Kapitel 9: Integritätssicherung und Zugriffskontrolle

9.1 Integritätsbedingungen

Ziel:
Die Datenbank soll zu jedem Zeitpunkt die Zusammenhänge und Regeln der realen (Geschäfts-) Welt so akkurat wie möglich widerspiegeln. Daher finden in Informationssystemen vielfältige Plausibilitätsprüfungen statt; die Realisierung dieser Prüfungen kann einen signifikanten Anteil der Anwendungsentwicklung ausmachen. Besser ist es, die Gewährleistung der Datenintegrität aus den Anwendungsprogrammen "herauszufaktorisieren" und in deklarativer Form an das Datenbanksystem selbst zu delegieren. Dieses Vorgehen hat zwei große Vorteile:

- eine effektivere Kontrolle der Integrität
- eine signifikante Vereinfachung der Anwendungsprogrammierung

9.1.1 Typen von Integritätsbedingungen

- Statische Integritätsbedingungen (Invarianten bzgl. des Datenbankzustands, in Form von prädikatenlogischen Formeln)
  - datenmodellinhärente Integritätsbedingungen
    - Primärschlüsselbedingung
    - Fremdschlüsselbedingung
  - anwendungsspezifische Integritätsbedingungen
    - für ein Attribut eines Tupels
    - für ein Tupel
    - für mehrere Tupel einer Relation
    - für mehrere Relationen
- Dynamische Integritätsbedingungen (Invarianten bzgl. erlaubter Änderungen des Datenbankzustands)

Achtung:
Die logische Widerspruchsfreiheit der spezifizierten Integritätsbedingungen muß (vom Datenbankdesigner) sichergestellt werden. Widerspruchsfreiheit bedeutet hier, daß die Konjunktion aller Integritätsbedingungen erfüllbar (im Sinne der Logik) sein muß.

Zeitpunkt der Prüfung von Integritätsbedingungen

- am Ende einer Datenbankoperation (SQL-Anweisung)
- am Ende einer Transaktion (beim COMMIT WORK)

Reaktion auf Integritätsverletzungen

- Nichtausführung bzw. Rückgängigmachen der Datenbankoperation
- Abbruch der Transaktion (implizites ROLLBACK WORK)
- Ausführung von Folgeänderungen zur Wiederherstellung der Integrität

Beispiele:

Statische Integritätsbedingungen

1) Der Rabatt eines Kunden darf nicht über 50 Prozent liegen.
2) Der Rabatt eines ausländischen Kunden darf nicht über 30 Prozent liegen.
   (Annahme: Es gibt ein zusätzliches Kundenattribut Land.)
3) Der durchschnittliche Rabatt aller Kunden darf 30 Prozent nicht überschreiten.
4) Der Gesamtwert aller Produkte im selben Lager darf 1 Mio. DM nicht überschreiten.
5) Es muß mindestens ein Produkt geben.
6) Die Rechnungssumme einer Bestellung ergibt sich aus dem Produkt von
   Preis und bestellter Menge des bestellten Produkts abzüglich des Kundenrabatts.
7) Der Saldo eines Kunden ist die (negative) Summe der Rechnungssummen aller noch
   nicht bezahlten Bestellungen des Kunden.

Dynamische Integritätsbedingungen

8) Der Rabatt eines Kunden darf nie reduziert werden.
9) Der Rabatt eines Kunden darf innerhalb eines Jahres um maximal 10 Prozent
   angehoben werden.
10) Von Kunden, deren Saldo unter - 100000 DM liegt, werden keine Bestellungen
    mehr angenommen.
11) Der Status einer Bestellung darf sich nur in "geliefert" ändern, der Status einer
    gelieferten Bestellung nur in "bezahlt". Der Status einer bezahlten Bestellung darf
    sich nie mehr ändern.

9.1.2 Ausdrucksmittel zur Spezifikation von Integritätsbedingungen in relationalen DBS

Achtung: Die Möglichkeiten, die der SQL-Standard vorsieht, sind nicht notwendigerweise in allen Produkten (identisch) implementiert.

