Autorisatieprincipes in het gegevensdomein

De gebruiker als 'enemy at the gates'

Frido van Orden

Een basisgegeven in elk informatiesysteem is autorisatiebeheer. Bepaalde gebruikers mogen geen handelingen verrichten die anderen wel mogen doen. Deze omschrijving bevat een paradox: moet een systeem taken afschermen voor onbevoegden, dan moet het over regels beschikken op grond waarvan besloten kan worden of een gebruiker al dan niet bevoegd is.

Wellicht is autorisatiemanagement niets anders dan het afdwingen van specifieke regels, en daarmee een onderdeel van constraint management. Het in voorgaande artikelen ontwikkelde gegevensmodel zou zijn kracht hier moeten kunnen bewijzen: een hele opluchting voor de lezers die na de constraint-trilogie vreesden een dergelijke verhandeling over autorisatie niet meer te kunnen volgen. Het klinkt haast te mooi om waar te zijn, en helaas is het dat ook.

OVEREENKOMSTEN EN VERSCHILLEN

Teruglezend in het eerste constraint-artikel (DB/M 3/2001) bekijken we nog eens de exacte definitie van de constraint management functionaliteit van een informatiesysteem:

het afdwingen van alle, maar dan ook werkelijk alle regels waaraan gegevens in een informatiesysteem moeten voldoen.

Het dbms voorbij (5)

In de afgelopen drie artikelen uit deze serie *Het dbms voorbij* heeft René Veldwijk aandacht besteed aan een van de belangrijkste taken van een informatiesysteem: het afdwingen van alle regels waaraan gegevens in een informatiesysteem moeten voldoen. Na de apotheose bereikt te hebben in de vorm van een gegevensmodel voor constraint management is de tijd nu rijp om de pijlen te richten op nog zo'n basaal stuk functionaliteit in elk informatiesysteem: autorisatiebeheer. Frido van Orden formuleert hiervoor een aantal principes en werkt die uit, voortbouwend op het gelegde constraint management-fundament.

De kneep zit 'm natuurlijk in de zinsnede 'waaraan gegevens moeten voldoen'. De regel 'medewerker sexe moet de waarde M (man) of V (vrouw) bevatten' valt duidelijk binnen deze definitie. Maar hoe zit dat met een regel als 'het salaris van een medewerker mag alleen worden geraadpleegd door medewerkers van de afdeling P&O' of 'het budget van een afdeling mag alleen worden gemuteerd door de manager van die afdeling'?

Laten we de overeenkomsten en verschillen tussen diverse autorisatieregels eens naast elkaar leggen. We maken weer gebruik van het eenvoudige afdeling medewerker voorbeeld dat we in deze serie als leidraad hanteren:

```
Medewerker(Med#, Naam, Sexe, Salaris, Afd#)
Afdeling(Afd#, Naam, Med#_Mgr, Budget, Afd#_Hoger)
```

Een van de opvallendste zaken -met het modelleren van validatieregels uit de vorige artikelen nog in het achterhoofd- is dat de regel 'sexe moet de waarde M (man) of V (vrouw) bevatten' (verder: V1) altijd te controleren is tegen een willekeurige status van de database door middel van een SQL-statement in de trant van:

```
SELECT DISTINCT 'Error'
FROM Medewerker
WHERE Sexe NOT IN ('M', 'V')
```

Voor de beide autorisatieregels 'het salaris van een medewerker mag alleen worden geraadpleegd door medewerkers van de afde-

ling P&O' (verder: A1) of 'het budget van een afdeling mag alleen worden gemuteerd door de manager van die afdeling' (verder: A2) geldt dit niet. Dit heeft een aantal oorzaken:

- voor het valideren van A2 hoeven we niet zozeer te weten hoe het budget is gewijzigd, maar wel dát het is gewijzigd. Deze informatie is niet in de database aanwezig, omdat het oude budget met het nieuwe wordt overschreven;
- voor het valideren van A1 hebben we helemaal een probleem, omdat selectie-operaties de database-inhoud ongewijzigd laten;
- de informatie over de gebruiker die de actie uitvoert ontbreekt.

