Beispiel zur referentiellen Integrität

```
CREATE TABLE T1(
k1 NUMERIC NOT NULL PRIMARY KEY);

CREATE TABLE T2(
k2 NUMERIC NOT NULL PRIMARY KEY,
k1 NUMERIC,
FOREIGN KEY(k1) REFERENCES T1(k1) ON DELETE CASCADE);

CREATE TABLE T3(
k3 NUMERIC NOT NULL PRIMARY KEY,
k1 NUMERIC,
FOREIGN KEY(k1) REFERENCES T1(k1) ON DELETE CASCADE);

CREATE TABLE T4(
k4 NUMERIC NOT NULL PRIMARY KEY,
k2 NUMERIC,
k3 NUMERIC,
FOREIGN KEY(k2) REFERENCES T2(k2) ON DELETE CASCADE,
FOREIGN KEY(k3) REFERENCES T3(k3) ON DELETE SET NULL);

delete from T1 where k1 = 1;
delete from T1 where k1 = 2;
```

T_1	$k1$
	1
	2

T_2	$k2$	$k1$
	21	1
	22	2

T_3	$k3$	$k1$
	31	1
	32	2

T_4	$k4$	$k2$	$k3$
	41	21	31
	42	22	32
	43	22	31

3.14.4 Dynamische Integrität und Trigger

- *Dynamische* Integrität beschäftigt sich mit der Formulierung von Integritätsbedingungen, die definieren, welche Zustandsübergänge auf den Tabellen zu einem Datenbank-Schema erlaubt sind.
- Sie müssen es uns dazu ermöglichen, in einem Ausdruck sowohl den alten, wie auch den neuen Zustand der Instanzen ansprechen zu können.
- Zur Gewährleistung der dynamischen Integrität bietet SQL einen mächtigen *Trigger*-Mechanismus. Trigger sind ein Spezialfall *aktiver Regeln*, in denen in Abhängigkeit von eingetretenen Ereignissen, sofern gewisse Bedingungen erfüllt sind, definierte Aktionen auf einer Datenbank ausgeführt werden (**E**vent**C**ondition**A**ction-Paradigma).
- Innerhalb SQL sind die auslösenden Operationen gerade Einfügungen, Löschungen und Änderungen von Zeilen der Tabellen.

Anwendungen

- Prüfen der Zulässigkeit von Werten vor der Durchführung von Änderungen, um so im Falle von Integritätsverletzungen diese korrigieren zu können.
- Protokollieren von auf sicherheitskritischen Tabellen vorgenommene Änderungen, z.B. mit Angabe der Benutzeridentifikation und Zugriffszeit.
- Implementierung von Änderungsoperationen auf Sichten.
- Definition von für Anwendungen verbindlichen (Geschäfts-)Regeln.

Hinweis: Oracle hat für Trigger eine leicht andere Syntax.

Ändert sich die Einwohnerzahl einer Stadt, dann soll die Einwohnerzahl der betreffenden Provinz angepaßt werden.

```
CREATE TRIGGER EinwohnerzahlAnpassen
   AFTER UPDATE OF Einwohner ON Stadt
   REFERENCING OLD AS Alt NEW AS Neu
   FOR EACH ROW
   UPDATE Provinz P
      SET P.Einwohner=P.Einwohner-Alt.Einwohner+
         Neu.Einwohner
      WHERE P.LCode=Alt.LCode AND P.PName=Alt.PName
```

Die Tabelle Grenze soll antisymmetrisch sein; d.h., für je zwei Länder darf eine Nachbarschaftsbeziehung nur einmal enthalten sein.

```
CREATE TRIGGER antiSymGrenze
   BEFORE INSERT ON Grenze
   REFERENCING NEW AS Neu
   FOR EACH ROW
   WHEN EXISTS ( SELECT * FROM Grenze G
      WHERE G.LCode1=Neu.LCode2 AND
            G.LCode2=Neu.LCode1 )
   BEGIN
      SIGNAL SQLSTATE '75001';
      SET Message='Grenze bereits vorhanden'
   END
```

Ergibt die Summe der Anteile an den Kontinenten für ein Land einen kleineren Wert als 100, so wird die Differenz dem Kontinent Atlantis zugeordnet.

```
CREATE TRIGGER Atlantis
   AFTER INSERT ON Lage
   FOR EACH STATEMENT
   WHEN EXISTS ( SELECT * FROM Lage
                 GROUP BY LCode
                 HAVING (SUM(Prozent) < 100) )
   BEGIN
      INSERT INTO Lage
         SELECT L1.LCode, 'Atlantis', (
            100 - ( SELECT SUM(L2.Prozent)
               FROM Lage L2
               WHERE L2.LCode = L1.LCode) )
         FROM Lage L1
         GROUP BY L1.LCode
         HAVING (SUM(L1.Prozent) < 100)
   END
```

