Twee left-overs ter afsluiting

Flexibele interface en 'managed redundancy'

Frido van Orden

De pionierstijd van Database Magazine viel samen met de opkomst van relationele systemen. Gaandeweg werden de 'Tips en Truuks' bijdragen van de Clipper-goeroes vervangen door de voortrekkers van SQL, daarbij gebruik makend van dBase IV of misschien zelfs wel Oracle versie 4. Een welhaast onherkenbaar beeld voor de lezer die DB/M alleen van recente jaren kent als een vakblad met serieuze marktverkenningen, product evaluaties en conceptuele artikelen, of dan toch op zijn minst met artikelen die het niveau van 'handigheidjes' ver ontstijgen.

Weest u gerust, we zullen met dit laatste artikel van 'Het dbms voorbij' in stijl afsluiten. Ook de variëteit aan functionaliteit die deze serie afsluit, zal weer zijn gebaseerd op een robuust gegevensmodel, waarbij de alom bekende relationele sleutels deze keer de hoofdrol spelen. Een 'Tips en Truuks' nieuwe stijl dus.

DYNAMISCHE GEBRUIKERSINTERFACES

Over presentatie-aspecten, en de relevantie daarvan in een artikelserie die nota bene de titel 'Het dbms voorbij' draagt, is in het artikel over presentatie (aflevering 7 van de serie, DB/M 5/2002) al uitgebreid geschreven. Daarbij is voornamelijk aandacht besteed aan de invloed van regels (veelal validatie- of autorisatieregels) op de gebruikersinterface.

Een van de bekendste vormen van vertaling van regels in gedrag is zonder meer het master-detail scherm. Vrijwel geen enkele handleiding over applicatie-ontwikkeling ontkomt aan het klassieke Employee-Department gegevensmodel, dat er in essentie als volgt uitziet:

```
Afdeling(Afd#, Naam, Budget, Afd#_Hoger,
Med#_Manager)

Medewerker(Med#, Naam, Salaris, Afd#)
```

De attributen Afd# in Afdeling en Med# in Medewerker vormen de primaire sleutel van de respectievelijke tabellen, en het attribuut Afd# in Medewerker vormt een verwijzende sleutel naar het overeenkomstige attribuut in Afdeling. Het attribuut Afd#_Hoger in Afdeling vormt een verwijzende sleutel naar de tabel Afdeling zelf en geeft de hiërarchie in afdelingen weer. Het attribuut Med#_Manager tenslotte, vormt een verwijzende sleutel naar het attribuut Med# in Medewerker en geeft aan welke medewerker de manager van de afdeling is.

Dan is de gebruiker echt ontwikkelaar geworden, en vindt het nog leuk ook!

Het standaard master-detailscherm op basis van dit gegevensmodel toont bovenin afdelingen en onderin de op die afdeling werkzame medewerkers. De essentie van dit klassieke voorbeeld is de vertaling van een regel ('Medewerkers zijn werkzaam op een afdeling', of, in relationele termen, 'Het attribuut Afd# in de tabel Medewerker moet verwijzen naar een bestaande Afd# in de tabel Afdeling') in gedrag: niet alleen worden per afdeling netjes de aldaar werkzame medewerkers getoond, tevens is het onmogelijk om een medewerker toe te wijzen aan een niet bestaande afdeling. Immers, de semantiek van het scherm luidt dat de medewerkers

Het dbms voorbij (12 en slot)

Ter afsluiting van 'Het DBMS voorbij' voor de variatie eens geen uitgebreide verhandelingen over ingewikkelde, verstrekkende en soms politiek beladen concepten als integriteitmanagement, autorisatie of de symbiose tussen objectoriëntatie en relationele technologie, maar behandelen we twee onderwerpen die tot dusverre tussen wal en schip zijn gevallen, en toch een plekje verdienen in deze artikelserie over grensverleggend gebruik van de technologie van het (r)dbms.

die in het onderste deel van het scherm worden getoond, werkzaam zijn op de in het bovenste deel van het scherm getoonde afdeling (bij een single-record layout) of de in het bovenste deel van het scherm 'current' afdeling (bij een multi-record layout). Technisch wordt het een en ander afgedwongen door een aantal functionaliteiten:

- bij het wijzigen van de 'current' Afdeling, als gevolg van het opnieuw opvragen van gegevens, of het bladeren door het resultaat van een query, wordt in het Medewerker-deel automatisch een query uitgevoerd met restrictie 'Afd# = <Afd# van de 'current' Afdeling>';
- het attribuut Afd# in Medewerker wordt niet getoond of is niet wijzigbaar;
- bij toevoegen van een Medewerker wordt in het attribuut Afd# automatisch de Afd# van de 'current' Afdeling ingevuld.

