# 4.9 SQL und Programmiersprachen

- SQL hat eine tabellenorientierte Semantik.
- SQL hat eine im Vergleich zu einer Programmiersprache eingeschränkte Mächtigkeit, so dass es typischerweise ermöglicht wird, SQL von einem in einer gängigen Programmiersprache geschriebenen Programm aus aufzurufen.
- Anwendungen können so unter Ausnutzung der vollen Mächtigkeit einer Programmiersprache Datenbank-Instanzen verarbeiten.
- *Impedance-Mismatch*

## Ansätze der Integration

(A) **SQL-Erweiterung**:
Erweiterung von SQL um imperative Sprachelemente zur Formulierung von *in der Datenbank gespeicherten* benutzerdefinierten Funktionen und Prozeduren.

(B) **Statisches SQL**:
Einbettung von SQL-Ausdrücken in eine Programmiersprache.

(C) **Dynamisches SQL**:
Übergabe eines datenabhängig gebildeten SQL-Ausdrucks an eine Datenbank während der Ausführung eines Programms.

## (A) SQL-Erweiterung

- *Deklaration* von *Variablen*
- *Zuweisung* von Werten an Variable
- *Sequenz* von Anweisungen
- *bedingte Anweisungen* und *Wiederholungsanweisungen*
- Funktionen und Prozeduren.[1]

Funktionen können als parametrisierbare virtuelle Sichten betrachtet werden.

*Hinweis*: Oracle verwendet teilweise eine andere Syntax als der SQL-Standard!

[1] In Oracle sind keine tabellenwertigen Parameter erlaubt.

# Funktionen als parametrisierbare virtuelle Sichten

**Benachbart**

| LCode1 | LCode2 |
|---|---|
| CH | D |
| CH | F |
| CH | I |
| D | F |
| I | F |
| ⋮ | ⋮ |

**nachbarVon('D')**

| LCode |
|---|
| CH |
| F |
| ⋮ |

Berechne die benachbarten Länder zu einem gegebenen Land.

```
CREATE FUNCTION nachbarVon(X CHAR(4))
RETURNS TABLE ( LCode CHAR(4) )
RETURN (
      SELECT LCode2 AS LCODE FROM Benachbart WHERE LCODE1 = X
      UNION
      SELECT LCode1 AS LCode FROM Benachbart WHERE LCODE2 = X )

SELECT * FROM TABLE (nachbarVon('D')) T
```

*Hinweis*: Nicht direkt umsetzbar in Oracle, da Funktionen in Oracle keine Tabellen zurückgeben können!

# SQL-Standard vs. Oracle Syntax

## Berechne die Anzahl Städte zu einem gegebenen Land. (SQL-Standard)

```
CREATE FUNCTION anzahlStaedte(Land CHAR(2))
RETURNS NUMBER
RETURN (
      SELECT count(*) FROM Stadt WHERE LCode = Land
)
```

## Berechne die Anzahl Städte zu einem gegebenen Land. (Oracle)

```
CREATE FUNCTION anzahlStaedte(Land CHAR)
RETURN NUMBER IS
numCities NUMBER;
BEGIN
      SELECT count(*) INTO numCities FROM Stadt WHERE LCode = Land;
      RETURN numCities;
END;
```

