niet voor mag komen en dat is de situatie waarin een student geen opleiding volgt. De total role constraint impliceert namelijk dat elke student in ieder geval één soort opleiding volgt.

7.2.5 Exclusion constraint

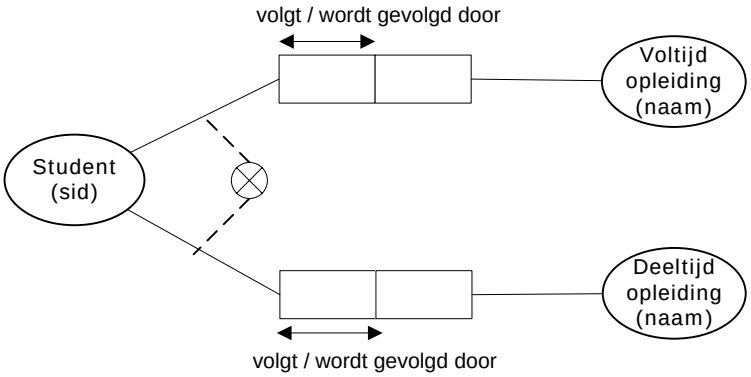
Een exclusion constraint wordt gebruikt om ervoor te zorgen dat de populatie van een objecttype óf de ene rol, óf de andere rol speelt maar niet allebei.



Figuur 17: ORM exclusion constraint

In Figuur 17 bestaan er personen die bepaalde functies uitoefenen. Een persoon kan óf student zijn, óf docent, óf onderzoeker. Combinaties hiervan zijn in deze situatie niet mogelijk. SQL kent geen exclusion constraint in deze vorm, maar via een omweg kan bovenstaande exclusion constraint toch bereikt worden in SQL. Voor een voorbeeld hiervan zie paragraaf 7.2.6 hieronder, waar te zien is dat de combinatie van een uniqueness constraint en een beperkt aantal mogelijke waarden tot hetzelfde resultaat leidt als een exclusion constraint.

Een variant van de exclusion constraint, is de externe exclusion constraint. Deze is weergegeven in Figuur 18, en vertoont veel overeenkomsten met de externe total role constraint.



Figuur 18: ORM externe exclusion constraint

In deze situatie volgen alle studenten óf een voltijd opleiding óf een deeltijd opleiding maar níet allebei. De bijbehorende tabel zijn dezelfde als die in het voorbeeld van de externe total role constraint. Om aan de externe exclusion constraint te voldoen, mag maar één van de twee opleidingsvelden ingevuld worden.

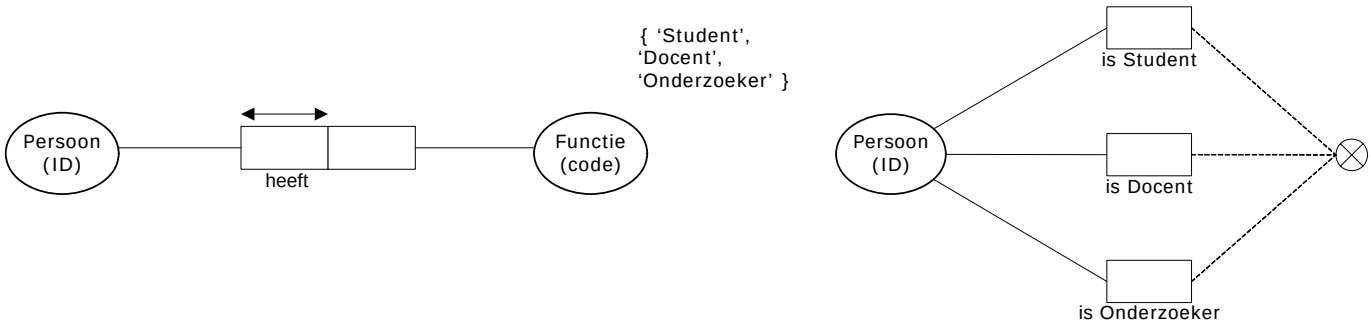
Ook hier kan dit gedaan worden door een CHECK constraint in het CREATE TABLE statement te plaatsen. Het CREATE TABLE statement zal er als volgt uit zien:

```
CREATE TABLE Student (
   sid                 integer     NOT NULL,  PRIMARY KEY,
   voltijd_opleiding   char(20)    NULL,
   deeltijd_opleiding  char(50)    NULL,
   CONSTRAINT opleiding            CHECK
   (voltijd_opleiding IS NULL AND deeltijd_opleiding IS NOT NULL)
   OR (voltijd_opleiding IS NOT NULL AND deeltijd_opleiding IS NULL))
```

De CHECK constraint controleert dat wanneer de ene kolom ingevuld wordt, de andere kolom leeg blijft. Als er niet aan deze voorwaarde wordt voldaan, worden de waarden niet toegevoegd aan de tabel.

7.2.6 Mogelijke waarden

In ORM kunnen heel gemakkelijk de mogelijke waarden waaruit de populatie van een objecttype kan bestaan, worden aangegeven door deze waarden tussen accolades te zetten. In Figuur 19 is dit gedaan in het linkse ORM model. Omdat er een uniqueness constraint op de eerste rol ligt, kan elke persoon maar één functie hebben. Dit komt dus overeen met de exclusion constraint in het ORM model rechts.



Figuur 19: ORM uniqueness constraint versus exclusion constraint

In SQL kunnen de mogelijke waarden van een kolom ook beperkt worden. Dit wordt gedaan door middel van een CHECK constraint, zoals deze ook al de revue heeft gepasseerd in paragraaf 4.3.

```
CREATE TABLE Universiteit
   ( PersoonID        integer     NOT NULL     PRIMARY KEY,
     voornaam         char(20)    NOT NULL,
     achternaam       char(50)    NOT NULL,
     woonplaats       char(50)    NOT NULL,
     leeftijd         integer     NOT NULL,
     functie          char(50)    NOT NULL,
     CONSTRAINT       pos_values  CHECK
    (functie IN ("Student", "Docent", "Onderzoeker"))
```

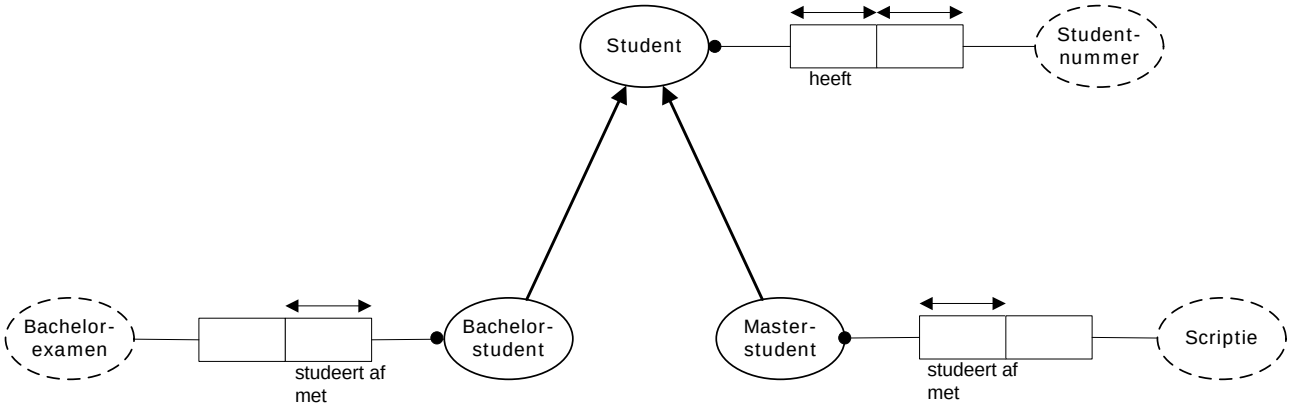
Door middel van bovenstaand CREATE TABLE statement, wordt er op de kolom functie van de tabel Universiteit een CHECK constraint gelegd, die ervoor zorgt dat de functie alleen student, docent of onderzoeker kan zijn.

7.3 Complexe modelleerconstructen

In deze paragraaf zullen enkele complexe modelleerconstructen bekeken worden, die in ORM gebruikt worden. Er zal onderzocht worden in hoeverre deze modelleerconstructen in SQL geïmplementeerd zijn of te benaderen zijn. Meer informatie over de hier besproken modelleerconstructen is te lezen in [HPW04].

7.3.1 Specialisatie (subtypering)

In ORM kennen we het modelleerconstruct specialisatie, ook wel subtypering genoemd. Dit wordt gebruikt wanneer men één of meer subtypen van een objecttype wil definiëren. Een subtype is een verfijning van het supertype waar het onder valt. Dit wil zeggen dat het subtype alle eigenschappen van het supertype erft [HPW04]. Het subtype heeft echter wel extra eigenschappen die alleen bij dat subtype behoren. Een voorbeeld van specialisatie in ORM is te zien in Figuur 20.



Figuur 20: ORM specialisatie (subtypering)

In deze figuur zien we dat een student een studentnummer heeft en dat een student twee subtypen heeft, namelijk bachelorstudent en masterstudent. Deze subtypen hebben dus ook een studentnummer. Subtypering wordt alleen toegepast als de subtypen een eigenschap hebben, dat het supertype niet heeft. In dit geval heeft een bachelorstudent de specifieke eigenschap dat hij/zij afstudeert met een bachelorexamen. De masterstudent doet dit door een scriptie te schrijven.

In SQL bestaat er sinds de SQL:1999 standaard ook een manier om subtypering toe te passen in een database. Het is namelijk mogelijk om subtabellen in een tabel aan te brengen. Een subtabel erft alle kolommen van zijn supertabel, iedere rij in de subtabel correspondeert met precies één rij in de supertabel en iedere rij in de supertabel komt overeen met maximaal één rij in de subtabel [ORDBMS]. Een subtabel erft ook alle constraints van de supertabel, waarbij de belangrijkste de `PRIMARY KEY` is. Een subtabel heeft dus geen eigen `PRIMARY KEY`.

Wanneer er een SELECT statement wordt uitgevoerd op een supertabel, zullen alle rijen van zowel de supertabel als de onderliggende subtabellen worden getoond. Dit is andersom niet het geval. Als er een SELECT statement wordt uitgevoerd op een subtabel, worden alleen de rijen van de subtabel getoond, met de bijbehorende rijen uit de supertabel. Eventuele rijen uit de supertabel die niet voorkomen in de subtabel, worden niet getoond [SQLU].

Het onderstaande SQL statement is nodig, om de subtypering uit Figuur 20 te creëren.

```
CREATE TABLE Student (
   studentnummer     CHAR(10)    PRIMARY KEY,
   ...
   ... )

CREATE TABLE Bachelorstudent UNDER Student
   bachelorexamen    BOOLEAN
   ...
   ... )

CREATE TABLE Masterstudent UNDER Student (
   scriptie          BOOLEAN
   ...
   ... )
```

Het SQL keyword UNDER wordt dus gebruikt om aan te geven dat een tabel een subtabel moet worden van een andere tabel. In dit voorbeeld staan de puntjes (…) uiteraard voor eventuele extra kolommen in de tabellen.

7.3.2 Generalisatie

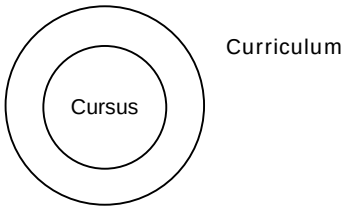
Generalisatie is in feite hetzelfde als specialisatie, alleen dan van een ander gezichtspunt bekeken. Specialisatie is een top-down benadering, waarbij er eerst een supertype wordt gedefinieerd. Daarna wordt er gekeken of dit supertype onder te verdelen is in meerdere subtypen met elk hun eigen specifieke attributen.

Generalisatie is een bottom-up benadering. Daarbij wordt bij een aantal subtypen gezocht naar een attribuut dat voor alle subtypen geldt. Dit attribuut wordt dan het attribuut van het supertype. In het voorbeeld van Figuur 20 is er bijvoorbeeld begonnen met twee typen studenten, bachelor- en masterstudenten. Tijdens het modelleren rijst de vraag of deze twee typen studenten een gemeenschappelijk attribuut hebben. Dit schijnt zo te zijn, alle studenten hebben namelijk een studentnummer. Het model is hierdoor uitgebreid met een supertype student dat als attribuut studentnummer heeft.

Het model van Figuur 20 blijft bij zowel specialisatie als generalisatie exact hetzelfde. Dit geldt ook voor de bijbehorende SQL statements. Het enige verschil tussen deze twee modelleerconstructen is de manier waarop er naar gekeken wordt.

7.3.3 Powertypen

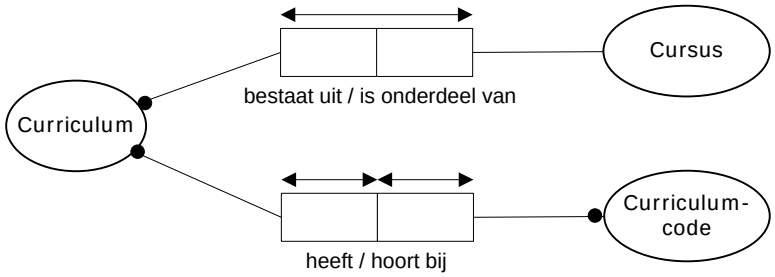
Een powertype, ook wel set type genoemd, is een objecttype dat een niet lege verzameling is van andere objecttypen. Een instantie van een powertype is een niet lege set van instanties van zijn element type. Elk element mag maar één keer voorkomen in de set. Instanties van een powertype worden geïdentificeerd door de elementen waarmee het is opgebouwd. Een powertype hoeft niet alle mogelijke sets te bevatten.



Figuur 21: PSM powertype

In het voorbeeld in Figuur 20 is een powertype weergegeven. Een curriculum bestaat uit een set van cursussen. Elke cursus is er één op zich, een cursus komt niet meerdere keren voor in een curriculum. Elk element van een powertype moet van hetzelfde type zijn. In dit geval mogen er alleen cursussen in het powertype curriculum voorkomen.

In standaard ORM komt het powertype in deze vorm niet voor. Er zijn wel uitbreidingen op ORM, zoals PSM (Predicator Set Model) [HPW04], waarin powertypen, sequentietypen en schematypen voorkomen. Deze complexe constructen zijn wel te vertalen naar standaard ORM. Het is namelijk mogelijk om een powertype uit te drukken door middel van twee binaire feittypen zoals hieronder is weergegeven [Web05].



Figuur 22: Powertype als binaire feittypen in ORM

Omdat een instantie van een powertype geïdentificeerd wordt door zijn elementen, is er in Figuur 22 een extra feittype toegevoegd, om de identificatie mogelijk te maken.

In SQL is een verzameling van unieke elementen van hetzelfde type te creëren door middel van het SET datatype (zie Tabel 4). Een poging om het powertype curriculum om te zetten naar SQL zou er als volgt uit kunnen zien:

```
CREATE TABLE Curriculum (
   curriculumcode    integer     NOT NULL     PRIMARY KEY,
   cursus            set(varchar(30) NOT NULL) )
```

In deze tabel kan in de kolom cursus een verzameling van cursusnamen worden opgeslagen.