1) beim CREATE TABLE und mit CREATE ASSERTION

"Grobsyntax":

```
    CREATE TABLE tablename
    ( colname datatype [DEFAULT {defaultconst | NULL}] [colconstraint {, colconstraint …}],
    {, colname datatype [DEFAULT {defaultconst | NULL}] [colconstraint {, colconstraint …}] …}
)
    [tabconstraint {, tabconstraint …}]

colconstraint ::= NOT NULL |
                  [CONSTRAINT constrname]
                     { UNIQUE | PRIMARY KEY | CHECK (searchcond) |
                       REFERENCES  tablename [(colname)] […]  }

tabconstraint ::= [CONSTRAINT constrname]
                     { UNIQUE colname {, colname …} |
                       PRIMARY KEY colname {, colname …} |
                       CHECK (searchcond) |
                       FOREIGN KEY colname {, colname …}
                          REFERENCES  tablename [(colname)] […]  }

CREATE ASSERTION assertname CHECK (searchcond)
                        [INITIALLY {DEFERRED | IMMEDIATE}]
```

Constraints und Assertions sind benannt, um auf sie bei ALTER TABLE ... DROP CONSTRAINT ... oder DROP ASSERTION ... Bezug nehmen zu können.

Semantik von Constraints und Assertions:

Die in der CHECK-Klausel erlaubten Search Conditions sind genau die, die in der WHERE-Klausel einer Anfrage erlaubt sind, können also, wenn man sich auf den Kern von SQL beschränkt, mittels sql2trc' auf Formeln des sicheren TRK abgebildet werden (siehe Kapitel 5). Diese sind von der Form F(t1, ..., tn) mit den freien Tupelvariablen t1, ..., tn, die jeweils einer der Relationen R1, ..., Rn zugeordnet werden können. (Im einfachsten Fall gibt es genau eine freie Variable für die Relation, zu deren CREATE-TABLE-Anweisung die Constraint gehört.) Die Semantik der CHECK-Klausel ist dann die folgende prädikatenlogische Formel ohne freie Variablen:

$$\forall t1 \ldots \forall tn \; (\; (t1 \in R1 \wedge \ldots \wedge tn \in Rn) \Rightarrow F(t1, \ldots, tn) \;)$$

Da eine Datenbank als (Modell der) konjunktive Verknüpfung einer Menge H elementarer prädikatenlogischer Formeln (mit jeweils einer solchen Formel pro Tupel) interpretiert werden kann (siehe Kapitel 4), können Constraints und Assertions einfach als weitere konjunktiv mit H verknüpfte (nichtelementare) Formeln interpretiert werden. Für Integritätsbedingungen I1, ..., Im ist die Datenbank damit eine Formel $H \wedge I1 \wedge \ldots \wedge Im$ bzw. ein Modell dieser Formel.

Dabei gelten Constraints, die beim CREATE TABLE spezifiziert werden,

für leere Relationen immer als erfüllt.

Mögliche Implementierung von Integritätsprüfungen:

Für jede SQL-Änderungsanweisung, die eine in einer CHECK-Klausel definierte Integritätsbedingung potentiell verletzen könnte, wird eine SQL-Anfrage wie folgt generiert:

Die searchcond der CHECK-Klausel entspreche der prädikatenlogischen Formel F(t1, ..., tn) mit freien Variablen t1, ..., tn, die Relationen R1, ..., Rn zugeordnet sind. Dann wird die Anfrage SELECT t1, ..., tn FROM R1, ..., Rn WHERE NOT searchcond generiert. Wenn das Anfrageergebnis nichtleer ist, wird die Integritätsbedingung verletzt, ansonsten ist sie eingehalten.

Zeitpunkt der Integritätsprüfung:
am Ende jeder SQL-Änderungsanweisung (bei ASSERTION ... IMMEDIATE)
oder am Ende der Transaktion (bei ASSERTION ... DEFERRED).

Reaktion bei Integritätsverletzung:
Die SQL-Änderungsanweisung wird nicht ausgeführt bzw. rückgängig gemacht;
bei verzögerter Prüfung wird die gesamte Transaktion zurückgesetzt.
Eine flexiblere Reaktion ist nur für Verletzungen der referentiellen Integrität vorgesehen.

Beispiele:

Bedingungen 1, 2, 3:
```
CREATE TABLE Kunden ( ...
        Rabatt FLOAT
                CONSTRAINT Rabattbedingung CHECK (Rabatt BETWEEN 0.0 AND 0.5)
                CONSTRAINT Auslandsrabattbedingung CHECK (Land = 'D' OR Rabatt <= 0.3),
        ...,
        CONSTRAINT Durchschnittsrabattbedingung CHECK
                (0.3 >= (SELECT AVG(Rabatt) FROM Kunden))  )
```
Bedingung 4:
```
CREATE TABLE Produkte ( ...,
        CONSTRAINT Lagerwertbedingung CHECK
        (1000000.0 >=ALL (SELECT SUM(Vorrat*Preis) FROM Produkte GROUP BY Lager))  )
```
Bedingung 5:
```
CREATE ASSERTION Produktexistenzbedingung CHECK
        (EXISTS (SELECT * FROM Produkte) )
```
Bedingung 6:
```
CREATE ASSERTION Rechnungssummenbedingung CHECK
        ( Bestellungen.Summe =
        (SELECT B.Menge * Produkte.Preis * (1.0 - Kunden.Rabatt)
        FROM Bestellungen B, Produkte, Kunden
        WHERE B.PNr=Produkte.PNr AND B.KNr=Kunden.KNr
        AND B.BestNr=Bestellungen.BestNr ) )
```
Bedingung 7:
```
CREATE ASSERTION Saldobedingung CHECK
        ( Kunden.Saldo =     (SELECT SUM(Summe) FROM Bestellungen WHERE
                                  Bestellungen.KNr=Kunden.KNr AND Status <> 'bezahlt') )
```

Die Semantik von Bedingung 6 beispielsweise ist dann die logische Invariante:

$\forall$ best (best $\in$ Bestellungen $\Rightarrow$

($\exists$ b $\exists$ p $\exists$ k (b $\in$ Bestellungen $\wedge$ p $\in$ Produkte $\wedge$ k $\in$ Kunden $\wedge$

$b.PNr = P.PNr \wedge b.KNr = k.KNr \wedge b.BestNr = best.BestNr \wedge$
$best.Summe = b.Menge * p.Preis * (1 - k.Rabatt))))$

Flexible Reaktionsmöglichkeit bei Verletzung der referentiellen Integrität

"Grobsyntax" der FOREIGN-KEY-Klausel:

```
... FOREIGN KEY ( column {, column ...} )
    REFERENCES [user .] table [ ( column {, column ...} ) ]
    [ ON DELETE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
    [ ON UPDATE { NO ACTION | CASCADE | SET NULL | SET DEFAULT } ]
    [ INITIALLY { IMMEDIATE | DEFERRED } ] ...
```

Semantik von CREATE TABLE R ...
FOREIGN KEY A1, …, Am REFERENCES R1 (B1), …, Rm (Bm):

$\forall t \, (t \in R \Rightarrow ((t.A1=\omega \wedge \ldots \wedge t.Am=\omega) \vee$
$(\exists t1 \ldots \exists tm \, (t1 \in R1 \wedge \ldots \wedge tm \in Rm \wedge t1.B1=t.A1 \wedge ... \wedge tm.Bm=t.Am))))$

Zeitpunkt der Integritätsprüfung:
am Ende jeder SQL-Anweisung oder am Ende der Transaktion

Reaktion bei Integritätsverletzung:
Zurückweisung der Löschung/Änderung (bei NO ACTION)
Löschen/Ändern aller "abhängigen" Tupel (bei CASCADE)
Fremdschlüssel in allen "abhängigen" Tupeln auf Nullwert/Default-Wert setzen

Beispiel:

Kunden

KNr	...
1	
2	
...	

Produkte

PNr	...
1	
2	
...	

Bestellungen
(Fremdschlüssel KNr)

BestNr	Monat	Tag	KNr	Summe	Status
1001			1		
1002			2		
1003			1		
...					

Bestellposten
(Fremdschlüssel BestNr, PNr)

BestNr	PNr	Menge
1001	1	
1002	2	
1003	1	
1003	2	
...		

```
CREATE TABLE Bestellungen ( ... ,
        FOREIGN KEY KNr REFERENCES Kunden (KNr)
        ON DELETE SET NULL )
CREATE TABLE Bestellposten ( ... ,
        FOREIGN KEY PNr REFERENCES Produkte (PNr)
        ON DELETE CASCADE,
        FOREIGN KEY BestNr REFERENCES Bestellungen (BestNr)
        ON DELETE CASCADE )
```

Löschen von Kunde 1 ⇒ Bestellungen 1001 und 1003 erhalten den Nullwert als KNr
Löschen von Produkt 1 ⇒ Die Bestellposten für Produkt 1 werden gelöscht.

2) Trigger

Kernidee:
Vor oder nach einer bestimmten Art von Änderungsoperationen wird bei Erfüllung einer spezifizierten Bedingung eine Folge von SQL-Anweisungen automatisch ausgeführt. (Sprechweise: Der Trigger "feuert".)