De doorgewinterde DB/M-lezer zal bij dit lijstje direct associaties krijgen met het begrip 'audit trail'. Een audit trail is in feite niets anders dan een tabel -idealiter in de database zelf- waarin van elke raadpleeg- en mutatieactie wordt vastgelegd hoe en door wie deze is uitgevoerd. Een dergelijke tabel ziet er bijvoorbeeld uit als figuur 1.

Te zien is dat gebruiker 'Orden' op 14 januari 2002 een raadpleegactie op de kolommen 'Naam' en 'Sexe' van de tabel Medewerker heeft uitgevoerd; dat het een enkele in plaats van twee losstaande raadpleegacties betreft, is af te leiden uit het attribuut 'ID' dat voor elke actie (lees: SQL-statement) wordt opgehoogd. In de tabel is ook te zien dat op dezelfde dag gebruiker 'Storm' het budget van een afdeling heeft verhoogd van € 10.000,- naar € 15.000,-.

Met de audit trail-tabel in de hand komen we een stap verder in ons streven naar de mogelijkheid autorisatieregels te vangen in ons model voor constraint-validatie.

Laten we echter eerst eens kijken wat we nu precies hebben gedaan en waarom deze oplossing ons verder helpt. De audit trail-tabel is een goed voorbeeld van een tabel met procesgegevens: niet relevant voor het eigenlijke informatiesysteem -en bovendien voor een groot deel redundant, zoals u natuurlijk direct al zag-, maar louter bedoeld om een bepaald sturend of controlerend proces van informatie te voorzien; in dit geval het proces van preventief of correctief autorisatiemanagement. Dergelijke procestabellen kunnen overigens ook heel goed op een wat hoger abstractieniveau in het gegevensmodel voorkomen. Denk bijvoorbeeld aan een incassobureau dat bijhoudt welke brieven en telefoontjes zijn verstuurd en ontvangen om een wanbetaler tot betaling te dwingen en op grond van deze informatie bijvoorbeeld kan besluiten een deurwaarder in te schakelen of het dossier juist als oninbaar af te sluiten.

We zien ook dat de audit trail-tabel zowel handelt over gegevenselementen uit het 'echte' informatiesysteem (de zogenaamde instance-gegevens) als over gegevenselementen uit de beschrijving van dat informatiesysteem (de zogenaamde metagegevens). Zo vinden we in de tabel instance-gegevens, als 'timestamp' en 'oude waarde', maar ook metagegevens, zoals 'gebruiker', 'tabel' en 'kolom'.

Vrijwel alle SQL-databases hebben 'connectie' en 'authenticatie' samengevoegd, wat het autorisatiebeheer frustreert

Het is van belang te beseffen hoe het audit trail-mechanisme, het stukje software in het dbms dat de audit trail-tabel vult, aan zijn informatie komt. Dit mechanisme analyseert voor elk afgevuurd SQL-statement wat het precies doet: insert, update, delete, select; welke kolommen? En het schrijft deze informatie weg, samen met die over de huidige gebruiker. Zo wordt informatie verzameld uit verschillende bronnen:

- het SQL-statement, geschreven door de programmeur of gegenereerd door de frontend, dat de structuur van de actie heeft vastgelegd (bijvoorbeeld: het budget van een afdeling wordt gewijzigd);
- de aan het SQL-statement meegegeven waarden (het nieuwe budget, de afdelingsidentificatie), opgegeven door de gebruiker;
- de gebruikersidentificatie, aangeleverd door het dbms, dat bijhoudt wie de aangelogde gebruiker is. De gebruikersidentificatie is door de gebruiker meegegeven bij het aanloggen aan het systeem, terwijl het feit dat de gebruiker bestaat is opgegeven door de applicatiebeheerder en/of dba.

