Beperking heeft te maken met hoge leeftijd van de taal

Silly SQL (5): Gebruikers op de tweede plaats

Rick van Rein

Gebruikers komen in SQL op de tweede plaats. En dat is niet altijd handig. Er zijn daardoor dingen die wel met LDAP kunnen, waar het datamodel gebruikers wel als eerste-orde elementen respecteert.

Een programmeertaal bevat componenten als eerste-orde data-elementen, en dingen die op een andere wijze zijn opgenomen, vaak met expliciete maar beperktere mogelijkheden. Een object-georiënteerde programmeertaal wordt zo genoemd omdat objecten er eerste-orde elementen zijn; een functionele programmeertaal heet zo omdat het functies als eerste-orde data-elementen behandelt. Een voorbeeld van dat laatste is bijvoorbeeld de map-functie:

```
map f [ ]    = [ ]
map f (x:xs) = f x : map f xs
```

Deze functie definieert dat een functie f op alle elementen in een lijst moet worden toegepast. Dus een lege lijst blijft leeg (de eerste regel), maar een regel met een kopelement x en een vervolglijst xs wordt omgerekend naar een kop die door de functie f is bewerkt en een restlijst waarop f weer via map wordt toegepast.
Merk op dat hier een functie f wordt doorgegeven als parameter. Dat maakt deze taal een functionele taal – functies worden als eerste-orde elementen beschouwd. (In de praktijk bedoelt men met een functionele taal ook een applicatieve taal, dat wil zeggen een taal zonder zij-effecten door assignment, maar dat is bezijden het punt dat we hier willen maken.)

Gebruikers zitten tweede rang

We noemen gebruikers in SQL tweede-orde componenten van de taal omdat er geen datastructuur is die gebruikers definieert. Weliswaar zijn er commando's om met gebruikers te werken, maar dat zijn commando's die gebruikers als heel andere dingen beschouwen dan de rest van de data. Er zijn daardoor alleen de voorgestanste ondersteunende mogelijkheden, en daar kan eigenlijk niet op worden voortgebouwd.
Merk op dat we hier nadrukkelijk spreken van SQL als standaard, en dat we het niet hebben over een concrete invulling van die taal in een DBMS-product. Het ligt voor de hand om gebruikers in een tabelvorm te administreren, en veel databases doen dat in de praktijk ook. Alleen is dat geen onderdeel van de standaard, en dus kunnen daar geen uitwisselbare aannames over worden gemaakt; van die aannames die je ook in je JDBC-code kunt gebruiken, bijvoorbeeld.
Gebruikersgegevens worden vaak in een afzonderlijke database bijgehouden, omdat daarmee wordt voorkomen dat die gegevens door de data zelf heen gaan lopen. Juist dit voorkomen van vermenging betekent dat er niet in standaard SQL kan worden gewerkt met gebruikers als element van een gewone query.

SQL is een standaard, maar wel een oude. In een serie 'Silly SQL' artikelen bespreken we de dingen die, in retrospect bezien, een stuk handiger hadden gekund.

Gebonden aan handen en voeten

Heel veel toepassingen van SQL kunnen prima overweg met een beperkt aantal gebruikers die los van de data zelf worden bijgehouden. Maar er zijn wel degelijk dingen die er onmogelijk door worden.
Denk bijvoorbeeld eens aan views op de database. Die dienen onder andere om te voorkomen dat een gebruiker informatie te zien krijgt die hem niet aangaat. Dat kan nu alleen categorisch worden gedaan, zoals een medewerker P&O die salarissen mag overzien, terwijl andere medewerkers dat niet te zien mogen krijgen. Dit is zo gebruikelijk dat het haast niet opvalt dat we hier maar wat roeien met de riemen die we hebben.

> Het ligt voor de hand om gebruikers in een tabelvorm te administreren

Waarom mag elke werknemer niet zijn eigen salaris inzien? Het antwoord heeft meer te maken met implementatie dan met de werkelijkheid: Het is omdat de werknemer niet als zichzelf inlogt maar onder een klasse van algemene werknemers, of in elk geval niet in de klasse van P&O'ers. Of als elke werknemer wel een persoonlijke account heeft, dan is het in elk geval zo dat er geen

afzonderlijke view is per werknemer. Stel je voor, een view salaris_XXX per werknemeraccount XXX, dat loopt al snel uit de klauwen. Is het niet omdat de DBMS daar niet op gebouwd is, dan wel op de administratieve kluwen die dat oplevert.
De oplossing is juist heel simpel wanneer gebruikersaccounts als eerste-orde elementen werken, vergelijkbaar met hoe functies in een functionele taal overgedragen kunnen worden.
Stel je voor dat SQL dat had gedaan, dan had je als viewdefinitie voor salarissen kunnen opzetten:

```
create view salaris as
select * from salariscore
where account=currentuser;
```

