Hoe theorie praktisch kan zijn

# De muteerbaarheid van views

Frido van Orden

**Deze keer staan views centraal. Views vormen op zich een van de meer bekende onderdelen van het relationele model, maar zijn in de dagelijkse praktijk toch een ernstig ondergeschoven kindje. Ook hebben views alles te maken met de concepten van fysieke en logische gegevensonafhankelijkheid.**

Wat is een view? Een ogenschijnlijk simpele vraag, niet? 'Een view is een afgeleide tabel', 'Een view is een soort voorgedefinieerd SQL statement', 'Een view is een niet-opgeslagen tabel', dat zijn meestal zo'n beetje de antwoorden die je krijgt. Al deze definities bevatten een zekere waarheid, maar toch zijn ze geen van allen juist. De eerste definitie, 'Een view is een afgeleide tabel', ligt nog het dichtst bij de formele definitie, als we tenminste in plaats van de SQL-term 'tabel' de officiële relationele term 'relatie' gebruiken. Het relationele model spreekt over *basisrelaties* en *afgeleide relaties*, waarbij de laatste veelal met de term 'view' worden aangeduid. Een view wordt gedefinieerd middels een relationele expressie, en in het geval van SQL door middel van een SQL SELECT-statement, zoals:

```
CREATE VIEW emp_dept_names AS
SELECT e.name emp_name, d.name dept_name
FROM employee e, department d
WHERE e.dept# = d.dept#
```

In het voorbeeld zijn 'employee' en 'department' tabellen waarin echte data ligt opgeslagen, en vormt de view 'emp_dept_names' afgeleide informatie, die niet hoeft worden opgeslagen maar 'at runtime' door het dbms wordt berekend. Simpel toch? Maar dan wordt het lastig. Wat te doen met een 'materialized view', ook wel bekend onder de naam 'snapshot view'? Materialized views komen niet voor in het relationele model maar wel in vele SQL-databases. Het idee is dat het resultaat van een view wordt opgeslagen in een tabel. Soms is deze tabel niet meer dan een snapshot van de database op een bepaald moment, en daarmee feitelijk te vergelijken met resultaat van het SQL statement 'create table t as select …'. In andere situaties wordt de materialized view op bepaalde tijdsintervallen automatisch bijgewerkt of zelfs direct bij mutaties van de onderliggende tabellen. Alleen in de laatste situatie is de inhoud van de materialized view in elke situatie identiek aan zijn non-materialized equivalent. Een materialized view is dus enerzijds een tabel (want data opslag) en anderzijds een view (want afgeleid).
Nu over naar het geavanceerdere werk. Veel relationele dbms'en kennen tegenwoordig de mogelijkheid om gegevens te partitioneren over meerdere schijven. Dit komt de performance ten goede, omdat veelal binnen 1 partitie kan worden gezocht of, indien meerdere partities benaderd moeten worden, deze benadering parallel op verschillende schijven kan plaatsvinden. In Oracle partitioneert men een tabel bijvoorbeeld zo:

```
CREATE TABLE sales
(invoice_no NUMBER,
sale_year  INT NOT NULL,
sale_month INT NOT NULL,
sale_day   INT NOT NULL )
```

## Relational Rules (8)

De vorig jaar overleden dr. E.F. Codd werd wereldberoemd met zijn serie publicaties over een gegevens(meta)model dat later bekend zou worden onder de naam 'Relationeel Model'. De eerste uit die serie publicaties was een intern IBM Research Report dat uitkwam in 1969. 2004 zal dus gelden als een lustrum – 35 jaar Relationeel Model. Frido van Orden schrijft in Database Magazine een serie artikelen over de betekenis en de erfenis van het gedachtegoed van Codd. In deze achtste aflevering worden regel 6, 8 en 9 gecombineerd besproken.

Regel 6: De view mutatie regel.
Alle views die theoretisch muteerbaar zijn dienen muteerbaar te zijn door het systeem.

Regel 8: Fysieke gegevensonafhankelijkheid.
De logische laag van de architectuur wordt afgebeeld op de fysieke laag van de architectuur. Gebruikers en programma's zijn niet afhankelijk van de fysieke structuur van de database.

