De relationele revolutie

Query's door de jaren heen

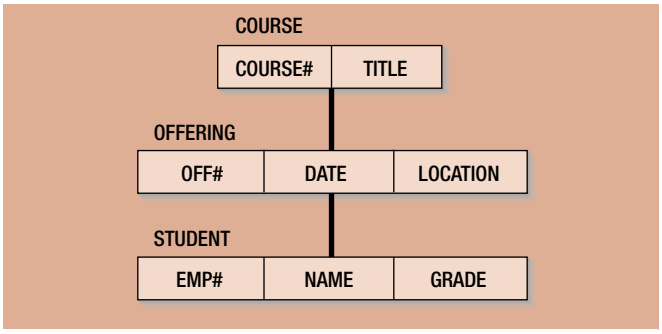
Frido van Orden

In de prerelationele dagen vergde zelfs het uitvoeren van de meest eenvoudige raadpleeg- of mutatieacties op een gegevensverzameling een aanzienlijke programmeerinspanning. Met name het specificeren van de toegangspaden betekende veel werk, waarbij zowel aspecten van het gegevensmodel als van de implementatie (indexen) in ogenschouw moesten worden genomen.

DE PRE(RELATIONELE)HISTORIE

Neem bijvoorbeeld het gegevensmodel zoals weergegeven in figuur 1 (de lezers van Date's 'An Introduction to Database Systems', en wie onder u is dat niet, zal dit bekend voorkomen). Voor de 40-minners onder u: het gegevensmodel van figuur 1 geeft een hiërarchie weer. Bovenaan in de hiërarchie staat een segment Course waarin records die cursussen beschrijven kunnen worden opgeslagen. Aan elke Course is een reeks Offerings gekoppeld, geïmplementeerd door middel van een pointerketting Course ➔ Offering 1 ➔ Offering 2 ➔ ... ➔ Offering n ➔ Course. Hetzelfde geldt voor Students binnen Offerings. In dit model is een Student, geheel tegen de intuïtie, dus géén zelfstandige entiteit! Een student die aan meerdere Offerings deelneemt wordt meermalen opgeslagen.

Een vraag als *Geef alle offerings van course 'Advanced Programming' die in Stockholm plaatsvinden* ziet er in bijvoorbeeld IMS DL/I (de 'querytaal' van het hiërarchische IMS-dbms) als volgt uit:



Figuur 1: Hiërarchisch model.

```
GU COURSE WHERE TITLE = 'Advanced Programming'
(using PCB-1) ;
Do until 'not found' on PCB-1 ;
  GNP OFFERING WHERE OFF# = '8' ;
  GNP OFFERING WHERE LOCATION = 'Stockholm';
end;
```

Hierin staat de operatie 'GU' voor 'Get unique' (wat op verwarrende wijze feitelijk 'Get first' betekent) en 'GNP' voor 'Get next within parent'. Merk op, het feit dat het statement expliciet afdwingt dat eerst de cursus wordt geselecteerd en vervolgens de offerings worden doorlopen. Dit komt overeen met de fysieke opslagstructuur van het gegevensmodel, waarin Offerings slechts benaderbaar zijn via een Course (tenzij separaat geïndexeerd) en Students slechts benaderbaar zijn via een Offering (wederom: tenzij separaat geïndexeerd).

De syntax van mutatie-statements kent een ogenschijnlijk minder rigide structuur, al valt bij nauwkeurige beschouwing ook hier niet aan de structuur van pointerkettingen in het gegevensmodel te ontkomen:

Voeg de student met EMP# 275404 en zonder GRADE toe aan Offering 8 van Course M23:

```
build new STUDENT segment in I/O area (EMP# =
'275404', GRADE = ' ');
ISRT COURSE WHERE COURSE# = 'M23' ,
  OFFERING WHERE OFF# = '8',
  STUDENT;
```

Het dbms voorbij (11)

In de voorgaande artikelen van 'Het DBMS voorbij' is reeds aandacht besteed aan diverse aspecten van object-oriëntatie, maar met name aan ondersteuning van typehiërarchieën. In deze aflevering gaan we nader in op de 'buitenkant' van object-oriëntatie: de programmeerinterface.