## Aufrufen der Funktion mittles Dummy Tabelle

```
SELECT anzahlStaedte('D') FROM Dual
```

Gib zu einem Land alle erreichbaren Länder mit der Mindestanzahl Grenzübergänge an (SQL-Standard; Oracle Version in den Übungen).

```
CREATE FUNCTION ErreichbarVon(Start CHAR(4))
RETURNS TABLE ( Nach CHAR(4), Anzahl INTEGER )
BEGIN
CREATE TABLE Erreichbar ( Nach CHAR(4), Anzahl INTEGER );
DECLARE alt, neu INTEGER;

/* Initialisiere mit den direkt erreichbaren Ländern */
INSERT INTO Erreichbar
      SELECT T.LCode2 AS Nach, 1 AS Anzahl
      FROM symBenachbart T WHERE T.LCode1 = Start;

/* Initialisiere Abbruchbedingung */
SET alt = 0;
SET neu = ( SELECT COUNT(*) FROM Erreichbar );

/* Berechne iterativ die indirekt erreichbaren Länder */
WHILE (alt <> neu) DO
      SET alt = neu;
      INSERT INTO Erreichbar
            SELECT DISTINCT B.LCode2, (A.Anzahl + 1)
            FROM Erreichbar A, symBenachbart B
            WHERE A.Nach = B.LCode1
                  AND B.LCode2 <> Start
                  AND B.LCode2 NOT IN ( SELECT Nach FROM Erreichbar );
      SET neu = ( SELECT COUNT(*) FROM Erreichbar );
END WHILE;

RETURN Erreichbar;
END
```

Initialisiere `Erreichbar` mit den *direkt* erreichbaren Ländern.

```
INSERT INTO Erreichbar
      SELECT T.LCode2 AS Nach, 1 AS Anzahl
      FROM symBenachbart T WHERE T.LCode1 = Start;
```

Die Berechnung soll terminieren, wenn nach einer Iteration kein neues *indirekt* erreichbares Land hinzugekommen ist.

```
SET neu = ( SELECT COUNT(*) FROM Erreichbar );
WHILE (alt <> neu) DO
      SET alt = neu;
      INSERT INTO Erreichbar
      ...
      SET neu = ( SELECT COUNT(*) FROM Erreichbar )
END WHILE;
```

Zyklen und Mehrfachberechnungen sollen vermieden werden.

- `DISTINCT` berücksichtigt den Fall, dass zum selben Land mehrere Wege gleicher Länge existieren können.
- Zyklen werden berücksichtigt, indem ein weiteres Land nur dann hinzugefügt wird, wenn das neu berechnete Land nicht bereits als erreichbares Land bekannt ist.
- Zyklen zum Start werden ebenfalls ausgeschlossen.

```
INSERT INTO Erreichbar
    SELECT DISTINCT B.LCode2, (A.Anzahl + 1)
    FROM Erreichbar A, symBenachbart B
    WHERE A.Nach = B.LCode1
        AND B.LCode2 <> Start
        AND B.LCode2 NOT IN ( SELECT Nach FROM Erreichbar );
```

## (B) Statisches SQL

- Stehen die auszuführenden SQL-Anfragen bereits zur Übersetzungszeit eines Programms als Teil des Programmcodes fest, dann redet man von einer *statischen* Einbettung von SQL in eine Programmiersprache.
- Eine datenabhängige Änderung der Anfragen während der Ausführung des Programms ist dann nicht mehr möglich.
- Die Ergebnisse einer SQL-Anfrage werden innerhalb des Programms mittels eines Cursors zugänglich gemacht.

### Anlegen eines Cursors

```
EXEC SQL DECLARE StadtCursor CURSOR FOR
   SELECT DISTINCT S.SName, L.LName
      FROM Stadt S, Land L
      WHERE S.LCode = L.LCode;
```

### Öffnen einen Cursors

```
EXEC SQL OPEN StadtCursor;
```

### Aktuelle Zeile eines Cursors auslesen

```
EXEC SQL FETCH StadtCursor INTO :stadtName, :landName;
```

### Cursor schließen

```
EXEC SQL CLOSE StadtCursor;
```

# (C) Dynamisches SQL

- Können wir einen SQL-Ausdruck während der Ausführung eines Programms in Form einer Zeichenkette an das Datenbanksystem übergeben, so redet man von *dynamischem* SQL.
- Standardisierte Schnittstellen: ODBC und JDBC.
  Erweiterung von Java: SQLJ.

## Beispiel: Berechnen einer Adjazenzmatrix in PHP

```
<?php
sql_connect('Mondial','lausen','buch');
$AdjazenzMatrix = array();
$query = 'SELECT * FROM Benachbart';
$result = sql_query($query);
while ($row = sql_fetch_assoc($result)) {
      $i = $row['von']; $j = $row['nach'];
      $AdjazenzMatrix[$i][$j] = 1; }
      ...
?>
```

# 4.10 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.

## 4.10.1 Schlüsselbedingungen

Zu jeder Tabelle werden typischerweise ein *Primärschlüssel* und möglicherweise weitere Schlüssel festgelegt (`UNIQUE`-Klausel).