Regel 9: Logische gegevensonafhankelijkheid.
Gebruikers en programma's zijn zoveel mogelijk onafhankelijk van de logische structuur van de database, dat wil zeggen, de logische structuur van gegevens kan wijzigen met minimale invloed op de applicatie.

```
PARTITION BY RANGE (sale_year, sale_month, sale_day)
(PARTITION sales_q1 VALUES LESS THAN (1997, 04, 01)
                                    TABLESPACE tsa,
PARTITION sales_q2 VALUES LESS THAN (1997, 07, 01)
                                    TABLESPACE tsb,
PARTITION sales_q3 VALUES LESS THAN (1997, 10, 01)
                                    TABLESPACE tsc,
PARTITION sales_q4 VALUES LESS THAN (1998, 01, 01)
                                    TABLESPACE tsd)
```

Als u goed kijkt ziet u dat dit eigenlijk helemaal niets bijzonders is: met views kunnen we precies hetzelfde bereiken:

```
CREATE TABLE sales_q1
(invoice_no NUMBER,
sale_year  INT NOT NULL,
sale_month INT NOT NULL,
sale_day   INT NOT NULL )
CHECK (sale_year < 1997 and sale_month < 04 and
                                     sale_day < 01)
CREATE TABLE sales_q2 ...
CREATE TABLE sales_q3 ...
CREATE TABLE sales_q4 ...
CREATE VIEW sales AS
SELECT invoice_no, sale_year, sale_month, sale_day
FROM sales_q1
UNION
SELECT invoice_no, sale_year, sale_month, sale_day
FROM sales_q2
UNION
SELECT invoice_no, sale_year, sale_month, sale_day
FROM sales_q3
UNION
SELECT invoice_no, sale_year, sale_month, sale_day
FROM sales_q4
```

Toch betaalt u bij heel wat leveranciers een heleboel geld voor deze 'syntactic sugar'. En ze lijken nog gelijk te hebben ook, want de onderverdeling in sales_q1, sales_q2, sales_q3 en sales_q4 is iets wat we graag 'onder de motorkap' houden: applicaties moeten gewoon op 'sales' geschreven kunnen worden. En wil de view-variant van 'sales' zich net zo gedragen als de gepartitioneerde tabel-variant dan moet uw dbms nogal wat kunnen:

- Selecties op 'sales' moeten om efficiënt uitgevoerd te worden gebruik maken van de kennis dat de onderliggende data gepartitioneerd zijn op sale_year, sale_month en sale_day. Deze informatie is aanwezig in de gedefinieerde check constraint, maar gaat u er maar niet vanuit dat de query-optimizer automatisch gebruik maakt van die kennis;
- Inserts, updates en deletes moeten vertaald worden naar operaties op de onderliggende tabellen. Dat is bepaald niet makkelijk en veel dbms'en verbieden dan ook het updaten van alle, behalve de allersimpelste, views. De gebruikte UNION in ons voorbeeld is voor Oracle in elk geval voldoende reden om er mee op te houden.

Het goed omgaan met views is niet bepaald het sterkste punt van hedendaagse rdbms-leveranciers. In plaats van hun huiswerk goed te doen bedenken ze liever een patch. Feest natuurlijk voor de marketing-afdeling, want een 'advanced partitioning option' verkoopt natuurlijk veel beter dan 'union view updatability', en reclame maken voor je optimizer is al helemaal 'not done'.
Een ander voorbeeld: de nieuw SQL3-standaard definieert het concept van zogenaamde 'subtabellen' als antwoord op de overervings-hiërarchieën uit de object-oriëntatie.
U maakt dan bijvoorbeeld deze database:

```
CREATE TABLE person(
person# NUMBER(4) NOT NULL,
name VARCHAR(30) NOT NULL)
PRIMARY KEY(person#);
CREATE TABLE employee UNDER person (
dept# NUMBER(4) NOT NULL,
salary NUMBER(7,2) NOT NULL);
CREATE TABLE sales_employee UNDER employee (
bonus NUMBER(6, 2));
```