Verplaats Offering 8 van Course M23 naar Helsinki:

```
GU COURSE WHERE COURSE# = 'M23' ;
  GHNP OFFERING WHERE OFF# = '8' ;
  Change OFFERING segment in I/O area
    (LOCATION = 'Helsinki') ;
  REPL ;
```

Verplaats Offering 8 van alle Courses 'Advanced Programming' naar Helsinki:

```
GHU COURSE WHERE TITLE = 'Advanced Programming'
(using PCB-1) ;
Do until 'not found' on PCB-1 ;
  GNP OFFERING WHERE OFF# = '8' ;
  Change OFFERING segment in I/O area
    (LOCATION = 'Helsinki') ;
  REPL ;
End ;
```

Merk op, het feit dat er niet noodzakelijk slechts één 'Advanced Programming' Course is, tot gevolg heeft dat de programmatuur aanzienlijk moet worden aangepast: er moet een loop-structuur worden toegevoegd.

Verwijder Offering 8 van Course M23:

```
GHU COURSE WHERE COURSE# = 'M23' ,
  OFFERING WHERE OFF#= '8' ;
DLET ;
```

DE SPRONG VOORWAARTS: DE RELATIONELE REVOLUTIE

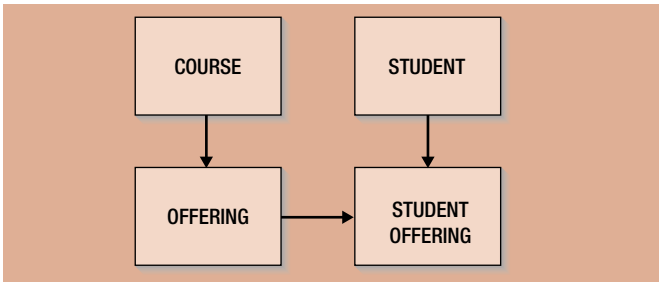
Met de komst van het relationele model werden we verlost van de fysieke constructies die zichtbaar waren in het – logische! – gegevensmodel. Het relationele gegevensmodel dat overeenkomt met het hiërarchische model uit figuur 1 is weergegeven in figuur 2. 'Student' is daarin een zelfstandige entiteit is geworden.

De queries van zojuist zien er in SQL als volgt uit:

Geef alle offerings van course 'Advanced Programming' die in Stockholm plaatsvinden:

```
SELECT O.*
FROM OFFERING O, COURSE C
WHERE C.TITLE = 'Advanced Programming'
  AND O.LOCATION = 'Stockholm'
  AND C.COURSE# = O.COURSE#;
```

De strenge hiërarchie en record-at-a-time toegang van DL/I is vervangen door de set-a-time toegang van SQL. De navigatie van



FIGUUR 2: RELATIONEEL/OO-MODEL.

Course naar Offering in het update-statement is vervangen door een join. Belangrijk verschil is dat deze join een puur logische constructie is en niets zegt over fysiek structuren als indexen of pointerkettingen. In het algemeen zegt een join zelfs niets over een verwijzende sleutel-relatie – het relationele equivalent van de pointer – al wordt in dit specifieke geval natuurlijk gewoon over de verwijzende sleutel van Offering naar Course gejoind.

Voeg de student met EMP# 275404 en zonder GRADE toe aan Offering 8 van Course M23:

```
Insert into STUDENT_OFFERING(COURSE#,OFF#,EMP#)
Values ('M23','8','275404');
```

Verplaats Offering 8 van Course M23 naar Helsinki:

```
UPDATE OFFERING
SET LOCATION = 'Helsinki'
WHERE COURSE# = 'M23'
AND OFF# = '8';
```

Verplaats Offering 8 van alle Courses 'Advanced Programming' naar Helsinki:

```
UPDATE OFFERING
SET LOCATION = 'Helsinki'
WHERE COURSE# IN (SELECT COURSE#
  FROM COURSE
  WHERE TITLE = 'Advanced Programming')
AND OFF# = '8';
```

Hier heeft SQL toch tenminste in de syntax de schijn tegen. De ogenschijnlijk geringe aanpassing in de vraag leidt tot een aanzienlijk verschillend statement. Het tweede statement wekt bovendien sterk de – misleidende – suggestie van een bepaalde volgorde bij de uitvoering.