Vrijwel al deze informatie is vluchtig en nergens aanwezig in de database, zowel voor als na uitvoering van de actie. In feite doet de audit trail niets anders dan het beschikbaar stellen van deze informatie in een formaat waarmee iedere database en applicatie iets kan: de relationele tabel.

Terug naar de oorspronkelijke vraag: kunnen we de autorisatieregels A1 en A2 valideren door middel van een SQL-statement?

```
AUDIT_TRAIL(
    ID              INTEGER NOT NULL,
    TIMESTAMP       DATETIME NOT NULL,
    USER            VARCHAR(30) NOT NULL,
    TABEL           VARCHAR(30) NOT NULL,
    DML             CHAR(1) NOT NULL,
    KOLOM           VARCHAR(30),
    OLDVAL          VARCHAR(100000),
    NEWVAL          VARCHAR(100000),
    PRIMARY KEY(ID, KOLOM)
)
```

ID	Timestamp	User	Tabel	DML	Kolom	OldVal	NewVal
1	14-01-2002	Orden	Medewerker	S	Naam		
1	14-01-2002	Orden	Medewerker	S	Sexe		
2	14-01-2002	Storm	Afdeling	U	Budget	10000	15000

FIGUUR 1: EEN AUDIT TRAIL.

Welnu, we komen dicht in de buurt. De enige informatie die we nog missen is de relatie tussen de gebruikers die database-acties uitvoeren en de gebruikersgroepen die we gebruiken in de terminologie van onze autorisatieregels. Gelukkig is deze informatie veelal beschikbaar in de catalogus van onze relationele database. Zonder verder op de details in te gaan, poneren we hier het bestaan van de volgende catalogustabellen:

```
SYSUSERS(
        USER_NAME VARCHAR(30),
        ...
PRIMARY KEY(USER_NAME)
)

SYSROLES(
        ROLE_NAME VARCHAR(30),
        ...
        PRIMARY KEY(ROLE_NAME)
)

SYSUSERROLES(
        USER_NAME VARCHAR(30),
        ROLE_NAME VARCHAR(30),
...
PRIMARY KEY(USER_NAME, ROLE_NAME)
)
```

De aan de database bekende gebruikers zijn vastgelegd in de tabel SYSUSERS. Gebruikersgroepen zijn opgenomen in de tabel SYSROLES, terwijl in de tabel SYSUSERROLES te vinden is tot welke groepen gebruikers behoren[1]. Met deze informatie in handen kunnen we nu de validatiequery schrijven voor de regel A1:

```
SELECT 'Error'
FROM AUDIT_TRAIL AT
WHERE AT.TABEL='Medewerker'
  AND AT.KOLOM='Salaris'
  AND AT.DML = 'S'
AND NOT EXISTS(
SELECT 'x'
FROM SYSUSERROLES UR
WHERE AT.USER = UR.USER_NAME
AND UR.ROLE_NAME = 'P&O')
```

Met regel A2 komen we echter opnieuw in de problemen: de relatie tussen 'manager van de afdeling' en 'gebruiker' ontbreekt. We zullen de tabel medewerker daarom uitbreiden met het volgende attribuut:

```
        USER_NAME VARCHAR(30)
```

Tevens missen we in de audit trail-tabel nog een identificatie van het record waarop de actie betrekking heeft. De audit trails van de grote dbms'en bieden op dit gebied, als met meezit, iets als een rowid of een SQL where clause. Maar laten wij het netjes doen en een extra tabel toevoegen:

```
AUDIT_TRAIL_KEY_COL(
        TRAIL_ID       INTEGER NOT NULL,
        TABELNAAM      VARCHAR(30) NOT NULL,
        KOLOMNAAM      VARCHAR(30) NOT NULL,
        WAARDE VARCHAR(100000) NOT NULL,
PRIMARY KEY (TRAIL_ID)
)
```

