2 SCHEMA-DEFINITION

Das *Datenbankschema* umfasst alle Informationen *über* die Datenbank (d.h., alles *außer* den eigentlichen Daten) und wird mit Hilfe der *Data Definition Language (DDL)* manipuliert.

Die DDL-Kommandos in SQL umfassen die Befehle `CREATE` / `ALTER` / `DROP` für das Anlegen, Ändern und Entfernen von Schemaobjekten. Relationen können als *Tabellen* (vgl. Abschnitt 2.1) *extensional* gespeichert werden, oder als *Views (Sichten)* (vgl. Abschnitt 2.2) *intensional* definiert sein.

Weiterhin enthält das Datenbankschema verschiedene Arten von Bedingungen, u. a. Wertebereichbedingungen einzelner Attribute, Bedingungen innerhalb einer Tabelle (intrarelational), Schlüsselbedingungen, sowie Bedingungen die mehrere Tabellen betreffen (interrelational).

Ferner werden in Abschnitt 13 die Konzepte `INDEX` und `CLUSTER` zur optimierten Speicherung von Tabellen, sowie in Abschnitt 14 `PROZEDUREN`, `FUNKTIONEN` und `TRIGGER` eingeführt. Zur Schemadefinition gehört außerdem noch die Vergabe von Zugriffsrechten durch `GRANT`-Befehle (vgl. Abschnitt 10) sowie die Definition von `SYNONYM`en (ebenfalls Abschnitt 10).

2.1 Definition von Tabellen

In der einfachsten Form besteht eine Tabellendefinition nur aus der Angabe der Attribute sowie ihres Wertebereichs:

```
CREATE TABLE <table>
  (<col> <datatype>,
   ⋮
   <col> <datatype>);
```

Die Bezeichnungen der erlaubten Datentypen können dabei abhängig von dem verwendeten System unterschiedlich sein. ORACLE erlaubt unter anderem

`CHAR`(n): Zeichenkette fester Länge n. Kürzere Zeichenketten werden ggf. aufgefüllt.

`VARCHAR`(n): Zeichenkette variabler Länge $\leq n$. Jede Zeichenkette benötigt nur soviel Platz, wie sie lang ist. Auf `CHAR` und `VARCHAR` ist der Operator || zur Konkatenation von Strings definiert. In ORACLE ist `VARCHAR2` eine verbesserte Implementierung des Datentyps `VARCHAR` früherer Versionen.

`NUMBER`: Zahlen mit Betrag $1.0 \cdot 10^{-30} \leq x < 1.0 \cdot 10^{125}$ und 38-stelliger Genauigkeit. Auf `NUMBER` sind die üblichen Operatoren $+$, $-$, $*$ und $/$ sowie die Vergleiche $=$, $>$, $>=$, $<=$ und $<$ erlaubt. Außerdem gibt es `BETWEEN x AND y`. Ungleichheit wird je nach System (auch innerhalb ORACLE noch plattformabhängig) mit $!=$, $\wedge=$, $\neg=$ oder $<>$ beschrieben.

`LONG`: Zeichenketten von bis zu 2GB Länge.

`DATE`: Datum und Zeiten: Jahrhundert – Jahr – Monat – Tag – Stunde – Minute – Sekunde. U.a. wird auch Arithmetik für solche Daten angeboten (vgl. Abschnitt 4).

weitere Datentypen findet man im ORACLE SQL Reference Manual.

Das folgende SQL-Statement erzeugt z. B. die Tabelle *City* (noch ohne Integritätsbedingungen):

```
CREATE TABLE City
      ( Name          VARCHAR2(35),
        Country       VARCHAR2(4),
        Province      VARCHAR2(32),
        population    NUMBER,
        Longitude     NUMBER,
        Latitude      NUMBER );
```

Darüberhinaus können mit der Tabellendefinition noch Eigenschaften und Bedingungen an die jeweiligen Attributwerte formuliert werden. Dabei ist weiterhin zu unterscheiden, ob eine Bedingung nur ein einzelnes oder mehrere Attribute betrifft. Erstere sind unter anderem Wertebereichseinschränkungen, die Angabe von Default-Werten, oder die Forderung, dass ein Wert angegeben werden muss. Bedingungen an die Tabelle als Ganzes ist z. B. die Angabe von Schlüsselbedingungen.

```
CREATE TABLE <table>
  (<col > <datatype> [DEFAULT <value>]
        [<colConstraint> ... <colConstraint>],
  ⋮
  <col> <datatype> [DEFAULT <value>]
        [<colConstraint> ... <colConstraint>],
  [<tableConstraint>,]
  ⋮
  [<tableConstraint>]);
```

(Die in `[...]` aufgeführten Teile der Spezifikation sind optional)

- Als `DEFAULT <value>` kann dabei ein beliebiger Wert des erlaubten Wertebereichs spezifiziert werden, der eingesetzt wird, falls vom Benutzer an dieser Stelle kein Wert angegeben wird.

