Aufgabe 3.1:

Lösung:

Wir beweisen durch Induktion über die Anzahl Operatoren $\pi, \sigma, \bowtie, \cup, -, \delta$ eines Ausdrucks der Relationenalgebra, dass zu jeder Anfrage der Relationenalgebra eine äquivalente Anfrage in SQL existiert.

Induktionsbasis: Sei Q_{RA} eine Anfrage der Relationenalgebra ohne Operator. Q_{RA} hat dann die Form R, wobei R ein Relationsbezeichner. Sei r die betreffende Instanz zu R. Q_{RA} hat dann die Antwort r und die äquivalente SQL-Anfrage Q_{SQL} hat die Form:

```
SELECT *
FROM R
```

Induktionsschluss: Sei Q_{RA} eine Anfrage der Relationenalgebra mit $n > 0$ Operatoren. Q_{RA} ist dann von einer der folgenden Formen, wobei Q, Q' Ausdrücke mit höchstens $n - 1$ Operatoren:

$$\pi[X]Q,\ \sigma[\alpha]Q,\ Q \bowtie Q',\ Q \cup Q',\ Q - Q' \text{ und } \delta[X, Y]Q.$$

Seien Q_{SQL}, bzw. Q'_{SQL} die äquivalenten SQL-Anfragen zu Q, bzw. Q'. Die durch Q_{SQL}, bzw. Q'_{SQL} definierten Tabellen seien TQ, bzw. TQ', wobei TQ Spalten $A_1, \ldots, A_n$, $n \geq 1$ besitzt und TQ' Spalten $B_1, \ldots, B_m$, $m \geq 1$.

Die äquivalente SQL-Anfrage zu Q_{RA} ergibt sich jetzt in Abhängigkeit von der Form von Q_{RA} wie folgt:

$\pi[X]Q$, $X = \{A'_1, \ldots, A'_{n'}\}$:

```
WITH
TQ(A_1, ..., A_n) AS (
Q_SQL )
SELECT A'_1, ..., A'_n'
FROM TQ
```

$\sigma[\alpha]Q$:

```
WITH
TQ(A_1, ..., A_n) AS (
Q_SQL )
SELECT *
FROM TQ WHERE α
```

$Q \bowtie Q'$:

```
WITH
TQ(A1,...,An) AS (
QSQL ),
TQ'(B1,...,Bm) AS (
Q'SQL )
SELECT *
FROM TQ NATURAL JOIN TQ'
```

$Q \cup Q'$, $\{A_1, \ldots, A_n\} = \{B_1, \ldots, B_m\}$:

```
WITH
TQ(A1,...,An) AS (
QSQL ),
TQ'(B1,...,Bm) AS (
Q'SQL )
SELECT * FROM TQ
UNION
SELECT * FROM TQ'
```

$Q - Q'$, $\{A_1, \ldots, A_n\} = \{B_1, \ldots, B_m\}$:

```
WITH
TQ(A1,...,An) AS (
QSQL ),
TQ'(B1,...,Bm) AS (
Q'SQL )
SELECT * FROM TQ
EXCEPT
SELECT * FROM TQ'
```

$\delta[X, Y]Q$, $X = \{A_1, \ldots, A_n\}$, $Y = \{B_1, \ldots, B_n\}$, $\delta(X_i) = Y_i$, $1 \leq i \leq n$:

```
WITH
TQ(A1,...,An) AS (
QSQL )
SELECT A1 AS B1, ..., An AS Bn
FROM TQ
```

Aufgabe 3.2: [2]

(a)

Lösung:

Wenn in Anfrage (3.33) und Anfrage (3.34) keine Mitgliedschaften zu Österreich existieren, d.h. die Anfrage

```
SELECT *
FROM Mitglied
WHERE M2.LCode = 'A'
```

liefert eine leere Tabelle, dann liefern Anfrage (3.33) und Anfrage (3.34) gerade eine Tabelle mit dem Inhalt der gesamten `LCode`-Spalte der Tabelle `Mitglied`.