Op de database kunnen vragen worden geformuleerd als:

```
SELECT person#, name, dept#, salary, bonus
FROM sales_employee
```

Geweldig! Maar met views kon u dit al jaren:

```
CREATE TABLE person(
person# NUMBER(4) NOT NULL,
name VARCHAR(30) NOT NULL)
PRIMARY KEY(person#);
CREATE TABLE employee_t (
Person# NUMBER(4) NOT NULL,
dept# NUMBER(4) NOT NULL,
salary NUMBER(7,2) NOT NULL)
PRIMARY KEY(person#)
FOREIGN KEY(person#) REFERENCES person;
CREATE TABLE sales_employee_t (
person# NUMBER(4) NOT NULL,
bonus NUMBER(6, 2))
PRIMARY KEY(person#)
FOREIGN KEY(person#) REFERENCES employee_t;
CREATE VIEW employee AS
SELECT p.person#, p.name, e.dept#, e.salary
FROM person p, employee_t e
WHERE p.emp# = e.emp#
CREATE VIEW sales_employee AS
SELECT p.person#, p.name, e.dept#, e.salary,
                                          se.bonus
FROM person p, employee_t e, sales_employee se
WHERE p.emp# = e.emp#
AND e.emp# = se.emp#
```

Met het gebruik van geavanceerde features als bijvoorbeeld Oracle clusters (die ervoor zorgen dat de rows van tabellen fysiek bij elkaar opgeslagen worden) kunt u de tabellen person, employee_t en sales_employee_t nog optimaal performant opslaan ook. Alleen nu maar hopen dat het dbms toestaat om join-views te updaten. Oracle geeft in elk geval niet thuis.

## Fysieke gegevensonafhankelijkheid

Met de zojuist besproken voorbeelden zijn we automatisch aangeland bij de concepten van logische en fysieke gegevensonafhankelijkheid. Fysieke gegevensonafhankelijkheid is een direct gevolg van het feit dat het relationele model een logisch model is, dat geen uitspraak doet over fysieke implementaties. Het is een van de dragende peilers van het relationele model en een van de grote zwaktes van de relationele dbms'en van vandaag, zoals we spoedig zullen zien.
Het meest bekende voorbeeld van fysieke gegevensonafhankelijkheid in het relationele model is het gebruik van indexen. Het al dan niet leggen van indexen in de database heeft een grote invloed op de wijze waarop het dbms intern functioneert. Voor gebruikers en applicaties is dit echter volkomen transparant: er is geen enkel verschil te bespeuren behalve dan in performance.
Het klinkt zo eenvoudig en vanzelfsprekend dat we al bijna weer vergeten zijn hoe kort het nog maar geleden is dat we bij de bouw van Cobol-, Btrieve- of Clipper-applicaties uitgebreide toegangspad-analyses maakten om voor elke database-operatie het meest optimale indexgebruik te specificeren.