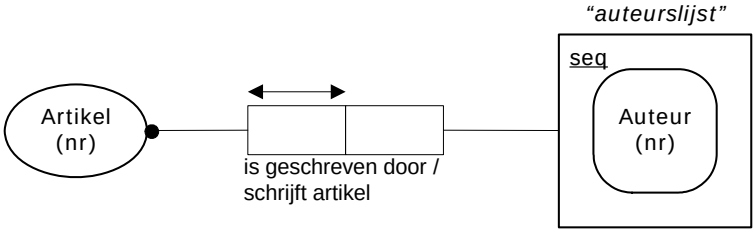
SQL Collectie Datatypen

	Unieke elementen	Volgorde van belang	Lege elementen
LIST	+/-	+	-
MULTISET	+/-	-	-
SET	+	-	-
ARRAY	+/-	+	+

Tabel 4: SQL collectie datatypen [IBM05]

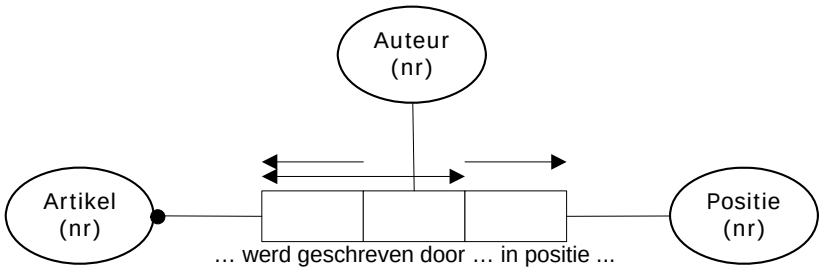
7.3.4 Sequentietypen

Het ORM sequentietype lijkt in veel opzichten op het powertype. Het zijn allebei typen om verzamelingen mee uit te drukken en bij allebei de typen kunnen de verzamelingen uit unieke elementen bestaan. In een sequentietype mogen, in tegenstelling tot het powertype, dubbele elementen voorkomen. Het belangrijkste verschil tussen het powertype en het sequentietype is dat bij een sequentietype de volgorde van de elementen belangrijk is.



Figuur 23: PSM sequentietype

In Figuur 23 is een sequentietype weergegeven dat een verzameling auteurs bevat die samen een artikel hebben geschreven. Het woordje seq in het sequentietype geeft aan dat alle elementen in het sequentietype uniek moeten zijn. Het kan namelijk niet zo zijn dat één auteur twee keer hetzelfde artikel heeft geschreven. Dit woordje mag ook worden weggelaten, wanneer de elementen wel vaker voor mogen komen. Het sequentietype is ook onderdeel van PSM, de uitbreiding op ORM. Een manier om het sequentietype in ORM te modelleren is als volgt [Halp99]:



{for each Artikel: Positie values are sequential from 1}

Figuur 24: ORM sequentietype

De uniqueness constraint over de eerste twee rollen geeft aan dat voor elk artikel een auteur maar één positie in kan nemen. De uniqueness constraint over de eerste en de derde rol geeft aan dat voor elk artikel een positie wordt ingenomen door maximaal één auteur. Onder het ternaire feittype in Figuur 24 staat nog een tekstuele constraint. Deze zegt dat voor elk artikel de nummering van Positie begint bij 1.

SQL kent een datatype LIST, dat een geordende verzameling van elementen kan bevatten. Net als bij het sequentietype kan het LIST datatype dubbele waarden bevatten. Het LIST datatype kent een specifieke volgorde van elementen binnen de verzameling. Een voorbeeld van het datatype LIST volgt hieronder.

```
CREATE TABLE Artikelen (
   artikelnr         integer       NOT NULL     PRIMARY KEY,
   auteurnaam        list(varchar(30) NOT NULL) )
```

8. Verschillen metamodellen

ORM en SQL hebben ontzettend veel overeenkomsten, waarvan er in het vorige hoofdstuk een groot aantal zijn besproken. In de praktijk blijkt dit ook uit het feit dat er tools op de markt zijn, die bij een door de gebruiker geproduceerd ORM model, een compleet SQL databasescript kunnen genereren, inclusief constraints en relaties. Dit is alleen mogelijk als de twee talen goed op elkaar aansluiten.

Uiteraard moeten er ook verschillen zijn tussen ORM en SQL, anders zou het onzin zijn om een onderscheid te maken. In dit hoofdstuk zullen de verschillen tussen ORM en SQL worden belicht.

8.1 Type en doel van de talen

ORM en SQL zijn twee verschillende typen talen. ORM is een modelleertaal, dat gebruikt wordt om informatiesystemen te modelleren op het conceptuele niveau. ORM wordt vaak gebruikt om databases te modelleren. SQL is een implementatietaal, en zit ingebakken in veel database management systemen. Met SQL is het mogelijk om een database te creëren, de aangemaakte tabellen van een populatie te voorzien en deze populatie op te vragen en te muteren.

ORM en SQL zijn dus wel degelijk met elkaar verbonden en kunnen in elkaars verlengde worden gebruikt. Een informatiesysteem kan namelijk gemodelleerd worden in ORM en vervolgens geïmplementeerd worden door middel van SQL. De twee talen blijven echter verschillend van aard en worden voor verschillende doeleinden gebruikt.

8.2 Logische verschillen

Er zijn uiteraard een hoop verschillen tussen ORM en SQL die te maken hebben met het doel waarvoor de taal gebruikt wordt. Zo heeft ORM specifieke kenmerken en eigenschappen van een modelleertaal en heeft het SQL metamodel weer onderdelen die alleen op databases van toepassing zijn. De verschillen in naamgeving zijn onvermijdelijk en zullen niet verder worden beschreven.

ORM Metamodel 2 in Figuur 38 beschrijft veel ORM specifieke constructies. Er is eerder al aangegeven dat rollen in ORM vergelijkbaar zijn met kolommen van tabellen in een RDBMS. ORM Metamodel 2 beschrijft zaken die samenhangen met rollen, die specifiek zijn voor ORM en niet te vertalen zijn naar eigenschappen van kolommen in SQL. Zo zijn in ORM de posities van rollen binnen een feittype van belang voor de Relationship Reading, de manier waarop feittypen gelezen moeten worden. Binnen SQL tabellen zijn de posities van kolommen van geen enkel belang. De feittypen hebben ook een Arity, oftewel een aantal rollen wat wordt uitgedrukt in unair, binair, ternair enzovoorts. Er wordt dan een relatie gelegd tussen twee of drie objecttypen. Het aantal rollen in een feittype is beperkt. Meestal worden er alleen binaire feittypen gebruikt, soms ook ternaire en heel soms komt het voor dat er quartaire feittypen aanwezig zijn in een ORM model. Het aantal kolommen in een tabel in een database, kan veel meer zijn. De Relationship Reading van feittypen in ORM (zoals Student heeft Voornaam) is ook typisch ORM gerelateerd en komt in SQL niet voor.

8.3 Complexe modelleerconstructen

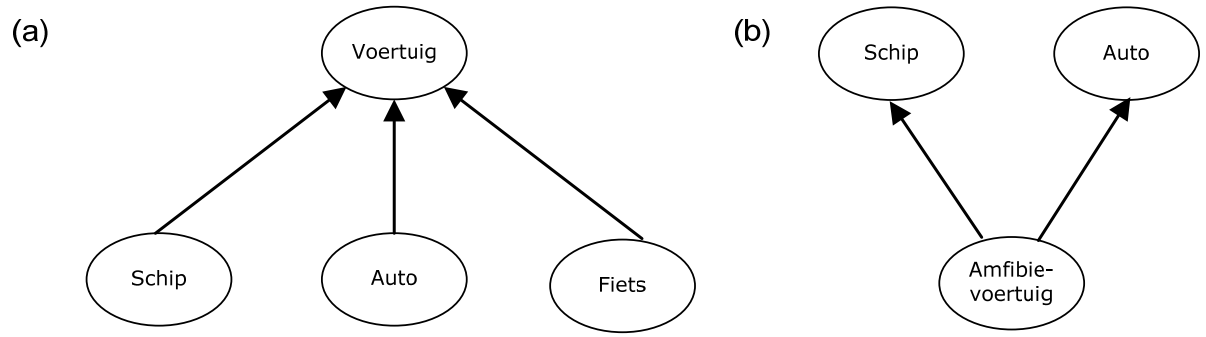
In het vorige hoofdstuk is specialisatie oftewel subtypering besproken. Er werd aangegeven dat de specialisatie binnen ORM modellen te vertalen is naar SQL door middel van het `UNDER` statement. Er is echter wel een verschil te ontdekken tussen specialisatie in ORM en SQL.

ORM schema's waarin subtypering wordt toegepast worden voorzien van een aantal subtype definiërende regels. Deze regels zijn van de vorm **`elke`** `Carnivoor` **`is een`** `Dier` **`van`** `Dier_type 'carnivoor'`. Deze regel zou onder een ORM model over de dierenwereld kunnen staan. Het schrijft voor wanneer een dier tot een bepaald subtype behoort. Alleen dieren van het type `'carnivoor'` behoren tot het subtype `Carnivoor`. Dieren van een ander `Dier_type` worden niet tot dit subtype toegelaten, maar behoren tot een ander subtype.

In SQL kunnen wel subtabellen worden aangemaakt binnen een tabel. Hier zou dus een tabel `Dieren` aangemaakt kunnen worden met daarbinnen de subtabellen `Herbivoor` en `Carnivoor`. Echter, de subtype definiërende regels uit ORM komen niet voor in SQL waardoor het systeem niet automatisch bij een `INSERT` in de tabel `Dieren` een dier in de juiste subtabel kan plaatsen. Dit moet allemaal handmatig door de gebruiker worden gedaan.

Het hele idee achter subtypering is dat de subtypes eigenschappen overerven van hun supertype. De subtypes zijn op zichzelf ook speciaal genoeg om eigen eigenschappen te bezitten. Dit zijn eigenschappen waarmee ze zich onderscheiden van de andere subtypes. Een `Dier` kan als eigenschap bijvoorbeeld een `Leefgebied` hebben. De subtypes van `Dier`, `Carnivoor` en `Herbivoor`, erven deze eigenschap over. Zij hebben dus ook een `Leefgebied`, maar van deze subtypes worden nog aparte eigenschappen vastgelegd. Van een `Carnivoor` kan er bijvoorbeeld worden vastgelegd wat zijn `Prooidieren` zijn, oftewel waar de carnivoor van leeft.

Er bestaan twee vormen van overerving; single inheritance, waarbij een subtype overerft van één supertype, en multiple inheritance, waarbij een subtype overerft van meerdere supertypes. In Figuur 25 is van beide vormen een voorbeeld gegeven.

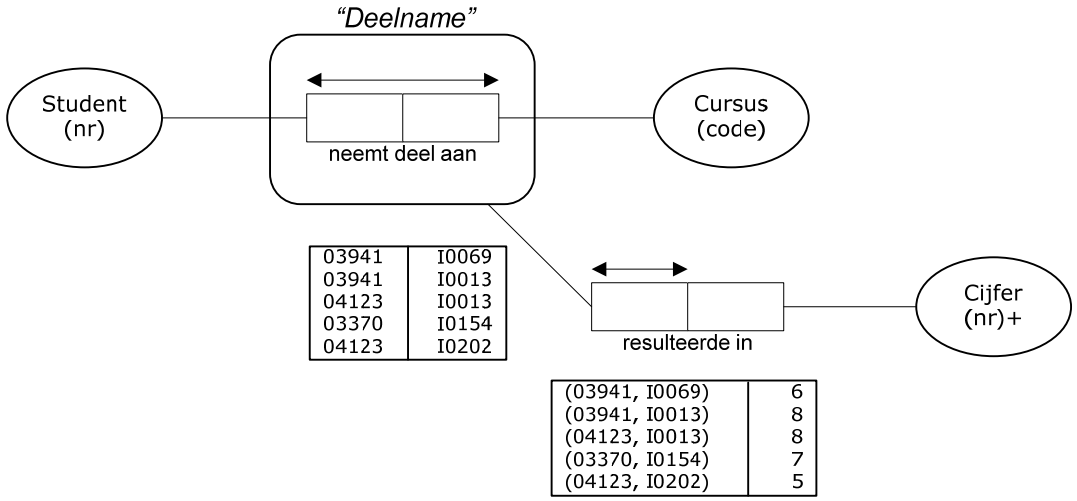


Figuur 25: Voorbeeld van (a) single inheritance en (b) multiple inheritance

Schip, Auto en Fiets zijn alle drie voertuigen. Zij erven dus de eigenschappen van Voertuig over. Daarnaast bezitten ze alle drie nog specifieke eigenschappen. Een Amfibievoertuig bezit eigenschappen van zowel Schip als Auto aangezien het kan rijden én varen. Het Amfibievoertuig erft dus eigenschappen over van zowel het supertype Schip als het supertype Auto.

Zowel SQL als ORM ondersteunen single inheritance. ORM ondersteunt ook multiple inheritance, zoals in Figuur 25 te zien is, maar SQL kent deze vorm van overerving niet. In SQL is het niet mogelijk dezelfde subtabel in meerdere tabellen te laten terugkomen.

Een ander complex construct uit ORM is objectificatie of nesting [ORMObj]. Hierbij wordt een feittype omgezet tot een objecttype, zodat dit feittype in zijn geheel weer mee kan doen in een ander feittype. In Figuur 26 wordt hier een voorbeeld van gegeven.



Figuur 26: ORM objectificatie

Een student neemt deel aan een bepaalde cursus. De uniqueness constraint over beide rollen geeft aan dat elke student maar één keer aan een bepaalde cursus mag deelnemen. Dit feittype is geobjectificeerd, waardoor het mogelijk wordt de volledige populatie van het feittype te laten participeren in een ander feittype. In dit geval worden de gegevens van cursusdeelname gekoppeld aan een cijfer. De combinatie van een student en cursus levert dus een cijfer op.

Als je dit principe zou vertalen naar SQL gebeurt er ongeveer het volgende. Er bestaat een tabel `StudentCursus` waarin staat welke studenten aan welke cursussen mee doen. Deze tabel wordt vervolgens getransformeerd tot een kolom die gebruikt wordt in de tabel `Cijfers`, zodat aan elke combinatie van student en cursus een cijfer kan worden gekoppeld. Deze transformatie van tabel naar kolom, komt in SQL niet voor. Het mechanisme van objectificatie is een puur modelleringsaspect en komt derhalve niet voor in een taal als SQL. Bovenstaande situatie zou in een database waarschijnlijk gewoon opgelost worden door één tabel te creëren met daarin drie kolommen; `studentID`, `cursuscode` en `cijfer`. De unieke combinatie van `studentID` en `cursuscode` zou dan middels een `CHECK` constraint moeten worden afgedwongen.

8.4 SQL/XML

De belangrijkste toevoeging in de SQL:2003 standaard is SQL/XML. In ORM wordt XML in zijn geheel niet gebruikt. In [Halp05] is te lezen dat er momenteel een tool ontwikkeld wordt waarmee ORM2 modellen (de opvolger van ORM) omgezet kunnen worden naar XML schema's. Dit is echter iets wat op het moment van schrijven nog niet mogelijk is.

8.5 Semantiek

Een groot verschil tussen ORM en SQL is de soort semantiek die bij beide talen gebruikt wordt. ORM maakt gebruik van ‘zachte semantiek’. ORM modellen worden opgebouwd aan de hand van zinnen in natuurlijke taal. Hierdoor zijn ORM modellen zeer geschikt als communicatiemiddel. Een gemaakt ORM model moet altijd gevalideerd worden. De opdrachtgever zal kijken of het model overeenkomt met de werkelijkheid. Om dit goed te kunnen beoordelen zal het model gemakkelijk te begrijpen moeten zijn. Ook voor mensen die niet bekend zijn met de ORM modelleertechniek. Door de natuurlijke taal zinnen kan een ORM model letterlijk ‘gelezen’ worden. `Student met Studentnr heeft Cijfer` geeft direct een helder beeld van wat er bedoeld wordt. Met een beetje extra hulp van de modelleur, die bijvoorbeeld aangeeft dat de combinatie Student-Studentnr uniek moet zijn, kan de opdrachtgever goed beoordelen of het model juist is.

SQL kent een zekere mate van ‘zachte semantiek’ maar queries zijn niet zonder meer goed te begrijpen. `SELECT Voornaam FROM Student` is een voorbeeld van een zin die vergelijkbaar is met de natuurlijke taal zinnen uit ORM. Maar wanneer er complexe constructies met `JOINS` en constraints aan bod komen, worden de queries een stuk minder overzichtelijk en een stuk moeilijker te begrijpen. Om SQL écht goed te kunnen begrijpen, is een stukje opleiding noodzakelijk.