(b)

Lösung:

Die Anfrage berechnet ebenfalls eine relationale Division. Existieren keine Mitgliedschaften zu Österreich, dann liefert die Anfrage eine leere Tabelle.

[2] Mehr zur Division findet man in (Celko, 2000, Chapter 19).

Aufgabe 3.3:

(a)

Lösung:

1) Die in der `Mondial`-Datenbank bestehenden Beziehungen zwischen Kontinenten, Ländern, Provinzen und Städten werden in der Sicht durch einzelne Zeilen dargestellt. Es werden nur solche Beziehungen aufgeführt, die ausgehend von `Lage`, d.h. einem Kontinent, ohne Lücken rekonstruiert werden können. Kann eine Beziehung nicht weiter verfolgt werden, beispielsweise die Beziehung von einem Kontinent zu einem Land, weil zu diesem Land keine Provinz existiert, so werden die Angaben zu Provinz und Stadt durch Nullwerte repräsentiert.
2) Es werden alle möglichen Aggregierungen über Kontinenten, Ländern, Provinzen und Städten gebildet und jeweils die Anzahl Zeilen in der entsprechenden Gruppe bestimmt.
3) Die Verwendung der `GROUPING`-Spalten erlaubt eine Unterscheidung, ob ein Nullwert aufgrund einer Verdichtung oder aufgrund fehlender Beziehungen in einer Zeile auftritt. Beispielsweise können wir mit der folgenden Anfrage gerade alle Zeilen bestimmen, die mindestens einen Nullwert enthalten und keiner Aggregierung entstammen.

```
SELECT Kontinent, LCode, PName, SName, Num
FROM myCube
WHERE (KontTest + LandTest + ProvTest + StadtTest = 0)
AND (Kontinent IS NULL OR LCode IS NULL OR
       PName IS NULL OR SName IS NULL)
```

Hinweis: Für die folgenden Teile (b) – (d) soll `myCube` wie eine materialisierte Sicht betrachtet werden, bzgl. der die Anfragen formuliert werden sollen. Der direkte Zugriff zu den Basisrelationen der Datenbank `Mondial` ist somit nicht im Sinne der Aufgabenstellung.

(b)

Lösung:

```
SELECT Kontinent, count(DISTINCT LCode) AS AnzLänder
FROM myCube
WHERE Kontinent IS NOT NULL
GROUP BY Kontinent
```

(c)

Lösung:

```
SELECT Distinct C.Kontinent,
   (select count(DISTINCT LCode)
    FROM myCube CC
   WHERE C.Kontinent = CC.Kontinent) AS Länder
FROM myCube C
WHERE C.Kontinent IS NOT NULL
```

(d)

Lösung:

```
SELECT PName, Num
FROM myCube
WHERE PName IS NOT NULL AND Konttest = 0 AND
   LandTest = 0 AND ProvTest = 0 AND StadtTest = 1
AND Num >= ALL
( SELECT Num
FROM myCube
WHERE PName IS NOT NULL AND Konttest = 0 AND
    LandTest = 0 AND ProvTest = 0 AND StadtTest = 1 )
```

Aufgabe 3.4:

(a)

Lösung:

```
CREATE TABLE Land (
      ⋮
      CHECK ( (
      SELECT SUM(S.Einwohner) FROM Stadt S
      WHERE S.LCode = Land.LCode ) <
      1000 * Land.Fläche )
```

(b)

Lösung:

```
CREATE ASSERTION AssertLand2 (
      CHECK ( NOT EXISTS (
            SELECT * FROM Land L
            WHERE ( SELECT SUM(S.Einwohner) FROM Stadt S
                  WHERE S.LCode = L.LCode ) >=
                  1000 * L.Fläche ) )
```

Aufgabe 3.5:

(a)

Lösung:

```
CREATE TABLE Land (
      ⋮
      CHECK ( (
      SELECT S.Einwohner FROM Stadt S
      WHERE S.SName = Land.HStadt AND
            S.LCode = Land.LCode) > (
      SELECT MAX(Einwohner) FROM Stadt T
      WHERE T.LCode = Land.LCode AND T.SName <> S.SName ) )
```