Trigger

- Ein Trigger ist einer Tabelle zugeordnet. Er wird aktiviert durch das Eintreten eines Ereignisses (SQL-Anweisung): Einfügung, Änderung und Löschung von Zeilen.
- Der Zeitpunkt der Aktivierung ist entweder vor oder nach der eigentlichen Ausführung der entsprechenden aktivierenden Anweisung in der Datenbank. Ein Trigger kann die Ausführung der ihn aktivierenden Anweisung verhindern.
- Ein Trigger kann einmal pro aktivierender Anweisung (Statement-Trigger) oder einmal für jede betroffene Zeile (Row-Trigger) seiner Tabelle ausgeführt werden.
- Mittels Transitions-Variablen `OLD` und `NEW` kann auf die Zeilen- und Tabellen-Inhalte vor und nach der Ausführung der aktivierenden Aktion zugegriffen werden. Im Falle von Tabellen-Inhalten handelt es sich dabei um hypothetische Tabellen, die alle betroffenen Zeilen enthalten.
- Ein aktivierter Trigger wird ausgeführt, wenn seine Bedingung erfüllt ist.
- Der Rumpf eines Triggers enthält die auszuführenden SQL-Anweisungen.
- Bei einem `BEFORE`-Trigger sind die *einzufügenden* Tupel nicht sichtbar in der Tabelle; es kann jedoch zu ihnen mittels `NEW` oder `NEW TABLE` zugegriffen werden. Bei einem `AFTER`-Trigger sind sie zusätzlich in der Tabelle zugreifbar.
- Bei einem `BEFORE`-Trigger sind die zu *löschenden* Tupel sichtbar in der Tabelle, bei einem `AFTER`-Trigger nicht; es kann zu ihnen mittels `OLD` oder `OLD TABLE` zugegriffen werden.
- Bei einem `BEFORE`- oder `AFTER`-Trigger kann zu den alten und neuen Werten der zu *ändernden* Tupel mittels `OLD/NEW` oder `OLD TABLE/NEW TABLE` zugegriffen werden. Bei einem `BEFORE`-Trigger sind die Änderungen nicht sichtbar in der Tabelle, bei einem `AFTER`-Trigger jedoch.

Bemerkungen

- Trigger können selbst weitere Trigger aktivieren, wenn ein ausgelöster Trigger eine Tabelle modifiziert, über der selbst Trigger definiert sind. Eine Transaktion kann somit während ihrer Ausführung eine ganze Reihe von Triggern auslösen.
- Die Reihenfolge der Ausführung dieser Trigger ist ohne weitere Kontrolle nicht vorhersehbar.
- Um eine deterministische Ausführung zu gewährleisten, sind Einschränkungen an die möglichen Triggerdefinitionen, bzw. Anforderungen an ihre Ausführung zu berücksichtigen.
- Eine Aktivierungsfolge von Triggern kann insbesonder zyklisch sein (*rekursive* Trigger); die Terminierung einer solchen Folge ist im Allgemeinen nicht gesichert.
- Seit SQL:2008 können auch `INSTEAD OF`-Trigger verwendet werden.

  Wird ein solcher Trigger aktiviert, dann werden *anstelle* der auslösenden Operation die Operationen des Triggers ausgeführt.

  Ein sinnvolles Anwendungsgebiet für `INSTEAD OF`-Trigger ist die Realisierung von Änderungsoperationen auf Sichten auf den zugehörenden Basistabellen.

Beispiel: Mit Triggern nachgebildete referentielle Aktionen.

T1	K1
	1

T2	K2	K1
	a	1

T3	K3	K1
	b	1

T4	K4	K2	K3
	c	a	b

```
/* DELETE CASCADE bei T2->T1 und T3->T1 */
/* Alternative 1: referentielle Aktion scheitert */
 CREATE TRIGGER t1delete_t2undt3
    AFTER DELETE ON T1 REFERENCING OLD as oldrow
    FOR EACH ROW
    BEGIN DELETE FROM T3 WHERE k1=oldrow.k1;
          DELETE FROM T2 WHERE k1=oldrow.k1; END
/* Alternative 2: referentielle Aktion erfolgreich */
 CREATE TRIGGER t1delete_t2undt3
    AFTER DELETE ON T1 REFERENCING OLD as oldrow
    FOR EACH ROW
    BEGIN DELETE FROM T2 WHERE k1=oldrow.k1;
          DELETE FROM T3 WHERE k1=oldrow.k1; END

/* DELETE CASCADE bei T4->T2 */
 CREATE TRIGGER t2delete_t4
    AFTER DELETE ON T2 REFERENCING OLD as oldrow
    FOR EACH ROW DELETE FROM T4 WHERE k2=oldrow.k2

/* DELETE RESTRICT bei T4->T3 */
 CREATE TRIGGER t3delete_t4
    AFTER DELETE ON T3 REFERENCING OLD as oldrow
    FOR EACH ROW WHEN (0 < (SELECT count(*) FROM T4 WHERE k3 = oldrow.k3))
        SIGNAL SQLSTATE '666' SET MESSAGE_TEXT='Dangling Reference'
```