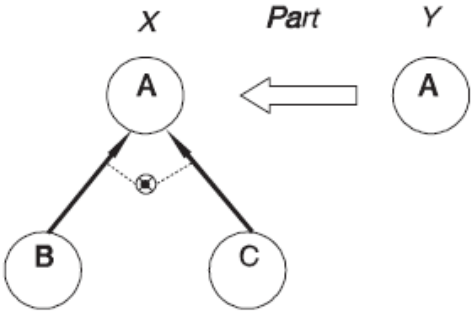
ORM modellen zijn dus goed te begrijpen zonder enige kennis van modelleertechnieken. Alleen voor de constraints en ingewikkelde constructies is hulp van een deskundige noodzakelijk. De hoofdlijnen kunnen in ieder geval direct uit het model afgelezen worden. SQL is een taal die echt aangeleerd moet worden. Ook al kent SQL een zekere mate van ‘zachte semantiek’ zijn queries niet zomaar goed te begrijpen.

9. Formele onderbouwing

In dit hoofdstuk zal een formele onderbouwing worden gegeven bij vergelijkingen van verschillende fragmenten uit de metamodellen van ORM en SQL. Dit zal gedaan worden aan de hand van de operaties zoals deze beschreven worden in [OHFB92].

9.1 Partitioning

Een partitie is een onderverdeling in kleinere stukjes. In metamodel y bestaat er bijvoorbeeld een objecttype. Dit objecttype komt ook voor in metamodel x, alleen in dit metamodel wordt het objecttype nog onderverdeeld in meerdere subtypen. Een voorbeeld hiervan is weergegeven in Figuur 27.

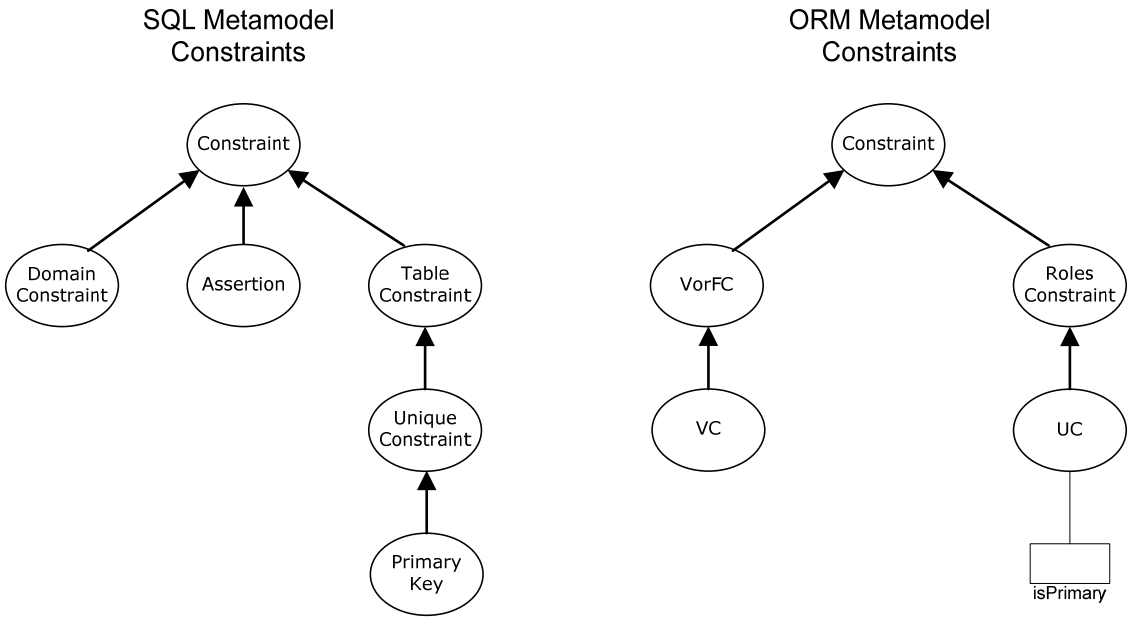


Figuur 27: Partitioning [OHFB92]

De algemene definitie van de partitie relatie is als volgt te omschrijven:

$$
\begin{aligned}
&x\ Part\ y \equiv x\ Part'\ y \vee \\
&\qquad \exists z{:}\ z\ Part'\ y \wedge x\ Part\ z \\
&\text{where} \\
&x\ Part'\ y \equiv \quad Spec(x) \supset Spec(y) \wedge \\
&\qquad\qquad\quad Object(x) \supset Object(y) \wedge \\
&\qquad\qquad\quad Constr(x) \backslash Constr(y) = \quad \{total(Object(x) \backslash Object(y)), \\
&\qquad\qquad\qquad\qquad\qquad\quad exclusion(Object(x) \backslash Object(y))\}
\end{aligned}
$$

In deze definitie staat *Spec(x)* voor de specialisatie relaties in metamodel *x*, *Object(x)* voor de objecten die voorkomen in metamodel *x* en *Constr(x)* voor de constraints in metamodel *x*. Hier geldt dus *Object(x) = (A, B, C)* en *Object(y) = (A)*.



Figuur 28: Fragmenten metamodellen (constraints)

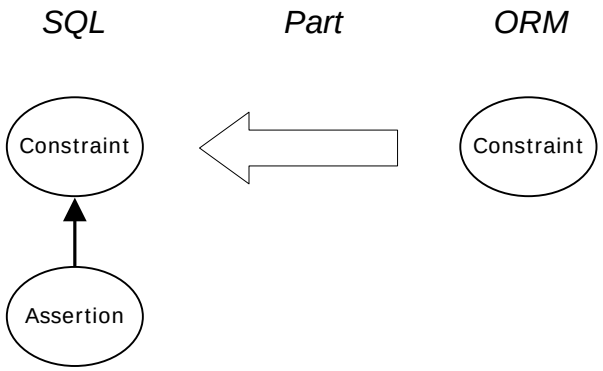
In Figuur 28 staan twee fragmenten afkomstig uit de metamodellen van ORM en SQL. De gelijkenissen tussen de twee zijn groot. In het ORM metamodel wordt er gesproken over constraints. Deze worden onderverdeeld in VorFC (constraints van het type Value Domain of Frequency) en Roles Constraints (constraints die over bepaalde rollen liggen zoals de total role constraint). De Roles Constraint wordt onder meer onderverdeeld in UC, wat staat voor Uniqueness Constraint. Ten slotte bestaat de mogelijkheid dat de uniqueness constraint een Primary Uniqueness Constraint is, wat wil zeggen dat deze uniqueness constraint het objecttype identificeert.

Ook in het SQL metamodel worden constraints behandeld. Hier worden de constraints onderverdeeld in Domain Constraints, Assertions en Table Constraints. De Domain Constraint uit het SQL metamodel en VorFC uit het ORM metamodel zijn vergelijkbaar aangezien ze beide constraints over bepaalde domeinen beschrijven. De SQL Table Constraint en de ORM Roles Constraint zijn ook te vergelijken. Een rol is namelijk de tegenhanger van een kolom in een tabel. De SQL Table Constraint wordt in een ander sub-metamodel gekoppeld aan een bepaalde kolom, waardoor deze vergelijking ook op gaat. De Table Constraint wordt net als bij ORM onderverdeeld in een Unique Constraint. Deze unique constraint wordt ten slotte nog onderverdeeld in een Primary Key, wat ook overeenkomt met isPrimary in het ORM metamodel. Zij identificeren namelijk beiden een bepaald object of entiteit.

Het enige verschil tussen deze twee fragmenten is dat het SQL metamodel ook nog Assertions als onderverdeling van constraints zien. Dit zien we in het ORM metamodel niet terug.

In bovenstaande vergelijking is een partitie relatie te ontdekken. Het meta concept Constraint wordt onderverdeeld in verschillende andere meta concepten. Een partitie is een onderverdeling in kleinere stukjes. Dit is hier het geval, aangezien subtypes deelverzamelingen zijn van supertypes.

Kort samengevat kan de partitie relatie in onderstaande figuur als volgt worden weergegeven.



Figuur 29: Partitioning toegepast op metamodellen

Dit fragment van het SQL metamodel is dus een partitie van het bijbehorende fragment uit het ORM metamodel omdat het SQL metamodel nog een verdere onderverdeling van Constraint kent.

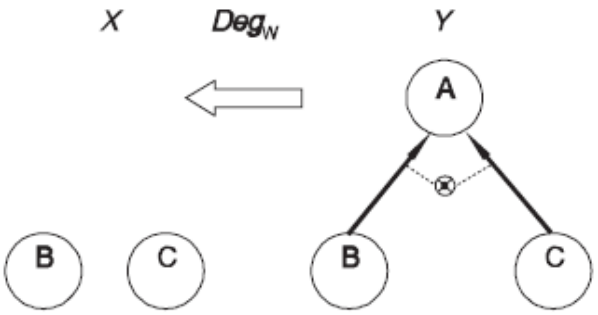
Voor deze specifieke situatie kan de definitie als volgt worden aangepast:

$$SQL\ Part\ ORM \equiv SQL\ Part'\ ORM \vee \exists z: z\ Part'\ ORM \wedge SQL\ Part\ z$$
where
$$SQL\ Part'\ ORM \equiv Spec(SQL) \supset Spec(ORM) \wedge Meaning(SQL) \approx Meaning(ORM) \wedge Object(SQL) \supset Object(ORM)$$

In deze definitie is de clausule over de constraints uit de algemene definitie weggelaten, aangezien er in dit voorbeeld geen total role of exclusion constraints worden gebruikt. Tevens is er een clausule aan de definitie toegevoegd. $Meaning(SQL) \approx Meaning(ORM)$ houdt in dat de betekenis van de objecten uit het SQL metamodel vrijwel overeen moeten komen met de betekenis van de objecten uit het ORM metamodel. In het voorbeeld in Figuur 27 staat in zowel metamodel x als y een objecttype A. In Figuur 28 worden er twee metamodellen vergeleken waarin de objecttypen verschillende namen hebben. Eigenlijk zou er gestreefd moeten worden naar een situatie waar $Meaning(SQL) = Meaning(ORM)$ geldt, maar aangezien er toch twee verschillende metamodellen worden vergeleken is dit niet haalbaar. De ORM en SQL meta concepten uit de metamodel fragmenten in Figuur 28 zijn vergelijkbaar. Er bestaan echter wel verschillen, die ook hierboven al beschreven zijn. De betekenissen van de meta concepten zijn wel vrijwel gelijk. De toegevoegde clausule is een versoepeling van de originele partitie relatie zoals beschreven in [OHFB92]. Deze clausule maakt het mogelijk om ook partitie relaties te definiëren tussen twee metamodellen die niet 100% gelijk aan elkaar zijn (wat in de meeste gevallen zo is, anders heeft een vergelijking ook geen zin), maar waarbij de metaconcepten wel een soortgelijke betekenis hebben.

9.2 Degeneration

Een metamodel kan een degeneratie van een ander metamodel zijn. Dit wil zeggen dat er in het ene metamodel eigenschappen verloren zijn gegaan ten opzichte van het andere metamodel. Er bestaan twee varianten van degeneratie; een sterke en een zwakke variant.



Figuur 30: Weak degeneration [OHFB92]

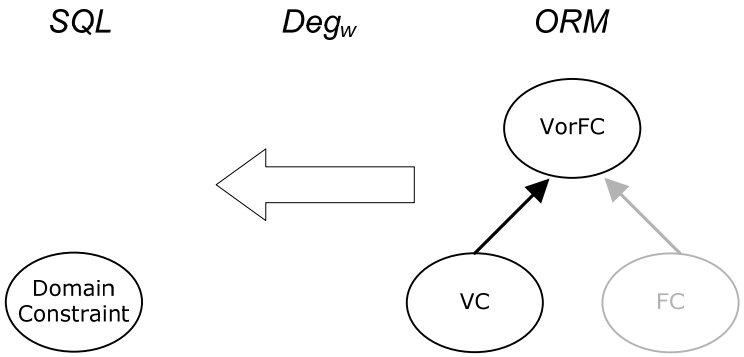
De algemene definitie van een zwakke degeneratie is de volgende:

$$x\ Deg_w\ y \equiv \quad Object(x) \subset Object(y) \wedge \forall o \in Object(y) \backslash Object(x): not(Leaf(o,y))$$
$$\text{where } Leaf(o,x) \equiv o \in Object(x) \wedge \forall \langle o_j, o_i \rangle \in Spec(x) \Rightarrow o_i \neq o$$

In Figuur 30 is een totale rol- en exclusion constraint gebruikt, die niet af te leiden is uit de algemene definitie. Het is vaak niet nodig om dit expliciet aan te geven in het formalisme. In dit geval lijkt dit wel wenselijk. Als de figuur goed bekeken wordt, valt op dat er zónder gebruik te maken van deze constraints gegevens verloren kunnen gaan. De constraints geven namelijk aan dat alle gegevens in objecttype A verdeeld zijn over de subtypen B en C. Als er geen constraints zouden zijn aangegeven, is het mogelijk dat er gegevens in objecttype A zitten die niet voorkomen in subtype B of C. Door middel van de zwakke degeneratie relatie wordt objecttype A uit het metamodel verwijderd. Hiermee kunnen dus ook gegevens verloren gaan. Hieronder wordt dan ook een voorstel gedaan voor een definitie van zwakke degeneratie waarin de constraints verplicht moeten worden opgenomen. Het zal in de praktijk per situatie verschillen of het ernstig is dat er gegevens verloren kunnen gaan. Er kan dan een keuze gemaakt worden tussen de oorspronkelijke en de aangepaste definitie. Voor nu zal er gebruik gemaakt worden van de oorspronkelijke definitie.

$$x\ Deg_w\ y \equiv \quad Object(x) \subset Object(y) \wedge \forall o \in Object(y) \backslash Object(x): not(Leaf(o,y)) \wedge Constr(y) \backslash Constr(x) = \{total(Object(y) \backslash Object(x)), exclusion(Object(y) \backslash Object(x))\}$$
$$\text{where } Leaf(o,x) \equiv o \in Object(x) \wedge \forall \langle o_j, o_i \rangle \in Spec(x) \Rightarrow o_i \neq o$$

In de zwakke variant, waarvan hierboven een voorbeeld is weergegeven, worden de generaliserende metaconcepten uit het metamodel verwijderd. In Figuur 31 worden twee fragmenten uit de metamodellen van ORM en SQL weergegeven. In het ORM metamodel is VorFC een subtype van Constraint. VorFC houdt in dat deze constraint van het type Value Domain Constraint (VC) of Frequency Constraint (FC) is. In het ORM metamodel wordt FC niet als subtype weergegeven aangezien deze geen aparte eigenschappen heeft. Het SQL metamodel kent Domain Constraints, maar kent géén Frequency Constraints. In het SQL metamodel bestaat er dan ook geen generaliserend subtype zoals VorFC in het ORM metamodel. Dit maakt dat dit fragment van het SQL metamodel een zwakke degeneratie is van het bijbehorende fragment uit het ORM metamodel.



Figuur 31: Weak degeneration toegepast op metamodellen

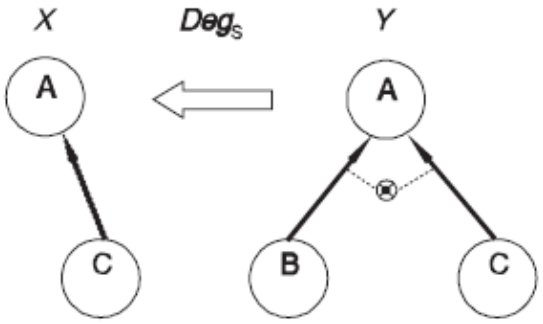
Toegepast op dit specifieke voorbeeld kan de definitie als volgt worden aangepast:

$$SQL\ Deg_w\ ORM \equiv Object(SQL) \subset Object(ORM) \wedge Meaning(SQL) \approx Meaning(ORM) \wedge \forall o \in Object(ORM) \backslash Object(SQL): not(Leaf(o,ORM))$$
$$where\ Leaf(o,SQL) \equiv o \in Object(SQL) \wedge \forall \langle o_j,o_i \rangle \in Spec(SQL) \Rightarrow o_i \neq o$$

Ook hier is de clausule $Meaning(SQL) \approx Meaning(ORM)$ toegevoegd. In deze situatie komen de meta concepten niet 100% overeen, maar zijn ze wel degelijk vergelijkbaar en de relatie tussen de vergelijkbare meta concepten uit de twee metamodellen is goed te leggen. Deze clausule ziet er op toe dat de gebruikte meta concepten in de degeneratie relatie een grote gelijkenis moeten hebben, maar een 100% overeenkomst is niet noodzakelijk.