  Beispiel 1 (`DEFAULT`-Wert) Ein Mitgliedsland einer Organisation wird als volles Mitglied angenommen, wenn nichts anderes bekannt ist:

```
CREATE TABLE is_member
      ( Country       VARCHAR2(4),
        Organization  VARCHAR2(12),
        Type          VARCHAR2(30)
                      DEFAULT 'member')

INSERT INTO is_member VALUES
   ('CZ', 'EU', 'membership applicant');
INSERT INTO is_member (Land,Organization)
   VALUES ('D', 'EU');
```

Country	**Organization**	**Type**
CZ	EU	membership applicant
D	EU	member
⋮	⋮	⋮

□

- `<colConstraint>` ist eine Bedingung, die nur *eine* Spalte betrifft, während `<tableConstraint>` mehrere Spalten betreffen kann. Jedes `<colConstraint>` bzw. `<tableConstraint>` ist von der Form

  `[CONSTRAINT <name>] <bedingung>`

  Dabei ist der Teil `CONSTRAINT <name>` optional, fehlt er, so wird dem Constraint ein interner Name zugeordnet. Im allgemeinen ist es sinnvoll, `NULL`-, `UNIQUE`-, `CHECK`- und `REFERENCES`-Constraints einen Namen zu geben, um sie ggf. ändern oder löschen zu können (`PRIMARY KEY` kann man ohne Namensnennung löschen, vgl. Abschnitt 7).

Beide Arten von Constraints verwenden in `<bedingung>` prinzipiell dieselben Schlüsselwörter:

1. `[NOT] NULL`: Gibt an, ob die entsprechende Spalte Nullwerte enthalten darf (nur als `<colConstraint>`).
2. `UNIQUE (<column-list>)`: Fordert, dass jeder Wert nur einmal auftreten darf.
3. `PRIMARY KEY (<column-list>)`: Bestimmt diese Spalte zum Primärschlüssel der Relation. Damit entspricht `PRIMARY KEY` der Kombination aus `UNIQUE` und `NOT NULL`. (`UNIQUE` wird von `NULL`-Werten *nicht* unbedingt verletzt, während `PRIMARY KEY` `NULL`-Werte *verbietet.*)

   Da auf jeder Relation nur ein `PRIMARY KEY` definiert werden darf, wird die Schlüsseleigenschaft von Candidate Keys üblicherweise über `NOT NULL` und `UNIQUE` definiert.

   In Oracle wird bei einer `PRIMARY KEY`-Deklaration automatisch ein Index über die beteiligten Attribute angelegt.
4. `FOREIGN KEY (<column-list>) REFERENCES <table>(<column-list2>) [<referential actions>]`: gibt an, dass ein Attributtupel Fremdschlüssel (d.h. Schlüssel in einer (nicht unbedingt anderen!) Relation) ist; vgl. Abschnitt 8 (Referentielle Integrität).

   Das referenzierte Attributtupel `<table>(<column-list2>]` muss ein Schlüsselkandidat der referenzierten Relation sein. In Oracle kann nur der deklarierte `PRIMARY KEY` referenziert werden, damit ist die Angabe der Spalten optional (aber zur besseren Dokumentation zu empfehlen).

   Eine `REFERENCES`-Bedingung wird durch `NULL`-Werte nicht verletzt; sogar dann nicht, wenn der Rest absolut sinnlos ist:

   Beispiel 2 In der Tabelle City(Name,Country,Province) müssen die Einträge (Country,Province) einen entsprechenden Eintrag in *Province* referenzieren. Dennoch kann man ('Brussels','XX',NULL) in die Relation *City* einfügen, obwohl 'XX' kein gültiges Land ist. □

   Der optionale Zusatz `<referential actions>` wird in Abschnitt 8 (Referentielle Integrität) behandelt.
5. `CHECK (<condition>)`: Keine Zeile darf `<condition>` verletzen. `NULL`-Werte ergeben dabei ggf. ein *unknown*, also *keine Bedingungsverletzung.*

Bei einem `<colConstraint>` ist die Spalte implizit bekannt, somit fällt der `(<column-list>)` Teil weg.

Beispiel 3 (`PRIMARY` und `UNIQUE`) Für die Tabelle *Country* wird das Kürzel *Code* als `PRIMARY KEY` definiert, und die Eindeutigkeit des Namens durch `NOT NULL` (column constraint) und `UNIQUE (Name)` (table constraint) als Candidate Key garantiert:

```
CREATE TABLE Country
  ( Name        VARCHAR2(32) NOT NULL UNIQUE,
    Code        VARCHAR2(4) PRIMARY KEY);
```

Code als `PRIMARY KEY` wird in den verschiedenen Relationen referenziert, d.h. ist dort *Fremdschlüssel*:

```
CREATE TABLE is_member
      ( Country       VARCHAR2(4) CONSTRAINT MemberRefsCountry
           REFERENCES Country(Code),
        Organization  VARCHAR2(12) CONSTRAINT MemberRefsOrg
           REFERENCES Organization(Abbreviation),
        Type          VARCHAR2(30) DEFAULT 'member');
```

□

Beispiel 4 (`CHECK`-Bedingungen) `CHECK`-Bedingungen werden verwendet um Werte von Attributen einzuschränken; ihre Syntax ist für column constraints dieselbe wie für table constraints. Die vollständige Definition der Relation *City* mit Bedingungen und Schlüsseldeklaration lautet wie folgt:

```
CREATE TABLE City
  ( Name        VARCHAR2(35),
    Country VARCHAR2(4) References Country(Code),
    Province VARCHAR2(32),
    Population NUMBER CONSTRAINT CityPop CHECK (Population >= 0),
    Longitude NUMBER CONSTRAINT CityLong
                CHECK ((Longitude >= -180) AND (Longitude <= 180)) ,
    Latitude NUMBER CONSTRAINT CityLat
                CHECK ((Latitude >= -90) AND (Latitude <= 90)) ,
    CONSTRAINT CityKey PRIMARY KEY (Name, Country, Province),
    FOREIGN KEY (Country,Province)
                REFERENCES Province (Country,Name));
```

Die Bedingungen `CityPop`, `CityLong` und `CityLat` definieren Wertebereichseinschränkungen, `CityKey` deklariert `(Name, Country, Province)` als Schlüssel. Dies verlangt a) dass der jeweilige Wert nur einmal auftreten darf und b) ein Wert angegeben werden muss. □

Im Zusammenhang mit `FOREIGN KEY` bzw. `REFERENCES`-Deklarationen ist folgendes zu beachten:

- Wenn eine Tabelle mit einer Spalte, die eine `REFERENCES <table>(<column-list>)`–Klausel enthält, erstellt wird, muss `<table>` bereits definiert und `<column-list>` dort als `PRIMARY KEY` deklariert sein.
- Beim Einfügen von Daten müssen die jeweiligen referenzierten Tupel bereits vorhanden sein.
- Zu diesem Zweck können Constraints zeitweise deaktiviert werden (vgl. Abschnitt 7).

