Kapitel 5: Integritätssicherung und Zugriffskontrolle

5.1 Semantische Datenintegrität

Ziel:
Die Datenbank soll zu jedem Zeitpunkt die Zusammenhänge und Regeln der realen (Geschäfts−) Welt so akkurat wie möglich widerspiegeln.
Die Gewährleistung dieser Datenintegrität soll aus den Anwendungsprogrammen "herausfaktorisiert" werden und durch das Datenbanksystem selbst übernommen werden.

→ effektivere Kontrolle der Integrität
→ einfachere Anwendungsprogrammierung

Typen von Integritätsbedingungen

- Statische Integritätsbedingungen (Prädikate über dem Datenbankzustand)
  - datenmodellinhärente Integritätsbedingungen
    - Primärschlüsselbedingung
    - Fremdschlüsselbedingung
  - anwendungsspezifische Integritätsbedingungen
    - für ein Attribut eines Tupels
    - für ein Tupel
    - für mehrere Tupel einer Relation
    - für mehrere Relationen
- Dynamische Integritätsbedingungen (Prädikate über Zustandsänderungen)

Achtung:
Die logische Widerspruchsfreiheit der spezifizierten Integritätsbedingungen muß (vom Datenbankdesigner) sichergestellt werden.

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.

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

Zulässige Integritätsbedingungen sind alle in der WHERE−Klausel der SELECT−Anweisung zulässigen Suchprädikate.

→ Prüfung einer Integritätsbedingung durch Ausführung einer Query

Bedingungen beim CREATE TABLE gelten für leere Relationen immer als erfüllt.

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

Reaktion bei Integritätsverletzung:
Die SQL−Anweisung 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, Kunden WHERE
                        Bestellungen.KNr=Kunden.KNr AND Status <> 'bezahlt') )
```

Flexible Reaktionsmöglichkeit bei Verletzung der referentiellen Integrität

”Grobsyntax”:

... 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)] ...

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

”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 | FOR EACH STATEMENT ]
[ WHEN ”(” condition ”)” ]
”(” statement−sequence ”)”
```

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

Beispiele:

Bedingung 7:

```
CREATE TRIGGER Saldoeintrag
AFTER INSERT ON Bestellungen WHEN ( Status = 'neu' ) FOR EACH ROW
( UPDATE Kunden SET Saldo = Saldo − Summe WHERE Kunden.KNr=Bestellungen.KNr )

CREATE TRIGGER Saldoausgleich
AFTER UPDATE OF Status ON Bestellungen
REFERENCING OLD As BOld
FOR EACH ROW
WHEN ( 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 )
```

Trigger sind nicht nur zur unmittelbaren Integritätssicherung nützlich.

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 )
```

Statt der expliziten Angabe einer durch den Trigger ausgelösten Anweisungsfolge kann auch in einigen Systemen der Aufruf einer "Stored Procedure" spezifiziert werden.
Solche Prozeduren sind typischerweise in einer 4GL−artigen Sprache des jeweiligen Systems zu schreiben (z.B. PL/SQL für Oracle) und werden selbst in der Datenbank gespeichert.

Weiterentwicklung des Trigger−Konzepts für "Aktive Datenbanken"

Allgemeinere ECA−Regeln der Form:
on event **if** condition **do** action

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.

5.2 **Views** (Sichten, Virtuelle 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 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.

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 (Fach, Student, 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 SQL−Systemen nicht zulässig
(wird aber z.B. von Oracle unterstützt).

Die einfachere Lösung mittels View:

```
CREATE VIEW Fachstatistik AS
   SELECT Fach, AVG(Note) 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,
im künftigen SQL3−Standard aber vorgesehen ist.

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.)

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 in intuitiver Weise nicht möglich !
6) UPDATE KundenInfo SET Saldo = Saldo + 1000.0 WHERE KNr=1
   ist in intuitiver Weise nicht möglich !

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 neu laden):

```
ALTER TABLE Kunden DROP Name, ADD Vorname CHAR(20), Nachname CHAR(20)
```

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!

5.3 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

Zugriffskontrolle / Autorisierung (engl.: data security, authorization):
Verhinderung von unbefugten Zugriffen auf gespeicherte Daten

Maßnahmen der Zugriffskontrolle

1) Organisatorische Maßnahmen (z.B. kontrollierter Zugang zu den Rechnerräumen)
2) Technische Maßnahmen (Datenverschlüsselung 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)
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 5:

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

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

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

W. Gerhardt, Zugriffskontrolle bei Datenbanken, Oldenburg–Verlag, 1993