Vorteile gegenüber rein deklarativer Spezifikation von Integritätsbedingungen:
- flexible Reaktion auf Integritätsverletzungen
- spezifischere Wahl der Überprüfungszeitpunkte
  (und damit u.U. eine effizientere, wenngleich eher prozedurale, Realisierung der Integritätssicherung)

"Grobsyntax":

CREATE TRIGGER trigger-name
{BEFORE | AFTER } { DELETE | INSERT | UPDATE [OF column {, column …}] }
ON table [REFERENCING OLD AS correlation-var NEW AS correlation-var]
[FOR EACH ROW | <u>FOR EACH STATEMENT</u>]
[WHEN (condition)]
(statement-sequence)

Dabei kann die REFERENCING-Klausel nur in Kombination mit FOR EACH ROW stehen.

Semantik:

Vor oder nach der spezifizierten Änderungsoperation wird die Condition in der WHEN-Klausel ausgewertet. Wenn Condition wahr ist, wird die spezifizierte Statement-Sequence ausgeführt, und zwar entweder wiederholt für jedes von der Änderungsoperation geänderte Tupel (Option FOR EACH ROW) oder nur einmal am Ende der gesamten Änderungsoperation (Option FOR EACH STATEMENT).

Die Condition der WHEN-Klausel ist dabei von der syntaktischen Form einer WHERE-Klausel (bzw. Constraint), im Kern also einer prädikatenlogischen Formel F entsprechend.

Bei der Option FOR EACH STATEMENT darf F keine freien Variablen haben. Nur wenn die Auswertung von F den Wert wahr ergibt, wird die vorgesehene Statement-Sequence ausgeführt.

Bei der Option FOR EACH ROW kann die der WHEN-Condition entsprechende prädikatenlogische Formel F ein oder zwei freie Variablen haben, die sich beide auf die von der Änderungsoperation betroffene Tabelle beziehen. Diese beiden Tupelvariablen - told und tnew - sind die in der REFERENCING-Klausel bei OLD bzw. NEW spezifizierten Variablen; fehlt die REFERENCING-Klausel, so gibt es nur eine implizite Tupelvariable t (den Tabellennamen), und zwar in der NEW-Rolle, falls AFTER spezifiziert wurde, und in der OLD-Rolle, falls BEFORE spezifiziert wurde. Die Tupelvariablen tnew, told bzw. t sind syntaktisch freie Variablen in t, sie werden aber wie Konstanten behandelt, bei denen die Attributwerte durch den alten bzw. neuen Zustand des aktuell geänderten Tupels gesetzt. Nur wenn mit dieser Substitution die Auswertung von F den Wert wahr ergibt, wird die vorgesehen Statement-Sequence ausgeführt.

Bei INSERT oder DELETE als Trigger-Ereignis ist - je nach Angabe von BEFORE oder AFTER - nur jeweils eine der beiden Tupelvariablen told und tnew relevant und die Angabe einer REFERENCING-Klausel sinnlos.

Beispiele:

Bedingung 7:

```
CREATE TRIGGER Saldoeintrag
AFTER INSERT ON Bestellungen FOR EACH ROW WHEN ( Status = 'neu' )
( UPDATE Kunden SET Saldo = Saldo - Summe WHERE Kunden.KNr=Bestellungen.KNr )
CREATE TRIGGER Saldoausgleich
AFTER UPDATE OF Status ON Bestellungen
REFERENCING OLD AS BOld  NEW AS BNew
FOR EACH ROW
WHEN ( BNew.Status = 'bezahlt' AND BOld.Status <> 'bezahlt' )
( UPDATE Kunden SET Saldo = Saldo + Summe WHERE Kunden.KNr=Bestellungen.KNr )
```

Bedingung 8:

```
CREATE TRIGGER Rabattmonotonie
AFTER UPDATE OF Rabatt ON Kunden
REFERENCING OLD AS KOld  NEW AS KNew
FOR EACH ROW
WHEN ( KNew.Rabatt < KOld.Rabatt )
        ( ROLLBACK WORK )
```

Bedingung 10:

```
CREATE TRIGGER Kundensperrung
BEFORE INSERT ON Bestellungen FOR EACH ROW
WHEN (    (SELECT Saldo FROM Kunden
           WHERE Kunden.KNr=Bestellungen.KNr) < -100000.0 )
        ( <Fehlermeldung ausgeben>; ROLLBACK WORK )
```

Achtung: Die Reihenfolge, in der die Trigger "feuern", ist u.U. essentiell.
Die durch einen Trigger ausgelöste Anweisungsfolge kann selbst wieder andere oder denselben Trigger "feuern".
(Da die Termination der so ausgelösten Anweisungsketten i.a. unentscheidbar ist und somit nicht prüfbar ist, beschränken kommerzielle Datenbanksystem zur Laufzeit einfach die maximale Schachtelung von „feuernden" Trigger-Ausführungen.)