2.2 Definition von Views

Mit *Views (Sichten)* können die in einer Datenbank gespeicherten Daten einem Benutzer in einer von der tatsächlichen Speicherung verschiedenen Form dargestellt werden. Dies ist z. B. wünschenswert, wenn verschiedenen Benutzern verschiedene Ausschnitte aus der Datenbasis oder verschieden aufbereitete Daten zur Verfügung gestellt werden sollen. Bei Anfragen kann auf Sichten genauso wie auf Tabellen zugegriffen werden. Lediglich Änderungsoperationen sind nur in eingeschränktem Umfang möglich. Views werden nicht zum Zeitpunkt ihrer Definition berechnet, sondern jedesmal, wenn auf sie zugegriffen wird. Sie spiegeln also stets den aktuellen Zustand der ihnen zugrundeliegenden Relationen wieder.

```
CREATE [OR REPLACE] VIEW <name> (<column-list>) AS
<select-clause>;
```

Durch die (optionale) Angabe von `OR REPLACE` wird eine View, falls es bereits definiert ist, einfach durch die neue Definition ersetzt.

Beispiel: Angenommen, ein Benutzer benötigt häufig die Information, welche Stadt in welchem Land liegt, sei jedoch weder an Landeskürzeln noch Einwohnerzahlen interessiert. Dann ist es sinnvoll, eine entsprechende Sicht zu definieren.

```
CREATE VIEW CityCountry (City, Country) AS
   SELECT City.Name, Country.Name
   FROM City, Country
   WHERE City.Country = Country.Code;
```

Wenn der Benutzer nun nach allen Städten in Kamerun sucht, so kann er die folgende Anfrage stellen:

Views können außerdem zur Zugriffskontrolle eingesetzt werden (vgl. Abschnitt 10).

2.3 Löschen von Tabellen und Views

Tabellen bzw. Views werden mit `DROP TABLE` bzw. `DROP VIEW` gelöscht:

```
DROP TABLE <table-name> [CASCADE CONSTRAINTS];
DROP VIEW <view-name>;
```

- Tabellen müssen nicht leer sein, wenn sie gelöscht werden sollen.
- eine Tabelle, auf die noch eine `REFERENCES`-Deklaration zeigt, kann mit dem einfachen `DROP TABLE`-Befehl nicht gelöscht werden.
- Mit DROP TABLE <table> CASCADE CONSTRAINTS wird eine Tabelle mit allen auf sie zeigenden referentiellen Integritätsbedingungen gelöscht.

Veränderungen an der Struktur von Tabellen/Views (`ALTER`) werden in Abschnitt 7 behandelt.

3 EINFÜGEN, LÖSCHEN UND ÄNDERN VON DATEN

3.1 Einfügen von Daten

Das Einfügen von Daten erfolgt mittels der INSERT-Klausel. Von dieser Klausel gibt es im wesentlichen zwei Ausprägungen: Daten können von Hand einzeln eingefügt werden, oder das Ergebnis einer Anfrage kann in eine Tabelle übernommen werden:

```
INSERT INTO <table> [<column-list>]
VALUES (<value-list>);
```

oder

```
INSERT INTO <table> [<column-list>]
<subquery>;
```

Fehlt die optionale Angabe <column-list>, so werden die Attributwerte in der durch die Tabellendefinition gegebenen Reihenfolge eingesetzt. Enthält eine <value-list> weniger Werte als in der Tabellendefinition bzw. <column-list> angegeben, so wird der Rest mit Nullwerten aufgefüllt.

So kann man z. B. das folgende Tupel einfügen:

```
INSERT INTO Country
(Name, Code, Population)
VALUES ('Lummerland', 'LU', 4);
```

Eine (bereits definierte) Tabelle Metropolis (Name, Country, Population) kann man z. B. mit dem folgenden Statement füllen:

```
INSERT INTO Metropolis
SELECT Name, Country, Population
FROM City WHERE Population > 1000000;
```

Beim Einfügen von Daten in eine Tabelle, für die Fremdschlüsselbedingungen definiert sind müssen die jeweiligen referenzierten Tupel bereits vorhanden sein.