(b)

Lösung:

```
CREATE ASSERTION AssertLand3 (
CHECK ( NOT EXISTS (
      SELECT S.Einwohner FROM Stadt S, Land L
      WHERE S.SName = L.HStadt AND
            S.LCode = L.LCode AND S.Einwohner <= (
            SELECT MAX(Einwohner) FROM Stadt T
            WHERE T.LCode = L.LCode AND T.SName <> S.SName ) ) ) )
```

(c)

Lösung: (nur für Updates)

```
CREATE TRIGGER EinwohnerzahlStadtTesten
BEFORE UPDATE OF Einwohner ON Stadt
REFERENCING OLD AS Alt NEW AS Neu
FOR EACH ROW
WHEN ( (
      Neu.SName <> (
      SELECT HStadt FROM Land L
      WHERE L.LCode = Neu.LCode ) ) AND (
      Alt.Einwohner < Neu.Einwohner ) AND ( (
      SELECT Einwohner FROM Stadt S
      WHERE S.SName = ( SELECT HStadt FROM Land L
            WHERE L.LCode = S.LCode ) ) <
      Neu.Einwohner ) )
BEGIN
      SIGNAL SQLSTATE '75001';
      SET Message='Ungültige Erhöhung einer Einwohnerzahl'
END

CREATE TRIGGER EinwohnerzahlHauptstadtTesten
BEFORE UPDATE OF Einwohner ON Stadt
REFERENCING OLD AS Alt NEW AS Neu
FOR EACH ROW
WHEN ( (
      Neu.SName = (
      SELECT HStadt FROM Land L
      WHERE L.LCode = Neu.LCode ) ) AND (
      Alt.Einwohner > Neu.Einwohner) AND ( (
      SELECT MAX(Einwohner) FROM Stadt S
      WHERE S.LCode = Neu.LCode ) >
      Neu.Einwohner ) )
BEGIN
      SIGNAL SQLSTATE '75001';
      SET Message='Ungültige Verringerung der
            Einwohnerzahl der Hauptstadt'
END
```

Aufgabe 3.6: *Für die Wurzel des Baumes ist Elter gleich dem Nullwert.*

(a)

Lösung:

```
CREATE FUNCTION Kinder(k INTEGER)
RETURNS TABLE (
      Kind INTEGER,
      Elter INTEGER )
RETURN (
      SELECT * FROM KBaum
      WHERE Elter = k )
```

(b)

Lösung:

```
CREATE FUNCTION Elter(k INTEGER)
RETURNS INTEGER
RETURN (
      SELECT Elter FROM KBaum
      WHERE Kind = k )
```

(c)

Lösung:

mittels Rekursion:

```
CREATE FUNCTION Nachfahren(k INTEGER)
RETURNS TABLE (
      Knoten INTEGER,
      Distanz INTEGER )
RETURN
WITH RECURSIVE Nach(Knoten, Distanz) AS (
      SELECT Kind AS Knoten, 1 AS Distanz
      FROM KBaum WHERE Elter = k
      UNION ALL
      SELECT TT.Kind AS Knoten, T.Distanz + 1 AS Distanz
      FROM Nach T, KBaum T WHERE T.Knoten = TT.Elter )
SELECT * FROM Nach
```

mittels Iteration:

```
CREATE FUNCTION Nachfahren(k INTEGER)
RETURNS TABLE (
      Knoten INTEGER,
      Distanz INTEGER )
BEGIN
CREATE TABLE tmp (
      Knoten INTEGER,
      Distanz INTEGER )
DECLARE alt, neu, Tiefe INTEGER;
INSERT INTO Temp
      SELECT Kind, '1' from KBaum WHERE Elter = k;
SET alt=0;
SET neu = (SELECT COUNT(*) FROM Temp);
SET Tiefe = 1;
WHILE (alt <> neu) DO
      SET alt = neu;
      INSERT INTO Temp
            SELECT Kind, (Tiefe + 1) FROM KBaum K
            WHERE K.Elter IN (
                  SELECT * FROM Temp
                  WHERE Distanz = Tiefe )
      SET neu = (SELECT COUNT(*) FROM Temp);
      SET Tiefe = Tiefe + 1;
END WHILE ;
RETURN Temp;
END
```