3.15 Arbeiten mit Schema-Definitionen

Alle mit `CREATE` definierten Konstrukte sind Teil eines *Datenbankschemas*.

- SQL bietet Anweisungen an, mit denen existierende Schemata erweitert, oder auch einmal festgelegte Definitionen innerhalb eines Schemas wieder entfernt oder geändert werden können.
- Um einen nachträglichen Bezug zu existierenden Definitionen zu haben, müssen diese Definitionen mit einem Namen versehen werden.
- Die Zuordnung eines Namens ist auch sinnvoll, um im Falle von auftretenden Datenbankfehlern, wie Integritätsverletzungen, einen konkreten Bezug innerhalb einer Fehlernachricht zu bekommen.

Definition eines Schemas.

```
CREATE SCHEMA MondialDatenbank
```

Änderungen eines Schemas

- Mittels einer `DROP`-Anweisung können existierende mittels `CREATE` erzeugte Wertebereiche, Tabellen, Sichten und Assertions entfernt werden.
- Mittels `ALTER` können nachträglich Änderungen vorgenomen werden.
- Spalten und Integritätsbedingungen können mittels `DROP` entfernt, bzw. mittels `ADD` nachträglich eingefügt werden.

Die Tabelle `Land` wird um eine Spalte `Einwohner` erweitert; des Weiteren wird die Spalte `Hauptstadt` entfernt.

```
ALTER TABLE Land
   ADD COLUMN Einwohner NUMBER

ALTER TABLE Land
   DROP COLUMN HStadt
```

DEFERRED und IMMEDIATE Constraints.

Zwischen Tabellen T1 und T2 bestehe eine zyklische referentielle Beziehung. Aufgrund des gegenseitigen Bezuges ist die Definition der Tabellen nicht ohne Weiteres möglich. Beim Einfügen von Zeilen mit gegenseitigem Bezug ist eine Verletzung der Integrität bei direkter Überprüfung ohnehin nicht vermeidbar. Jedoch:

```
CREATE TABLE T1 (
... );

CREATE TABLE T2 (
...
   CONSTRAINT C2
      FOREIGN KEY ... REFERENCES T1
      INITIALLY DEFERRED );

ALTER TABLE T1 ADD CONSTRAINT C1
   FOREIGN KEY ... REFERENCES T2
   INITIALLY DEFERRED;

   :
INSERT INTO T1 ( ... ) VALUES ( ...);
INSERT INTO T2 ( ... ) VALUES ( ...);
SET CONSTRAINTS C1, C2 IMMEDIATE;
```

- Bei zyklischen Beziehungen kann zunächst die zuerst zu definierende Tabelle ohne `REFERENCES`-Klausel definiert werden.
- Nach erfolgter Definition der zweiten Tabelle die `REFERENCES`-Klausel mittels `ALTER Table` nachholen.
- Außerdem kann festgelegt werden, wann eine Integritätsbedingung überprüft werden soll.
- Zu jeder Integritätsbedingung kann mittels `IMMEDIATE` und `DEFERRED` festgelegt werden, ob sie direkt nach Ausführung einer SQL-Anweisung, oder nach Ausführung einer sie enthaltenden Transaktion überprüft werden soll.
- Diese Angaben können nachträglich modifiziert werden. Bedingungen können als `DEFERRABLE` oder `NOT DEFERRABLE` definiert werden. `INITIALLY` definiert den gültigen Modus für eine Transaktionen zu Beginn ihres Ablaufs. Mittels der `SET CONSTRAINTS`-Anweisung kann dann während der Ausführung einer Transaktion eine Bedingungen `IMMEDIATE` oder `DEFERRED` werden.