Dit gebruikt een verwijzing naar de huidige gebruiker in currentuser. Het gevolg van deze mogelijkheid is dat dezelfde view salaris opeens voor elke werknemer toegankelijk is, maar dat er een limiet zit op wat die view aan die persoon toont.
In een praktische personeels-database zal de accountnaam niet zomaar gebruikt worden, maar de koppeling is te leggen met gewone SQL-tabellen. Er kan zelfs voor worden gekozen om een rollenmechanisme op te zetten, en query's toe te laten zoals deze:

```
create view salaris as
select * from salariscore
where account=currentuser
or 'poadmin' in (
        select name
        from roles
        where account=currentuser );
```

Merk op dat hier gebruik wordt gemaakt van de standaard-mogelijkheden die SQL biedt. Dit is typerend voor de mogelijkheden van een eerste-orde element in een taal, en gebruikersaccounts zijn dat dus typisch niet in SQL.
De toegang tot gebruikers, en dus ook tot hun rechten, wordt bepaald door de specifieke beheercommando's die voor dit doel zijn gemaakt, dus vooral tot grant en revoke. Uitbreidingen van specifieke fabrikanten lossen dit niet op; niet omdat gebruikers meestal buiten de database blijven en niet omdat het geen standaard SQL is.

Hoe had het wel gemoeten?

De conclusie uit bovenstaande is dat gebruikers als eerste-orde element in de SQL-standaard hadden moeten worden opgenomen. Daarbij dient het zo te gebeuren dat allerlei standaard-mechanismen mogelijk zijn; er moet in where-clausules kunnen worden gesproken over gebruikers, en met name ook over de huidige gebruiker.
Met name het spreken van 'de huidige gebruiker' is bijzonder. Dat is namelijk een stukje data dat verschilt van wat anderen zien, maar zoals in bovenstaande voorbeelden duidelijk zal zijn geworden, is dat juist erg nuttige informatie om te hebben. De manier waarop een DBMS zoiets zou kunnen uitwerken kan verschillen. Een DBMS zou het hele accountgebeuren kunnen opslaan in een gewone tabel (met een naam als dbuser die uit de standaard zou volgen) waarop een gebruiker dan een view krijgt die alleen de eigen data toont, behalve voor gebruikers die gemachtigd zijn om alles te zien. Het klinkt logisch als een dbadmin met toegang tot alle gebruikersrecords, ook alle data te zien krijgt die elk van die gebruikers te zien kan krijgen. Een alternatieve implementatie zou zijn een tabel op te slaan in tijdelijk geheugen (RAM), of die zelfs te simuleren door middel van software. In beide gevallen zou in elk geval de huidige gebruikersnaam opvraagbaar zijn, en er kan mee worden gegoocheld. Met name is het nuttig dat het hele toegangsbeheer expliciet kan worden uitgeprogrammeerd in SQL – en dus zit het niet meer vast aan de beperkingen die volgen uit grant en revoke. Eén ding is wel belangrijk om te realiseren in een alternatieve aanspraak van gebruikersaccounts, en dat is het verwijzen naar hosts waarvan wordt ingelogd. Ook gehashte passwords dienen te worden opgeslagen en uiteraard zijn die alleen aanpasbaar voor hen die toegang hebben tot een systeem.

Concreet alternatief

Op tabelniveau horen we over SQL te denken als een fysieke laag – en dus tellen ontwerpoverwegingen als efficiency mee. Om die reden is het een goed idee om een scheiding aan te brengen in dynamische data en statische data. De dynamische data zouden een dbsession tabel vormen. Als we die met de hand zouden aanmaken (wat natuurlijk niet de bedoeling is) dan zou dat gebeuren met:

```
create table dbsession (
        username varchar (100) not null,
        hostname varchar (100) not null,
        sessionid char(16) not null,
        primary key (username) );
```

Deze tabel zou bij voorkeur in RAM worden beheerd, als een DBMS het efficiënt zou willen doen althans. Daarnaast kan er een statische tabel worden aangemaakt met daarin exact de zaken die grant en revoke normaal gesproken beheren:

```
create table _dbaccess (
        username varchar (100),
        hostname varchar (100),
        passwd_sha1 char (40) );

create view dbaccess as
select _dbaccess.* from _dbaccess
where dbsession.username = _dbaccess.username;
```

Zoiets. Een entry die NULL is wordt niet getest. En natuurlijk is het in het algemeen een slecht idee om de password hash vast te pinnen op SHA1. Ook moet er iets gedaan worden om _dbaccess niet algemeen zichtbaar te maken. Enfin, details.