Tot slot: *verwijder Offering 8 van Course M23:*

```
DELETE FROM OFFERING
WHERE COURSE# = 'M23' AND OFF# = '8';
```

De structuur van delete en update statements in SQL is, afgezien van het keyword en de SET clause, identiek, zodat de opmerkingen bij updates ook bij deletes kunnen worden geplaatst.

DE OO-TOEKOMST?

De overgang van pre-relationeel naar relationeel kenmerkt zich enerzijds door de overgang van procedureel (hoe) naar declaratief (wat) en de bijkomende abstractie van fysieke opslagstructuren. Hierdoor kunnen programma's worden geschreven die onafhankelijk zijn van de fysieke opslag en automatisch meeprofiteren van optimalisaties in die fysieke opslagstructuur, bijvoorbeeld de toevoeging van een index.

Object-oriëntatie wordt vaak verweten dat zij 'oude wijn in nieuwe zakken' is, en geheel ten onrechte is dat verwijt niet. Zo zagen we reeds dat de pointerketting uit het hiërarchische model werd vervangen door het abstracte en implementatie-onafhankelijke concept van de relationele verwijzende sleutel. In het object-georiënteerde paradigma wordt in zekere zin weer een stap teruggedaan. In ons voorbeeldgegevensmodel is dan bijvoorbeeld sprake van een verwijzing van een Offering naar een Course (de term 'verwijzing' is de OO-term voor een pointer). Deze verwijzing is opgenomen als een attribuut 'course' van de klasse Offering en heeft als type 'Course' (of preciezer gezegd: pointer-naar-Course). Evenzo bevat de klasse Course een attribuut 'offerings' van type lijst-van-Offerings (of array-van-Offerings, of set-van-Offerings). De bezwaren van deze benadering zijn:

- Er zijn nu twee attributen ('course' in de klasse Offering en 'offerings' in de klasse Course) die dezelfde informatie bevatten, namelijk de associatie tussen Courses en Offerings.
- Het attribuut 'offerings' in de klasse Course kan verschillende gegevenstypen aannemen (lijst-van-Offerings, array-van-Offerings, set-van-Offerings) die dezelfde informatie uitdrukken maar een verschillende programmeerinterface kennen (zo zijn lijsten en arrays geordend en sets niet, en zijn lijsten en is het aantal elementen in sets en lijsten onbeperkt variabel en in arrays over het algemeen niet).

Een ander bezwaar is dat in de OO-wereld voor raadpleegacties gebruik gemaakt kan worden van een vraagtaal die enigszins vergelijkbaar is met SQL (veelal OQL of iets dergelijks genaamd), terwijl voor mutaties gebruik wordt gemaakt van de faciliteiten van een OO-programmeertaal als C++ of Java. Deze faciliteiten zijn vrijwel zonder uitzondering record-georiënteerd en ontberen de mogelijkheden voor set-level manipulatie zoals het relationele model en SQL die bieden.

