4.14 Integrität und Trigger

- Im Allgemeinen sind nur solche Instanzen einer Datenbank erlaubt, deren Relationen die der Datenbank bekannten *Integritätsbedingungen* (IB) erfüllen.
- Integritätsbedingungen können *explizit* durch den Benutzer definiert werden, durch die Definition von konkreten Schemata *implizit* erzwungen werden, oder bereits dem relationalen Datenmodell *inhärent* sein.
- *Inhärente Bedingungen:* Attributwerte sind skalar; Relationen, abgesehen von Duplikaten, verhalten sich wie Mengen, d.h. ohne weitere Angaben haben sie insbesondere keine Sortierung.
- *Implizite Bedingungen:* Werte der Attribute eines Primärschlüssels dürfen keine Nullwerte enthalten
- *Explizite Bedingungen:* Werden als Teil der `CREATE TABLE`-Klausel, bzw. der `CREATE SCHEMA`-Klausel definiert.

- Integritätsbedingungen sind von ihrer Natur aus deklarativ: sie definieren die zulässigen Instanzen, ohne auszudrücken, wie eine Gewährleistung der Integrität implementiert werden kann.

  Eine wichtige Klasse von deklarativen Integritätsbedingungen sind *Fremdschlüsselbedingungen*, die gewährleisten, dass keine *dangling* Referenzen zwischen den Tupeln in den Tabellen bestehen.
- Komplementär zu den deklarativen Bedingungen bieten Datenbanksysteme einen *Trigger*-Mechanismus an, mit dem in Form von Regeln definiert werden kann, welche Aktionen zur Gewährleistung der Integrität vorgenommen werden sollen, bzw., wie Verletzungen behandelt werden sollen.
- Mittels Trigger können wir insbesondere die Zulässigkeit von *Zustandsübergängen* kontrollieren, was mit Integritätsbedingungen aufgrund ihres Bezugs zu gerade einem Zustand nicht möglich ist.

4.14.1 Fremdschlüsselbedingungen

- *Fremdschlüsselbedingungen* werden als Teil der `CREATE TABLE`-Klausel definiert.
- Sie sind formal sogenannte *Inklusionsabhängigkeiten*: zu jedem von `null` verschiedenen Fremdschlüsselwert in einer Zeile der *referenzierenden* Tabelle, der C- (child-) Tabelle, existiert ein entsprechender Schlüsselwert in einer Zeile der *referenzierten* Tabelle, der P- (parent-) Tabelle.
- Man redet hier auch von *referentieller* Integrität. Zur Definition von Fremdschlüsselbedingungen steht die `FOREIGN KEY`-Klausel zur Verwendung in der C-Tabelle zur Verfügung.

Die Spalte `LCode` innerhalb der Tabelle `Provinz` enthält Werte des Schlüssels `LCode` der Tabelle `Land`. Zu jeder Zeile in `Provinz` muss eine Zeile in `Land` existieren, deren Schlüsselwert gleich dem Fremdschlüsselwert ist.

```
CREATE TABLE Provinz (
   PName    VARCHAR(35),
   LCode    VARCHAR(4),
   Fläche   NUMBER
   PRIMARY KEY (PName, LCode),
   FOREIGN KEY (LCode) REFERENCES Land (LCode) )
```

Die Zeilen der Tabelle Grenze enthalten jeweils zwei unterschiedliche Fremdschlüssel, die beide Werte des Schlüssels LCode der Tabelle Land annehmen.

```
CREATE TABLE Grenze (
   LCode1   VARCHAR(4),
   LCode2   VARCHAR(4),
   Länge    INTEGER,
   PRIMARY KEY (LCode1, LCode2),
   FOREIGN KEY (LCode1) REFERENCES Land (LCode),
   FOREIGN KEY (LCode2) REFERENCES Land (LCode) )
```


Für die Tabelle Stadt sind zwei Fremdschlüsselbeziehungen relevant. Einmal müssen die referenzierten Länder in der Tabelle zu Land existieren, und zum andern entsprechend die Provinzen. Letzterer Fremdschlüssel besteht aus zwei Spalten. Die Zuordnung der einzelnen Spalten des Fremd- und Primärschlüssels ergeben sich aus der Reihenfolge des Hinschreibens.

```
CREATE TABLE Stadt (
   ⋮
   PRIMARY KEY (SName, LCode, PName),
   FOREIGN KEY (LCode) REFERENCES Land (LCode),
   FOREIGN KEY (LCode, PName) REFERENCES Provinz (LCode, PName) )
```

referentielle Aktionen im Überblick

Zur Gewährleistung der referentiellen Integrität werden sogenannte *referentielle Aktionen* zur Ausführung bezüglich der C-Tabellen definiert. Aufgabe dieser Aktionen ist die Kompensierung von durch `DELETE`- und `UPDATE`-Operationen auf der zugehörigen P-Tabelle verursachten Verletzungen der Integrität.

- Änderungen der P-Tabelle werden auf die C-Tabelle übertragen (Aktion `CASCADE`).
- Die Änderung der P-Tabelle wird im Falle einer Verletzung der referentiellen Integrität einer C-Tabelle abgebrochen (Aktion `NO ACTION` und Aktion `RESTRICT`).
- Der Fremdschlüsselwert der C-Tabelle wird angepaßt (Aktion `SET NULL` und Aktion `SET DEFAULT`).

Hinweis: Oracle unterstützt nur DELETE-Operationen. Als referentielle Aktionen kann CASCADE und SET NULL, SET DEFAULT gewählt werden - keine Angabe entspricht NO ACTION.