In jeder Instanz zu der Tabelle `Land` können alle Zeilen eindeutig durch ihren Spaltenwert zu `LCode`, oder alternativ, durch ihren Spaltenwert zu `LName` identifiziert werden.

```
CREATE TABLE Land (
    LName       VARCHAR(35) UNIQUE,
    LCode       VARCHAR(4) PRIMARY KEY,
    ...
```

Identifizierendes Kriterium für die Tabelle `Stadt` sei `SName`, `LCode`, `PName`, bzw. alternativ, `LGrad`, `BGrad`.

```
CREATE TABLE Stadt (
    ...
    PRIMARY KEY (SName,LCode,PName),
    UNIQUE (LGrad,BGrad))
```

# 4.10.2 Statische Integrität

## CHECK-Klausel

- *Statische* Integrität definiert unter Verwendung der `CHECK`-Klausel, welche Instanzen eines Schemas zulässig sind.
- Mittels der `CHECK`-Klausel können die zulässigen Werte eines Datentyps und die für eine Spalte einer konkreten Tabelle zu verwendenden Werte weiter eingeschränkt werden.
- Darüberhinaus können beliebige, mittels SQL-Anfrageausdrücken gebildete, Bedingungen über den Instanzen der Tabellen eines Schemas ausgedrückt werden.

## Wertebereichsbedingungen mittels NONNUll, DEFAULT und CREATE DOMAIN

```
LName VARCHAR(35) NONNULL
Prozent NUMBER DEFAULT 100
```

```
CREATE DOMAIN meineStädte VARCHAR(35) DEFAULT '?'

CREATE TABLE Stadt (
   SName meineStädte,
   ⋮
```

## Wertebereichsbedingungen mittels CHECK

```
CREATE DOMAIN meineStädte VARCHAR(35),
   DEFAULT 'Paris',
   CHECK (VALUE IN ('Berlin', 'Paris',
                    'London', 'Rom'))
```

## Spaltenbedingung mittels CHECK

```
CREATE TABLE Stadt (
   SName        meineStädte,
   ⋮
   LGrad        NUMBER
                CHECK (LGrad BETWEEN -180 AND 180),
   BGrad        NUMBER
                CHECK (BGrad BETWEEN -90 AND 90),
   ...)
```

## Spalten- und Tabellenbedingung mittels CHECK

Die Summe aller Anteile an unterschiedlichen Kontinenten eines Landes muss 100 ergeben.

```
CREATE TABLE Lage (
   LCode        VARCHAR(4),
   Kontinent    VARCHAR(35),
   Prozent      NUMBER
                CHECK (Prozent BETWEEN 0 AND 100),
   CHECK (100 = (SELECT SUM(L.Prozent) FROM Lage L
                   WHERE LCode = L.LCode)))
```

## Assertion

- Spalten- und Tabellenbedingungen sind erfüllt, wenn jede Zeile der betreffenden Tabelle sie erfüllt.
- Spalten- und Tabellenbedingungen sind somit *implizit* $\forall$-quantifiziert über den Zeilen der Tabelle.
- Alternativ können wir die explizitere Form einer `ASSERTION` wählen

`CHECK` Assertion:
Die Summe aller Anteile an unterschiedlichen Kontinenten eines Landes muss 100 ergeben.

```
CREATE ASSERTION AssertLage (
   CHECK ( NOT EXISTS (
      SELECT LCode FROM Lage
      GROUP BY LCode
      HAVING (SUM(Prozent) <> 100) )
   )
)
```

## CHECK und ASSERTION in ORACLE, SQL Server, etc.

- Typischerweise sind Sub-Queries in der CHECK-Klausel verboten.
- ASSERTION wird nicht unterstützt.