(d)

Lösung:

mittels Rekursion:

```
CREATE FUNCTION Vorfahren(k INTEGER)
RETURNS TABLE (
      Knoten INTEGER,
      Distanz INTEGER )
RETURN
WITH RECURSIVE Vor(Knoten, Distanz) AS (
      SELECT Elter AS Knoten, 1 AS Distanz
      FROM KBaum WHERE Kind = k
      UNION ALL
      SELECT TT.Elter AS Knoten, T.Distanz + 1 AS Distanz
      FROM Nach T, KBaum T WHERE T.Knoten = TT.Kind )
SELECT * FROM Nach
```

mittels Iteration:

```
CREATE FUNCTION Vorfahren(k INTEGER)
RETURNS TABLE (
      Knoten INTEGER,
      Distanz INTEGER )
BEGIN
CREATE TABLE Temp (
      Knoten INTEGER,
      Distanz INTEGER );
DECLARE alt, neu, Tiefe INTEGER;
INSERT INTO Temp
      SELECT Elter, 1 FROM KBaum WHERE Kind = k;
SET alt=0;
SET neu = (SELECT COUNT(*) FROM Temp);
SET Tiefe = 1;
WHILE (alt <> neu) DO
      SET alt = neu;
      INSERT INTO Temp
            SELECT Elter, (Tiefe + 1) FROM KBaum K
            WHERE K.Kind IN (
                  SELECT * FROM Temp
                  WHERE Distanz = Tiefe )
      SET neu = (SELECT COUNT(*) FROM Temp);
      SET Tiefe = Tiefe + 1;
END WHILE;
RETURN tmp;
END
```

(e)

Lösung:

```
CREATE FUNCTION Delete(k INTEGER)
RETURNS BOOLEAN
BEGIN
IF EXISTS (SELECT * FROM KBAUM WHERE Kind = k)
BEGIN
      DELETE FROM KBaum
      WHERE ((Kind, Elter) IN Nachfahren(k)) OR
                  Kind = k;
RETURN TRUE;
ENDIF
ELSE RETURN FALSE
END
```

(f)

Lösung:

```
CREATE FUNCTION Insert(k INTEGER, kk INTEGER)
RETURNS BOOLEAN
BEGIN
IF EXISTS((SELECT * FROM KBAUM WHERE Kind = k)
BEGIN
      INSERT INTO KBaum VALUES (kk, k);
      RETURN TRUE;
ENDIF
ELSE RETURN FALSE;
END
```

Aufgabe 3.7: [3]

Für die Wurzel des Baumes ist Elter gleich dem Nullwert. (a)

Lösung:

```
CREATE FUNCTION Kinder(k INTEGER)
RETURNS TABLE (
      Kind INTEGER,
      Elter INTEGER )
RETURN (
      SELECT * FROM IBaum
      WHERE Elter = k )
```

(b)

Lösung:

```
CREATE FUNCTION Elter(k INTEGER)
RETURNS INTEGER
RETURN
      SELECT Elter FROM IBaum
      WHERE Kind = k )
```

[3] Vergleiche mit (Celko 2000, Chapter 29).

(c)

Lösung:

```
CREATE FUNCTION Nachfahren(k INTEGER)
RETURNS TABLE (
      Knoten INTEGER,
      Distanz INTEGER )
BEGIN
CREATE TABLE tmp (
      Knoten INTEGER,
      Distanz INTEGER )
DECLARE kLevel, l, r INTEGER;
SET l = (SELECT links FROM IBaum WHERE Kind = k);
SET r = (SELECT rechts FROM IBaum WHERE Kind = k);
SET kLevel = ( SELECT COUNT(*) FROM IBaum T
            WHERE l BETWEEN T.links AND T.rechts );
INSERT INTO Temp
      SELECT T1.Kind AS Knoten, (
            SELECT COUNT(*) FROM IBaum T2
            WHERE T1.links BETWEEN T2.links AND T2.rechts ) -
            kLevel AS Distanz
      FROM (
            SELECT * FROM IBaum T2
            WHERE T2.links BETWEEN l AND r ) T1
RETURN Temp;
END
```

