UNIVERSITÄT DES SAARLANDES
Fachbereich 14 – Informatik
Lehrstuhl Prof. Weikum

Datenbanksysteme WS 1999/200

Beispiellösung zu Übung 4

Aufgabe 1: Integritätsbedingungen

Gegeben sei das aus früheren Übungen bekannte Schema einer Universitätsdatenbank:

Professor	(P_Name, Fachbereichs_Nr, Gebäude, Raum, Tel)
Fachbereich	(Fachbereichs_Nr, Fachbereich_Name, Prodekan)
Gebäude	(Gebäude, Hausmeister)
Student	(Matrikel_Nr, S_Name, Semester, Fachbereichs_Nr)
Prüfung	(Matrikel_Nr, Fach, Prüfer, Note)

a) Für die Tabelle 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) Integritätsbedigungen mit Hilfe von Assertion- oder Trigger-Deklarationen:

i) Studenten dürfen die Prüfung im Fach Datenbanksysteme erst ab dem 5. Semester ablegen.

```
CREATE ASSERTION CONSTRAINT DBS-Klausur
CHECK  NOT EXISTS (SELECT *
                   FROM   Prüfung p, Student s
                   WHERE  p.Matrikel_Nr = s.Matrikel_Nr
                   AND    Fach = 'DBS'
                   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 Datenbanksystemen 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.sem > 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 MAT-Nr
BEFORE UPDATE OF Matrikel_Nr ON STUDENT
(<Fehlermeldung>; ROLLBACK WORK)
```

v) Die Note eines Studenten darf nur erhöht, nicht aber erniedrigt werden.

```
CREATE Trigger Noten-Update
AFTER UPDATE OF Note ON Prüfung
REFERENCES OLD AS old-pruef NEW AS new-pruef
WHEN (new-pruef.Note > old-pruef.Note) (ROLLBACK WORK)
```

Aufgabe 2: Integritätsbedingungen und Trigger

Eine Electronic-Commerce-Vertriebsfirma (z.B. für Bücher oder CDs) betreibe eine Datenbank mit den folgenden Relationenschemata:

Bestellungen	(BestellNr, ProduktNr, KundenNr, Anzahl, Lieferdatum);
Kunden	(KundenNr, Name, Adresse, E-Mail, Bestellimit);
Rechnungen	(BestellNr, KundenNr, Rechnungsdatum, Zahldatum);
Lager	(ProduktNr, Bez, Preis, Vorrat);
Nachbestellung	(ProduktNr, Anzahl, Bestelldatum);
Lieferung	(BestellNr, ProduktNr, KundenNr, Anzahl, Liefdatum);

Die Kunden der Firma haben die Möglichkeit, Produkte über das Internet zu bestellen. Sie bekommen dazu eine BestellNr (also einen elektronischen Warenkorb) zugewiesen, unter deren Angabe sie nacheinander verschiedene Produkte bestellen können. Bei jeder einzelnen Bestellung wird ein Tupel mit der zugewiesenen Bestellnummer, der Kundennummer, der Produktnummer, sowie der Anzahl der bestellten Produkte in die Relation Bestellungen eingefügt. Das Lieferdatum wird stets mit null initialisiert.

Innerhalb der Relation Bestellungen sind immer nur die Bestellungen eingetragen, die noch nicht ausgeliefert wurden oder vom Kunden noch nicht bezahlt wurden.

Innerhalb der Firma sind folgende Abläufe vorgeschrieben:

- Neue Bestellungen von Kunden werden zurückgewiesen, falls der Gesamtbetrag aller noch nicht bezahlten Bestellungen ein in der Relation Kunden eingetragenes Limit überschreitet.