Het master-detailscherm kent vele gedaanten, met tabbladen, detailschermen of zelfs boomstructuren, maar is functioneel telkens tot hetzelfde basisprincipe te herleiden. Voor puur registratieve applicaties is de master-detailstructuur zelfs vaak de enige

De consistentie kan geheel door het systeem worden afgedwongen

'functionaliteit' die de applicatie biedt boven het absolute minimum van 1 scherm per tabel. In dergelijke situaties komt het dan ook niet zelden voor dat het aantal master-detailschermen een veelvoud bedraagt van het aantal tabellen in het systeem, eenvoudigweg omdat gebruikers telkens nieuwe uitsnedes en verbanden uit de gegevensverzameling in een kant en klaar en gebruiksvriendelijk scherm willen zien.

Hoe anders ziet de gebruiksinterface eruit in veel analyse- en warehouse-omgevingen. Deze read-only applicaties zijn natuurlijk het puurste voorbeeld van een registratieve applicatie, maar de gebruikersinterface is een wereld van verschil. In plaats van een veelheid aan complexe, maar kant en klare, schermen krijgt de gebruiker een soort toolbox ter beschikking, waarmee hij naar believen relaties kan leggen tussen de tabellen die de bouwstenen van de applicatie vormen (voor zover de term 'applicatie' hier nog van toepassing kan worden verklaard). De gebruiker is als het ware zijn eigen RAD-ontwikkelaar geworden.

Het is bepaald niet uit te sluiten dat de mogelijkheden van dergelijke flexibele gebruikersinterfaces een deel van de verklaring vormen voor de populariteit van warehouses bij sommige eindgebruikers. Voor ICT-managers is er de bijbehorende nachtmerrie, dat sommige techneuten voor de realisatie van al dat moois ook meteen sterschema's onontbeerlijk achten. Het blijft hoe dan ook eeuwig zonde dat de flexibiliteit van warehouse userinterfaces nooit is doorgedrongen tot de wereld van klassieke administratieve applicaties, temeer daar de onderliggende techniek verbluffend eenvoudig is. Want sinds rdbms-producten het concept 'verwijzende sleutel' ondersteunen, is alle informatie om op basis van verwijzende sleutels master-detailfunctionaliteit te genereren beschikbaar voor elk applicatieprogramma of voor elke eindgebruiker.

Domeinen

Helemaal uitgebreid worden de mogelijkheden wanneer we niet alleen naar relationele sleutels maar ook naar relationele domeinen kijken. Er ontstaan dan mogelijkheden als:

- Medewerkers met dezelfde naam als de afdeling;
- Medewerkers met hetzelfde salaris als het budget van de afdeling.

Toegegeven, in het beschreven Employee-Department voorbeeld zijn deze vergelijkingen onzinnig. Het is hier echter eerder de vraag of het gegevensmodel goed is, dan de vraag of het feature *overkill* is. Indien gesteld wordt dat namen van afdelingen en namen van medewerkers verschillende dingen zijn (en dat klinkt redelijk), dan dienen de attributen Afdeling.Naam en Medewerker.Naam feitelijk te worden gespecificeerd met verschillende domeinen. Het feit dat beide domeinen in technische zin wellicht worden gespecificeerd als 'vrije tekst met een maximum van 50 posities' doet daar niets aan af! We lopen hier tegen het probleem aan dat, anders dan het geval is bij verwijzende sleutels, relationele dbms'en nog steeds geen voorzieningen kennen voor het ondersteunen van domeinen. Zoals zo vaak in deze serie is betoogd, levert dat echter geen serieus beletsel op: definieer zelf het domeinconcept in uw applicatieomgeving en schrijf uw generieke software. Het bedrijf waarvoor ondergetekende werkt doet dat al jaren op basis van het dictionary model dat in het kader van de illustere Tien Geboden-serie midden jaren negentig in Database Magazine is beschreven door René Veldwijk.