In het formalisme wordt de functie $not(Leaf(o,y))$ en $Leaf(o,x)$ gebruikt. Een $Leaf$ is in deze definitie een synoniem voor subtype. $Leaf(o,x)$ is dan te vertalen naar een subtype o in metamodel x. Op dezelfde manier is $not(Leaf(o,y))$ te vertalen naar een supertype o in metamodel y, want niet supertype is subtype. De clausule $\forall o \in Object(y) \backslash Object(x): not(Leaf(o,y))$ zorgt er nu voor dat er in metamodel x ten opzichte van metamodel y een supertype wordt weggehaald.

De sterke variant van degeneratie wordt in het volgende voorbeeld weergegeven:



Figuur 32: Strong degeneration [OHFB92]

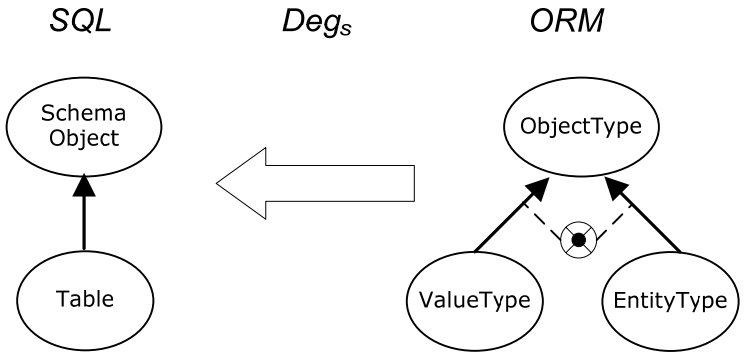
In de sterke variant van degeneratie wordt één van de specialisaties of subtypen van een supertype verwijderd. Dit resulteert dan in een nieuw metamodel, wat een sterke degeneratie is van het oude metamodel.

De algemene definitie van een sterke degeneratie ziet er als volgt uit:

$$x\ Deg_s\ y \equiv \quad Object(x) \subset Object(y) \wedge \forall o \in Object(y) \backslash Object(x): Leaf(o,y)$$
$$\text{where } Leaf(o,x) \equiv o \in Object(x) \wedge \forall \langle o_j, o_i \rangle \in Spec(x) \Rightarrow o_i \neq o$$

In het ORM metamodel worden objecttypen onderverdeeld in Value Types en Entity Types. Het SQL metamodel gebruikt het woord Schema Object om de verschillende typen mee aan te duiden. Dit komt overeen met de objecttypen uit ORM. De ORM entiteittypen komen overeen met de SQL tabellen en de labeltypen met de SQL kolommen. Echter, in het SQL metamodel wordt alleen Table als direct subtype van Schema Object gezien. Column is hier weer een subtype van Table.

In dit geval is het SQL metamodel een sterke degeneratie van het ORM metamodel. In het metamodel van SQL wordt namelijk het subtype Column (ValueType in het ORM metamodel) niet gezien als subtype van het supertype Schema Object (Objecttype in het ORM metamodel). Dit voorbeeld is te zien in Figuur 33.



Figuur 33: Strong degeneration toegepast op metamodellen

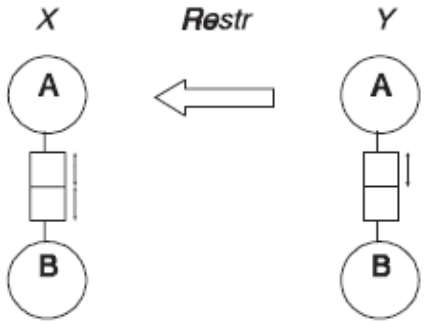
Toegepast op dit specifieke voorbeeld kan de definitie als volgt worden aangepast:

$$SQL\ Deg_s\ ORM \equiv Object(SQL) \subset Object(ORM) \wedge Meaning(SQL) \approx Meaning(ORM) \wedge \forall o \in Object(ORM) \backslash Object(SQL): Leaf(o,ORM)$$
$$\text{where } Leaf(o,SQL) \equiv o \in Object(SQL) \wedge \forall \langle o_j, o_i \rangle \in Spec(SQL) \Rightarrow o_i \neq o$$

Ook hier is de clausule $Meaning(SQL) \approx Meaning(ORM)$ toegevoegd om dezelfde reden als hierboven is aangegeven. In het voorbeeld van de zwakke degeneratie werd er door middel van $not(Leaf(o,y))$ een supertype verwijderd om tot een nieuw metamodel te komen. Bij de sterke variant wordt er juist een subtype verwijderd, wat te zien is aan de clausule $Leaf(o,y)$.

9.3 Restriction

De restrictie relatie tussen twee metamodellen houdt in dat er in het ene metamodel constraints worden toegevoegd ten opzichte van het andere metamodel, waardoor de mogelijke populatie beperkt wordt. Een voorbeeld van een dergelijke relatie is te zien in Figuur 34.



Figuur 34: Restriction [OHFB92]

In metamodel y ligt er een uniqueness constraint over de eerste rol van het feittype. Hierdoor moet de populatie van deze rol uniek zijn. De restrictie relatie zorgt er nu voor dat er een metamodel x afgeleid wordt van metamodel y, waarbij er extra constraints worden toegevoegd aan het metamodel ten opzichte van de al aanwezige constraints in metamodel y. In metamodel x ligt er nu op beide rollen een uniqueness constraint, waardoor de mogelijke populatie van het metamodel wordt beperkt (restrictie) ten opzichte van de mogelijke populatie in metamodel y.

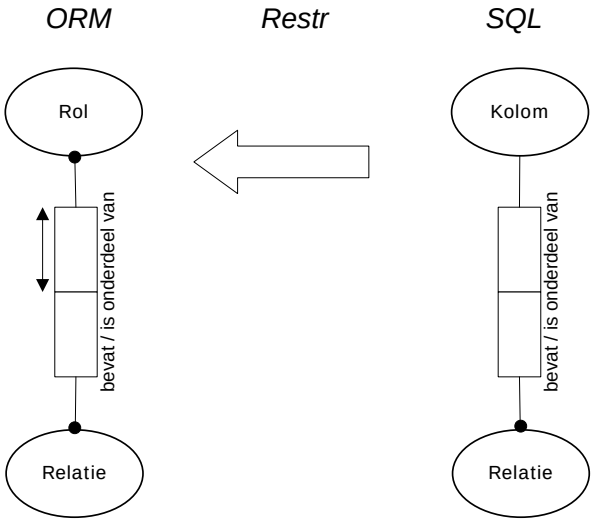
De algemene definitie van een beperkingsrelatie is de volgende:

$$x\ Restr\ y \equiv IsPop(x,Pop) \Rightarrow IsPop(y,Pop) \wedge x \neq y$$

Hierbij staat $IsPop(x,Pop)$ voor een geldige populatie in metamodel x. Deze definitie zegt dus eigenlijk dat als er een geldige populatie is voor metamodel x, dat deze populatie ook geldig moet zijn in metamodel y, waarbij beide metamodellen verschillend zijn.

In de metamodellen van ORM en SQL zijn geen restrictie relaties te vinden. Natuurlijk zijn er wel bepaalde mogelijkheden in de ene taal, die in de andere taal beperkt worden. Deze zijn alleen niet letterlijk in de metamodellen opgenomen. Om hier toch een voorbeeld uit te werken wordt het SQL metamodel hier even uitgebreid. Het in deze scriptie gebruikte SQL metamodel beschrijft niets over de relaties die middels SQL gelegd

kunnen worden tussen tabellen. In deze scriptie is echter al wel duidelijk naar voren gekomen dat dit kan. In het voorbeeld in Figuur 35 worden de relaties in ORM en SQL met elkaar vergeleken.



Figuur 35: Restriction toegepast op metamodellen (en daarbuiten)

In SQL worden er relaties gelegd tussen kolommen in verschillende tabellen. Niet iedere kolom hoeft persé onderdeel te zijn van een relatie. Het is mogelijk dat een kolom mee doet in meerdere relaties. Elke relatie wordt gelegd tussen kolommen. In de ORM situatie is het zo dat elke rol onderdeel is van een relatie en dat elke rol ook maar aan één relatie deel mag nemen. De mogelijke populaties worden hier dus beperkt ten opzichte van de mogelijkheden van SQL. Voor deze fragmenten geldt dus dat ORM een restrictie is van SQL.

De definitie van een restrictie relatie kan voor dit voorbeeld als volgt worden aangepast:

$$ORM\ Restr\ SQL \equiv IsPop(ORM,Pop) \Rightarrow IsPop(SQL,Pop) \wedge ORM \neq SQL \wedge Meaning(SQL) \approx Meaning(ORM)$$

10. Conclusie

In dit hoofdstuk zullen de onderzoeksvragen die aan het begin van het onderzoek zijn opgesteld, worden beantwoord. Tevens zal er een opmerking over eventueel vervolgonderzoek worden gegeven.

10.1 Antwoorden op de onderzoeksvragen

Gedurende de voorgaande hoofdstukken zijn alle (deel)onderzoeksvragen al beantwoord, maar hier zullen al deze antwoorden nog even kort onder de aandacht worden gebracht. Ook zullen er nu enkele conclusies aan deze antwoorden worden verbonden, die nog niet eerder in deze scriptie naar voren zijn gekomen.

10.1.1 Deelvraag 1

Deelvraag 1 luidde: *Wat is SQL?*

SQL is een databasetaal waarmee informatie uit relationele databases kan worden opgevraagd en gemuteerd. SQL is een ANSI/ISO standaard en zit in alle commerciële relationele database management systemen (RDBMS) ingebouwd[5]. Met SQL statements kunnen onder meer gegevens uit een database worden verwijderd, aan een database worden toegevoegd of worden gewijzigd. Ook kunnen door middel van SQL statements tabellen worden aangemaakt en gebruikersrechten worden ingesteld.

10.1.2 Deelvraag 2

Deelvraag 2 luidde: *Welke SQL standaarden zijn er?*

Er zijn tot nu toe vijf officiële ANSI/ISO SQL standaarden. De SQL standaarden worden geproduceerd door werkgroepen van ISO en ANSI. Op het internationale toneel werkt de ISO werkgroep “ISO/IEC JTC 1/SC 32 Data Management and Interchange/WG 3 – Database Languages” aan de SQL standaard. Deze werkgroep bestaat uit delegaties uit negen landen, waaronder Nederland [JCC06].

De ANSI werkgroep genaamd “ANSI NCITS H2 Technical Committee on Database” [ANSIH2] richt zich alleen op de Verenigde Staten. Deze werkgroep bestaat uit vertegenwoordigers van alle grote RDBMS ontwikkelaars, aangevuld met enkele consultants en gebruikers.

De eerste SQL standaard werd in 1986 door ANSI en in 1987 door ISO geaccepteerd. De hierop volgende versies werden gezamenlijk geaccepteerd en dit gebeurde in 1989, 1992, 1999 en 2003. De eerstvolgende SQL standaard zal waarschijnlijk in 2007 of 2008 gepresenteerd worden.

[5] De commerciële producten concurreren onderling door eigen functionaliteiten aan de SQL standaard toe te voegen, waardoor er enkele ‘dialecten’ zijn ontstaan zoals Transact-SQL voor Microsoft SQL Server en Sybase Adaptive Server Enterprise [Wiki07] en PL/SQL van Oracle [Wiki08].

10.1.3 Deelvraag 3

Deelvraag 3 luidde: *Wat zijn de verschillen tussen deze standaarden?*

Elke SQL standaard bevat weer nieuwe mogelijkheden (zoals nieuwe datatypen) en functionaliteiten ten opzichte van zijn voorganger. In hoofdstuk 4 zijn alle standaarden belicht. Voor de precieze verschillen verwijs ik u graag naar dit hoofdstuk.

SQL-86 was de allereerste officiële SQL standaard en daarmee legde deze standaard de basis voor alle hierop volgende standaarden. In SQL-86 worden de basisstatements van SQL geïntroduceerd. Voorbeelden hiervan zijn `CREATE TABLE` om tabellen aan te maken, `SELECT` om gegevens uit een database te selecteren en `INSERT`, `UPDATE` en `DELETE` om gegevens te muteren.
De SQL-89 standaard is een update van de SQL-86 standaard. Er was hierbij geen sprake van een compleet nieuwe standaard. SQL-89 voegde enkel referentiële integriteit toe aan de SQL-86 concepten. Referentiële integriteit is een database concept dat er voor zorgt dat relaties tussen tabellen in een database consistent zijn en blijven.
Twee technieken die bij referentiële integriteit horen zijn ‘cascading update’ en ‘cascading delete’. Deze technieken zorgen ervoor dat wanneer er wijzigingen zijn in de verwijzende tabel, deze wijzigingen ook worden doorgevoerd in de primaire tabel.

SQL-92 was een echt nieuwe standaard met veel nieuwe mogelijkheden ten opzichte van de oude standaard. Dynamic SQL werd geïntroduceerd waardoor het mogelijk werd om dynamisch queries te genereren. Een standaard query wordt dan voorzien van een parameter die steeds veranderd. Hierdoor worden steeds die gegevens opgevraagd of gemuteerd die als `ID` bijvoorbeeld een parameterwaarde hebben. Er hoeft op deze manier maar één enkele query geschreven te worden voor alle situaties waarbij dezelfde gegevens opgevraagd of gemuteerd moeten worden. In SQL-92 zijn ook de `JOINS` geïntroduceerd. Hiermee kunnen gegevens uit meerdere tabellen opgevraagd worden. Een variant op het `JOIN` statement is het `UNION` statement. Hiermee kunnen ook gegevens uit meerdere tabellen opgevraagd worden, alleen moeten nu de gegevens uit de tabellen van hetzelfde type zijn (tekst, numeriek, datum). Dit is bij het `JOIN` statement niet het geval. De volgende nieuwe mogelijkheid is de `CHECK` constraint. Deze constraint wordt toegevoegd aan het `CREATE TABLE` statement en wordt gebruikt om de mogelijke waarden die in een kolom kunnen worden ingevoerd, te beperken. De `CHECK` constraint is een belangrijke en krachtige constraint die, zoals in deze scriptie meerdere keren gedaan wordt, gebruikt kan worden om ORM constructies te implementeren in SQL.

De uitbreidingen in SQL:1999 richtten zich vooral op objectgeoriënteerd SQL. De belangrijkste toevoegingen ten opzichte van SQL-92 waren de mogelijkheden om triggers en stored procedures te gebruiken. Een trigger is een stuk code dat door het RDBMS geactiveerd wordt indien een bepaalde operatie op de database wordt uitgevoerd, en alleen dan als een bepaalde conditie waar is. Een stored procedure is een stuk code dat geactiveerd wordt door deze aan te roepen vanuit een programma, een trigger of een andere stored procedure. Zowel een trigger als een stored procedure zijn stukken code die opgeslagen liggen in de database. Het belangrijke verschil tussen een trigger en een stored procedure is de manier van aanroepen. Triggers kunnen namelijk niet direct door een programma of een andere stored procedure worden aangeroepen. Stored procedures kunnen wel op die manier aangeroepen worden.