In deze tabel komen voor elke audit trail id de waarde van de primary key-attributen van het relevante record te staan, bijvoorbeeld:

TRAIL_ID	TABELNAAM	KOLOMNAAM	WAARDE
2	Afdeling	Afd#	D/C

Nu zijn we waar we wezen willen en kunnen we regel A2 als volgt controleren:

```
SELECT DISTINCT 'Error'
FROM AUDIT_TRAIL AT, AUDIT_TRAIL_KEY_COL AC
WHERE AT.TABEL='Afdeling'
  AND AT.KOLOM='Budget'
  AND AT.DML = 'U'
  AND AT.ID = AC.TRAIL_ID
AND NOT EXISTS(
SELECT 'x'
FROM Medewerker M
WHERE M.USER_NAME = AT.USER
AND M.Afd# = AC.WAARDE)
```

ANALYSE

Ongemerkt zijn we al diep op een tweetal concrete voorbeelden van autorisatieregels ingegaan zonder dat we netjes hebben onderkend wat we nu precies onder autorisatie verstaan. Om dat te kunnen doen, dienen we eerst vast te stellen wat we bedoelen met de term 'autorisatie' en welke zaken daarbij een rol spelen. Ook leggen we weer de link naar het gedrag van de applicatie, zoals we dat ook bij validatie hebben gedaan. Autorisatie kunnen we definiëren als: *het geven of weigeren van rechten aan actoren op het uitvoeren van een bepaalde actie op een object.*

Deze algemene definitie kunnen we in termen van informatiesystemen op diverse wijzen invoeren, zoals weergeven in Tabel 1. In de eerste regel uit de tabel herkent u ongetwijfeld autorisatieregel A1. Deze vorm van autorisatie wordt aangeduid met de term verticale gebruikers-gegevensautorisatie.

Regel 2 uit de tabel is een specifieke uitwerking van onze voorbeeldregel A2. Deze vorm van autorisatie wordt aangeduid met de term *horizontale gebruikersgegevensautorisatie* en is een verfijning van de standaard *verticale autorisatie*. Regel 3 zullen de meesten wel herkennen als de aloude CRUD-matrix. We duiden hem aan met de term *verticale functiegegevensautorisatie*. Regel 4 is de horizontale

Dynamische constraints

De analyse van autorisatieregels brengt ons op een klasse validatieregels die in de voorgaande artikelen onbesproken is gebleven, maar die de lezer wellicht nog kent uit de Tien geboden-serie, namelijk die van de dynamische constraints.
Het standaard voorbeeld in de literatuur van een dynamische constraint is 'Het salaris van een medewerker kan nooit worden verlaagd' - over de semantische correctheid van deze regel in de huidige economische tijd doen we geen uitspraak. Bij het valideren van deze regel lopen we tegen hetzelfde probleem aan als bij de autorisatieregels, namelijk dat de regel informatie vereist over de waarde van een gegeven zowel voor als na een mutatie. Mogelijk is die informatie niet aanwezig in een separaat bijgehouden historietabel; een audit trail-tabel kunnen we voor deze discussie ook in deze categorie plaatsen. In dat geval is deze regel niet controleerbaar tegen een statische toestand van de database.

In de dagelijkse praktijk doet zich bij dynamische constraints gelukkigerwijs het fenomeen voor dat de oplossing er meestal eerder is dan het probleem. Nemen we de salarisverlagingsregel weer als voorbeeld. We kunnen ons afvragen waarom iemand een dergelijke regel bedenkt. Aangezien het merkwaardig is een regel te bedenken over iets waarin niemand geïnteresseerd is, kunnen we concluderen dat er tenminste behoefte bestaat aan inzicht in het salarisverloop van medewerkers. Wellicht zelfs is er programmatuur die niet goed functioneert wanneer salarissen worden verlaagd; een imperfect salariëringssysteem bijvoorbeeld. In beide gevallen geldt dat er behoefte is aan vastlegging van de historie van het salaris. De historietabel die we nodig hebben voor onze validatie is dus vrijwel zeker reeds in het systeem aanwezig! Uitgaande van de volgende tabel:

```
Medewerker_SalHist(Med#, DatumIn, DatumEind,
            Salaris)
```