Relationelen als Chris Date zijn dan ook van mening dat het object-georiënteerde gegevensmodel helemaal geen gegevensmodel is, waar het relationele model dat wel is, namelijk een formeel mechanisme om business-modellen (de visie van gebruikers) te vertalen in een representatie in een database. Object-oriëntatie is een verzameling regels voor het schrijven van 'goede' programma's en geen theorie over gegevensbeheer. Al deze kritieken op de OO-visie op databases (veelal aangeduid als 'persistente gegevens') zijn voor een groot deel waar. Nemen we als voorbeeld het mechanisme voor het omgaan met persistente gegevens zoals dat ontwikkeld is voor de programmeertaal Java onder de naam 'Java Data Objects'. Een typisch code-fragment ziet er zo uit:

```
// Maak een nieuwe query.
// pm = persistence manager = 'connectie'
Query qry = pm.newQuery();
// de kandidaten voor de query
// (vgl. SQL from clause) zijn alle
// persistent objecten van de klasse Offering
qry.setCandidates(pm.getExtent("Offering"));
// Query parameters, vergelijk placeholders in SQL
qry.declareParameters("String courseTitle,
String loc");
// Query filter, vergelijk SQL WHERE clause
qry.setFilter("COURSE.TITLE = courseTitle &&
LOCATION = loc");
// Resultaat van het uitvoeren van de query is een
// verzameling objecten
Collection c = qry.execute('Advanced Programming',
'Stockholm');
// Itereer over de zojuist opgehaalde Offering
// objecten
for (Iterator it = c.iterator(); it.hasNext(); ) {
   Offering off = (Offering) it.next();
   // Verplaats de Offering naar Helsinki
   off.setLocation('Helsinki');
   // Maak een nieuwe StudentOffering
   StudentOffering so = new StudentOffering();
   // Vul de verwijzing naar de Student middels een
   // sleutelattribuut
   so.setStudentEmpNo('275404');
   // Vul de verwijzing naar de Offering via een
   // reference attribuut
   // Impliciet: maak dit object persistent, want
   // off is persistent
   so.setOffering(off);
}
```

Wie verknocht is aan de kracht van SQL zal dit fragment afdoen als inferieur aan de equivalente SQL-statements:

```
UPDATE OFFERING
SET LOCATION = 'Helsinki'
WHERE LOCATION = ?
  AND COURSE# IN (SELECT COURSE# FROM COURSE WHERE
  TITLE = ?)

INSERT INTO STUDENT_OFFERING(EMP#, COURSE#, OFF#)
SELECT '275404', C.COURSE#, O.OFF#
FROM OFFERING O, COURSE C
WHERE O.LOCATION = ? AND C.TITLE = ?
  AND O.COURSE# = C.COURSE#
```

Toch valt ook op deze twee, relatief eenvoudige, SQL-statements het een en ander af te dingen:

- Het selectiecriterium op locatie en titel komt tweemaal voor, maar in verschillende gedaanten (1x als join, 1x als subselect).

- Wat gebeurt er indien de waarden uit het voorbeeld niet rechtstreeks worden gegeven, maar worden bepaald aan de hand van een procedurele berekening? De set-level insert van het tweede SQL-statement valt dan bijvoorbeeld uiteen in individuele inserts van telkens 1 record.
- De declaratieve SQL-statements beslaan totaal negen regels code tegen elf voor het procedurele Java-fragment. Bedenk daarbij echter dat SQL zelden of nooit zelfstandig wordt uitgevoerd maar dat vrijwel altijd gebruik wordt gemaakt van een host-omgeving (bijvoorbeeld Cobol of Java). In de host-omgeving moeten op zijn minst weer enige regels worden gecodeerd voor het alloceren en opruimen van statements en het zetten van de variabelen voor locatie en titel. Per saldo moeten bij gebruik van SQL dus eerder meer dan minder regels worden gecodeerd.

Maar hoe staat het dan met het eerder genoemde bezwaar dat bijvoorbeeld het attribuut 'course' in de klasse Offering en het attribuut 'offerings' in de klasse Course dezelfde informatie uitdrukken?

Hier ligt een fraaie gelegenheid om de voordelen van het relationele model (dat kernachtig beschrijft welke betekenis aan gegevens moet worden toegekend) en het object-georiënteerde model (dat 'handig' is) te combineren. Wat specificeren we bijvoorbeeld als we de waarde van het attribuut 'course' in een Offering-object wijzigen'? Welnu, het relationele model leert ons dat we de associatie tussen de 'oude' Course en de Offering verwijderen en een associatie tussen de 'nieuwe' Course en de Offering toevoegen. Daarmee zeggen we tevens dat de Offering uit de verzameling Offerings in (de waarde van) het attribuut 'offerings' in de 'oude' Course moet worden verwijderd en moet worden toegevoegd aan de verzameling Offerings in (de waarde van) het attribuut 'offerings' in de 'nieuwe' Course. Het enige dat daarvoor 'onder de motorkap' bekend moet zijn is het feit dat het attribuut 'course' in de klasse Offering en het attribuut 'offerings' in de klasse Course gebaseerd zijn op hetzelfde relationele concept: de verwijzende sleutel van Offering naar Course.