EEN VOORBEELD

We beschouwen het Employee-Department voorbeeld eens vanuit het perspectief van een flexibele gebruikersinterface. De gebruiker opent twee schermen met daarin de gegevens over Afdelingen respectievelijk Medewerkers. Dit kan door de tabellen te selecteren in een lijst met in het systeem aanwezige tabellen. Vervolgens geeft de gebruiker aan dat hij deze schermen aan elkaar wil koppelen. Nu gaat de intelligentie van het systeem aan het werk. Het systeem moet uitvinden of er een zinvolle relatie te leggen is tussen Afdeling en Medewerker, en zo ja, welke dit dan zijn. Het systeem maakt hiervoor gebruik van de informatie over de sleutels die op de tabellen zijn gedefinieerd en komt tot de volgende mogelijkheden (we gaan ervan uit dat we Medewerker als 'detail' onder Afdeling willen hangen):

- Medewerkers van een Afdeling (Medewerker.Afd# = Afdeling.Afd#), de klassiek 1:n synchronisatie gebaseerd op een verwijzende sleutel;

- Manager van een Afdeling (Medewerker.Med# = Afdeling.Med#_Manager), een 1:1 synchronisatie gebaseerd op een 'omgekeerde' verwijzende sleutel;
- Medewerkers van een hogere afdeling (Medewerker.Afd# = Afdeling.Afd#_Hoger), een minder voor de hand liggende 1:n synchronisatie die in feite gebaseerd is op de verwijzende sleutels Medewerker.Afd# = Afdeling2.Afd# en Afdeling2.Afd# = Afdeling.Afd#_Hoger, waarbij Afdeling2 een tweede voorkomen is van de tabel Afdeling, vergelijkbaar met een SQL FROM clausule 'FROM Medewerker, Afdeling a1, Afdeling a2'.

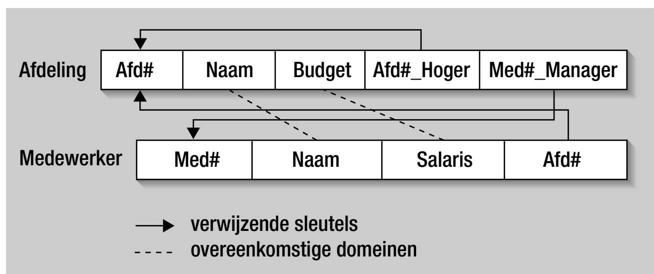
De kracht van een flexibele gebruikersinterface is feitelijk niets anders dan het slim gebruik maken van elementaire kennis over het gegevensmodel van de applicatie. Elementaire kennis, die beschikbaar is in de catalogus van de relationele database en steeds vaker ook in de repository's van applicatie-ontwikkelhulpmiddelen, maar vaak onvoldoende wordt benut.

Want laten we ons flexibele voorbeeld nog een stapje verder trekken; de gebruiker heeft *on the fly* een prachtig samenstel van onderling gesynchroniseerde schermen in elkaar geklikt, een simpele druk op de knop en het geheel is direct als samengesteld scherm kant en klaar beschikbaar. En dan is de gebruiker echt ontwikkelaar geworden, en vindt het nog leuk ook!

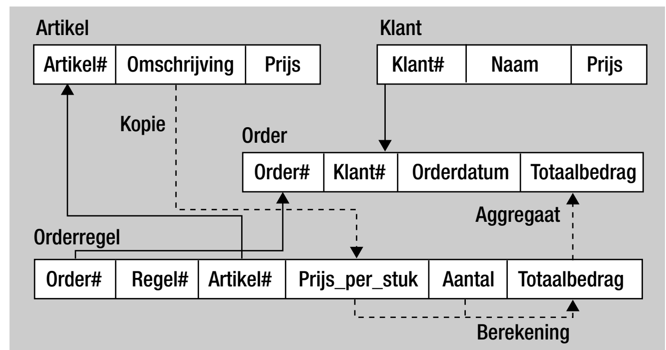
MANAGED REDUNDANCY: DE SLEUTEL VOORBIJ

Na het Elysische perspectief van de flexibele gebruikersinterface, tot slot een zachte landing in de modder van de 'echte' database. Een van de oudste technieken op het gebied van gegevensmodellering is het normaliseren. Het principe achter normaliseren is, zoals de lezer ongetwijfeld weet, het eenmalig vastleggen van elk gegeven in de database, of omgekeerd: het uitbannen van redundantie.