Lösung:
Annahme: In der Relation Bestellungen sind nur unbezahlte Bestellungen.

```
DROP TRIGGER Bestellungslimitierung;
CREATE TRIGGER Bestellungslimitierung
BEFORE INSERT on Bestellungen
FOR EACH ROW
DECLARE
   limit                NUMBER;
   Bestsumme            NUMBER;
   Bestsumme2           NUMBER;
   zaehler              INTEGER;
BEGIN
     dbms_output.put('Trigger Best.limit. hat gefeuert !' );
      /* Welches Limit hat der Kunde ?*/
     SELECT K.Bestellimit INTO limit
     FROM Kunden K where :new.KundenNr=K.KundenNr;
      /* Hat der Kunde vorher schon etwas bestellt ?
         (Alles in Tabelle Bestellungen ist unbezahlt.)*/
      /*Bem.:Ich nehme hier an, dass der Kunde auch existiert.*/
     SELECT COUNT(*) INTO zaehler
     FROM Bestellungen B
     WHERE B.KundenNr=:new.KundenNr;
     IF(zaehler>0)
     THEN
          /* falls ja, fuer welchen Wert ?*/
         SELECT sum(B.Anzahl*L.Preis)
         INTO Bestsumme
         FROM Bestellungen B, Lager L
         WHERE B.ProduktNr=L.ProduktNr
         AND :new.KundenNr=B.KundenNr;
     ELSE
         Bestsumme:=0;
     END IF;
      /* Da der Trigger BEFORE INSERT feuert,
         muss hier noch der Betrag der einzufuegenden
         Bestellung dazu addiert werden.*/
     SELECT sum(:new.Anzahl*L.Preis)
     INTO Bestsumme2
     FROM  Lager L
     WHERE :new.ProduktNr=L.ProduktNr;
      /* Addiere beide Summen */
     Bestsumme:=Bestsumme+Bestsumme2;
      /* Limit ueberschritten ? */
     IF(Bestsumme>limit)
     THEN
```

```
            raise_application_error(-20000, 'Fehler:Bestellung kann nicht angenommen
                                             werden, da der Kunde
                                             sein Limit ueberschritten hat !');
     END IF;
END;
/
```

- Bei jeder Bestellung durch den Kunden wird automatisch geprüft, ob das bestellte Produkt in ausreichender Anzahl im Lager enthalten ist. Ist das Produkt in ausreichender Anzahl im Lager vorhanden, so wird die bestellte Anzahl vom Lagerbestand abgezogen, es wird eine Lieferung für die Bestellung in der Relation Lieferungen vorgesehen und das Lieferdatum in der Bestellung auf das aktuelle Datum gesetzt. Ist das Produkt nicht in ausreichender Form vorhanden, so wird eine Nachbestellung für das Produkt in der Relation Nachbestellung eingetragen. Existiert schon eine Nachbestellung (aufgrund anderer Bestellungen) für das Produkt, so wird die Anzahl einfach inkrementiert. Ansonsten wird die Nachbestellung (erstmals) eingefügt.
  Lösung:

```
DROP TRIGGER autoNachBest;
CREATE TRIGGER autoNachBest
AFTER INSERT ON Bestellungen
FOR EACH ROW
DECLARE
    P_nummer              INTEGER;
    Best_menge            INTEGER;
    Lagerbestand          INTEGER;
    Kunden_nummer         INTEGER;
    zaehler               INTEGER;
    Datum                    DATE;
BEGIN
    dbms_output.put('Trigger autoNachBest. hat gefeuert !' );
    SELECT K.KundenNr INTO Kunden_nummer
    FROM Kunden K where :new.KundenNr=K.KundenNr;
    Best_menge:=:new.Anzahl;
    P_nummer:=:new.ProduktNr;
     /* Bestimme Vorrat */
    SELECT sum(L.Vorrat) INTO Lagerbestand
    FROM Lager L
    WHERE L.ProduktNr=:new.ProduktNr;
     /* Reicht der Vorrat ?*/
    IF(Lagerbestand>=Best_menge)
    THEN
        /* Vorrat reicht fuer die Bestellung */
```

```
        /* vom Lagerbestand abziehen */
       UPDATE Lager L SET Vorrat=(Lagerbestand-Best_menge)
       WHERE L.ProduktNr=P_nummer;
        /* Lieferung generieren */
       INSERT INTO Lieferung
       VALUES (:new.BestellNr,:new.ProduktNr,:new.KundenNr,:new.Anzahl);
       /* Lieferdatum setzen */
       UPDATE Bestellungen B SET Lieferdatum=SYSDATE
       WHERE B.BestellNr=:new.BestellNr
             AND B.ProduktNr=:new.ProduktNr;
    ELSE
        /* Vorrat reicht nicht fuer die Bestellung */
        /* Wir bestellen die bestellten Produkte komplett nach.*/
        /* existiert schon eine Nachbestellung fuer dieses Produkt?*/
       SELECT count(*) INTO zaehler
       FROM Nachbestellung
       WHERE ProduktNr=:new.ProduktNr;
       if(zaehler>0)
       THEN
           /* Nachbestellung existiert schon -> inkrementieren*/
          UPDATE Nachbestellung SET Anzahl=Anzahl+Best_menge
          WHERE ProduktNr=:new.ProduktNr;
       ELSE
           /* neue Nachbestellung generieren */
          /*Datum:=:new.Bestelldatum; ist null ! */
          INSERT INTO Nachbestellung
          VALUES (P_nummer,Best_menge,SYSDATE);
       END IF;
    END IF;
END;
/
```