Beispiel zur referentiellen Integrität

```
CREATE TABLE Z1 (
K1 CHAR(2),
K2 CHAR(2),
PRIMARY KEY (K1) );

CREATE TABLE Z2 (
K2 CHAR(2),
K1 CHAR(2),
PRIMARY KEY (K2) );

ALTER TABLE Z1 ADD CONSTRAINT cyclic1
FOREIGN KEY (K2)
REFERENCES Z2 (K2) ON DELETE CASCADE;

ALTER TABLE Z2 ADD CONSTRAINT cyclic2
FOREIGN KEY (K1)
REFERENCES Z1 (K1) ON DELETE CASCADE;

/* INSERT, DELETE, COMMIT */
```

```
CREATE TABLE Z2 (
K2 CHAR(2),
K1 CHAR(2),
PRIMARY KEY (K2) );

CREATE TABLE Z1 (
K1 CHAR(2),
K2 CHAR(2),
PRIMARY KEY (K1),
CONSTRAINT cyclic1
  FOREIGN KEY (K2)
  REFERENCES Z2 (K2) ON DELETE CASCADE
  DEFERRABLE INITIALLY DEFERRED);

ALTER TABLE Z2 ADD CONSTRAINT cyclic2
  FOREIGN KEY (K1)
  REFERENCES Z1 (K1) ON DELETE CASCADE
  DEFERRABLE INITIALLY DEFERRED;

/* INSERT, DELETE, COMMIT */
```

LIKE- und AS-Klausel

- SQL:2003 bietet die `CREATE TABLE LIKE`-, bzw. `CREATE TABLE AS`-Klausel an.
- Im ersteren Fall wird die komplette Spaltendefinition einer existierenden Tabelle in die neu zu definierende Tabelle übernommen, wobei zusätzlich weitere neue Spalten hinzugenommen werden können.
- Im zweiten Fall wird die neue Tabelle mittels einer beliebigen SFW-Anweisung definiert. Es können somit beliebige Spalten aus existierenden Tabellen ausgewählt werden und es wird gleichzeitig eine Instanz der neuen Tabelle erzeugt.
- In beiden Varianten der `CREATE`-Klausel sind die neuen Tabellen unabhängig von ihren Ursprüngen.

Die Tabelle `Stadt_1` ist wie `Stadt` definiert und enthält zusätzlich eine Spalte `Fläche`.

```
CREATE TABLE Stadt_1 (
   LIKE        Stadt
   Fläche      NUMBER)
```

Die Tabelle `Stadt_2` hat den Inhalt von `Stadt` und zusätzlich für jede Stadt den Anteil an der Gesamtbevölkerung ihres Landes.

```
CREATE TABLE Stadt_2 AS (
   SELECT S1.*, (
      SELECT S1.Einwohner/SUM(S2.Einwohner)
      FROM Stadt S2 WHERE S1.LCode = S2.LCode ) AS Anteil
   FROM Stadt S1 )
   WITH DATA
```

3.16 empfohlene Lektüre

ENFORCING INCLUSION DEPENDENCIES AND REFERENTIAL INTEGRITY

Marco A. Casanova, Luiz Tucherman

Rio Scientific Center - IBM Brasil
P.O. Box 4624 - Rio de Janeiro - Brasil

Antonio L. Furtado

Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente, 225 - Rio de Janeiro - Brasil

ABSTRACT

The general architecture of a monitor that enforces inclusion dependencies and referential integrity is described. The monitor traces the operations a user submits in a session and can either modify an operation or propagate it, depending on additional information the database designer provided at design time. Propagation is implemented by executing new operations when the session terminates, using summary data collected during normal processing.

1. INTRODUCTION

When the database designer specifies a conceptual schema, he may include a set of integrity constraints to capture when a database state correctly reflects the real world. A database state is consistent when it satisfies all integrity constraints. Therefore, any operation modifying the database must preserve consistency, that is, map consistent states into consistent states.

that accepts a user's program and produces a new program that has new tests and operations, depending on the constraints of the schema. The new program would have the same basic behavior as the old program, but it would preserve consistency of the database. Such strategy would then help produce correct pre-defined update programs. In a second scenario, the system may have a *constraint enforcement monitor*, acting as a front-end to the DBMS, that controls (interprets) streams of operations guaranteeing that the final database state is consistent. This second strategy would allow users to submit on-line streams of operations leaving to the monitor the problem of consistency preservation.

However, it is very difficult to find an optimized enforcement strategy due to the intrinsic complexity of general integrity constraints. Therefore, it is reasonable to concentrate on small, but significant classes of constraints that can be enforced efficiently.

This paper contributes to the investigation on the automatic enforcement of constraints by describing the general architecture of a monitor that enforces inclu-

1

[1] In: Proceedings of the 14th VLDB Conference Los Angeles. California 1988. Kann gegoogelt werden.