→ Trigger sind ein sehr mächtiges Konzept zur Integritätssicherung; Triggerspezifikationen sind aber auch potentiell sehr fehleranfällig.

Trigger sind nicht nur zur unmittelbaren Integritätssicherung nützlich, sondern können u.U. auch zur aktiven Steuerung der „Business-Logik“ verwendet werden..

Beispiel:

```
CREATE TRIGGER Kundenbeförderung
AFTER INSERT ON Bestellungen
REFERENCING NEW AS BNew
FOR EACH ROW
WHEN (   (  (SELECT SUM(Summe) FROM Bestellungen
             WHERE Bestellungen.KNr=BNew.KNr) > 100000.0 )
         AND
         (  (SELECT SUM(Summe) - BNew.Summe FROM Bestellungen
             WHERE Bestellungen.KNr=BNew.KNr) <= 100000.0 ) )
(  UPDATE Kunden SET Rabatt = Rabatt + 0.05 WHERE Kunden.KNr=BNew.KNr;
   INSERT INTO Stammkunden SELECT * FROM Kunden WHERE Kunden.KNr=BNew.KNr )
```

In der durch den Trigger ausgelösten SQL-Anweisungsfolge können auch Stored Procedures aufgerufen werden (die z.B. bei Oracle in PL/SQL oder in Java geschrieben sein können).

Weiterentwicklung des Trigger-Konzepts für "Aktive Datenbanken"

In sog. aktiven Datenbanken hat man das versucht, das Trigger-Konzept auf allgemeinere **ECA-Regeln** der Form

on event **if** condition **do** action

zu erweitern. Im Gegensatz zu einem Trigger können sich ECA-Regeln auf zusammengesetzte Ereignisse (also nicht nur auf einzelne SQL-Änderungsoperationen) beziehen (Beispiele s.u.). Diese Ansätze hatten Einflüsse auf Workflow-Management-Systeme (siehe Kapitel 14), sind in kommerziellen Datenbanksystemen selbst aber nur in geringem Maße und nur für interne Zwecke realisiert.

Beispiele:

1) Auffüllen des Lagers für knappe Produkte automatisch initiieren:
   on after update Produkte.Vorrat
   if Produkte.Vorrat < 100
   do execute LagerAuffüllen(...)
2) Werbebriefe drucken für Kunden, die seit einem Jahr nichts mehr bestellt haben:
   on 0:00 **daily**
   do execute JunkMail(...)
3) Verschicken eines Mahnbriefs an Kunden, die dreimal hintereinander die Zahlungsfrist überschreiten.
4) Ausgabe einer Meldung, wenn der Kurs einer Aktie innerhalb der letzten 5 Minuten um mehr als 50% variiert.

9.2 Views (Sichten, virtuelle Relationen, intensionale Relationen)

Idee (eine von mehreren Motivationen für das View-Konzept):

Integritätssicherung wird einfacher, wenn weniger abgeleitete Daten gespeichert werden. Solche abgeleiteten Daten (z.B. Saldo) sollen vielmehr nur bei Bedarf berechnet werden. Um die Formulierung der entsprechenden Anfragen so einfach wie möglich zu machen, können abgeleitete Daten als "Views" zur Verfügung gestellt werden. Views erscheinen gegenüber dem SQL-Programmierer praktisch wie gespeicherte Relationen, ohne daß die Tupel der View wirklich gespeichert sind.

"Grobsyntax" zur Definition von Views:

```
CREATE VIEW view-name [ ( column {, column …} ) ]
AS select-block [ WITH CHECK OPTION ]
```

Beispiel:

gespeicherte Relationen:

```
Kunden ( KNr, Name, Stadt, Rabatt )
Bestellungen ( BestNr, Monat, Tag, KNr, PNr, Menge, Summe, Status )
```

zusätzliche View:

```
CREATE VIEW KundenInfo ( KNr, Name, Stadt, Rabatt, Saldo ) AS
    SELECT Kunden.KNr, Name, Stadt, Rabatt, SUM(Summe)
    FROM Kunden, Bestellungen
    WHERE Kunden.KNr = Bestellungen.KNr
    AND Status <> 'bezahlt'
    GROUP BY KNr, Name, Stadt, Rabatt
```

Abfrage des Saldos:

```
SELECT Saldo FROM KundenInfo WHERE KNr=1
```

oder beispielsweise auch:

```
SELECT Saldo FROM KundenInfo, Bestellungen
WHERE KundenInfo.KNr=Bestellungen.KNr AND BestNr=1001
```

Views können generell zur Vereinfachung von Abfragen definiert werden
(analog zu Zuweisungen in der Relationenalgebra).
Auf Views können wiederum weitere Views definiert werden.