kunnen we onze dynamische constraint dan ook zonder problemen vertalen in een equivalente statische constraint:

```
      SELECT 'Error'
      FROM Medewerker m,
     Medewerker_SalHist sh1,
Medewerker_SalHist sh2
      WHERE m.Med# = sh1.Med#
        AND m.Med# = sh2.Med#
  // Aansluitende tijdsperioden
        AND sh1.DatumEind = sh2.DatumIn
  AND sh2.Salaris < sh1.Salaris
```

Dat een dynamische constraint als zodanig voorkomt -en dus niet in de vertaalde statische vorm- is een indicatie voor een imperfect gegevensmodel. Dat kan bij voortschrijdende ontwikkeling en aanpassing van informatiesystemen heel valide zijn, maar dient bij nieuwe ontwikkeling -waarop de generieke mechanismen in deze artikelserie zich toch richten- te worden vermeden.

variant daarop. Regel 5 ten slotte is de verreweg meest gebruikte -en in vele oudere, maar ook nieuwe applicaties enige- vorm van autorisatie: het toegang geven aan gebruikers tot bepaalde schermen, lijsten etcetera. De term is *gebruikersfunctie-autorisatie*.

VERTICALE GEBRUIKERSGEGEVENS-AUTORISATIE

Verticale gebruikersgegevensautorisatie is de meest bekende vorm van autorisatie. Aan gebruikers (of groepen van gebruikers) wordt het recht toegekend gegevens aan tabellen toe te voegen, gegevens uit een tabel te verwijderen, bepaalde kolommen van een tabel te raadplegen of bepaalde kolommen te muteren. In SQL komen we deze vorm van autorisatie tegen in het `GRANT`-statement:

```
GRANT INSERT ON Medewerker TO Orden
GRANT SELECT(Med#, Naam) ON Medewerker TO Storm
```

De meeste dbms-producten ondersteunen deze vorm van autorisatie in het dbms zelf, al bestaat de mogelijkheid om bij update en select specifieke kolommen te autoriseren meestal pas sinds kort.

Een uitermate vervelend verschijnsel treedt echter op indien het dbms wordt gebruikt in combinatie met applicaties die voor de toegang tot het dbms gebruik maken van een mechanisme van *connection pooling*. Een dergelijk mechanisme maakt gebruik van een 'pool' van een beperkt aantal fysieke connecties naar de database. Indien een gebruiker een databaseconnectie wenst, opent hij een logische connectie naar de connection pool, die hem vervolgens tijdelijk een connectie uit de pool ter beschikking stelt. Hiermee wordt zowel voorkomen dat moet worden gewacht op het leggen van een fysieke connectie naar de database -een relatief trage operatie- als dat applicaties nodeloos connecties 'bezet' houden en daarmee de databaseserver overbelasten. Het probleem ontstaat doordat vrijwel alle SQL-databases het concept 'connectie' (het leggen van een verbinding naar de database) en 'authenticatie' (het bekend maken van de identiteit van de gebruiker) hebben samengevoegd. Dit leidt ertoe dat de connecties uit een connection pool zich aan de database aanmelden onder een account met universele rechten. En de database kan niet meer achterhalen wie de 'echte' gebruiker is. Het hoeft geen betoog dat alle vormen van autorisatiemanagement in het dbms hiermee in het water vallen. Iets om eens met uw dbms-leverancier over te praten bij de volgende ronde van licentie-onderhandelingen!

Het gewenste gedrag van applicaties om aan verticale autorisatie tegemoet te komen wijkt af van wat we eerder zagen bij constraint-validatie. Is bij constraints in principe sprake van een 'piepsysteem', aangevuld met dwingende constructies als drop down listboxen, bij autorisatie is het gedrag gebaseerd op het simpelweg niet toestaan van acties. Dit is weergegeven in tabel 2.