- Wenn eine Bestellung (ggf. ohne die nachbestellten Produkte) per Lkw den Hof des Lagers verläßt, soll der entsprechende Eintrag in der Relation Lieferungen gelöscht werden.
  Lösung: ECA-Regel
  - EVENT: Ein LKW verläßt den Hof mit der Menge von Lieferungen L2.
  - CONDITION: TRUE
  - ACTION:

```
DELETE FROM Lieferung L
WHERE EXISTS(SELECT * FROM L2
             WHERE L.BestellNr=L2.BestellNr
             AND   L.ProduktNr=L2.ProduktNr
```

```
AND   L.KundenNr=L2.KundenNr
AND   L.Anzahl=L2.Anzahl
);
```

- Nachbestellungen für Produkte werden immer komplett an den jeweiligen Kunden geliefert. Sowie eine Nachbestellung bei der Firma eintrifft, wird die Nachbestellung aus der Datenbank ausgetragen und der aktuelle Lagerbestand muß entsprechend erhöht werden. Nach Lieferung einer Nachbestellung sollen für alle Bestellungen, die auf das gelieferte Produkt warten und jetzt vollständig ausgeliefert werden können, Lieferungen generiert werden.
  Lösung:

```
DROP TRIGGER Nachbest_liefern;
CREATE TRIGGER Nachbest_liefern
AFTER DELETE ON Nachbestellung
FOR EACH ROW
DECLARE
   P_nummer             INTEGER;
   Best_menge           INTEGER;
   zaehler              INTEGER;
   Datum                   DATE;
   aktueller_vorrat     INTEGER;
   benoetigte_menge     INTEGER;
   CURSOR bestellungscursor IS SELECT * FROM Bestellungen B
                            WHERE  B.ProduktNr=:old.ProduktNr
                            AND (B.Lieferdatum IS NULL);
BEGIN
   dbms_output.put('Trigger Nachbest_liefern hat gefeuert !' );
   Best_menge:=:old.Anzahl;
   P_nummer:=:old.ProduktNr;
    /* Aktualisiere Lagerbestand*/
   UPDATE Lager L
   SET Vorrat=Vorrat+Best_menge
   WHERE L.ProduktNr=P_nummer;
    /*Pruefe fuer alle Bestellungen, die dieses Produkt enthalten, ob
      sie jetzt komplett ausgeliefert werden koennen.*/
    /*Idee: M=select from Bestllgn where pnr=old.pnr
      Mit Cursor fuer jede Bestellung im M eine Lieferung g
      generieren und in die Bestellung
      das Lieferdatum eintragen, solange
      der Vorrat reicht.*/
   SELECT L.Vorrat INTO aktueller_vorrat
```

```
    FROM Lager L
    WHERE L.ProduktNr=P_nummer;
    FOR btupel IN bestellungscursor LOOP
       dbms_output.put('Trigger Nachbest_liefern ist in for-schleife');
       IF(btupel.Anzahl<=aktueller_vorrat)
       THEN
          aktueller_vorrat:=aktueller_vorrat-btupel.Anzahl;
          INSERT INTO Lieferung VALUES(btupel.BestellNr,btupel.ProduktNr,
                                       btupel.KundenNr,btupel.Anzahl);
          /* update vom Datum */
          UPDATE Bestellungen B
          SET B.Lieferdatum=SYSDATE
          WHERE B.BestellNr=btupel.BestellNr;
       END IF;
    END LOOP;
    /*Die fuer die Auslieferung benoetigten Produkte wieder
      aus dem Lager entfernen.*/
    UPDATE Lager L
    SET L.Vorrat=aktueller_vorrat
    WHERE L.ProduktNr=P_nummer;
END;
/
```