Met enige regelmaat, recentelijk nog in Computable, verschijnen artikelen waarin de bijl in de wortels van dit geloof wordt gezet en wordt gepredikt dat redundantie 'goed' is (en passant wordt met een volstrekt onzuivere argumentatie het relationele model vaak tegelijk naar de prullenbak verwezen). De argumentatie komt er in het kort op neer dat redundantie in de echte wereld ook vaak voorkomt en dat het aanbeveling verdient om het gegevensmodel zo natuurlijk mogelijk bij de werkelijkheid te laten aansluiten. Soms stopt het verhaal daar en wordt niet nader op de



Afbeelding 1: Het klassieke Employee-Department gegevensmodel.



Afbeelding 2: Het klassieke model van order en orderregels.

implicaties in gegaan. Doet men dat wel, dan betreft van de rest van het betoog zo'n 90% de voordelen van redundantie voor het raadplegen van gegevens, en wordt in het beste geval voor 10% aandacht besteed aan de problemen van het consistent houden van die redundante gegevens bij mutaties op de database, zonder dat daarover meer wordt gezegd dan 'het valt in de praktijk wel mee'.

Laten we eens een praktijkcasus nemen, wederom een echte modelleer-klassieker: het model van orders en orderregels. Het model ziet er als volgt uit en moge voor zichzelf spreken:

```
Klant(Klant#, Naam, Adres)

Artikel(Artikel#, Omschrijving, Prijs)

Order(Order#, Klant#, Orderdatum, Totaalbedrag)

Orderregel(Order#, Regel#, Artikel#, Aantal,
Totaalbedrag)
```

We zien natuurlijk direct dat er twee attributen in het gegevensmodel afgeleide gegevens bevatten: Order.Totaalbedrag (zijnde de som van de totaalbedragen van de orderregels) en Orderregel.Totaalbedrag (zijnde het product van Aantal en de Prijs van het bijbehorende Artikel). Als we ervoor kiezen om deze afgeleide gegevens in de database vast te leggen (bijvoorbeeld om redenen van performance), dan rijst de vraag hoe de consistentie tussen de afgeleide gegevens en de brongegevens kan worden gewaarborgd.

Een gangbare oplossing is het gebruik maken van *database triggers*. Feitelijk betekent dit dat de acties die benodigd zijn om de gegevens consistent te houden, procedureel worden gespecificeerd. In ons voorbeeld kan dit met twee triggers:

```
CREATE TRIGGER Upd_Regeltotaal
AFTER INSERT OR UPDATE OF Orderregel
BEGIN
   SELECT :new.Aantal * Artikel.Prijs
   INTO :new.Totaalbedrag
   FROM Artikel
   WHERE Artikel.Artikel# = :new.Artikel#;
END;
```

```
CREATE TRIGGER Upd_Ordertotaal
AFTER INSERT OR UPDATE OR DELETE OF Orderregel
BEGIN
   UPDATE Order
   SET Totaalbedrag = (SELECT SUM(Totaalbedrag)
                      FROM Orderregel
                      WHERE Orderregel.Order# =
                      Order.Order#
                      )
   WHERE Order# = :new.Order#;
END;
```

Er is echter ook een zienswijze waarin de consistentie grotendeels declaratief wordt bewaakt. De kern van deze zienswijze is het

Of er indexen bestaan of niet, is vanuit applicatieperspectief verder irrelevant

beschouwen van het relationele concept 'verwijzende sleutel' als een bijzonder geval van het concept 'verband', waarbij 'verband' wordt gedefinieerd als het voorkomen van een bepaalde vorm van consistentie tussen verzamelingen attributen in twee verschillende tabellen.