	Actor	Actie	Object
1	Medewerkers P&O	Raadplegen	Salaris van een medewerker
2	Gebruiker 'Storm'	Muteren	Budget afdeling 'Debiteuren/crediteuren'
3	Functie 'Raadplegen Afdelingen'	Raadplegen	Attributen van afdelingen
4	Functie 'Muteren medewerkers'	Muteren	Medewerkers behalve de directie
5	Gebruiker 'Orden'	Uitvoeren	Functie 'Raadplegen Afdelingen'

Tabel 1: Vormen van autorisatie.

De meeste moderne frontend-tools kunnen met insert, delete en update wel overweg. Bij selecties ligt dat al een stuk lastiger: het herhaaldelijk bouwen van hetzelfde scherm voor verschillende autorisatieprofielen is eerder regel dan uitzondering. Toch is het vanuit een technisch oogpunt zeker in OO omgevingen niet al te lastig om velden die toch niet geraadpleegd mogen worden, simpelweg weg te halen van het scherm. Dit is bij een lijst layouts (multirecord) ook het fraaist, bij formulier-layouts (single record) is het uitschakelen van het veld -dat is meer dan alleen 'disabled' aanzetten!- vanuit een visueel oogpunt veelal fraaier.

Het lijkt alsof bij de ontwikkelingen van vandaag de lessen uit het verleden vrolijk opnieuw worden geleerd

In omgevingen die QBE (query by example) ondersteunen is het veelal gebruikelijk ook te kunnen specificeren op welke velden een query-restrictie mag worden geplaatst. Zo kan bijvoorbeeld het zoeken op medewerkernummer worden toegestaan, maar op salaris niet. Nog mooier is natuurlijk dat bovendien onderscheid gemaakt kan worden tussen exacte (Med# = 12345, Naam = 'Jansen') en inexacte vergelijkingen (Salaris < 100.000, Naam LIKE '%s%').

In feite introduceren we hier een vijfde actie naast de vier bekende. Ook op het niveau van het dbms zelf zou dit een uiterst handige toevoeging aan de reeds aanwezige autorisatiemechanismen kunnen zijn. Voorkomen moet worden dat gebruikers die met SQL of een query-tool de database benaderen deze met ongebreidelde query's kunnen overbelasten. Iets wat overigens niet zelden de reden is om tot inrichting van een datawarehouse over te gaan.

HORIZONTALE GEBRUIKERSGEGEVENS-AUTORISATIE

Bij de horizontale autorisatievormen betreffen de regels niet alleen de structuur, maar ook de inhoud van relevante tupels. Hierbij wordt veelal wordt gebruik gemaakt van constructies waarin de gebruiker gekoppeld wordt aan een tabel uit het gegevensmodel, zoals in de voorbeeldregel A2, waarin 'de eigen afdeling' een koppeling legt tussen de gebruiker en de tabel Afdeling. Natuurlijk kunnen de regels arbitrair complex worden, zoals 'Managers mogen alleen het budget van hun eigen afdeling wijzigen, tenzij op die afdeling medewerkers werken die voor een uurtarief van meer dan € 150,- zijn ingezet voor een klant van accountmanager Fritsen'. De problematiek van validatie van user defined constraints komt hier weer in zijn volle omvang terug. Dat is goed nieuws als we met generieke zaken bezig zijn, maar slecht nieuws in alle andere gevallen!