- Sowie alle bestellten Produkte einer Bestellung an den Kunden ausgeliefert sind, soll automatisch eine Rechnung über den Gesamtbetrag der Bestellung an den Kunden geschickt werden und in der Relation Rechnungen registriert werden. Das Zahldatum wird initial auf NULL gesetzt.
  Lösung:
  Idee:
  Falls ein LKW den Hof verläst, löscht die ECA-Regel in der Tabelle Lieferung die entsprechenden Einträge. Ein Trigger, der 'AFTER DELETE ON Lieferung' feuert, prüft, ob noch Nachbestellungen oder weitere Lieferungen für diese Bestellung vorhanden sind. Falls nein, generiert er eine Rechnung.
  Problem: Wie kann man einer Bestellung ansehen, ob sie geliefert wurde oder gerade nachbestellt wird (Eine Nachbestellung kann von mehreren Bestellungen unterschiedlicher Kunden ausgelöst worden sein.)?
  Lösung: Das Attribut Lieferdatum ('Bestelldatum' im Schema auf dem Übungsblatt wird in 'Lieferdatum' umbenannt.) wird beim Einfügen von neuen Bestellungen mit null initialisiert und falls später eine entsprechende Lieferung generiert wird, mit dem Datum, an dem die Lieferung generiert wurde belegt. Bestellungen, die auf Nachbestellungen warten, erkennt man dann am Nullwert.

```
DROP TRIGGER Rechnung_erstellen;
CREATE TRIGGER Rechnung_erstellen
```

```
AFTER DELETE ON Lieferung
FOR EACH ROW
DECLARE
    anz_lieferungen  INTEGER;
    anz_nachbest     INTEGER;
BEGIN
   /* Existieren weitere Lieferungen ?*/
   SELECT count(*) INTO anz_lieferungen
   FROM Lieferung L
   WHERE L.BestellNr=:old.BestellNr;
   IF (anz_lieferungen=0)
   THEN
       /* falls nein Existieren Nachbestellungen ?*/
      SELECT count(*) INTO anz_nachbest
      FROM Bestellungen B
      WHERE B.BestellNr=:old.BestellNr
      AND   NOT( B.Lieferdatum IS null);
       IF (anz_nachbest=0)
       THEN
           /* Es gibt an dieser Stelle keine Lieferungen,
              fuer diese Bestellnummer, die den Hof
              noch nicht verlassen haben und
              keine Nachbestellungen fuer diese BestNr.
              -> dies ist die letzte Lieferung : mache Rechnung
             */
           INSERT INTO Rechnungen
           VALUES(:old.BestellNr,:old.KundenNr,
                  SYSDATE, null);
       END IF;
   END IF;
END;
/
```

- Sowie der Kunde bezahlt hat, trägt die Rechnungsabteilung das Zahldatum in Rechnungen ein. Dabei sollen alle bezahlten Bestellungen aus der Relation Bestellungen gelöscht werden.
  Lösung:

```
/*Idee: Trigger auf Update von Rechnungen, der prueft, ob old.zahldatum==null*/
DROP TRIGGER bezahlteBestLoeschen;
CREATE TRIGGER bezahlteBestLoeschen
AFTER UPDATE ON Rechnungen
FOR EACH ROW
```

```
BEGIN
   dbms_output.put('Trigger bezahlteBestLoeschen hat gefeuert !' );
   if ((:old.Zahldatum IS NULL) AND (NOT(:new.Zahldatum IS NULL)))
   THEN
       /* Loesche alle bezahlten Bestellungs-Tupel*/
      dbms_output.put('Trigger bezahlte_best_loeschen loescht ein Tupel aus Bestellungen !' );
      DELETE FROM Bestellungen B
      WHERE B.BestellNr=:new.BestellNr;
   END IF;
END;
/
```

- Manchmal macht die Rechnungsabteilung Fehler und markiert fälschlich Rechnungen als bezahlt. Deshalb soll das Zahldatum nur setzbar sein, wenn der Kunde tatsächlich alle Produkte der entsprechenden Bestellung empfangen hat.
  Lösung:
  Da eine Rechnung in Rechnungen erst eingefügt wird, wenn alle bestellten Produkte einer Bestellung an den Kunden ausgeliefert sind (vgl. vorheriger Punkt), existieren keine Rechnungen, die fälschlich markiert werden könnten. Es sei denn die Rechnungsabteilung fügt selbst Rechnungen ein.

Aufgabe 3: Views

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 TABELLE *Produkte* in der ursprünglichen Datenbank und daher gibt es keine syntaktischen 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-Abfrage sowohl mit der VIEW *Produkte*, als auch der ursprünglichen TABELLE *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 Datenbanksystemen meist der konservative Weg beschritten und ein Update von Views grundsätzlich verboten. Alle anderen Attribute werden durch den Join dupliziert und sind daher nicht über die VIEW *Produkt* änderbar.