(d)

Lösung:

```
CREATE FUNCTION Vorfahren(k INTEGER)
RETURNS TABLE (
      Knoten INTEGER,
      Distanz INTEGER )
BEGIN
CREATE TABLE tmp (
      Knoten INTEGER,
      Distanz INTEGER )
DECLARE kLevel, l INTEGER;
SET l = (SELECT links FROM IBaum WHERE Kind = k);
SET kLevel = ( SELECT COUNT(*) FROM IBaum T
            WHERE l BETWEEN T.links AND T.rechts );
INSERT INTO Temp
      SELECT T1.Kind AS Knoten, kLevel - (
            SELECT COUNT(*) FROM IBaum T2
            WHERE T1.links BETWEEN T2.links AND T2.rechts )
            AS Distanz
      FROM (
            SELECT * FROM IBaum T2
            WHERE l BETWEEN T2.links AND T2.rechts ) T1
RETURN Temp;
END
```

(e)

Lösung:

```
CREATE FUNCTION Delete(k INTEGER)
RETURNS BOOLEAN
BEGIN
IF EXISTS (SELECT * FROM IBAUM WHERE Kind = k)
BEGIN
      DELETE FROM IBaum
      WHERE (Kind IN SELECT Knoten FROM Nachfahren(k)) OR
                  Kind = k;
RETURN TRUE;
ENDIF
ELSE RETURN FALSE
END
```

(f)

Lösung:

Wir fügen den neuen Kindknoten k' so ein, dass k' der am weitesten links stehende Kindknoten von k wird.

```
CREATE FUNCTION Insert(k INTEGER, kk INTEGER)
RETURNS BOOLEAN
BEGIN
DECLARE l INTEGER;
IF EXISTS(SELECT * FROM IBAUM WHERE Kind = k)
BEGIN
      SET l = (SELECT links FROM IBaum WHERE Kind = k);
      UPDATE IBaum
            SET links = links + 2
            WHERE links > l;
      UPDATE IBaum
            SET rechts = rechts + 2
            WHERE rechts >= l+1;
      INSERT INTO IBaum VALUES (kk, k, l+1, l+2);
      RETURN TRUE;
ENDIF
ELSE RETURN FALSE;
END
```

Aufgabe 3.8: *(nicht im Buch)*
Streiche in der bisherigen Tabelle `IBaum` *aus Aufgabe 3.7 die* `Elter`*-Spalte.*
Gib Anfragen in SQL an, die zu einem Knoten seinen Elterknoten, bzw. alle seine Kinder bestimmen.

Lösung:

Kinder(*k*) liefert die Tabelle aller Kindknoten von *k*.

```
CREATE FUNCTION Kinder(k INTEGER)
RETURNS TABLE (Knoten INTEGER)
RETURN
SELECT Kind FROM IBaum T1
WHERE (
      T1.links > (SELECT T2.links FROM IBaum T2 WHERE T2.Kind = k) AND
      T1.links < (SELECT T2.rechts FROM IBaum T2 WHERE T2.Kind = k) ) AND
      NOT EXISTS (
            SELECT * FROM IBaum T3
            WHERE T1.links > T3.links AND T1.links < T3.rechts AND
                  T3.links > (SELECT T2.links FROM IBaum T2 WHERE T2.Kind = @k) )
```

Elter(*k*) liefert den Elterknoten zu *k*; ist *k* Wurzel, so ist das Resultat der Nullwert.

```
CREATE FUNCTION Elter(k INTEGER)
RETURNS INTEGER
RETURN
SELECT Kind FROM IBaum T1
WHERE T1.links = (
      SELECT Max(T2.links) FROM IBaum T2
      WHERE T2.links BETWEEN
            (SELECT T3.links FROM IBaum T3 WHERE T3.Kind = k) AND
            (SELECT T3.rechts FROM IBaum T3 WHERE T3.Kind = k) )
```