Wird der Code eines Landes geändert oder das Land gelöscht, so sollen die neuen Codes bei den zugehörigen Provinzen nachgezogen werden, bzw. auch die Provinzen des gelöschten Landes gelöscht werden.

```
CREATE TABLE Provinz (
   ⋮
   FOREIGN KEY (LCode) REFERENCES Land (LCode)
      ON DELETE CASCADE ON UPDATE CASCADE )
```

Werden Provinzen gelöscht, so sollen ihre Städte weiter in der Datenbank bestehen bleiben, wobei der betreffende Fremdschlüsselwert Nullwerte erhält. Änderungen eines Provinzschlüssels sollen auf die betroffenen Städte übertragen werden.

```
CREATE TABLE Stadt (
   ⋮
   PRIMARY KEY (SNAME)
   FOREIGN KEY (LCode, PName)
      REFERENCES Provinz (LCode, PName)
      ON DELETE SET NULL ON UPDATE CASCADE )
```

Warum werden nur DELETE- und UPDATE-Operationen auf den zugehörigen P-Tabellen betrachtet?

- Einfügen bezüglich der P-Tabelle oder Löschen bezüglich der C-Tabelle ist für die referentielle Integrität immer unkritisch.
- Einfügen bezüglich der C-Tabelle oder Ändern bezüglich der C-Tabelle, die einen Fremdschlüsselwert erzeugen, zu dem kein Schlüssel in der P-Tabelle existiert, sind immer primär unzulässig, da von Änderungen in den C-Tabellen im Allgemeinen kein sinnvoller Rückschluss auf Änderungen der P-Tabellen möglich ist; anderenfalls sind die Änderungen unkritisch.

referentielle Aktionen

`NO ACTION:`	Die Operation auf der P-Tabelle wird zunächst ausgeführt; ob Dangling References in der C-Tabelle entstanden sind wird erst nach Abarbeitung aller durch die Operation auf der P-Tabelle direkt oder indirekt ausgelösten referentiellen Aktionen überprüft.
`RESTRICT:`	Die Operation auf der P-Tabelle wird nur dann ausgeführt, wenn durch ihre Anwendung keine Dangling References in der C-Tabelle entstehen.
`CASCADE:`	Die Operation auf der P-Tabelle wird ausgeführt. Erzeugt die DELETE/UPDATE-Operation Dangling References in der C-Tabelle, so werden die entsprechenden Zeilen der C-Tabelle ebenfalls mittels DELETE entfernt, bzw. mittels UPDATE geändert. Ist die C-Tabelle selbst P-Tabelle bezüglich einer anderen Bedingung, so wird das DELETE/UPDATE bezüglich der dort festgelegten Lösch/Änderungs-Regel weiter behandelt.
`SET DEFAULT:`	Die Operation auf der P-Tabelle wird ausgeführt. In der C-Tabelle wird der entsprechende Fremdschlüsselwert durch die für die betroffenen Spalten in der C-Tabelle festgelegten DEFAULT-Werte ersetzt; es muss jedoch gewährleistet sein, daß entsprechende Schlüsselwerte in den P-Tabellen existieren.
`SET NULL:`	Die Operation auf der P-Tabelle wird ausgeführt. In der C-Tabelle wird der entsprechende Fremdschlüsselwert spaltenweise durch NULL ersetzt. Vorausetzung ist hier, daß Nullwerte zulässig sind.

Bei Verwendung von `RESTRICT` können in Abhängigkeit von der Reihenfolge der Abarbeitung der `FOREIGN KEY`-Klauseln in Abhängigkeit vom Inhalt der Tabellen potentiell unterschiedliche Ergebnisse resultieren.

```
CREATE TABLE T1 (... PRIMARY KEY K1)
CREATE TABLE T2 ( ... PRIMARY KEY K2
     FOREIGN KEY (K1) REFERENCES T1 (K1)
          ON DELETE CASCADE)
CREATE TABLE T3 (... PRIMARY KEY K3
     FOREIGN KEY (K1) REFERENCES T1 (K1)
          ON DELETE CASCADE)
CREATE TABLE T4 (... PRIMARY KEY K4
     FOREIGN KEY (K2) REFERENCES T2 (K2)
          ON DELETE CASCADE
     FOREIGN KEY (K3) REFERENCES T3 (K3)
          ON DELETE RESTRICT)
```

Das Beispiel `DELETE FROM T1 WHERE K1 = 1` demonstriert, dass bzgl. `T4` die `RESTRICT`-Aktion scheitert, sofern nicht vorher bzgl. `T4` die `CASCADE`-Aktion durchgeführt wurde.

T1	K1
	1

T2	K2	K1
	a	1

T3	K3	K1
	b	1

T4	K4	K2	K3
	c	a	b

zum potentiellen Nichtdeterminismus

- Um nichtdeterministische Ausführungen dieser Art auszuschließen, wird vorgeschlagen, die Implementierung nach einer Strategie vorzunehmen, in der im Wesentlichen vor Berücksichtigung einer `RESTRICT`-Aktion alle `CASCADE`-Aktionen ausgeführt werden.
- Diese Strategie klärt offensichtlich obige Unbestimmtheit.
- Alternativ können gewisse Kombinationen von referentiellen Aktionen verboten werden. Ersetzt man `RESTRICT` durch `NO ACTION` in obigem Beispiel, so wird das Endergebnis wieder eindeutig, unabhängig von der Reihenfolge der betrachteteten referentiellen Aktionen.

4.14.2 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 (s. Beispiel Seite 84).

```
/* 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 ATOMIC 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 ATOMIC 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'
```