SQL:2003 is de meest recente SQL standaard. Het belangrijkste nieuwe bestanddeel van de SQL standaard is SQL/XML. XML (eXtensible Markup Language) is een taal voor de representatie van gestructureerde gegevens in de vorm van platte tekst. XML heeft een duidelijk verband met SQL. Beiden kunnen worden gebruikt om structuur in gegevens

aan te brengen. SQL/XML is in het leven geroepen om steun te bieden bij het gebruiken van XML in de context van een SQL database systeem. SQL/XML maakt het mogelijk om XML documenten op te slaan in een database. Vervolgens kunnen deze XML documenten door middel van queries opgevraagd en gewijzigd worden. Ook kunnen uit tabellen van een database, XML bestanden worden gegenereerd. SQL:2003 legt de laatste hand aan collectietypen met de invoering van het `MULTISET` datatype. In voorgaande standaarden waren al datatypen als `ARRAY` en `SET` geïntroduceerd. In deze collectietypen kunnen verzamelingen van gegevens worden opgeslagen. SQL:2003 voegt ook een manier om data te muteren toe aan de al bestaande manieren zoals `INSERT`, `UPDATE` en `DELETE`. Deze nieuwe manier is het `MERGE` statement. `MERGE` is een combinatie van `INSERT` en `UPDATE`. Er worden hiermee rijen aan een tabel toegevoegd, maar rijen die ingevoerd worden terwijl ze al in de tabel aanwezig zijn, worden bijgewerkt.

10.1.4 Deelvraag 4

Deelvraag 4 luidde: *Wat is een conceptueel model?*

Een conceptueel model is een model dat vaak gebruikt wordt tijdens het ontwerpen van een database. Uit een dergelijk model kan vaak de databasestructuur worden afgeleid. Er bestaan ook tools, zoals Microsoft VisioModeler, die vanuit een conceptueel ORM model de bijbehorende SQL statements kan genereren waarmee tabellen, kolommen, relaties en constraints kunnen worden aangemaakt in een database. Een conceptueel model komt tot stand door een intensieve samenwerking met de klant. De klant bezit de kennis over de bedrijfsprocessen en werkwijzen binnen het bedrijf. De klant wordt in deze situatie ook vaak domeinexpert genoemd. Met de informatie die de domeinexpert geeft, en de informatie die uit bedrijfsdocumenten te halen, kan er een conceptueel model opgesteld worden. Dit model bevat dan alle informatie dat belangrijk is voor het bedrijf.

Een model is vaak niet in één keer goed of compleet op te stellen. In overleg met de domeinexpert wordt er net zo lang aan geschaafd tot de domeinexpert tevreden is met het model. Later in het systeem ontwikkeltraject zal er uit dit conceptuele model een databasestructuur worden afgeleid.

10.1.5 Deelvraag 5

Deelvraag 5 luidde: *Wat is een metamodel?*

Een metamodel is een model van een model. Het beschrijft de structuur en eigenschappen van een modelleertaal. In een metamodel van bijvoorbeeld ORM wordt beschreven waar ORM modellen aan moeten voldoen.

10.1.6 Deelvraag 6

Deelvraag 6 luidde: *Welke overeenkomsten zijn er te vinden tussen de metamodellen van ORM en SQL?*

Er zijn ontzettend veel overeenkomsten te vinden tussen de metamodellen van ORM en SQL. Te veel om hier allemaal nog een keer op te noemen. In hoofdstuk 7 van deze scriptie worden de gevonden overeenkomsten beschreven. In deze scriptie is het niet de bedoeling geweest om tot op elk detail nauwkeurig alle mogelijke overeenkomsten uiteen te zetten. Er zijn voldoende overeenkomsten gegeven, zodat de onderzoeksvraag beantwoordt kan worden, en de lezers van deze scriptie een goed beeld krijgen van de overeenkomsten tussen de metamodellen van ORM en SQL.

Alle constraintsoorten uit ORM zijn te vertalen naar SQL. Ook complexe modelleerconstructen als subtypering, powertypen en sequentietypen zijn in SQL na te

bootsen. Veel overeenkomsten die er gevonden zijn, zijn niet één-op-één te vertalen. Het is in de meeste gevallen niet zo dat constructie A in ORM gelijk staat aan constructie B in SQL. Vaak was het wel mogelijk om via omwegen, bijvoorbeeld door toevoeging van extra constraints, de werking en betekenis van een ORM constructie na te bootsen in SQL.

10.1.7 Deelvraag 7

Deelvraag 7 luidde: *Welke verschillen zijn er te vinden tussen de metamodellen van ORM en SQL?*

Er zijn maar weinig verschillen te ontdekken tussen de metamodellen van ORM en SQL. Zoals hierboven ook al is aangegeven zijn de meeste constructies door omwegen toch te vertalen van de ene naar de andere taal. Op een gegeven moment rees de vraag wanneer er van een verschil gesproken kan worden. Soms moeten er dermate veel extra handelingen worden verricht dat, ook al kan de werking en betekenis van een constructie uiteindelijk dan toch vertaald worden naar een andere taal, dit gevoelsmatig niet tot de overeenkomsten gerekend kan worden. Uiteindelijk is er voor gekozen om alle ‘vertaalbare constructies’ tot de overeenkomsten te rekenen, welke omwegen er dan ook voor nodig zijn. Dit heeft tot gevolg dat er weinig verschillen overbleven tussen de metamodellen van ORM en SQL. Het was al redelijk snel duidelijk dat er waarschijnlijk weinig verschillen gevonden zouden worden. Er bestaan tools die ORM modellen om kunnen zetten naar SQL statements. Dit impliceert dat deze twee talen behoorlijk met elkaar verbonden zijn. Uit het aantal overeenkomsten en verschillen komt dit ook naar voren.

Natuurlijk zijn er altijd verschillen tussen twee talen. De talen zijn immers niet identiek. Zo hebben beiden talen een ander doel. ORM is een modelleertaal, dat gebruikt wordt om informatiesystemen te modelleren op het conceptuele niveau. ORM wordt vaak gebruikt om databases te modelleren. SQL is een implementatietaal, en zit ingebakken in veel database management systemen. Met SQL is het mogelijk om een database te creëren, de aangemaakte tabellen van een populatie te voorzien en deze populatie op te vragen en te muteren.

ORM en SQL zijn dus wel degelijk met elkaar verbonden en kunnen in elkaars verlengde worden gebruikt. Een informatiesysteem kan namelijk gemodelleerd worden in ORM en vervolgens geïmplementeerd worden door middel van SQL. De twee talen blijven echter verschillend van aard en worden voor verschillende doeleinden gebruikt.

Ook ORM specifieke constructies als subtype definiërende regels en objectificatie zien we niet terug in SQL. De subtype definiërende regels bepalen wanneer een bepaalde waarde tot een bepaald subtype behoort. In SQL kan dit niet automatisch gedaan worden. De gebruiker zal hier altijd zelf moeten zorgen dat de goede waarden in de goede subtabellen komen.

Subtypering kan in beide talen gebruikt worden. Er is echter wel één groot verschil te ontdekken. ORM maakt gebruik van multiple inheritance, wat inhoudt dat een subtype van meerdere supertypen kan overerven. SQL kent alleen single inheritance. Een SQL subtabel kan niet in twee supertabellen voorkomen.

Een ander verschil is de semantiek die in beide talen gebruikt wordt. ORM modellen zijn door de natuurlijke taal zinnen voor een ieder te begrijpen. Om SQL echt goed te kunnen begrijpen is een stukje opleiding noodzakelijk.

10.1.8 Hoofdvraag

De hoofdvraag luidde: *Hoe kunnen conceptueel model en RDBMS dichter bij elkaar gebracht worden met gebruik van de standaard SQL:2003?*

In deze scriptie is er een vergelijking gedaan tussen twee metamodellen. Een metamodel van ORM en een metamodel van SQL. Het uiteindelijke doel was om deze twee metamodellen dichter bij elkaar te brengen. Hiermee wordt bedoeld de expressiviteit van enerzijds verschillende SQL standaarden en anderzijds Object Role Modeling (ORM) te inventariseren en vergelijken. Het was bij voorbaat al duidelijk dat er een samenhang bestaat tussen ORM en SQL. Dit onderzoek heeft getracht deze samenhang concreet te maken door de overeenkomsten en verschillen tussen deze twee talen te beschrijven.

Allereerst zijn er twee metamodellen gezocht waar dit onderzoek op gebaseerd zou worden. De keuze voor deze metamodellen is gemaakt omdat dit de meest complete metamodellen waren. Hierdoor zou er op zoveel mogelijk punten vergeleken kunnen worden.

Het onderzoek keek op twee verschillende niveaus naar het probleem. Op een praktisch niveau werden concrete voorbeelden gegeven bij de vergelijking tussen de twee metamodellen. Op een technisch niveau werd er getracht een formele benadering te geven bij de vergelijking. Hier zijn de metamodellen vergelijken aan de hand van verschillende formalismen. Door steeds kleine delen van de metamodellen te vergelijken en daar een voorbeeld bij te geven, is er op systematische wijze vergeleken. Het is niet de bedoeling geweest om een complete vergelijking te doen van deze twee talen. Er zijn voldoende overeenkomsten en verschillen gegeven, zodat de onderzoeksvraag beantwoord kan worden, en de lezers van deze scriptie een goed beeld krijgen van de overeenkomsten tussen de metamodellen van ORM en SQL.

De conclusie van dit onderzoek is dat beide talen inderdaad behoorlijk veel samenhang hebben. Heel veel ORM modellen zijn zonder moeite te vertalen naar SQL databaseschema's. Dit is een in de praktijk beproefde methodiek gebleken. Er zijn verschillende tools op de markt die de gebruiker ondersteunen bij de conversie van een ORM model naar een SQL relationele database. De samenhang tussen deze twee talen is een krachtig middel in het systeem ontwikkeltraject.

De in deze scriptie gehanteerde vergelijkingsmethode is mijns inziens een goede methode omdat er op een systematische wijze en op meerdere niveaus vergeleken is. Deze vergelijkingsmethode is deels gebaseerd op de methode die besproken wordt in [OHFB92]. Daar worden alleen geen concrete voorbeelden gegeven bij de formalismen. Op deze manier is er meer inzicht verkregen in de precieze samenhang tussen de twee talen. Van tevoren was het al bekend dat deze twee talen een zekere samenhang hadden. Hierdoor was het ook een goed onderzoeksdomein. Het heeft namelijk geen zin om twee totaal verschillende talen te vergelijken. Deze scriptie geeft duidelijk bewijs voor de samenhang tussen deze twee talen. Door de vergelijking zowel te staven met concrete voorbeelden als met formalismen (tot op zekere hoogte) wordt er een breed publiek bediend. Minder technisch onderlegde mensen zullen de concrete voorbeelden goed kunnen begrijpen, mede door de begeleidende teksten. De echte technici zullen wat meer waarde hechten aan de formalismen.

Het eigenlijke doel van de scriptie was om te kijken in hoeverre ORM en SQL dichter bij elkaar gebracht konden worden met gebruik van de standaard SQL:2003, zoals de hoofdvraag ook aangeeft. Het gebruik van de SQL:2003 standaard is achteraf minimaal gebleken. De vergelijking is gedaan door SQL als geheel (dat wil zeggen dat alle SQL standaarden in de vergelijking betrokken zijn) te vergelijken met ORM. De belangrijkste toevoeging die de SQL:2003 standaard doet ten opzichte van de eerder verschenen standaarden, is de introductie van SQL/XML. XML wordt in ORM echter in zijn geheel niet ondersteund. Hierdoor wordt de invloed van de SQL:2003 standaard beperkt tot enkele nieuwe datatypen en een nieuwe data mutatie methode.

Is de conclusie verrassend? Ja en nee. Van de ene kant wel aangezien het toch twee verschillende talen zijn. Ik had verwacht méér verschillen tussen de twee talen te kunnen ontdekken. Van de andere kant was het voor mij van tevoren wel duidelijk dat er een bepaalde mate van samenhang tussen de twee talen bestond. Ik wist dat ORM modellen werden gebruikt als basis voor een databasestructuur. SQL is een databasetaal. Hieruit blijkt al een overeenkomst. Ik heb me echter wel verbaasd over de verhouding tussen de overeenkomsten en de verschillen. Ik had niet verwacht dat SQL en ORM zóveel samenhang zouden hebben.

10.2 Verder onderzoek

Deze scriptie geeft een mogelijke benadering om twee metamodellen te vergelijken. In de toekomst kan er wellicht een complete vergelijking gedaan worden tussen de talen SQL en ORM. Beide talen zijn constant in ontwikkeling. In 2007 of 2008 wordt er een nieuwe SQL standaard verwacht en ook ORM 2, de opvolger van ORM, wordt al gebruikt en is nog volop in ontwikkeling. De verwachting is dat de talen door deze ontwikkelingen steeds verder naar elkaar toe zullen groeien. De nieuwe SQL standaard zal waarschijnlijk weer leiden tot een uitbreiding van ORM.

Deze scriptie richt zich alleen op de hier gebruikte metamodellen. Er bestaan meerdere metamodellen van ORM en SQL. Deze scriptie is alleen geldig bij gebruik van de hier gebruikte metamodellen en bij een vergelijking van de hier vergeleken talen. Het zal duidelijk zijn dat andere metamodellen of andere talen tot andere resultaten zullen leiden. De hier gebruikte aanpak is naar mijn mening wel toepasbaar bij vergelijkbare onderzoeken. Een vergelijking tussen UML en SQL zou bijvoorbeeld volgens hetzelfde stramien gedaan kunnen worden. Toekomstig onderzoek zal eventueel een standaard vergelijkingsmethode kunnen opleveren, die bij elke vergelijking gebruikt kan worden.

11. Evaluatie

In dit hoofdstuk zal in aangeven waarom volgens mij deze scriptie van voldoende niveau is. Ook zal ik het verloop van het onderzoek beschrijven: welke problemen ben ik tegen gekomen tijdens het onderzoek en wat ging me goed af.

11.1 Informatiekunde vs informatica

Als informatiekundige begeef je je constant in een gebied tussen techniek en mens. Je neemt vaak de rol aan van intermediair tussen klant en programmeur. Om deze rol goed te kunnen uitvoeren is kennis uit beide 'werelden' noodzakelijk. Een informatiekundige moet dus in staat zijn om te begrijpen wat een programmeur doet (en dus een bepaald niveau programmeerkennis hebben), zodat hij dit kan vertalen naar begrijpbare taal voor de vaak a-technische klant. Omgedraaid moet een informatiekundige ook in staat zijn om klantwensen te beoordelen op haalbaarheid en te vertalen naar ontwerpen voor software die uiteindelijk mede door de programmeur ontwikkeld zal worden. Een informatiekundige is niet alleen tolk tussen twee werelden, maar participeert ook mee in beide werelden. Zo wordt een informatiekundige ook geacht zelfstandig (kleine) applicaties te kunnen ontwikkelen en klanten van een deskundig advies te kunnen voorzien (consultancy).

Een informaticus begeeft zich veelal in het gebied van de techniek en formalismen. Hij zal dan ook veel technischer onderlegt zijn dan bij een informatiekundige het geval is. In deze scriptie is getracht op enkele plaatsen een formele onderbouwing te geven. De formalismen nemen in deze scriptie echter geen al te prominente rol in. Dit kan denk ik ook niet van een informatiekundige verlangd worden, en behoord meer bij de informaticus thuis. Tijdens mijn masteropleiding is deze opvatting ook bij de vakken Beweren en Bewijzen en Formeel Denken uitgesproken. Daar hoefden de informatiekundigen ook minder stof te beheersen dan de informatici. Uiteraard wil dit alles niet zeggen dat een informatiekundige dit in zijn geheel niet hoeft te beheersen; een bepaald niveau formalismen moet ook voor een informatiekundige haalbaar zijn.