Het voorbeeld van een gepartitioneerde tabel is ook een goede illustratie van fysieke gegevensonafhankelijkheid: vanuit applicatieperspectief is de gepartitioneerde tabel 'sales' volkomen identiek aan zijn niet-gepartitioneerde broertje. Jammer is wel dat de verschillende partities een puur technische constructie zijn: het is niet mogelijk om bijvoorbeeld een rechtstreekse selectie 'select count(*) from sales_q3' op een specifieke partitie uit te voeren.
In het equivalente view voorbeeld is dit wel mogelijk. Een duidelijk geval van 'meer is minder': de toegevoegde taalconstructie om tabellen in partities te verdelen is minder krachtig dan de reeds aanwezige functionaliteit van views!
Ook het subtabel-voorbeeld roept vragen op. U ziet dat bij de uitwerking van een variant met gebruik van views uitgegaan is van het gebruik van 1 tabel per 'klasse'. Een alternatief zou kunnen zijn het opslaan van de hele 'klasse-hiërarchie' in 1 tabel die alle kolommen bevat:

```
CREATE TABLE person_t
type VARCHAR(30) NOT NULL,
person# NUMBER(4) NOT NULL,
name VARCHAR(30) NOT NULL,
dept# NUMBER(4),
salary NUMBER(7,2),
bonus NUMBER(6, 2))
PRIMARY KEY(person#)
CHECK (
IF type='person' THEN
dept# IS NULL AND salary IS NULL AND bonus IS NULL
ELSE
dept# IS NOT NULL AND salary IS NOT NULL AND
IF type='sales_employee' THEN
bonus IS NOT NULL
ELSE
bonus IS NULL
END IF
END IF
)

CREATE VIEW employee AS
SELECT person#, name, dept#, salary
FROM person_t
WHERE type in ('employee', 'sales_employee');
CREATE VIEW sales_employee AS
SELECT person#, name, dept#, salary, bonus
FROM person_t
WHERE type = 'sales_employee';
CREATE TRIGGER employee_i
INSTEAD OF INSERT ON employee
INSERT INTO person_t(type, person#, name, dept#,
                                                salary)
VALUES('employee', :new.person#, :new.name,
                              :new.dept#, :new.salary);
CREATE TRIGGER sales_employee_i
INSTEAD OF INSERT ON sales_employee
INSERT INTO person_t(type, person#, name, dept#,
                                        salary, bonus)
VALUES('sales_employee', :new.person#, :new.name,
                :new.dept#, :new.salary, :new.bonus)
```

Een dergelijk alternatief wordt in de praktijk vaak toegepast, waarbij opvallend vaak performance als argument geldt: er hoeft immers nooit een join gelegd te worden. U ziet echter dat de constructie aanzienlijk complexer is; met een wilde check constraint en een aantal triggers om de type-kolom goed gevuld te krijgen. Hoewel beide alternatieven vanuit applicatieperspectief nu equivalent lijken is dit toch niet het geval. Wat gebeurt er bijvoorbeeld als we een employee tot sales-employee willen promoveren (of demoveren)? In het ene model doet u dan:

```
INSERT INTO sales_employee_t(person#, bonus)
VALUES('P1', 130)
```

en in het andere:

```
UPDATE person
SET type='sales_employee', bonus=130
WHERE person#='P1'
```

Wat we eigenlijk zouden willen is het elegante model met 1 tabel per 'klasse' en de performance van 1 tabel per 'klasse-hiërarchie'.

En het moet helaas gezegd worden, de hedendaagse rdbms'en laten u hier vreselijk in de kou staan. De oorzaak hiervan is het ooit ingenomen uitgangspunt dat 'tabellen' gelijk stonden aan 'basis-relaties' en die weer aan 'opgeslagen relaties', en anderzijds 'views' gelijk stonden aan 'afgeleide relaties' en die weer aan 'niet-opgeslagen relaties'. En hoewel afgeleide relaties met opslag al weer tijden bestaan in de vorm van materialized views, schitteren de basis-relaties zonder opslag door afwezigheid. Schematisch is dit weergegeven in tabel 1.

| | Opslag | Geen opslag |
|---|---|---|
| **Basis-relatie** | Tabel | ??? |
| **Afgeleide relatie** | Materialized view | View |

Tabel 1.

Men is met open ogen in de val getrapt dat het onderscheid tussen basis en afgeleide relaties een *logisch* onderscheid is, terwijl het verschil tussen opslag en geen opslag een *fysiek* onderscheid is. En nogmaals: het relationele model is een logisch model dat geen enkele uitspraken doet over fysieke implementaties! Herinnert u zich nog de begintijd van relationele dbms'en, toen om het luidst werd geroepen dat relationele databases niet konden performen omdat het gebruik van joins veel inefficiënter was dan het overlopen van pointer-kettingen?

Die stelling mag inmiddels ontkracht heten, maar helaas is indertijd de enkeling niet gehoord die doorzag dat niets een relationele database belet om een join (of beter: de onderliggende verwijzende sleutelrelatie) te *implementeren* als een pointer-ketting. Was op dat besef verder geborduurd, dan hadden we nu wellicht het sql statement 'CREATE VIRTUAL TABLE' of iets dergelijks gehad om basisrelaties mee te definiëren die geen eigen opslag kennen.

