UNIVERSITÄT DES SAARLANDES
Fachrichtung 6.2 - Informatik
Lehrstuhl Prof. Weikum

Informationssysteme SS 2002

Übung 5 Beispiellösung

Aufgabe 1: Integritätsbedingungen

Gegeben sei das aus früheren Übungen bekannte Schema einer Universitätsdatenbank:

Professor	(P_Name, Fachrichtung_Nr, Gebäude, Raum, Tel)
Fachrichtung	(Fachrichtung_Nr, F_Name, Studiendekan)
Gebäude	(Gebäude, Hausmeister)
Student	(Matrikel_Nr, S_Name, Semester, Fachrichtung_Nr)
Prüfung	(Matrikel_Nr, Fach, Prüfer, Note)

a) Für die Relation Prüfung seien die folgenden Fremdschlüsselbedingungen spezifiziert:

```
CREATE TABLE Prüfung (
…
FOREIGN KEY Matrikel_Nr REFERENCES Student (Matrikel_Nr)
        ON DELETE CASCADE
FOREIGN KEY Prüfer REFERENCES Professor (P_Name)
        ON DELETE SET NULL)
```

Simulieren Sie mit Hilfe geeigneter Trigger-Spezifikationen den Effekt dieser Fremdschlüsselbedingungen.

```
CREATE TRIGGER LöschePrüfungen
AFTER DELETE ON Student
FOR EACH ROW
DELETE FROM     Prüfung
        WHERE Prüfung.Matrikel_Nr = Student.Matrikel_Nr

CREATE TRIGGER LöschePrüfer
AFTER DELETE ON Professor
FOR EACH ROW
UPDATE  Prüfung
        SET         Prüfer = NULL
        WHERE Prüfer = Professor.P_Name
```

b) Spezifizieren Sie die folgenden (nicht notwendigerweise mit der realen Welt übereinstimmenden) Integritätsbedingungen mit Hilfe von Assertion- oder Trigger-Deklarationen:

i) Studenten dürfen die Prüfung im Fach „Softwaretechnik“ erst ab dem 5. Semester ablegen.

```
CREATE ASSERTION CONSTRAINT Sem-5
CHECK NOT EXISTS   (SELECT   *
                    FROM     Prüfung p, Student s
                    WHERE    p.Matrikel_Nr = s.Matrikel_Nr
                    AND      Fach = 'Softwaretechnik'
                    AND      Semester < 5)
```

ii) Studenten, die bereits im 15. Semester sind, müssen mindestens eine Prüfung abgelegt und bestanden haben.

```
CREATE ASSERTION CONSTRAINT Sem-15
CHECK NOT EXISTS   ((SELECT   Matrikel_Nr
                     FROM     Student
                     WHERE    Semester > 14)
                     MINUS
                     (SELECT  Matrikel_Nr
                     FROM     Prüfung
                     WHERE    Note ≤ 4.3))
```

Da MINUS in früheren Versionen einiger DBS nicht implementiert war, hier noch eine zweite Variante:

```
CREATE ASSERTION CONSTRAINT Sem-15b
CHECK 1 ≤ ALL      (SELECT   COUNT(*)
                    FROM     Student s, Prüfung p
                    WHERE    s.Semester > 14
                    AND      s.Matrikel_Nr = p.Matrikel_Nr
                    AND      p.Note ≤ 4.3
                    GROUP BY s.Matrikel_Nr)
```

iii) Professoren sollen sich ihr Büro nicht mit Kollegen teilen müssen.

```
CREATE ASSERTION CONSTRAINT Professor-Büro
CHECK 1 = ALL  (SELECT    COUNT(*)
                FROM      Professor
                GROUP BY Gebäude, Raum)
```

Eine Variante ohne Group By kann wie folgt formuliert werden:

```
CREATE ASSERTION CONSTRAINT Professor-Büro-b
CHECK NOT EXISTS   (SELECT   *
                    FROM     Professor p1, Professor p2
                    WHERE    p1.P_Name ≠ p2.P_Name
                    AND      p1.Gebäude = p2.Gebäude
                    AND      p1.Raum = p2.Raum)
```

iv) Die Matrikel_Nr eines Studenten darf nie verändert werden.

```
CREATE TRIGGER MatNr
BEFORE UPDATE OF Matrikel_Nr ON Student
```

(<Fehlermeldung>; ROLLBACK WORK)

v) Die Note eines Studenten darf bei nachträglicher Änderung nur verbessert, nicht aber verschlechtert werden.