**Gewährleistung von Integritätsbedingungen in solchen Fällen mittels TRIGGER**

# 4.10.3 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) )
```

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

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. (`CASCADE`)
- Die Änderung der P-Tabelle wird im Falle einer Verletzung der referentiellen Integrität einer C-Tabelle abgebrochen. (`NO ACTION` oder `RESTRICT`)
- Der Fremdschlüsselwert der C-Tabelle wird angepaßt. (`SET NULL` oder `SET DEFAULT`)

*Hinweis*: Oracle unterstützt bei UPDATE-Operationen nur die Aktion NO ACTION und bei DELETE-Operationen die Aktionen CASCADE, SET NULL und NO ACTION (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 entsprechenden P-Tabellen betrachtet?

- Einfügen bezüglich der P-Tabelle ist für die referentielle Integrität immer unkritisch.
- 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 im Überblick

**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 betrachteten referentiellen Aktionen.

# 4.10.4 Dynamische Integrität und Trigger

- Komplementär zu deklarativen Integritätsbedingungen 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.
- Damit können auch die Einschränkungen kommerzieller DBS bei der Verwendung der `CHECK`-Klausel umgangen werden.
- Mittels Trigger können wir insbesondere die Zulässigkeit von *Zustandsübergängen* kontrollieren, was mit deklarativen Integritätsbedingungen bisher aufgrund ihres Bezugs zu gerade einem Zustand nicht möglich ist.

- *Dynamische Integrität* beschäftigt sich somit 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 von 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 Fläche einer Provinz, dann soll auch das entsprechende Land angepaßt werden. (SQL-Standard)

```
CREATE TRIGGER FlaecheAnpassen
   AFTER UPDATE OF Flaeche ON Provinz
   REFERENCING OLD AS Alt NEW AS Neu
   FOR EACH ROW
      UPDATE Land L
      SET L.Flaeche = L.Flaeche - Alt.Flaeche + Neu.Flaeche
      WHERE L.LCode = Alt.LCode
```

(Oracle)

```
CREATE TRIGGER FlaecheAnpassen
   AFTER UPDATE OF Flaeche ON Provinz
   REFERENCING OLD AS Alt NEW AS Neu
   FOR EACH ROW
   BEGIN
      UPDATE Land L
      SET L.Flaeche = L.Flaeche - :Alt.Flaeche + :Neu.Flaeche
      WHERE L.LCode = :Alt.LCode;
   END;
```

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
```

*Hinweis*: Oracle bietet eine Funktion zum Erzeugen eines Laufzeitfehlers:

`raise_application_error(error_number, message)`

Die Fehlernummer muss eine negative Nummer kleiner -20000 sein, z.B.:

`raise_application_error(-20001, 'Grenze bereits vorhanden')`

*Hinweis*: `WHEN` darf kein SFW enthalten.

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 LCode, 'Atlantis' AS Kontinent,
            (100 - SUM(Prozent)) AS Prozent
         FROM Lage
         GROUP BY LCode
         HAVING (SUM(Prozent) < 100)
   END
```

*Hinweis*: Statement Trigger sind der Standardfall in Oracle. Die Angabe von `FOR EACH STATEMENT` entfällt. Weitere Einschränkungen in Oracle.

# Eigenschaften von Triggern

- 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.
- Ein aktivierter Trigger wird ausgeführt, wenn seine Bedingung erfüllt ist.
- Der Rumpf eines Triggers enthält die auszuführenden SQL-Anweisungen.
- Mittels Transitions-Variablen `OLD/NEW` bzw. `OLD/NEW TABLE`[2] kann auf die Zeilen- bzw. 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.
- Bei einem `BEFORE`-Trigger sind die *einzufügenden* Tupel nicht sichtbar in der betreffenden Tabelle; bei einem `AFTER`-Trigger sind sie sichtbar.
- Bei einem `BEFORE`-Trigger sind die zu *löschenden* Tupel sichtbar in der betreffenden Tabelle; bei einem `AFTER`-Trigger sind sie nicht sichtbar.

[2] `OLD/NEW TABLE` wird von Oracle nicht unterstützt.

## 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 Trigger-Definitionen, bzw. Anforderungen an ihre Ausführung zu berücksichtigen.
- Eine Aktivierungsfolge von Triggern kann insbesondere 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.