Bemerkung 1 Es ist allerdings sinnvoller, die oben genannte Tabelle Metropolis als View anzulegen:

```
CREATE VIEW Metropolis (Name, Country, Population) AS
SELECT Name, Country, Population
FROM City
WHERE Population > 1000000;
```

Der Unterschied liegt darin, dass ein View *Metropolis zu jeder Zeit* aktuell (bezüglich der Tabelle *City*) ist, während die Lösung mit der festen Tabelle jeweils die Situation zu dem Zeitpunkt beschreibt, wo das `INSERT ...`-Statement ausgeführt wurde. □

3.2 Löschen von Daten

Tupel können mit Hilfe der `DELETE`-Klausel aus Relationen gelöscht werden:

```
DELETE FROM <table>
WHERE <predicate>;
```

Mit einer leeren `WHERE`-Bedingung kann man z. B. alle Tupel einer Tabelle löschen (die Tabelle bleibt bestehen, sie kann mit `DROP TABLE` entfernt werden):

```
DELETE FROM City;
```

Der folgende Befehl löscht sämtliche Städte, deren Einwohnerzahl kleiner als 50.000 ist.

```
DELETE FROM City
WHERE Population < 50000;
```

Beim Löschen eines referenzierten Tupels muss die referentielle Integrität erhalten bleiben (vgl. Abschnitt 8).

3.3 Ändern von Daten

Die Änderung einzelner Tupel erfolgt mittels der `UPDATE`-Operation.

Syntax:

```
UPDATE <table>
SET <attribute> = <value> | (<subquery>),
                    ⋮
    <attribute> = <value> | (<subquery>),
    (<attribute-list>) = (<subquery>),
                    ⋮
    (<attribute-list>) = (<subquery>)
WHERE <predicate>;
```

Beispiel: Sankt-Peterburg wird in Leningrad umbenannt, außerdem steigt die Einwohnerzahl um 1000:

```
UPDATE City
SET Name = 'Leningrad',
    Population = Population + 1000,
WHERE Name = 'Sankt-Peterburg';
```

Oft ist `<subquery>` eine *korrelierte Subquery*, die abhängig von den Werten des zu verändernden Tupels die neuen Werte der zu verändernden Attribute berechnet:

Beispiel: Die Einwohnerzahl jedes Landes wird als die Summe der Einwohnerzahlen aller Provinzen gesetzt:

```
UPDATE Country
SET Population = (SELECT SUM(Population)
                  FROM Province
                  WHERE Province.Country=Country.Code);
```

4 ZEIT- UND DATUMSANGABEN

Für jeden Wert vom Typ `DATE` werden Jahrhundert, Jahr, Monat, Tag, Stunde, Minute und Sekunde gespeichert. Das Default-Format wird mit dem Parameter `NLS_DATE_FORMAT` gesetzt, als Default ist `'DD-MON-YY'` eingestellt, d.h. man gibt ein Datum z. B. als `'20-Okt-97'` an. Für die fehlenden Komponenten setzt ORACLE dabei Default-Werte ein.

- Gibt man ein Datum ohne Zeitkomponente an, wird dafür als Default 12:00:00 a.m. eingesetzt.
- Gibt man für ein Datum nur eine Zeitkomponente an, so wird der erste Tag des aktuellen Monats des aktuellen Jahres eingesetzt.
- partielle Zeit-/Datumsangaben werden nach diesem Muster aufgefüllt.

Die zulässigen Formate für Datums- und Zeitangaben findet man im ORACLE SQL Reference Guide.

MONDIAL enthält die Gründungsdaten der Länder in der Relation *politics*. Um diese Daten eingeben zu können muss u. a. die Angabe des Jahres auf 4 Ziffern erweitert werden:

```
CREATE TABLE Politics
  ( Country VARCHAR2(4),
    Independence DATE,
    Government VARCHAR2(120));
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM YYYY';

INSERT INTO politics VALUES ('AL','28 11 1912','emerging democracy');
INSERT INTO politics VALUES ('A','12 11 1918','federal republic');
INSERT INTO politics VALUES ('B','04 10 1830','constitutional monarchy');
INSERT INTO politics VALUES ('D','18 01 1871','federal republic');
```

Beispiel 5 Alle Länder, die zwischen 1200 und 1600 gegründet wurden lassen sich damit folgendermaßen bestimmen:

```
SELECT Country, Independence FROM politics
    WHERE Independence BETWEEN '01 01 1200' AND '31 12 1599';
```