CREATE TRIGGER Noten-Update
AFTER UPDATE OF Note ON Prüfung
FOR EACH ROW
REFERENCES OLD AS old-prüf NEW AS new-prüf
WHEN (new-prüf.Note > old-prüf.Note) (ROLLBACK WORK)

Aufgabe 2: Integritätsbedingungen

Formulieren Sie die folgenden Integritätsbedingungen als Assertion oder Trigger in SQL:

a) Ein Buch kann frühestens ab seinem Erscheinungsdatum geliefert werden.
Standardlösung (mit Assertion bzw. Check-Klausel):

```
CREATE TABLE verkaeufe(
/* ... */
  CREATE ASSERTION CHECK (Lieferdatum>=ALL(SELECT edatum
  FROM Buecher WHERE Buecher.isbn=isbn))
);
```

Alternativlösung mit Trigger:

```
CREATE OR REPLACE TRIGGER A3a BEFORE INSERT OR UPDATE
   OF lieferdatum,isbn ON verkaeufe REFERENCING NEW AS new
   FOR EACH ROW
DECLARE ed DATE;
BEGIN
  SELECT edatum INTO ed FROM Buecher b WHERE b.isbn=:new.isbn;
  IF :new.lieferdatum<ed
  THEN
    ROLLBACK WORK;
  END IF;
END;
```

b) Für einen Kunden mit einer offenen Bestellung (also einem Bestelldatum ungleich NULL, aber Lieferdatum gleich NULL) darf die Bankverbindung nicht geändert werden.

```
CREATE OR REPLACE TRIGGER A3b BEFORE UPDATE
   OF bankverbindung ON kunden
   REFERENCING NEW AS new FOR EACH ROW
DECLARE num_best LONG;
BEGIN
  SELECT COUNT(*) INTO num_best FROM Verkaeufe V WHERE
V.email=:new.email AND
  v.Bestelldatum IS NOT NULL AND v.Lieferdatum IS NULL;
  IF num_best>0
  THEN
    ROLLBACK WORK;
  END IF;
END;
```

Aufgabe 3: Views

Betrachten Sie das Beispielschema der Vorlesung mit den Relationen:

Kunden (KNr, Name, Stadt, Saldo, Rabatt)
Produkte (PNr, Bez, Gewicht, Preis, Lagerort, Vorrat)
Bestellungen (BestNr, Monat, Tag, KNr, PNr, Menge, Summe, Status)

Nehmen Sie an, die Organisation des Unternehmens wird derart geändert, daß ein Produkt an verschiedenen Orten vorrätig gehalten werden kann. Bestellungen sollen grundsätzlich von dem Lager geliefert werden, das den größten Vorrat des entsprechenden Produkts hat.

Die Relation Produkte wird dazu in die zwei folgenden Relationen aufgespalten:

Prod (PNr, Bez, Gewicht, Preis)
Lager (PNr, Lagerort, Vorrat)

Wie können Sie erreichen, daß trotz dieser einschneidenden Änderung des Datenbankschemas und der Datenbank die für den Vertrieb essentielle Abfrage der Produktverfügbarkeit:

SELECT Vorrat FROM Produkte WHERE PNr = ...

und das Programm zur Erfassung von Lieferungen (siehe S. 88/89 Vorlesungsskript) unverändert weiterlaufen können?

Folgende UPDATE- und SELECT-Anweisungen müssen mindestens unterstützt werden:

```
SELECT Vorrat
FROM   Produkte
WHERE  PNr = ...

UPDATE   Produkte
SET      Vorrat = Vorrat – :menge
WHERE    PNr = :pnr
AND      Vorrat ≥ :menge
```

Wenn möglich, sollten jedoch alle auf der Tabelle Produkte möglichen SQL-Abfragen auf die View Produkte anwendbar sein.

```
CREATE   VIEW Produkte (PNr, Lagerort, Vorrat, Bez, Gewicht, Preis) AS
         SELECT L1.PNr, L1.Lagerort, L1.Vorrat, Prod.Bez, Prod.Gewicht, Prod.Preis
         FROM   Prod, Lager L1
         WHERE  Prod.PNr = L1.PNr
         AND    L1.Lagerort = (SELECT   MAX(L2.Lagerort)
                               FROM     Lager L2
                               WHERE    L2.PNr = L1.PNr
                               AND      L2.Vorrat = (SELECT   MAX(L3.Vorrat)
                                                     FROM     Lager L3
                                                     WHERE    L2.PNr = L3.PNr));
```

Das Attributschema der View Produkte ist gleich dem Attributschema der Relation Produkte in der ursprünglichen Datenbank und daher gibt es keine syntaktsichen Probleme.

1. SELECT-Anweisungen: Es wird zu jedem Produkt nur ein Tupel mit maximalem Vorrat und zugehörigem Lagerort ausgegeben. Falls es mehrere Lagerorte mit maximalem Vorrat gibt, wird einfach der alphabetisch letzte Lagerort ausgewählt. Daher liefert eine SQL-Anfrage sowohl mit

der View Produkte, als auch mit der ursprünglichen Relation Produkte vergleichbare Ergebnisse.

2. UPDATE-Anweisungen: Das Attribut Vorrat der View Produkte ist nicht abgeleitet. Durch den Join werden keine Werte des Attributs dupliziert, die Unterabfragen selektieren lediglich Datensätze aus dem Join. Also ist ein Update des Attributs Vorrat prinzipiell möglich, doch wird in kommerziellen DBS meist der konservative Weg beschritten und ein Update einer solchen View grundsätzlich verboten. Alle anderen Attribute werden durch den Join dupliziert und sind daher nicht über die View Produkte änderbar.