Beispiele:

1) CREATE VIEW BestellungsInfo
   (BestNr, Monat, Tag, KNr, Kundenname, Rabatt, PNr, Produktbez, Menge, Summe, Status)
   AS
```
SELECT    BestNr, Monat, Tag, Kunden.KNr, Name, Rabatt,
          Produkte.PNr, Bez, Menge, Summe, Status
FROM Bestellungen, Kunden, Produkte
WHERE Bestellungen.KNr=Kunden.KNr AND Bestellungen.PNr=Produkte.PNr
```
   Anfrage z.B.:
```
SELECT Kundenname FROM BestellungsInfo WHERE Produktbez='Platte'
```
2) CREATE VIEW SuperBestellungsInfo AS
```
SELECT * FROM BestellungsInfo WHERE Summe > 10000.0
```

Ausführung von Operationen auf Views

Anfragen auf Views werden DBS-intern durch Substitution in Anfragen auf gespeicherten Relationen transformiert.

Für CREATE VIEW VIRT AS viewquery ist
sql2ra [SELECT A1, …, Am FROM TAB1, …, TABk, VIRT WHERE F] =
π[A1, ..., Am] (sql2ra' [F] (TAB1 $\times$ … $\times$ TABk $\times$ sql2ra[viewquery]))

Einfaches Beispiel:
σ[Rabatt>0.3] (SuperBestellungsInfo)
= σ[Rabatt>0.3] (σ[Summe>10000.0] (BestellungsInfo))
= σ[Rabatt>0.3] (σ[Summe>10000.0] (π[BestNr, ...] (Kunden |x| Bestellungen |x| Produkte)))

Mit Hilfe von Views sind u.U. sogar Anfragen möglich, die mit der SELECT-Anweisung des ursprünglichen SQL-Standards nicht oder nur schwierig ausdrückbar wären.

Beispiel:

Gegeben sei die Relation Prüfungen (<u>Fach, Student</u>, Note).
Welches ist das Fach mit der besten Durchschnittsnote?

Die Lösung
```
SELECT Fach FROM Prüfungen GROUP BY Fach
HAVING AVG(Note) <= ALL   (   SELECT AVG(Note) FROM Prüfungen
                              GROUP BY Fach )
```
war lange Zeit in vielen Systemen unzulässig (wird aber vom SQL-Standard erlaubt und z.B. von Oracle unterstützt).

Die einfachere Lösung mittels View:
```
CREATE VIEW Fachstatistik AS
    SELECT Fach, AVG(Note) AS Durchschnitt FROM Prüfungen
    GROUP BY Fach;
SELECT Fach FROM Fachstatistik
WHERE Durchschnitt = (  SELECT MIN(Durchschnitt) FROM Fachstatistik )
```
entspricht einer geschachtelten Subquery in der FROM-Klausel,
was in SQL92 in direkter Form (also ohne View) unzulässig wäre
und auch nur von einem Teil der kommerziell wichtigen Produkte unterstützt wird.

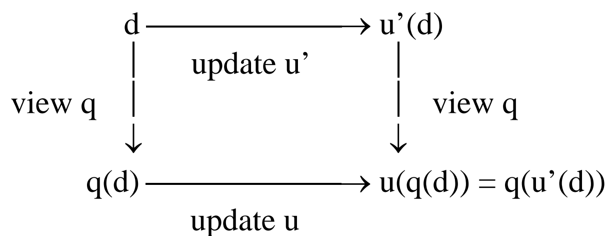
Ausführung von Update-Operationen auf Views

Änderungen eines Tupels einer View sind nur möglich, wenn sie eindeutig auf ein Tupel einer gespeicherten Relation abgebildet werden können (analog für Einfügen und Löschen).

Definitionen:

Sei D die Menge aller Datenbanken. Eine View ist eine partielle Funktion q: D → D; ein Update ist eine partielle Funktion u: D → D. Eine View q heißt *änderbar* genau dann, wenn es für jede Datenbank d, auf der q definiert ist, und jeden auf q(d) definierten Update u einen eindeutigen Update u' gibt, der auf d definiert ist und für den u(q(d)) = q(u'(d)) gilt.
(Zusätzlich wird häufig informell gewünscht, daß u' "intuitiv" definiert ist.)