Ik heb mijn uiterste best gedaan om de formalismen in deze scriptie zo nauwkeurig mogelijk uit te werken. Er zullen echter ongetwijfeld nog incomplete formalismen aanwezig zijn in deze scriptie, mede als gevolg van een mindere formele en technische onderlegging. De begeleidende tekst bij de formalismen zal duidelijk moeten maken wat het doel is van de formalismen.

11.2 HBO vs WO

Vorig jaar heb ik mijn vierjarige HBO opleiding Bedrijfskundige Informatica afgerond aan de Hogeschool van Arnhem en Nijmegen. Tijdens mijn laatste jaar HBO heb ik ook alle schakelvakken voor de universitaire opleiding Informatiekunde gevolgd en afgerond. Dit jaar heb ik de éénjarige masteropleiding Informatiekunde gevolgd en deze hoop ik middels deze scriptie met goed gevolg af te ronden.

Aan het einde van de rit bekijk ik wat de verschillen zijn tussen HBO en WO en waarom deze scriptie volgens mij van voldoende niveau is.

HBO staat voor Hoger Beroeps Onderwijs, en zoals de naam al doet vermoeden wordt je hier klaargestoomd voor de beroepspraktijk. Het HBO is erg praktijkgericht en dit komt tot uiting in het vele werken in projectgroepen en het stagelopen en afstuderen in het bedrijfsleven. Toen ik colleges ging volgen aan de Radboud Universiteit kon ik al gauw het verschil met het HBO merken. De colleges waren veel theoretischer. Meer hoorcolleges en minder practica. Colleges worden gegeven door docenten die vaak al

vele jaren ervaring hebben in het specifieke vakgebied, waar op het HBO docenten soms een spoedcursus moesten volgen om een vak te kunnen geven met als gevolg dat in sommige situaties een student deskundiger was dan de docent.

Eén van de grootste verschillen vond ik de wiskundige en formele onderbouwing van theorieën, die bij alle cursussen terug kwam. Hier had ik het in het begin erg moeilijk mee, aangezien hier op het HBO totaal geen aandacht aan werd besteed en ik een HAVO Wiskunde A1 achtergrond had. Op een gegeven moment heb ik wel geleerd hoe ik naar deze notaties moest kijken en begreep ik ze ook aanzienlijk beter. Maar in alle eerlijkheid: een expert ben ik niet.

Ik vind deze scriptie van WO niveau om een aantal redenen. Ten eerste vanwege de gebruikte formalismen. Ze zijn in aantal en waarschijnlijk ook in correctheid minder dan die van een informaticus, maar de redenering hierachter is al eerder uitgelegd. Deze formalismen had ik tijdens mijn HBO opleiding nooit en te nimmer op papier kunnen krijgen.

Ten tweede de omvang van de scriptie. Dit is al aanzienlijk meer dan er op het HBO van me werd verlangd. De tijd die ik nodig heb gehad is vergelijkbaar met die van mijn HBO stage. Het eindresultaat daarvan was qua omvang de helft van deze scriptie. Maar kwaliteit gaat nog altijd boven kwantiteit en dat brengt me bij punt drie.

Tijdens mijn onderzoek heb ik veelal gebruik gemaakt van wetenschappelijke artikelen, waardoor de kwaliteit van de scriptie omhoog gaat. Je refereert dan namelijk aan auteurs die zelf ook op WO niveau onderzoek hebben gedaan. Op het HBO werd er weinig aandacht besteed aan de bronnen die je gebruikte voor je werk.

Ten slotte heb ik getracht het onderzoek op een wetenschappelijke manier uit te voeren door vooraf een doelstelling te formuleren, een probleem te definiëren en onderzoeksvragen op te stellen. Gedurende het onderzoek worden de onderzoeksvragen beantwoord en uiteindelijk beschrijft de conclusie de oplossing voor het probleem.

Al deze redenen maken voor mij dat deze scriptie van voldoende niveau is. Ik weet dat ik dit zonder de aan de universiteit opgedane kennis niet had kunnen doen.

Deze scriptie is ook een informatiekunde scriptie en geen informatica scriptie. Dit is uit een aantal zaken af te leiden. In deze scriptie is de nadruk gelegd op het geven van praktische voorbeelden bij de vergelijking. Er wordt steeds een voorbeeld model gegeven met een stuk begeleidende tekst om de vergelijking duidelijk te maken. In het bedrijfsleven worden modellen vaak voorgelegd aan klanten om een versimpelde weergave van de werkelijkheid te geven. Veel modelleertalen zijn zo opgebouwd, dat ze voor leken (met de nodige uitleg) goed te begrijpen zijn. Dit in tegenstelling tot formalismen, waar een grondige uitleg vaak gewenst is. Omdat je als informatiekundige de vertaalslag maakt van technisch- of vakjargon naar voor de klant begrijpbare taal, is de in deze scriptie gehanteerde vergelijkingsmethode volgens mij een goede. Als informatiekundige moet je echter ook de formalismen van een informaticus kunnen begrijpen. Vandaar dat er in deze scriptie ook enkele formalismen zijn gebruikt. Als informatiekundige begeef je je constant in een gebied tussen techniek en mens, en in deze scriptie worden beide gebieden behandeld. Een informaticus zal waarschijnlijk veel meer gericht zijn op de formele kant en zal minder moeite doen om de scriptie voor leken nog enigszins begrijpbaar te maken.

Het vakgebieden waar deze scriptie zich op richt (informatieanalyse, modelleren) zijn ook typische informatiekunde vakgebieden. Om tot een model te komen moet er eerst een analyse van bestaande informatie gedaan worden (zoals documenten) en er zal intensief contact zijn met de klant om informatie in te winnen en een uiteindelijk model te valideren. Aan de hand van een model kan er dan uiteindelijk een systeem gebouwd

worden. Dit laatste is de fase waarin de informatiekundige het stokje overgeeft aan de informaticus. De informaticus kan dan echter nog wel vragen stellen aan de informatiekundige met betrekking tot de informatie binnen het bedrijf of het opgestelde model.

11.3 Verloop van het onderzoek

Ik ben erg tevreden over het verloop van het onderzoek. Het onderzoek kende een vliegende start. Na twee weken waren er al rond de vijftien bladzijden geproduceerd. Hierna werd de snelheid waarmee de scriptie in omvang groeide beduidend minder. Het eerste deel was dan ook het gemakkelijkst. De informatie over de verschillende SQL standaarden was redelijk snel gevonden. Het schrijven van een hoofdstuk hierover kostte dan ook weinig moeite. Hierna werd de moeilijkheidsgraad van de scriptie steeds een stukje meer opgevoerd. Op een gegeven moment wordt er van de onderzoeker verwacht dat hij zelf met ideeën over de materie op de proppen komt.

Ik heb (in overleg) besloten om voor de vergelijking een methode te gebruiken die beschreven wordt in [OHFB92]. Deze vergelijkingsmethode vergelijkt twee talen aan de hand van hun metamodellen. Op de metamodellen worden dan enkele formele relaties toegepast om de overeenkomsten en verschillen boven water te krijgen. Ik heb deze methode uitgebreid door naast de formalismen ook concrete voorbeelden van overeenkomsten en verschillen te geven. Hierdoor wordt de scriptie naar mijn idee te begrijpen voor een grotere groep mensen.

De vergelijking aan de hand van de concrete voorbeelden ging me wel goed af. Soms kostte het wat tijd om een geschikt voorbeeld te vinden die ik kon gebruiken om een overeenkomst of verschil te illustreren. Ik kwam er ook achter dat het SQL metamodel dat ik heb gebruikt in deze scriptie op sommige punten niet volledig was. Het was echter wel het beste SQL metamodel wat ik heb kunnen vinden. De echte problemen kwamen bij de vergelijking aan de hand van de formalismen. In [OHFB92] worden verschillende formele relaties met hun formele definities gegeven om metamodellen mee te vergelijken. Ik begreep de formele definities wel redelijk. Er waren enkele onduidelijkheden die door navraag bij mijn begeleider opgelost werden. Ik wilde de standaard definities aanpassen aan een specifieke ORM/SQL situatie. Hier had ik problemen mee. De standaard definities waren namelijk zo compact beschreven dat er weinig ruimte was voor aanpassingen. Ik had ook moeite met het vinden van geschikte voorbeelden die aansloten bij de voorbeelden uit [OHFB92]. Eigenlijk moesten er delen van de metamodellen worden vergeleken, maar soms wist ik dat er een overeenkomst of verschil was, maar was deze niet terug te zien in de metamodellen. Hierdoor heb ik soms buiten de metamodellen om moeten werken. Uiteindelijk had ik voldoende onderzoek gedaan om de onderzoeksvragen zonder al te veel moeite te kunnen beantwoorden.

Ik ben blij met het verloop van het onderzoek. Ik kende weinig problemen en van de moeilijkheden die ik had met de formalismen heb ik alleen maar geleerd.

Appendix A: SQL tijdlijn

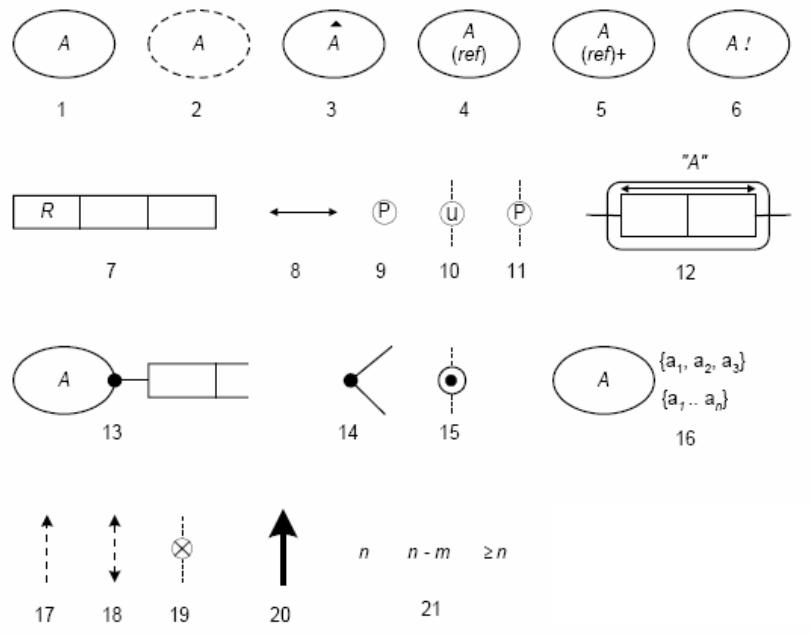
Part	SQL 1992	SQL 1999
Part 1: SQL/Framework		Completed
Part 2: SQL/Foundation	Completed (includes bindings and schema information tables)	Completed
SQL/OLAP		Processed as an amendment to SQL/Foundation
Part 3: SQL/CLI	Completed in '95 as an expansion	Completed
Part 4: SQL/PSM	Completed in '96 as an expansion	Completed (stored routines and call statement moved to SQL/Foundation)
Part 5: SQL/Bindings		Specification for embedding SQL in programming languages moved to a separate part.
Part 6: SQL/Transaction		Project cancelled
Part 7: SQL/Temporal		
Part 8: SQL/Objects - Extended Objects		Merged into SQL/Foundation
Part 9: SQL/MED		ISO version based on SQL'99 in process
Part 10: SQL/OLB	completed in '98 as an ANSI only standard	ISO version based on SQL'99 completed
Part 11: SQL/Schemata		
Part 12: SQL/Replication		Became a project in 2000, with goal of defining syntax and semantics to support definition of replication schemes and rules, including rules for resolution of update conflicts.
Part 13: SQL/JRT	completed in '99 as an ANSI only standard	
Part 14: SQL/XML		

Part	SQL 2003	SQL 2007
Part 1: SQL/Framework	Completed	
Part 2: SQL/Foundation	Completed (SQL/Bindings merged in; schema information tables separated out)	
SQL/OLAP		
Part 3: SQL/CLI	Completed	
Part 4: SQL/PSM	Completed	
Part 5: SQL/Bindings	SQL/Bindings merged back into SQL/Foundation	
Part 6: SQL/Transaction		
Part 7: SQL/Temporal	Withdrawn due to difference of opinion on Part 7	
Part 8: SQL/Objects - Extended Objects		
Part 9: SQL/MED	Revision completed	
Part 10: SQL/OLB	Revision completed	
Part 11: SQL/Schemata	Schema information tables extracted from SQL'99 completed	
Part 12: SQL/Replication	Dropped due to lack of progress.	
Part 13: SQL/JRT	Revision completed	
Part 14: SQL/XML	Completed	Expansion targeted for completion in late 2005. Revision coordinated with the other parts for 2007.

Tabel 5: SQL tijdlijn [JCC06]

Toelichting: bij sommige delen van de SQL standaard staat er in meerdere kolommen dat dat deel 'completed' is. Als dit er staat zonder verdere toelichting (bijvoorbeeld revision completed), houdt het gewoon in dat het al af was en dat er verder niets aan is toegevoegd. In de andere gevallen staat er steeds vermeld wat er is veranderd aan dat deel.

Appendix B: ORM symbolen



Legenda

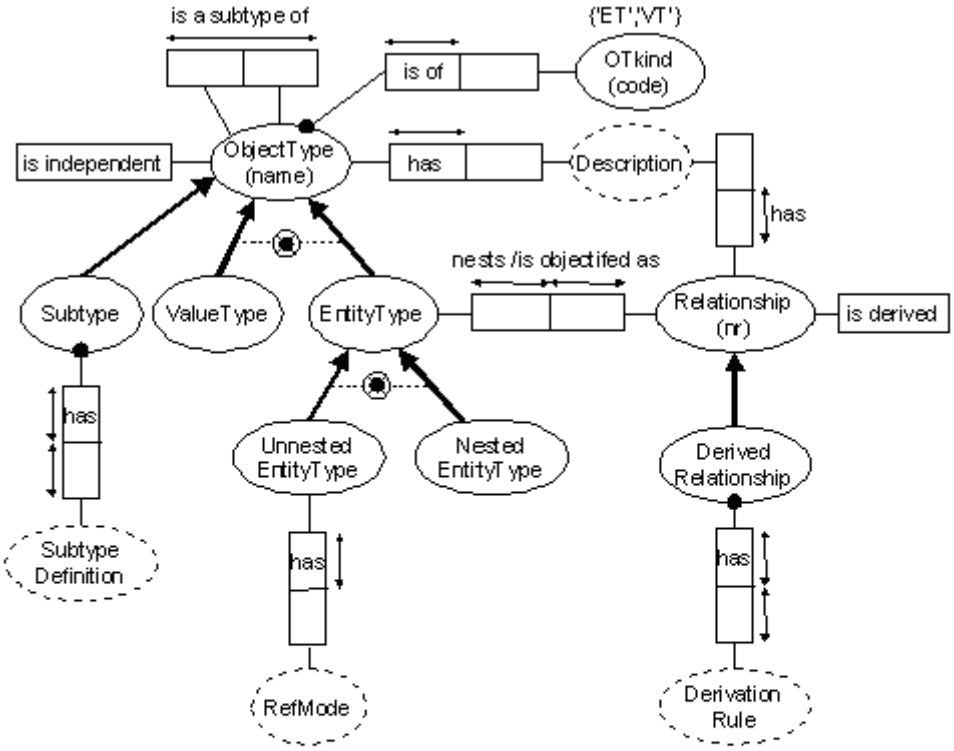
Nummer	Omschrijving
1	Objecttype
2	Labeltype
3	Object type dat vaker voorkomt in één schema
4	Objecttype A wordt geïdentificeerd door ref
5	Objecttype A wordt geïdentificeerd door ref en ref is numeriek
6	Objecttype is onafhankelijk; het hoeft geen onderdeel te zijn van een feittype
7	Een ternair feittype bestaande uit drie rollen. Elke rol hoort bij precies één objecttype
8	Interne uniqueness constraint. Wordt geplaatst boven één of meerdere rollen en geeft aan dat de rolpopulatie uniek moet zijn.
9	De belangrijkste uniqueness constraint wordt aangegeven met de P van primary
10	Externe uniqueness constraint. Wordt geplaatst tussen twee of meer rollen en geeft aan dat de combinatie van rolpopulaties uniek moet zijn.
11	De belangrijkste uniqueness constraint wordt aangegeven met de P van primary
12	Objectificatie; van een feitt ype een objecttype maken
13	Total role constraint. De gehele populatie va n het objecttype A moet die rol spelen
14	Total role constraint. Wordt geplaatst tussen twee of meerdere rollen en geeft aan dat de populatie van een objecttype minstens één van die rollen moet spelen
15	Total role constraint. Wordt geplaatst tussen twee of meerdere rollen en geeft aan dat de populatie van een objecttype minstens één van die rollen moet spelen
16	De mogelijke populatie van objecttype A kan alleen de waarden tussen haakjes bevatten
17	Subset constraint; de populatie van objecttype B is een subset van de populatie van objecttype A, waarbij de pijlpunt naar een rol van objecttype A wijst
18	Equality constraint; de populaties van de rollen waartussen deze pijl geplaatst wordt, moeten gelijk zijn
19	Exclusion constraint. Wordt geplaatst tussen twee of meerdere rollen en geeft aan dat de populatie van een objecttype maar één van deze rollen kan spelen
20	Subtypering; objecttype B is een subtype van objecttype A waarbij de pijlpunt naar objecttype A wijst
21	Frequency constraint. Geeft aan hoe vaak een bepaalde rol gespeeld moet worden: n keer, tussen n en m keer of tenminste n keer