Deze gedachtegang kunnen we verder doortrekken. Nemen we als voorbeeld de klasse/entiteit 'Student Offering'. Deze bestaat in de relationele visie uit de attributen 'Emp#', 'Course#' en 'Off#'. Het eerste attribuut vormt een verwijzende sleutel naar de entiteit Student en de laatste twee attributen vormen tezamen een verwijzende sleutel naar de entiteit 'Offering'. In de object-georiënteerde visie bestaat de klasse uit de attributen 'student' en 'offering' die als waarde een verwijzing (pointer) naar respectievelijk een object van klasse Student en een object van klasse Offering vormen. Waarom kiezen tussen deze beide representatievormen? We zagen dat het gelijktijdig voorkomen van het attribuut 'course' in de klasse Offering en het attribuut 'offerings' in de klasse Course geen probleem is als deze attributen onderling consistent worden gehouden. Dan mag het ook geen probleem zijn om de klasse 'Student Offering' te voorzien van alle vijf de attributen 'Emp#',

'Course#', 'Off#, 'student' en 'offering', mits we kunnen garanderen dat bij wijzigingen van 'Emp#' de waarde van 'student' navenant wordt aangepast (en omgekeerd). Hetzelfde geldt voor de combinatie 'Course#' en 'Off#' enerzijds en 'offering' anderzijds.

Hierbij liggen wel twee adders onder het gras, die gelukkig eenvoudig onschadelijk zijn te maken. De eerste adder is wat er moet gebeuren indien aan 'Emp#' een waarde wordt toegekend die niet voorkomt in de tabel 'Student'. Dit is het equivalent van een schending van een verwijzende sleutel en er zijn daarom twee mogelijkheden om hiermee om te gaan. Ofwel de mutatie wordt verboden (vergelijkbaar met de SQL constraint-optie 'CHECK IMMEDIATE'), ofwel de mutatie wordt tijdelijk toegestaan, waarna bij het committen van de transactie alsnog een fout wordt teruggegeven indien de inconsistentie niet is verholpen (vergelijkbaar met de SQL constraint-optie 'CHECK DEFERRED'). In dat geval moet de waarde van het reference attribuut 'student' tijdelijk op de waarde 'null' (de null pointer, niet te verwarren met het concept NULL in SQL) worden gesteld.

De tweede adder is een variant op de eerste: wat als van een 'Student Offering' object zowel het Course# als het Off# worden gewijzigd, en wel zo dat alleen het wijzigen van beide attributen weer een geldige Offering-verwijzing oplevert. Dit lijkt sterk op de vorige situatie, ware het niet dat we in SQL ook als de constraint-optie 'CHECK IMMEDIATE' wordt gekozen de mutatie door kunnen voeren door middel van:

```
UPDATE STUDENT_OFFERING
SET COURSE# = ?, OFF# = ?
WHERE ...
```

Dit kunnen we niet simuleren in OO, tenzij we naast de standaardmethoden setCourse#() en setOff#() een extra methode setCourse#EnOff#() toevoegen, waarbij vanuit constraint management-perspectief beide mutaties atomair worden beschouwd.

WAT KUNT U VANDAAG KOPEN?

Zoals bij alle onderwerpen die in deze serie de revue zijn gepasseerd geldt, ook hier weer 'close, but no cigar'. Vele leveranciers brengen tools op de markt waarmee de programmeermodellen van object-georiënteerde applicaties zijn af te beelden op robuuste, relationele database-infrastructuren. De visie is echter veelal gebaseerd op het primaat van object-georiënteerde modellen die op den duur moeten worden afgebeeld op relationele 'legacy'. Dat is jammer, want juist het verrijken van het object-georiënteerde paradigma met enige oude relationele wijsheden zou de acceptatie en het gebruik van dergelijke tools aanzienlijk kunnen vergroten. ●

Ir. F.G.W. van Orden (fridoo@faapartners.com) is managing partner van FAA Partners BV