Country	Independence
MC	01 01 1419
NL	01 01 1579
E	01 01 1492
THA	01 01 1238

□

Eine Konversion von `CHAR` und `DATE` ist mit `TO_DATE` und `TO_CHAR` möglich. Im Zusammenspiel mit `NLS_DATE_FORMAT` kann man damit die einzelnen Komponenten von Datumsangaben extrahieren:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD MM';
SELECT Country, TO_DATE(TO_CHAR(Independence,'DD MM'),'DD MM') FROM Politics;
```

Country	**TO_DA**
RSM	01 01
YU	11 04
SK	01 01
SLO	25 06
E	01 01

ORACLE bietet einige Funktionen um mit dem Datentyp `DATE` zu arbeiten:

- `SYSDATE` liefert das aktuelle Datum.
- Addition und Subtraktion von Absolutwerten auf `DATE` ist erlaubt, Zahlen werden als Tage interpretiert: `SYSDATE` + 1 ist morgen, `SYSDATE` + (10/1440) ist "in zehn Minuten".
- `ADD_MONTHS`(d, n) addiert n Monate zu einem Datum d. Ist d der letzte Tag eines Monats, so ist das Ergebnis der letzte Tag des resultierenden Monats.[1]
- `LAST_DAY`(d) ergibt den letzten Tag des in d angegebenen Monats.
- `MONTHS_BETWEEN`(d_1, d_2) gibt an, wieviele Monate zwischen den angegebenen Daten liegen (Dezimalwert, ein Monat wird mit 31 Tagen angenommen).
- weitere Funktionen siehe ORACLE SQL Reference Guide.

[1] Interessant: `ADD_MONTHS`(30-JAN-97,1).

5 OBJEKTTYPEN ALS KOMPLEXE ATTRIBUTE UND GESCHACHTELTE TABELLEN

Neben den bereits genannten Standard-Datentypen können mit Hilfe von ORACLE's objektorientierter Funktionalität komplexe Attribute, geschachtelte Tabellen (oft auch als *Collections* bezeichnet) sowie "richtige" Objekttypen definiert werden. "Richtige" Objekttypen soll hier bedeuten, dass ein Objekt Methoden bereitstellt, mit dem seine Daten manipuliert und abgefragt werden können. Dazu sind prozedurale Konzepte notwendig, die erst in Abschnitt 14 vorgestellt werden – entsprechend werden "richtige" Objekttypen erst in Abschnitt IV behandelt.

Bereits ohne Nutzung prozeduraler Konzepte erlaubt der "strukturelle" objektorientierte Ansatz die Erweiterung der deklarativen Konzepte um komplexe Attribute und geschachtelte Tabellen. Erstere sind bereits aus klassischen nicht-objektorientierten Programmiersprachen als *Records* bekannt, letztere stellen sich als "syntactic sugar" für Joins, Subqueries und Gruppierung heraus.

Beide Konzepte werden in ORACLE durch die Einführung von Schemaobjekten des Typs `TYPE` implementiert. `TYPE`n bestehen aus zwei Teilen:

- einer *Spezifikation*, die die Signatur des Typs angibt und mit `CREATE TYPE` erstellt wird. Dabei werden die Attribute (wie von `CREATE TABLE` bekannt) sowie die Signatur der Methoden definiert.
- ggf. einem *Body*, der die Implementierung der Methoden (in PL/SQL formuliert; vgl. Abschnitt 14) enthält und mit `CREATE TYPE BODY` erstellt wird. Der Body muss nur angegeben werden, falls für ein Objekt Methoden definiert werden sollen.

Komplexe Attribute und geschachtelte Tabellen sind Typen ohne Methoden und ohne Body:

```
CREATE [OR REPLACE] TYPE <type> AS OBJECT
  (  <attr> <datatype>,
       ⋮
     <attr> <datatype>);
/
 oder
CREATE [OR REPLACE] TYPE <type> AS TABLE OF <type'>;
/
```

Hierbei entsprechen die Attribute <attr> den Spalten <col> bei der Definition von Tabellen – man sieht, es ist eigentlich in erster Linie einmal eine Frage der Terminologie. Unter ORACLE ist es wichtig, die Eingabe zusätzlich zu den ";" noch durch "/" abzuschließen.

5.1 Komplexe Attribute

Ein komplexes Attribut ist ein Attribut, das aus mehreren Teilen besteht, z. B. "geographische Koordinaten" als Paar (Länge, Breite). Die Definition eines komplexen Attributs geschieht analog zu der Definition einer Tabelle.

Beispiel 6 (Komplexe Attribute) Geographische Koordinaten werden wie folgt als komplexes Attribut modelliert:

```
CREATE TYPE  GeoCoord AS OBJECT
  (Longitude NUMBER,
   Latitude  NUMBER);
/
```

Komplexe Attribute werden dann genauso wie andere Attribute in Tabellendefinitionen verwendet:

```
CREATE TABLE Mountain
  (Name        VARCHAR2(20),
   Height      NUMBER,
   Coordinates GeoCoord);
```

Durch `CREATE TYPE <type> AS OBJECT (...)` mit n Attributen wird automatisch eine n-stellige *Konstruktormethode* mit Namen `<type>` definiert, mit der Objekte dieses Typs generiert werden können:

```
INSERT INTO Mountain VALUES ('Feldberg', 1493, GeoCoord(8,48));
```

Entsprechend ist auch die Ausgabe:

Beispiel:

```
SELECT * FROM Mountain;
```

Name	Height	Coordinates(Longitude, Latitude)
Feldberg	1493	GeoCoord(8,48)

Die einzelnen Komponenten von komplexen Attributen werden in Anfragen mit der aus anderen Programmiersprachen bereits bekannten *dot*-Notation adressiert:

Beispiel:

```
SELECT Name, Coordinates.Longitude, Coordinates.Latitude FROM Mountain;
```

Name	Coordinates.Longitude	Coordinates.Latitude
Feldberg	8	48

Da komplexe Attribute prinzipiell *Objekte* sind (vgl. Abschnitt 15), können sie (zumindest in ORACLE 8.0.3 und 8.0.4) nur unter Verwendung eines Tabellen-Alias angesprochen werden. Die obige Anfrage muss umformuliert werden zu

```
SELECT Name, B.Coordinates.Longitude, B.Coordinates.Latitude FROM Mountain B;
```

(Bemerkung: "normale" Qualifizierung als `Mountain.Coordinates.Longitude` genügt nicht.)

5.2 Geschachtelte Tabellen

Manchmal ist es sinnvoll, in einem Relation ein Attribut zu definieren, das nicht höchstens einen Wert annehmen kann, sondern eine *Menge von Werten*:

Nested_Languages		
Country	**Languages**	
D	German	100
CH	German	65
	French	18
	Italian	12
	Romansch	1
FL	NULL	
F	French	100
⋮	⋮	

Mit geschachtelten Tabellen lassen sich Zusammenhänge modellieren, die sonst in verschiedenen Tabellen abgelegt werden in Anfragen per Join ausgewertet werden müssten. Eine solche Lösung bietet sich für mengenwertige Attribute oder 1:n-Beziehungen an, oft sogar für m:n-Beziehungen. Dazu wird zuerst der Typ der Tupel der inneren Tabelle als Objekt definiert, dann der Tabellentyp der inneren Tabelle auf Basis des eben definierten Tupeltyps. Dieser kann dann bei der Definition der äußeren Tabelle verwendet werden. Dabei wird angegeben, dass es sich um eine geschachtelte Tabelle handelt und diese bekommt einen Namen zugewiesen:

```
CREATE [OR REPLACE] TYPE <inner_type> AS OBJECT (...);
CREATE [OR REPLACE] TYPE <inner_table_type> AS
  TABLE OF <inner_type>;
/
CREATE TABLE <table_name>
  (  ... ,
     <table-attr> <inner_table_type> ,
     ... )
NESTED TABLE <table-attr> STORE AS <name >;
```

In der Mondial-Datenbank sind z. B. die in den einzelnen Ländern gesprochenen Sprachen durch geschachtelte Tabellen modelliert:

```
CREATE TYPE Language_T AS OBJECT
  ( Name VARCHAR2(50),
    Percentage NUMBER );
/
CREATE TYPE Languages_list AS
  TABLE OF Language_T;
/
CREATE TABLE NLanguage
  ( Country VARCHAR2(4),
    Languages Languages_list)
  NESTED TABLE Languages STORE AS Languages_nested;
```

ORACLE behandelt geschachtelte Tabellen ähnlich wie Cluster (vgl. Abschnitt 13): Es werden zwei *logische* Tabellen angelegt, die in demselben physikalischen Datensegment abgelegt werden.

Geschachtelte Tabellen unterstützen auch keine (referentiellen) Integritätsbedingungen, die den Inhalt der inneren Tabelle mit dem umgebeneden Tupel oder einer anderen Tabelle verknüpfen.

Analog zu komplexen Attributen ist auch für geschachtelte Tabellen eine Konstruktormethode desselben Namens definiert, die als Argument eine *Liste* von Objekten des Type der Listenelemente (natürlich wiederum mit der entsprechenden Konstruktormethode des Listenelement-Typs erzeugt) enthält. Diese wird dazu verwendet, `VALUES` in eine solche Tabelle einzufügen (Zu `INSERT INTO <table> SELECT ...` siehe weiter unten):

```
INSERT INTO NLanguage VALUES ('SK',
  Languages_list
  ( Language_T('Slovak',95),
    Language_T('Hungarian',5)));
```

Die übliche Anfragesyntax liefert geschachtelte Tabellen jeweils als ganzes zurück – d.h. als Liste von Objekten unter Verwendung der entsprechenden Objektkonstruktoren:

Beispiel:

```
SELECT *
FROM NLanguage
WHERE Country='CH';
```

Country	**Languages(Name, Percentage)**
CH	Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

Beispiel:

```
SELECT Languages
FROM NLanguage
WHERE Country='CH';
```

Languages(Name, Percentage)
Languages_List(Language_T('French', 18), Language_T('German', 65), Language_T('Italian', 12), Language_T('Romansch', 1))

Um *innerhalb* geschachtelter Tabellen arbeiten zu können werden diese mit dem Schlüsselwort `THE` angesprochen. `THE (SELECT ...)` teilt ORACLE mit, dass das Ergebnis einer Anfrage eine Tabelle ist:

```
SELECT ...
FROM THE (<select-statement>)
WHERE ... ;

INSERT INTO THE (<select-statement>)
VALUES ... / SELECT ... ;

DELETE FROM THE (<select-statement>)
WHERE ... ;
```

wobei `<select-statement>` eine *Tabelle* selektieren muss.

Beispiel:

```
SELECT Name, Percentage
FROM THE  (SELECT Languages
           FROM NLanguage
           WHERE Country='CH');
```

Name	**Percentage**
German	65
French	18
Italian	12
Romansch	1

```
INSERT INTO THE (SELECT Languages FROM NLanguage where Country= A.Country)
   VALUES ('Hungarian',2);
```

Wenn man das Ergebnis einer Subquery, die eine Menge von Tupel liefert, als geschachtelte Tabelle in eine Tabelle einfügen will, muss man diese mit Hilfe der Befehle `CAST` und `MULTISET` strukturieren:

```
INSERT INTO <table>
VALUES (..., CAST(MULTISET(SELECT ...) AS <nested-table-type>),...);
```

wobei `<nested-table-type>` der Typ der geschachtelten Tabelle an der entsprechenden Stelle ist:

Dennoch ist `CAST/MULTISET` zum Füllen von geschachtelten Tabellen nicht so geeignet wie man auf den ersten Blick vermuten könnte:

Beispiel 7 (`CAST` und `MULTISET`) Mit

```
INSERT INTO NLanguage - zulaessige, aber falsche Anfrage !!!!
 (SELECT Country,
        CAST(MULTISET(SELECT Name, Percentage
                      FROM Language
                      WHERE Country = A.Country)
             AS Languages_List)
 FROM Language A);
```

wird jedes Tupel (Land, Sprachenliste) x-mal eingefügt, wobei x die Anzahl der für das Land gespeicherten Sprachen ist. □

Daher ist es notwendig, einzeln vorzugehen:

```
INSERT INTO NLanguage (Country)
  SELECT Country
  FROM Language;

UPDATE NLanguage B
SET Languages =
  CAST(MULTISET(SELECT Name, Percentage
                FROM Language A
                WHERE B.Country = A.Country)
       AS Languages_List);
```

Liefert eine Subquery bereits eine Tabelle, kann diese ohne casting direkt (unter Verwendung von `THE`) eingefügt werden:

```
INSERT INTO <table>
VALUES (..., THE (SELECT <attr>
                  FROM <table'>
                  WHERE ...));
```

wobei das Attribut `<attr>` von `<table'>` eine geschachtelte Tabelle geeigneten Typs ist.