Figuur 36: ORM symbolen [Halp98]

Appendix C: Symbolen formalismen

$\equiv$	$x \equiv y$	*x* en *y* zijn identiek; *x* is een andere naam voor *y*
$=$	$x = y$	*x* is gelijk aan *y*
$\neq$	$x \neq y$	*x* is niet gelijk aan *y*
$\subset$	$A \subset B$	Subset: elk element uit *A* is ook een element uit *B*, maar *A* is niet gelijk aan *B*.
$\subseteq$	$A \subseteq B$	Subset: elk element uit *A* is ook een element uit *B*.
$\cup$	$A \cup B$	Vereniging: de verzameling die zowel alle elementen van *A* als die van *B* bevat, maar geen andere.
$\cap$	$A \cap B$	Doorsnede: De verzameling die alle elementen bevat, die zowel in *A* als in *B* zitten.
$\setminus$	$A \setminus B$	Verschilverzameling: de verzameling van alle elementen uit *A*, die niet in *B* zitten.
$\supset$	$A \supset B$	Superset: elk element uit *B* is ook een element uit *A*, maar *A* is niet gelijk aan *B*.
$\supseteq$	$A \supseteq B$	Superset: elk element uit *B* is ook een element uit *A*.
$\in$	$x \in S$	Element van: *x* is een element van de verzameling *S*.
$\notin$	$x \notin S$	Element van: *x* is geen element van de verzameling *S*.
$\neg$	$\neg A$	Negatie: waar als *A* onwaar is.
$\wedge$	$A \wedge B$	Conjunctie: waar als *A* én *B* waar zijn, anders onwaar
$\vee$	$A \vee B$	Disjunctie: waar als *A* óf *B* (of allebei) waar zijn, als geen van beide waar is dan onwaar.
$\Leftrightarrow$	$A \Leftrightarrow B$	Gelijkwaardigheid (besta): *A* is waar als *B* waar is, *A* is onwaar als *B* onwaar is
$\Rightarrow$	$A \Rightarrow B$	Implicatie: als *A* waar is, dan is *B* ook waar
$\rightarrow$	$f: X \rightarrow Y$	De functie *f* beeldt de verzameling *X* af op de verzameling *Y*
$\forall$	$\forall$	Universele kwantor: voor alle … geldt …
$\exists$	$\exists$	Existentiële kwantor: er is een … waarvoor geldt …
Part	MM1 Part MM2	Partitioning relatie tussen twee metamodellen. MM1 voegt subtypering toe aan de meta concepten uit MM2
Restr	MM1 Restr MM2	Restriction relatie tussen twee metamodellen. MM1 vermindert (door toevoeging van constraints) de mogelijke populatie van meta concepten uit MM2
Deg_w	MM1 Deg_w MM2	Weak degeneration. MM1 bevat alle meta concepten uit MM2, met uitzondering van de generaliserende meta concepten (of supertypen)
Deg_s	MM1 Deg_s MM2	Strong degeneration. MM1 bevat alle generaliserende meta concepten uit MM2, maar bevat een deelverzameling van de specialiserende meta concepten (of subtypen)

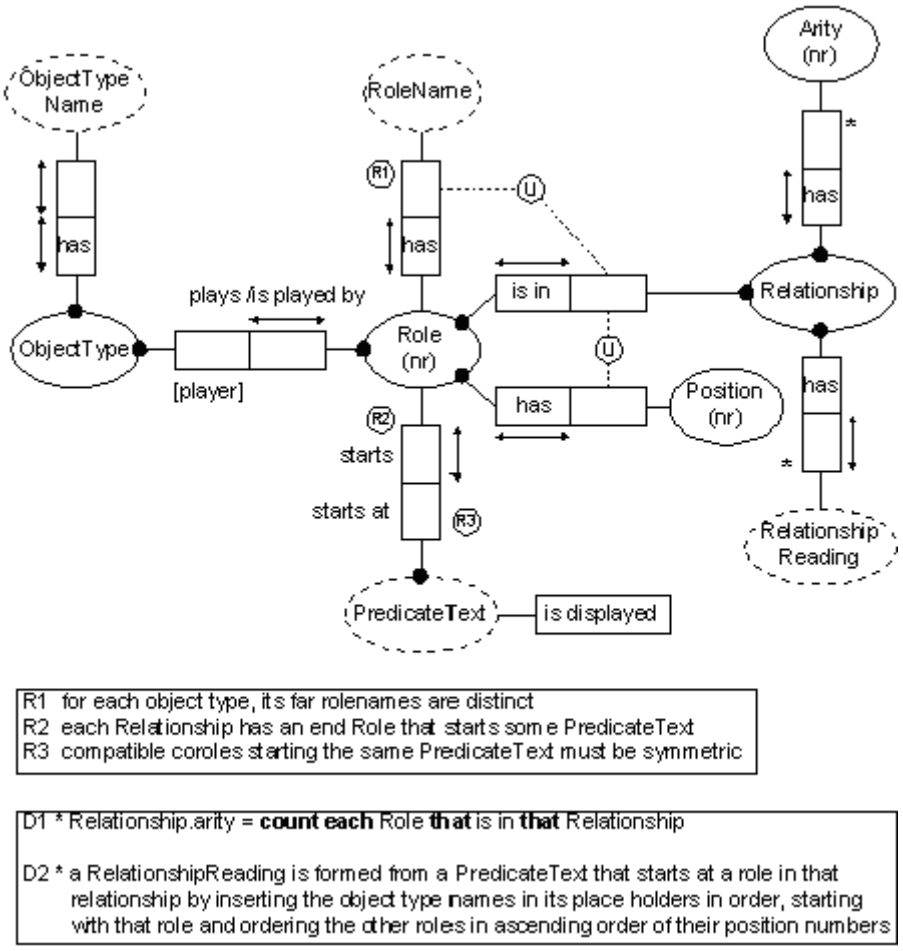
Appendix D: Metamodel ORM



each Subtype **is an** ObjectType **that** is a subtype of **some** ObjectType
each ValueType **is an** ObjectType **that** is of OTkind 'VT'
each EntityType **is an** ObjectType **that** is of OTkind 'ET'
each NestedEntityType **is an** Entity type **that** nests **some** Relationship
each UnnestedEntityType **is an** EntityType **that is not a** NestedEntityType

Apart from FCO-IM, ORM notations typically do not allow nesting of unary relationships. Unlike UML, most ORM dialects require the association to either have a spanning uniqueness constraint or to be 1:1. These restrictions can be introduced to the metamodel by attaching the nesting role to an appropriate subtype of Relationship.

Figuur 37: ORM Metamodel 1: Objecttypen en relaties [Halp00]

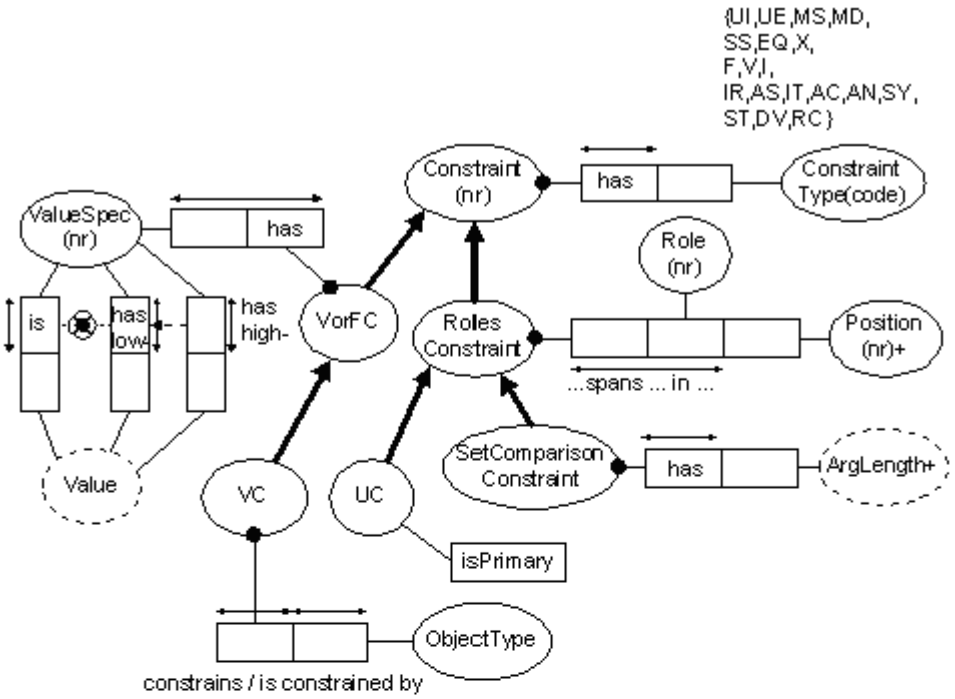


R1 for each object type, its far rolenames are distinct
R2 each Relationship has an end Role that starts some PredicateText
R3 compatible coroles starting the same PredicateText must be symmetric

D1 * Relationship.arity = **count each** Role **that** is in **that** Relationship

D2 * a RelationshipReading is formed from a PredicateText that starts at a role in that relationship by inserting the object type names in its place holders in order, starting with that role and ordering the other roles in ascending order of their position numbers

Figuur 38: ORM Metamodel 2: Objecttypen, predikaten en rollen [Halp00]



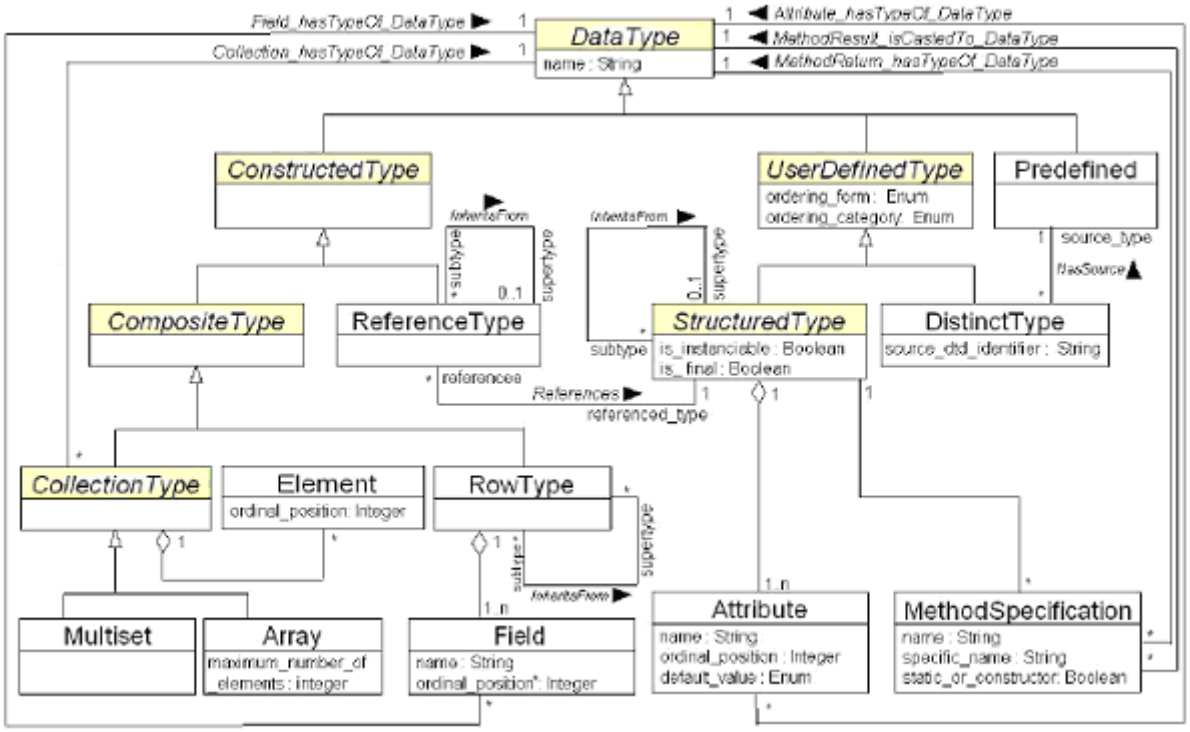
each SetComparisonConstraint **is a** Constraint **that** is of ConstraintType **in** {'SS','EQ','X'}
each UC **is a** Constraint **that** is of ConstraintType **in** {'UI','UE'}
each VC **is a** Constraint **that** is of ConstraintType 'V'
each VorFC **is a** Constraint **that** is of ConstraintType **in** {'V','F'}
each RolesConstraint **is a** Constraint **that** is of ConstraintType **in**
{'UI','UE','MS','MD','SS','EQ','X','F','I','IR','AS','IT','AC','AN','SY','RC','DV'}

We ignore the possible but pathological case of the same role appearing more than once in the constraint. If ever we wish to cater for this, have the UC span the ternary and allow position markers in the UI to be a comma-delimited list

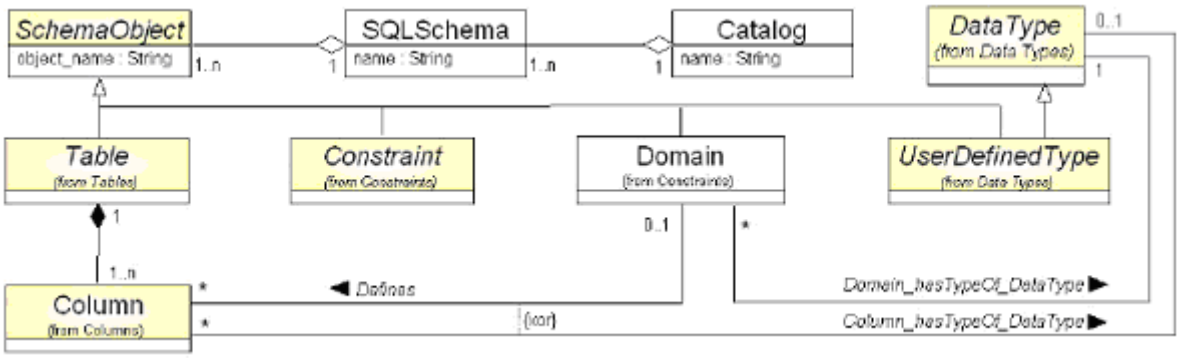
UI	Uniqueness Internal		
UE	Uniqueness External		
MS	Mandatory Simple		
MD	Mandatory Disjunctive		
SS	SubSet		
EQ	EQuality		
X	eXclusion		
F	Frequency		
I	Index	ST	SubType
IR	IRreflexive		– formal definitions with mapping
AS	ASymmetric	DV	Default Value
IT	InTransitive		– e.g. default nrBugs = 0
AC	ACyclic	RC	other Rule Constraint
AN	ANtisymmetric		– e.g. any rule specifiable via ConQuer
SY	SYmmetric		
V	Value domain		