Das folgende Abbildungsdiagramm muß also „kommutieren":



Beispiele:

1) UPDATE KundenInfo SET Stadt='Homburg' WHERE KNr=1
   ist erlaubt
2) UPDATE BestellungsInfo SET Produktbez = 'Druckerpapier' WHERE Monat=11
   ist nicht erlaubt !
3) UPDATE BestellungsInfo SET Produktbez = 'Druckerpapier' WHERE PNr=1
   ist theoretisch zulässig, aber in den meisten DBS nicht erlaubt !
4) UPDATE BestellungsInfo SET Produktbez = 'Druckerpapier' WHERE BestNr=1
   ist nicht erlaubt !
5) CREATE VIEW BestellungsKurzInfo (BestNr, Kundenname, Produktbez, Menge) AS
   SELECT BestNr, Name, Bez, Menge FROM Kunden, Bestellungen, Produkte
   WHERE Bestellungen.KNr=Kunden.KNr AND Bestellungen.PNr=Produkte.PNr
   INSERT INTO BestellungsKurzInfo VALUES (1111, 'Hempel', 'Maus', 5)
   ist nicht erlaubt (der zu generierende Update u' ist nicht eindeutig bestimmt) !
6) UPDATE KundenInfo SET Saldo = Saldo + 1000.0 WHERE KNr=1
   ist nicht erlaubt (der zu generierende Update u' ist nicht eindeutig bestimmt) !

Die Entscheidung, ob eine View änderbar ist oder nicht, ist ein schwieriges Problem. Daher erlauben die meisten Datenbanksysteme View-Updates nur, wenn die View durch Selektion und ggf. Projektion aus einer einzigen Tabelle erzeugt ist und den Primärschlüssel der Tabelle enthält. (Dies ist übertrieben konservativ.)

Änderungen eines View-Tupels, die dazu führen, daß das Tupel aus der View "verschwindet", können durch Spezifikation der CHECK OPTION verboten werden.

Beispiel:

CREATE VIEW SuperKunden AS SELECT * FROM Kunden WHERE Rabatt>0.3
WITH CHECK OPTION
INSERT INTO SuperKunden (KNr, Name, Stadt, Rabatt) VALUES (100, 'Meier', 'Homburg', 0.1)
wird zurückgewiesen

UPDATE SuperKunden SET Rabatt = Rabatt - 0.05 WHERE KNr=10
wird u.U. zurückgewiesen

Views als Mittel zur Datenunabhängigkeit bei Schemaänderungen

bisherige Anfrage:
SELECT * FROM Kunden WHERE KNr=1
Schemaänderung (plus Datenbank aktualisieren oder neu laden):
ALTER TABLE Kunden ADD Vorname CHAR(20), Nachname CHAR(20)
UPDATE Kunden SET Vorname = SUBSTR (Name, ...), Nachname = SUBSTR (Name, ...)
ALTER TABLE Kunden DROP Name
Vorgehen zur "Bewahrung" der bisherigen Anfrage:
- Kunden in KundenDaten umbenennen
- CREATE VIEW Kunden (KNr, Name, Stadt, Rabatt, Saldo) AS
SELECT KNr, Vorname | | ' ' | | Nachname, Stadt, Rabatt, Saldo FROM KundenDaten

aber: Kunden.Name ist nicht änderbar!

9.3 Datenschutz und Zugriffskontrolle

Begriffe

Datenschutz (engl.: data privacy):
Einschränkungen bei der Speicherung und Verarbeitung "kritischer" Daten, insbesondere personenbezogener Daten (Schutz der Privatsphäre von Personen)
→ Datenschutzgesetz („schützt Personen vor Daten“)

Zugriffskontrolle / Autorisierung (engl.: data security, authorization):
Verhinderung von unbefugten Zugriffen auf gespeicherte Daten
(„schützt Daten vor Personen“)

Maßnahmen der Zugriffskontrolle

1) Organisatorische Maßnahmen (kontrollierter Zugang zu den Rechnerräumen, zum u.U. drahtlosen Netz, usw.; z.B. Zugriff nur von bestimmten Geräten oder IP-Nummern)
2) Technische Maßnahmen (Datenverschlüsselung, kryptographische Protokolle etc.)
3) Maßnahmen des Betriebssystems
(Die der Datenbank zugrundeliegenden Dateien bzw. Platten sind nur für das DBS zugreifbar, also z.B. nur vom Account "Oracle" aus.)
4) Authentikation des DB-Benutzers
(typischerweise durch Angabe eines Kennworts beim CONNECT;
besser durch digitalen Schlüssel oder andere kryptographische Protokolle)
5) Prüfung der Zugriffsrechte des DB-Benutzers beim Zugriff auf Daten

Prüfung von Zugriffsrechten ("Discretionary Access Control")

Prinzip:
Subjekte haben *Rechte* (zur Ausführung von Operationen) auf *Objekten*.