## Logische gegevensonafhankelijkheid

Logische gegevensonafhankelijkheid volgt vaak uit fysieke gegevensonafhankelijkheid. In het voorbeeld van de twee implementatiealternatieven zagen we al dat twee heel verschillende oplossingen zich vanuit applicatieperspectief soms moeilijk te onderscheiden zijn. Of iets een basis- of afgeleide relatie is is vaak volkomen arbitrair, zoals onderstaand (ik geef toe, enigszins gezocht) voorbeeld toont:

```
CREATE TABLE employee(emp#, name, sal);
CREATE VIEW emp_id AS SELECT emp FROM employee;
CREATE VIEW emp_name AS SELECT emp#, name
                                    FROM employee;
CREATE VIEW emp_sal AS SELECT emp#, sal
                                    FROM employee;
CREATE TABLE emp_id(emp#)
CREATE TABLE emp_name(emp#, name) FOREIGN KEY
                          (emp#) REFERENCES emp_id;
CREATE TABLE emp_sal(emp# sal) FOREIGN KEY(emp#)
                                REFERENCES emp_id;
CREATE VIEW employee as
SELECT e.emp#, en.name, es.sal
FROM employee e, emp_name en, emp_sal es
WHERE e.emp# = en.emp# AND e.emp# = es.emp#
```

Een ander voorbeeld van logische gegevensonafhankelijkheid is dat bij toevoegen van een kolom aan een tabel bestaande select- en delete-operaties altijd en insert- en update-operaties (bij gebruik van default waarden) meestal blijven werken.
Over het algemeen is logische gegevensonafhankelijkheid goed geregeld in de hedendaagse rdbms-producten. Een naar mijn mening opvallende uitzondering is echter het omgaan met joins. Hoewel joins niet per definitie over een verwijzende sleutel-relatie hoeven te gaan, is dit bijna altijd wel het geval. En hoewel de definitie van verwijzende sleutel relatie bij het dbms bekend is, programmeren we dat toch telkens weer uit:

```
SELECT e.name ename, d.name dname
FROM employee e INNER JOIN department d ON
                                    e.dept# = d.dept#
SELECT d.name, sum(e.salary)
FROM employee e INNER JOIN department d ON
                                    e.dept# = d.dept#
GROUP BY d.name
```

Indien nu ooit de definitie van de foreign key-relatie wijzigt, bijvoorbeeld doordat we in tabel employee de kolom dept# hernoemen naar dept_id, dan moet de gehele programmatuur worden aangepast. Hoeveel fraaier zou het volgende zijn:

```
SELECT e.name ename, d.name dname
FROM employee e JOIN department d ON emp_dept_fk
```

Merk op dat de specificatie 'INNER' is verdwenen: het dbms weet immers dat de verwijzende sleutel key relatie emp_dept_fk verplicht is en dat een outer join dus geen zin heeft!

## Vooruitblik

We zijn begonnen met views en gaan er nog even mee door.
U heeft inmiddels al wat kritische kanttekeningen over de muteerbaarheid van views in hedendaagse relationele dbms'en kunnen lezen. Zeker is dat de regel dat alle in theorie muteerbare views muteerbaar moeten zijn, door het systeem in de praktijk met voeten wordt getreden. Overigens is de muteerbaarheid van views de afgelopen decennia een vrijwel onuitputtelijke bron van wetenschappelijke publicaties geweest. Het merkwaardige daarbij is dat ondanks het fundamentele karakter van het onderwerp de verschillende gevolgde benaderingen wat betreft uitgangspunten soms mijlenver van elkaar verschilden.
Ook Chris Date heeft een duit in het view-mutatie zakje gedaan. De volgende keer zullen we zijn visie uitgebreid bespreken. En na lezing van dit artikel weet u waarom deze theorie praktisch is!

**Frido van Orden** (frido.van.orden@faapartners.com) is partner bij FAA Partners.