Terwijl applicatie-ontwikkelaars bij verticale gebruikersgegevensautorisatie nog in behoorlijke mate op ondersteuning door dbms en frontend kunnen rekenen, ziet het landschap er compleet anders uit indien de zaak iets geavanceerder wordt door het toevoegen van horizontale restricties. Van de grote dbms-leveranciers heeft alleen Oracle zich ingespannen om op dit gebied iets te betekenen. Oracle noemt haar implementatie 'Fine grained object access' en heeft het verstopt onder de PL/SQL package DBMS_RLS. Het idee is even simpel als elegant: een zelfgeschreven functie geeft een SQL where clause terug, die het dbms vervolgens automatisch toevoegt bij het uitvoeren van een willekeurig SQL-statement. Figuur 2 toont een voorbeeld. Het effect hiervan is dat het statement:

```
UPDATE Afdeling SET Budget=15000 WHERE Afd#='D/C'
```

wordt uitgevoerd als:

```
UPDATE Afdeling SET Budget=15000 WHERE Afd#='D/C'
AND Afd# IN (SELECT Afd# FROM Medewerker WHERE
USER_NAME = USER)
```

Merk op dat bij schending van de autorisatieregel geen foutmelding volgt maar alleen het resultaat '0 records updated'! Lezers die nog met het oude Ingres hebben gewerkt zullen direct de overeenkomsten zien met het constraint-afhandelingsmechanisme dat dit oer-rdbms-mechanisme vijftien jaar geleden reeds bezat en dat dezelfde 'genezen is beter dan voorkomen'-interpretatie gaf aan het afdwingen van databaseregels. Nog zo'n groot nadeel is dat het niet mogelijk is op kolomniveau horizontale select- of update-autorisaties toe te voegen, een euvel waaraan veel vroege implementaties van verticale database-autorisatie ook leden. Het lijkt alsof bij de ontwikkelingen van vandaag de lessen uit het verleden vrolijk opnieuw worden geleerd!

Ten slotte is het jammer dat de autorisatie niet via een SQL-state-

Verboden actie	Gedrag
Insert	Uitschakelen functie
Delete	Uitschakelen functie
Update	Read-only maken veld
Select	Veld weglaten of uitschakelen

Tabel 2: Applicatiegedrag bij verticale autorisatie.

ment ALTER TABLE of GRANT kan worden gelegd. Al kunnen we daar ook direct bij zeggen dat in ook in de nieuwste SQL-standaard niets voorkomt dat ook maar enigszins in de buurt komt van horizontale autorisatievormen.

Aan de frontend-kant lijkt horizontale autorisatie in grote mate op een record-level user defined constraint: in de meeste gevallen is het pas bij het verlaten van het record mogelijk de regel te valideren, veelal gekoppeld aan het wegschrijven naar de database. Uitzonderingen zijn ook hier natuurlijk weer mogelijk. Zo is het denkbaar dat een eenvoudige regel als 'Afdelingen met een budget gelijk aan 0 mogen door niemand worden gewijzigd' voor een frontend te implementeren is door een form read only te maken als het huidig getoonde record een budget van 0 heeft. De grens van het haalbare ligt hier bij regels die slechts bestaan uit eenvoudige vergelijkingen binnen één record, eigenlijk net zoals dat voor user defined constraints geldt.

VERTICALE EN HORIZONTALE FUNCTIEGEGEVENSAUTORISATIE

Is het bij de hiervoor besproken gebruikersautorisatie nog de gebruiker die in toom moet worden gehouden, bij de functiegegevensautorisatie is de ontwikkelaar het doelwit. Van oudsher is de CRUD-matrix een mooi documentatiehulpmiddel, dat ook als handleiding voor de technisch ontwerper en programmeur dient. In de goede oude tijd waarin een systeemfunctie geïmplementeerd werd door een programma te schrijven dat vervolgens als een listing kon worden bekeken, konden zelfs technisch onderlegde testers nog wel kijken of de programmeur niet buiten zijn boekje was gegaan. Met de komst van 4GL-tools en n-laags objectgeoriënteerde class library's is dat voorgoed verleden tijd, zo niet in theorie

```
CREATE FUNCTION A2PolicyFunc(obj_schema
   VARCHAR2, obj_name VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
   IF obj_name = 'AFDELING' THEN
          – USER bevat de naam van de gebruiker
     RETURN 'Afd# IN (SELECT Afd#
          FROM Medewerker
          WHERE USER_NAME = USER)';
     ELSE
          – Blokkeert toegang
          RETURN 1 = 2;
     END IF;
END A2PolicyFunc;

DBMS_RLS.ADD_POLICY('schema', 'Afdeling',
'A2Policy', 'schema', 'A2PolicyFunc', 'update');
```