Figuur 39: ORM Metamodel 3: Constraints [Halp00]

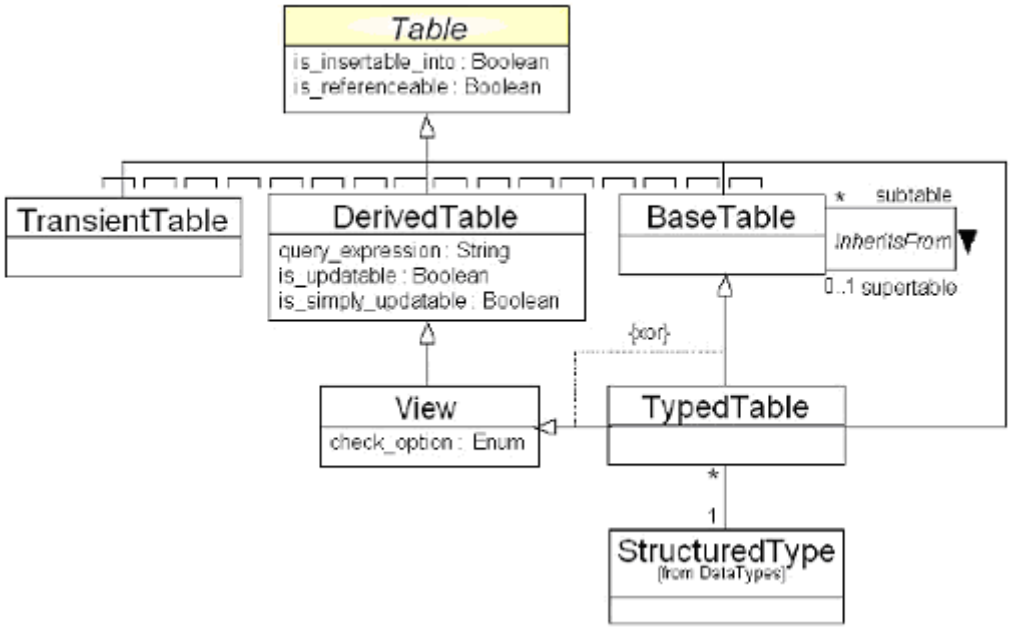
Appendix E: Metamodel SQL



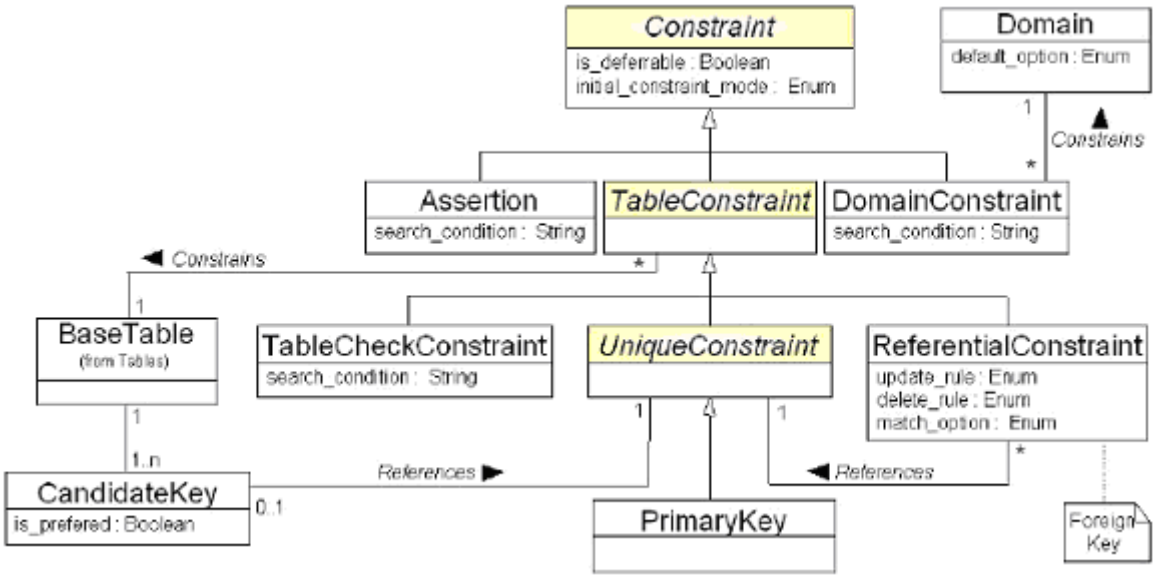
Figuur 40: SQL Metamodel 1: Datatypen [CRBaBrP]



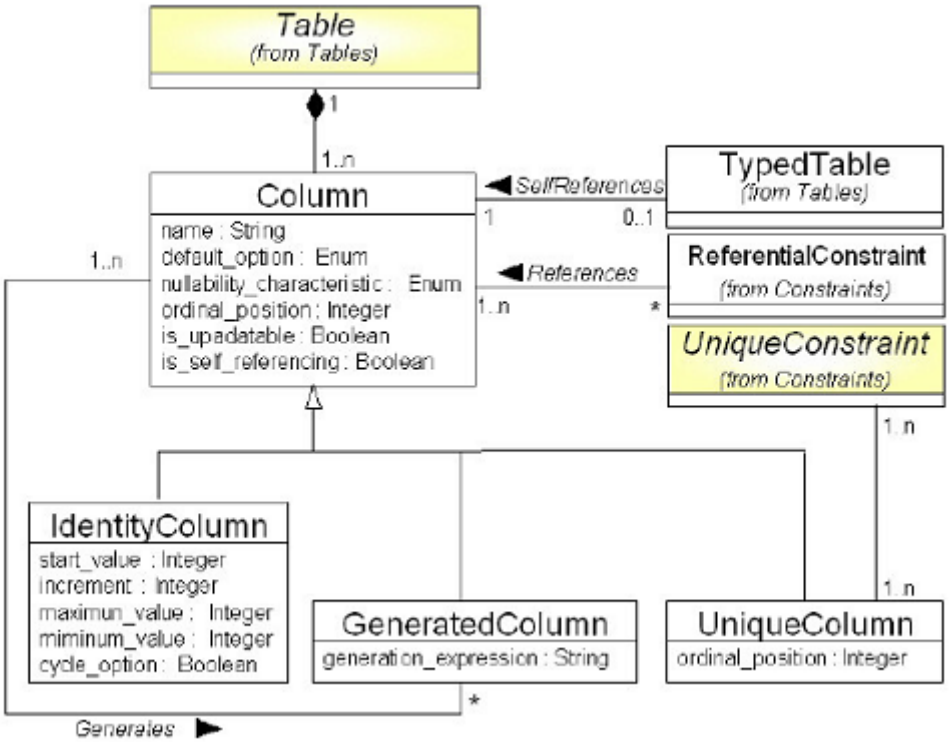
Figuur 41: SQL Metamodel 2: Schema objecten [CRBaBrP]



Figuur 42: SQL Metamodel 3: Tabellen [CRBaBrP]



Figuur 43: SQL Metamodel 4: Constraints [CRBaBrP]



Figuur 44: SQL Metamodel 5: Kolommen [CRBaBrP]

Bronvermelding

Internet

[Andries] Structured Query Language
http://docent.ehsal.be/vakken/infoburo/Access/SQL.html

[ANSIH2] ANSI NCITS H2 Technical Committee on Database ledenlijst 2005
http://www.incits.org/tc_home/a_reports2005/in050382/H2-SD-006_2005-05-06.pdf

[Chapple] Referential Integrity
http://databases.about.com/cs/administration/g/refintegrity.htm

[CLI01] Call Level Interfaces
http://www.canaimasoft.com/f90SQL/OnlineManual/Chapter01/Call%20level%20interfaces.htm

[EssSt99] Object Role Modeling
http://www.essentialstrategies.com/publications/modeling/orm.htm

[GaVa98] The Relational Data Model
http://www.soi.city.ac.uk/~tony/dbms/relational_dm.html

[Gorm05] Is SQL A Real Standard Anymore?
http://www.wiscorp.com/is_sql_a_real_standard.pdf

[IBM05] IBM Informix Dynamic Server v10.0 Information Center
http://publib.boulder.ibm.com/infocenter/idshelp/v10/index.jsp?topic=/com.ibm.sqlr.doc/sqlrmst143.htm

[JCC06] JCC's SQL Standards Page
http://www.jcc.com/sql.htm

[Mimer] Introduction to SQL
http://developer.mimer.com/documentation/Mimer_SQL_Reference_Manual/Intro_SQL_Stds3.html

[ModMet] Modeling methodologies (ORM)
http://www.aisintl.com/case/method.html#ORM

[MOey96] Inleiding SQL
http://www.cs.vu.nl/~michel/pracdb/SUPPL/node2.html

[ORDBMS] Object-Relational DBMS
http://www.his.se/upload/21980/ORDBMS.pdf

[ORMObj] ORM Objectification
http://www.orm.net/pdf/objectification.pdf

[SOracle] Dynamic SQL
http://searchoracle.techtarget.com/sDefinition/0,,sid41_gci927933,00.html

[SQLCT] SQL Standard Collection Types
http://farrago.sourceforge.net/design/CollectionTypes.html

[SQLTeam] Stored Procedures: Returning Data
http://www.sqlteam.com/item.asp?ItemID=2644

[SQLU] SQL UNDER statement
http://docs.openlinksw.com/virtuoso/CREATETABLE.html#under

[SQLX] SQL/XML
http://www.sqlx.org/

[SynSem] Syntaxis en Semantiek
http://biology.leidenuniv.nl/~zandee/algo/syllabus/h2.pdf

[Werf05] Hoe de database wiskundig werd; de acceptatie van het relationele model
http://www.win.tue.nl/~hemerik/2R930/Scripties/HoC_056_vdWerf/geschiedenis.pdf

[White99] Great News, The Relational Data Model is Dead! Presentation
www.ncb.ernet.in/education/modules/dbms/SQL99/Sql99_c2.pdf

[Wiki01] Wikipedia NL: SQL
http://nl.wikipedia.org/wiki/SQL
[Wiki02] Wikipedia NL: XML
http://nl.wikipedia.org/wiki/XML
[Wiki03] Wikipedia EN: SQL
http://en.wikipedia.org/wiki/SQL
[Wiki04] Wikipedia EN: Relational Database Management System
http://en.wikipedia.org/wiki/Relational_database_management_system
[Wiki05] Wikipedia EN: Relational Model
http://en.wikipedia.org/wiki/Relational_model
[Wiki06] Wikipedia NL: Relationele Model
http://nl.wikipedia.org/wiki/Relationele_model
[Wiki07] Wikipedia EN: Transact-SQL
http://en.wikipedia.org/wiki/Transact-SQL
[Wiki08] Wikipedia EN: PL/SQL
http://en.wikipedia.org/wiki/PL_SQL

Artikelen

[ChBo74] Donald D. Chamberlin, Raymond F. Boyce. SEQUEL: A structured English query language, *SIGMOD Workshop,* vol. 1:249-264, mei 1974
http://portal.acm.org/citation.cfm?id=811515
[Codd70] E.F. Codd. A Relational Model for Large Shared Data Banks. *Communications of the ACM*, 13(6):377-387, juni 1970
http://www.acm.org/classics/nov95/toc.html
[CRBaBrP] C. Calero, F. Ruiz, A. Baroni, F. Brito e Abreu, M. Piattini. An Ontological Approach to Describe the SQL:2003 Object-Relational Features. *Journal of Computer Standards & Interfaces*, Elsevier, 2005
http://www-ctp.di.fct.unl.pt/QUASAR/Resources/Papers/2005/baroniCSI.pdf
[EiMe01] A. Eisenberg, J. Melton. SQL/XML and the SQLX Informal Group of Companies. *ACM SIGMOD Record*, 30(3):105-108, september 2001
http://www.acm.org/sigmod/record/issues/0109/standards.pdf
[EiMe02] A. Eisenberg, J. Melton. SQL/XML is Making Good Progress. *ACM SIGMOD Record*, 31(2):101-108, juni 2002
http://www.sigmod.org/record/issues/0206/standard.pdf
[EiMe99] A. Eisenberg, J. Melton. SQL:1999, Formerly Known As SQL3. *ACM SIGMOD Record*, 28(1):131-138, maart 1999
http://www.cl.cam.ac.uk/Teaching/2003/Databases/sql1999.pdf
[EMeKMiZ] A. Eisenberg, J. Melton, K. Kulkarni, J. Michels, F. Zemke. SQL:2003 Has Been Published. *ACM SIGMOD Record*, 33(1):119-126, maart 2004
http://www.sigmod.org/sigmod/record/issues/0403/E.JimAndrew-standard.pdf
[Halp00] Dr. Terry Halpin. An ORM Metamodel. *Journal of Conceptual Modeling*, oktober 2000
http://www.inconcept.com/JCM/October2000/halpin.html
[Halp02] Dr. Terry Halpin. Modeling Collections in UML and ORM. *Proc. EMMSAD'00: 5th IFIP WG8.1 Int. Workshop on Evaluation of Modeling Methods in Systems Analysis and Design,* juni 2000
http://www.orm.net/pdf/EMMSAD2000.pdf
[Halp98] Dr. Terry Halpin. Object Role Modeling (ORM/NIAM). *Handbook on Architectures of Information System,* Chapter 4, 1998
http://www.orm.net/pdf/springer.pdf
[Halp99] Dr. Terry Halpin. UML Data Models from an ORM Perspective (Part 10). *Journal of Conceptual Modeling*, augustus 1999
http://www.inconcept.com/jcm/August1999/halpin.html

[Halp05] Dr. Terry Halpin. ORM 2 Graphical Notation. *Technical Report ORM2-01*, September 2005 http://www.orm.net/pdf/ORM2_TechReport1.pdf

[HPW04] A.H.M. ter Hofstede, H.A. Proper, Th.P. van der Weide. PSM: Datamodelleren in het Kwadraat, *DB/Magazine*, 3(4):37-41, juni 2004 https://osiris.cs.kun.nl/pms/iris-diglib/src/getContent.php?id=1992-Hofstede-PSMPromoNL

[KKR01] G. Kappel, E. Kapsammer, W. Retschitzegger. XML and Relational Database Systems – A Comparison of Concepts, *Proceedings of the 2001 International Conference on Internet Computing*, pp. 199-205, juni 2001 ftp://ftp.ifs.uni-linz.ac.at/pub/publications/2001/0501.pdf

[MEL03] J. Melton. SQL 2003 standard – ISO-ANSI Working Draft, juli 2003 http://www.wiscorp.com/sql_2003_standard.zip

[OHFB92] J. Oei, L. van Hemmen, E. Falkenberg, S. Brinkkemper. The Meta Model Hierarchy: A Framework for Information Systems Concepts and Techniques, juli 1992 http://mirror.eacoss.org/documentation/ITLibrary/IRIS/Data/1992/Oei/MMHierarchy/1992-Oei-MMHierarchy.pdf

[Web05] Jaap Weber. Afstudeerscriptie Ontologieën en PSM, december 2005 http://www.infstud.science.ru.nl/~jaapwebe/Ontologieen&PSM-JaapWeber.pdf

[Visio97] Visio Corporation. Guide to FORML, 1997-1998. http://www.cs.ru.nl/~gerp/IS0/dictaat/4a_Guide_to_Forml.pdf

Boeken

[Kroe02] David M. Kroenke. *Databases: beginselen, ontwerp en implementatie*, achtste editie, 2002

[Lans01] Rick F. van der Lans. *Het SQL leerboek*, vijfde herziene uitgave, 2001

[Halp01] T.A. Halpin. *Information Modeling and Relational Databases*, *From Conceptual Analysis to Logical Design*, 2001

Lijst van figuren