Vergabe von Rechten durch die GRANT-Anweisung in SQL.
"Grobsyntax":

```
GRANT { ALL | privilege {, privilege ...} } ON { table | view }
TO { PUBLIC | user {, user …} } [ WITH GRANT OPTION ]
```

Mögliche Rechte zum Zugriff auf relationale Datenbanken sind:

SELECT	lesender Zugriff auf eine Relation
INSERT	Einfügen in eine Relation
UPDATE	Ändern von Tupeln einer Relation (ggf. nur bestimmte Attribute)
DELETE	Löschen von Tupeln einer Relation
CONNECT	Verbindung zum DBS aufnehmen ("Login"-Recht)
RESOURCE	Anlegen neuer Relationen (ggf. mit Limit für den Plattenplatz)
DBA	Datenbankadministration (z.B. Aufruf von Dienstprogrammen)
EXECUTE	Ausführung eines Anwendungsprogramms
IO_LIMIT	Beschränkung des Ressourcenverbrauchs für SQL-Anweisungen

Beispiele:
1) Benutzer Meier hat das Recht zur Ausführung von SELECT-Anweisungen auf der Relation Bestellungen.
   GRANT SELECT ON Bestellungen TO Meier
2) Benutzer Meier hat das Recht zur Ausführung des Programms Lieferung.
   GRANT EXECUTE Lieferung TO Meier
3) Das Programm Lieferung hat das Recht zur Änderung der Relation Bestellungen.
   GRANT UPDATE Status ON Bestellungen TO Lieferung

Prädikatorientierte Verfeinerung von Rechten durch Views

Beispiel:

Benutzer Meier hat das Recht zum Lesen der Kundendaten der Stadt Homburg.
CREATE VIEW KundenHOM AS SELECT * FROM Kunden
WHERE Stadt='Homburg'
GRANT SELECT ON KundenHOM TO Meier

Weitergabe und Rücknahme von Rechten

Für jedes Objekt gibt es genau ein Subjekt, genannt *Eigentümer* (engl.: owner), das alle Rechte für das Objekt besitzt.
Rechte können mit GRANT an andere Subjekte weitergegeben werden.

Bei Angabe der GRANT OPTION darf der Empfänger eines Rechts dieses selbst wiederum an andere Subjekte weitergeben !
Weitergegebene Rechte können mit der Anweisung
REVOKE privilege FROM user
wieder zurückgenommen werden.

Problemszenario:

Meier sei der Eigentümer der Relation MeierTabelle

Meier:	GRANT SELECT ON MeierTabelle TO Schmid WITH GRANT OPTION
Schmid:	GRANT SELECT ON MeierTabelle TO SchmidFreund WITH GRANT OPTION
Meier:	REVOKE SELECT ON MeierTabelle FROM Schmid
SchmidFreund:	GRANT SELECT ON MeierTabelle TO Schmid

Lösung in relationalen DBS:

REVOKE wirkt transitiv, nimmt also auch die vom Empfänger eines Rechts an Dritte weitergegebenen Rechte wieder zurück.

Ergänzende Literatur zu Kapitel 9:

J. Melton, A.R. Simon, Understanding the New SQL: A Complete Guide, Morgan Kaufmann, 1993

A. Behrend, R. Manthey, B. Pieper, An Amateur's Introduction to Integrity Constraints and Integrity Checking in SQL, 9. GI-Fachtagung über Datenbanksysteme in Büro, Technik und Wissenschaft, Oldenburg, 2001, Springer-Verlag

J. Widom, S. Ceri (Editors), Active Database Systems, Morgan Kaufmann, 1996

S. Ceri, R.J. Cochrane, J. Widom, Practical Applications of Triggers and Constraints: Successes and Lingering Issues, International Conference on Very Large Data Bases (VLDB), Cairo, 2000

S. Castano, M. Fugini, G. Martella, P. Samarati, Database Security, Addison-Wesley, 1995