```
INSERT INTO NLanguage VALUES
 ('CHXX', THE (SELECT Languages from NLanguage
              WHERE Country='CH'));
```

Einige dieser Features scheinen in der derzeitigen Version noch nicht voll ausgereift zu sein. So darf eine Unterabfrage nur *eine einzige* geschachtelte Tabelle zurückgeben. Damit ist es also nicht möglich in Abhängigkeit von dem Tupel der äußeren Tabelle die jeweils passende innere Tabelle auszuwählen, bzw. nebeneinander Daten aus der äußeren und der inneren Tabelle auszugeben:

Beispiel 8 (Unzulässige Anfragen)

- Es sollen alle Länder ausgegeben werden, in denen Deutsch gesprochen wird:
  ```
  SELECT Country - UNZULAESSIG !!!! FROM NLanguage A,
  THE SELECT Languages FROM NLanguage B
  WHERE B.Country=A.Country) WHERE Name='German';
  ```
- Es sollen alle Paare (L, S) ausgegeben werden, so dass die Sprache S im Land L gesprochen wird:
  ```
  SELECT Country, Name, Percentage - UNZULAESSIG !!!!
  FROM NLanguage A,
       THE ( SELECT Languages
             FROM NLanguage B
             WHERE B.Country=A.Country);
  ```

□

Die zweite der oben aufgeführten Anfragen ist zulässig, wenn man nur ein Land selektiert – d.h., nur eine geschachtelte Tabelle verwendet wird:

```
SELECT Country, Name, Percentage
FROM NLanguage A,
     THE ( SELECT Languages
           FROM NLanguage B
           WHERE B.Country=A.Country)
WHERE A.Country = 'CH';
```

Die erste der oben aufgeführten Anfragen kann mit Hilfe einer Subquery und des `TABLE (<attr>)`-Konstrukts (wobei `<attr>` ein tabellenwertiges Attribut sein muss) gelöst werden. Dieses erlaubt, die innere Tabelle in einer Subquery zu verwenden:

Beispiel:

```
SELECT Country
FROM NLanguage
WHERE EXISTS
  (SELECT *
   FROM TABLE (Languages)
   WHERE Name='German');
```

Country
A
B
CH
D
NAM

wobei hier `(Languages)` immer die innere Tabelle *NLanguage.Languages* des aktuellen Tupels adressiert.

Da `TABLE (<attr>)` jedoch nicht in der `FROM`-Zeile des äußeren `SELECT`-Statements vorkommen kann, kann man auch keine Attribute der inneren Tabelle für das äußere `SELECT`-Statement angeben. Entsprechend ist es nicht möglich, in der obigen Ausgabe auch den prozentualen Anteil in den verschiedenen Ländern auszugeben.

Eine etwas bessere Formulierung liefert der `CURSOR`-Operator (verwandt mit dem PL/SQL-Cursorkonzept):

Beispiel:

```
SELECT Country,
       cursor (SELECT *
               FROM TABLE (Languages))
FROM NLanguage;
```

Country	CURSOR(SELECT...)
CH	CURSOR STATEMENT : 2
NAME	**PERCENTAGE**
French	18
German	65
Italian	12
Romansch	1

Dieses Problem, lässt sich erst mit PL/SQL (vgl. Abschnitt 14) und *Cursoren* (oder evtl. mit Version ORACLE 8.1) lösen, ebenso die zweite oben beschriebene Aufgabe.

Hat man eine Tabelle *All_Languages*, die alle Sprachen enthält, so ist immerhin die folgende korrelierte Subquery möglich:

```
SELECT Country, Name
FROM NLanguage, All_Languages
WHERE Name IN
  (SELECT Name
   FROM TABLE (Languages));
```

Fazit: Man sollte den Wertebereich von geschachtelten Tabellen in *einer* Tabelle zugreifbar haben.

Die so definierten Objekttypen kann man sich mit `SELECT * FROM USER_TYPES` ausgeben lassen:

Type_name	Type_oid	Typecode	Attributes	Methods	Pre	Inc
GeoCoord	_	Object	2	0	NO	NO
Language_T	_	Object	2	0	NO	NO
Languages_List	_	Collection	0	0	NO	NO

Typen können mit `DROP TYPE` gelöscht werden. Hat man *abhängige Typen*, d.h. die Definition von Type B nutzt die Definition von Type A, so muss man diese Abhängigkeit übergehen:

In dem obigen Szenario führt `DROP TYPE Language_T` zu der Fehlermeldung *"Typ mit abhängigen Typen oder Tabellen kann nicht gelöscht oder ersetzt werden"* – da der Typ `Languages_List` davon abhängig ist. `DROP TYPE Language_T FORCE` führt dazu, dass der Typ `Language_T` gelöscht wird, allerdings leidet darunter der abhängige Typ `Languages_List` auch etwas:

```
SQL> desc Languages_List;
FEHLER:
ORA-24372: Ungültiges Objekt für Beschreibung
```

6 TRANSAKTIONEN IN ORACLE

Beginn einer Transaktion.