- De waarden van attribuut A1, ..., An van tabel A verwijzen naar een voorkomen van attributen B1, ..., Bn in tabel B. Dit is de klassieke verwijzende sleutelrelatie. Voorbeeld: de waarde in attribuut Artikel# in de tabel Orderregel verwijst naar een voorkomen van attribuut Artikel# in de tabel Artikel;
- De waarde van attribuut A1 van tabel A is een kopie van de waarde van attribuut B1 in tabel B, waarbij de waarden in attribuut A2, ..., An de verwijzing naar een voorkomen van attribuut B2, ..., Bn in tabel B bepalen. Als voorbeeld kunnen we aan de tabel Orderregel een attribuut Prijs_per_stuk toevoegen, dat een kopie is van Artikel.Prijs volgens de relatie Orderregel.Artikel# = Artikel.Artikel#. Anders gezegd: de waarde van Artikel.Prijs wordt 'naar beneden gekopieerd' via de verwijzende sleutel van Orderregel naar Artikel.[RVE1];
- De waarde van attribuut A1 van tabel A is een aggregaat van de gegevens in attribuut B1 in tabel B, waarbij de waarden in kolom A2, ..., An de verwijzing naar een voorkomen van attributen B2, ..., Bn in tabel B bepalen. Een voorbeeld is het attribuut Order.Ordertotaal, dat een sommatie is van Orderregel.Ordertotaal via de relatie Order.Order# = Orderregel.Order. Anders gezegd: de waarde van Orderregel.Totaalbedrag wordt 'omhoog geaggregeerd' via de verwijzende sleutel van Orderregel naar Order.

Geen zuivere relatie (want betrekking hebbende op attributen uit 1 tabel), en ook niet geheel declaratief is de volgende regel: De waarde in attribuut A1 van tabel A is de uitkomst van een eenvoudige berekening op basis van de waarden in attributen A2, ..., An van tabel A. Een voorbeeld is het attribuut Orderregel.Totaalbedrag dat een vermenigvuldiging is van Orderregel.Aantal en Orderregel.Prijs_per_stuk (dat op zijn beurt weer een kopie is van Artikel.Prijs). Bedenk overigens wel dat we hier het procedurele vlak betreden: de specificatie van de berekening is een zuiver voorbeeld van schending van de 0-de normaalvorm.

Het voordeel van de declaratieve benadering is dat de consistentie geheel door het systeem kan worden afgedwongen, iets wat met triggers of andere procedurele logica niet mogelijk is. Dit is vergelijkbaar met het verschil tussen het afdwingen van verwijzende sleutels via relationele *foreign key constraints* of met database triggers. Beide benaderingen waarborgen (mits de trigger goed is gecodeerd) dat toekomstige mutaties adequaat worden gecontroleerd. De trigger controleert echter niet of de database zich reeds in een situatie van schending van de verwijzende sleutel bevindt. Evenmin is het mogelijk om in geval van twijfel aan de correctheid van de trigger de constraint te controleren voor alle voorkomens in de database, anders dan het handmatig coderen van een SQL SELECT statement.

In ons voorbeeld van 'managed redundancy' biedt de declaratieve oplossing daarnaast het voordeel dat het vanuit applicatieperspectief irrelevant is, of de afgeleide gegevens daadwerkelijk in de database worden opgeslagen of automatisch worden berekend bij het ophalen van records uit de database. Vergelijk het met indexen op een relationele database: het al dan niet leggen van indexen kan grote invloed hebben op de performance van de applicatie, maar het feit of er indexen bestaan of niet is vanuit applicatieperspectief verder irrelevant.

TENSLOTTE

Met dit artikel zijn we aan het einde gekomen van 'Het dbms voorbij'. Twaalf artikelen lang hebben we de grenzen verkend van wat een modern dbms minimaal zou moeten zijn en wat het geweest zou kunnen zijn.

Hoewel de technologie natuurlijk niet stilstaat zou het naïef zijn om te veronderstellen dat de zaken die we hebben besproken voor een significant deel tot een commodity zullen zijn geworden in het dbms van 2010. Bedenk echter dat alles wat is besproken herleidbaar is tot eenvoudige, relationele principes. Niemand vraagt van de lezer om een eigen dbms te maken, maar het verrijken van uw huidige lievelings-dbms (vul maar in) met de besproken ideeën is even uitvoerbaar als profijtelijk.

Speciale dank is verschuldigd aan mijn collega Marc Dierick, aan wiens geest de hier beschreven ideeën over 'managed redundancy' zijn ontsproten. ●

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