Rollen zouden een geschikte uitbreiding vormen op dit schema – en het is ook bij uitstek een SQL-techniek, omdat elke gebruiker in meerdere rollen kan zitten, al dan niet samen met andere gebruikers. Dit hoeft geen SQL-standaard te zijn overigens; hier begint de eigen controle over het gebruikersbeheer al:

```
create table dbuserroles (
        userrole varchar (100) not null,
        username varchar (100) not null );
```

De userrole zou ideaal zijn om te beslissen of iemand zijn gegevens mag inzien. Het bovenstaande voorbeeld zou in dit geval neerkomen op:

```
create view salaris as
select salariscore.* from salariscore
where account=dbsession.username
or 'poadmin' in (
        select userrole
        from dbuserroles, dbsession
        where dbsession.username=dbuserroles.
                                        username );
```

Het is prettig dat hier al veel flexibeler met gebruikersaccounts kan worden omgegaan. Een tabel als _dbaccess zou bijvoorbeeld prima met extra kolommen uitgebreid kunnen worden met informatie als personeelsnummers. En ook een sessietabel met daarin gegevens voor de lopende sessie kan op een dergelijke manier uitgebreid worden om andere sessiedata in op te slaan. De sessionid kan op andere manieren worden uitgewisseld, bijvoorbeeld via een cookie in een beveiligde webverbinding. Die selecteert dan in één klap alle van toepassing zijnde gegevens.

Op de bres voor gebruikers

SQL is beperkt in wat er met gebruikersaccounts mogelijk is. De commando's grant en revoke bieden summiere mogelijkheden die SQL in de praktijk minder vriendelijk maakt dan strikt noodzakelijk was geweest. Helaas echter, zijn gebruikers niet als eerste-orde data-element in de taal opgenomen. Dat zal te maken hebben met de hoge leeftijd van de taal.

> Op tabelniveau horen we over SQL te denken als een fysieke laag

Achteraf bezien had SQL nooit met tweederangs gebruikers ontworpen mogen zijn. Maar achteraf heb je natuurlijk makkelijk praten.

Rick van Rein
Dr. ir. H. van Rein (rick@openfortress.nl) is ontwikkelaar en beheerder bij OpenFortress Digital signatures.

Update

Informatica gaat ook ongestructureerde data integreren

Met de overname van Itemfield verkrijgt Informatica datatransformatie-technologie voor uitwisseling van en toegang tot ongestructureerde en semi-gestructureerde data.
De technologie van Itemfield integreert bronnen met ongestructureerde data (bijvoorbeeld e-mail, pdf, Word, Excel) en semi-gestructureerde data (zoals branchespecifieke op XML gebaseerde standaarden zoals SWIFT bij de banken en HL7 in de gezondheidszorg), met traditionele gestructureerde bronnen.
Informatica gaat de technologie van Itemfield integreren in de eigen data-integratie-software. Daarnaast biedt Informatica haar klanten de mogelijkheid Itemfield te embedden in infrastructuren voor EAI, ESB, B2Bi en applicatieservers.

Microsoft en Progress Apama leveren oplossing voor MiFID

Progress Software Corporation en Microsoft gaan samenwerken om het Progress Apama Event Processing Platform aan te bieden als het hart van de MiFID (Markten voor Financiële Instrumenten) Solution suite van Microsoft. Met deze oplossing kunnen financiële instellingen wereldwijd voldoen aan de Europese MiFID-vereisten, die op 1 november 2007 in werking zullen treden.
Het Apama platform biedt met Complex Event Processing (CEP) en Business Activity Monitoring (BAM) bedrijven de mogelijkheid hun activiteiten te analyseren en te monitoren wat betreft het voldoen aan de MiFID vereisten van 'best execution'. Tegelijkertijd voldoen ze aan de MiFID vereisten voor rapportage, referentiedata en historie van handelsactiviteiten. MiFID vereist dat relevante markt- en handelsgegevens meer dan vijf jaar opgeslagen en bewaard moeten worden.
Het Apama Event Processing platform stelt organisaties in staat om de snelle informatiestromen op de financiële markt in de gaten te houden, complexe patronen te herkennen en actie te ondernemen – en dat alles in minder dan een milliseconde.
Daarnaast kunnen compliance-analisten met de grafische CEP-ontwikkeltool (Apama Event Modeller) binnen enkele minuten elementen van hun MiFID infrastructuur ontwerpen, evolueren, testen en implementeren. Met deze eigenschappen kunnen financiële instellingen meteen op veranderende vereisten van de richtlijnen inspelen, in plaats van pas na dagen, weken of maanden, zoals bij andere oplossingen het geval is.