FIGUUR 2: VOORBEELD VAN DE WERKING VAN GEBRUIKERSGEGEVENSAUTORISATIE MET HORIZONTALE RESTRICTIES DOOR MIDDEL VAN ORACLES IMPLEMENTATIE 'FINE GRAINED OBJECT ACCESS'.

dan toch zeker in praktijk. Als de CRUD-matrix niet meer zou zijn dan een stuk systeemdocumentatie, na verloop van tijd als vanzelfsprekend verouderd, zouden de baten van een snellere systeemontwikkeling wellicht nog opwegen tegen de baten. De CRUD-matrix wordt echter ook door ontwerpers gebruikt om te bepalen welke constraints de operaties in een bepaalde functie kunnen schenden en die de programmeur dus moet afvangen. Uiteraard tenzij hij gebruik gemaakt van een mechanisme zoals besproken in voorgaande artikelen uit deze serie.

Het is dan ook niet verwonderlijk dat bij de huidige mode van iteratieve en dynamische systeemontwikkeling, waarbij specificaties dagelijks wijzigen, de programmeur met kennis van de business uitermate populair is. Immers, bij veranderende specificaties (lees: functionele eisen en/of CRUD-matrices) verwacht men van hem dat hij de lijst met aan te roepen validaties ook 'even' aanpast. De voorspelbaarheid van het eindresultaat wordt daar niet beter op. Helemaal gevaarlijk wordt het als applicatiebeheerders op grond van de opgeleverde CRUD-specificaties hun functionele autorisatie gaan inrichten. Gebruikt men niet tevens een autorisatie op gegevenselementniveau, dan betekent dit een reëel risico: op grond van niet up-to-date CRUD-documentatie kunnen gebruikers onbedoeld recht krijgen op het uitvoeren van bepaalde databasemutaties. Dat de traceerbaarheid achteraf van mutaties dan veel aandacht krijgt, is natuurlijk slechts een doekje voor het bloeden.

Uiteraard is het heel eenvoudig de correctheid van CRUD-matrices af te dwingen, namelijk door ze niet meer als passief maar als actief element in de applicatie op te nemen. Als een applicatie een mechanisme heeft voor de validatie van gebruikersgegevensautorisatie moet het ontwikkelen van een mechanisme voor de validatie van functiegegevensautorisatie relatief eenvoudig zijn. Een aantal ontwikkeltools doet dit reeds automatisch voor zover het schermfuncties betreft: de definitie van de schermen is bovenop de CRUD gebouwd, zodat de CRUD voor deze functies per definitie up-to-date is.

GEBRUIKERSFUNCTIE-AUTORISATIE

Met de laatste authorisatievariant, die voor de gebruikersfunctie, verlaten we definitief het randgebied tussen dbms en applicaties waarop deze artikelserie zich richt. Vanuit dbms-perspectief valt hier slechts te zeggen dat het wenselijk is te komen tot een duidelijke relatie tussen de gebruikers zoals de applicatie deze onderkent en de gebruikers op dbms-niveau. Hoe we dat kunnen vormgeven, komt aan de orde in het volgende artikel. Daarin zullen we ook weer een gegevensmodel ontwikkelen, ditmaal voor autorisatiemanagement. ●

Noot:

1. De term 'rol' voor een gebruikersgroep is standaard gebruik in de SQL-wereld. Ikzelf vind het gebruiker/rolmodel minder geslaagd, maar daarover meer in een van de volgende artikelen.

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