```
SET TRANSACTION READ [ONLY | WRITE];
```

Sicherungspunkte setzen. Für eine längere Transaktion können zwischendurch Sicherungspunkte gesetzt werden: Falls die Transaktion in einem späteren Stadium scheitert, kann man auf den Sicherungspunkt zurücksetzen, und von dort aus versuchen, die Transaktion anders zu einem glücklichen Ende zu bringen:

```
SAVEPOINT <savepoint>;
```

Ende einer Transaktion.

- `COMMIT`-Anweisung, macht alle Änderungen persistent,
- `ROLLBACK [TO <savepoint>]` nimmt alle Änderungen [bis zu `<savepoint>`] zurück,
- DDL-Anweisung (z. B. `CREATE, DROP, RENAME, ALTER`),
- Benutzer meldet sich von ORACLE ab,
- Abbruch eines Benutzerprozesses.

7 ÄNDERN DES DATENBANKSCHEMAS

Mit Hilfe der `ALTER`-Anweisung kann (unter anderem) das Datenbankschema verändert werden. Für alle Datenbankobjekte, die mit einem `CREATE`-Befehl erzeugt werden, gibt es den analogen `DROP`-Befehl um sie zu löschen, und den entsprechenden `ALTER`-Befehl zum Verändern des Objektes.

Mit `ALTER TABLE` können Spalten und Bedingungen hinzugefügt werden, bestehende Spaltendeklarationen verändert werden, und Bedingungen gelöscht, zeitweise außer Kraft gesetzt und wieder aktiviert werden:

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ⋮
  DROP <drop-clause>
  DISABLE <disable-clause>
  ⋮
  DISABLE <disable-clause>
  ENABLE <enable-clause>
  ⋮
  ENABLE <enable-clause>;
```

Die einzelnen Kommandos werden im folgenden behandelt:
Mit `ADD` können zu einer Tabelle Spalten und Tabellenbedingungen hinzugefügt werden:

```
ALTER TABLE <table>
  ADD (<col> <datatype> [DEFAULT <value>] [<colConstraint> ... <colConstraint>],
       ⋮
       <col> <datatype> [DEFAULT <value>] [<colConstraint> ... <colConstraint>],
       <tableConstraint>,
       ⋮
       <tableConstraint>)
  MODIFY (<modify-clause>)
  DROP <drop-clause>
  ... ;
```

Beispiel: Die Relation *economy* wird um eine Spalte *unemployment* erweitert. Außerdem wird zugesichert, dass die Summe der Anteile von Industrie, Dienstleistung und Landwirtschaft am Bruttosozialprodukt maximal 100% ist:

```
ALTER TABLE Economy
  ADD (Unemployment NUMBER,
       CHECK (Industry + Services + Agriculture <= 100));
```

Beim Einfügen einer neuen Spalte wird diese mit NULL-Werten gefüllt. Damit kann eine solche Spalte nicht als NOT NULL deklariert werden (das kann man aber später per ALTER TABLE ADD (CONSTRAINT ...) noch nachholen).

Mit ALTER TABLE ... MODIFY lassen sich Spaltendefinitionen verändern:

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<col> [<datatype>] [DEFAULT <value>] [<colConstraint> ... <colConstraint>],
          ⋮
          <col> [<datatype>] [DEFAULT <value>] [<colConstraint> ... <colConstraint>])
  DROP <drop-clause>
  ... ;
```

Hier sind allerdings als <colConstraint> nur NULL und NOT NULL erlaubt. Alle anderen Bedingungen müssen mit ALTER TABLE ... ADD (<tableConstraint>) hinzugefügt werden.

Beispiele:

```
ALTER TABLE Country MODIFY (Capital NOT NULL);
ALTER TABLE encompasses ADD (PRIMARY KEY (Country,Continent));
ALTER TABLE Desert ADD (CONSTRAINT DesertArea CHECK (Area > 10));
```

Wird bei ADD oder MODIFY eine Bedingung formuliert, die der aktuelle Datenbankzustand nicht erfüllt, liefert ORACLE eine Fehlermeldung. Im obigen Beispiel wird die dritte Zeile akzeptiert, obwohl Null-Werte (die per Definition keine Bedingung verletzen können) auftreten.

Mit ALTER ... DROP/DISABLE/ENABLE können Bedingungen an eine Tabelle entfernt oder zeitweise außer Kraft gesetzt und wieder aktiviert werden:

```
ALTER TABLE <table>
  ADD (<add-clause>)
  MODIFY (<modify-clause>)
  DROP     PRIMARY KEY [CASCADE] | UNIQUE (<column-list>) |
           CONSTRAINT <constraint>
  DISABLE  PRIMARY KEY [CASCADE] | UNIQUE (<column-list>) |
           CONSTRAINT <constraint>| ALL TRIGGERS
  ENABLE   PRIMARY KEY | UNIQUE (<column-list>) |
           CONSTRAINT <constraint>| ALL TRIGGERS;
```

Mit ENABLE/DISABLE ALL TRIGGERS werden alle zu einer Tabelle definierten Trigger (siehe Abschnitt 14.7) aktiviert/deaktiviert.

Eine PRIMARY KEY-Bedingung kann nicht gelöscht/disabled werden solange dieser Schlüssel durch einen Fremdschlüssel in einer REFERENCES-Deklaration referenziert wird. Wird CASCADE angegeben, werden eventuelle Fremdschlüssel-Bedingungen ebenfalls gelöscht/disabled. Beim enable'n muss man allerdings kaskadierend disable'te Constraints manuell einzeln wieder aktivieren werden.

In dem Abschnitt über referentielle Integrität wird gezeigt, dass die Definition von referentiellen Integritätsbedingungen das Verändern von Tupeln erheblich behindert. Deshalb müssen in solchen Fällen vor dem Update die entsprechenden Integritätsbedingungen deaktiviert und nachher wieder aktiviert werden.

`ALTER SESSION` wurde bereits in Abschnitt 4 (Datumsangaben) verwendet, um die Parameter einer ORACLE-Sitzung zu verändern.