Typ- und Tabellenhierarchie

- **Ziel:**
  Modellierung von Hierarchien von Objektmengen

- **Ohne Typhierarchien keine Tabellenhierarchien!**
  - Strukturierter Typ kann andere strukturierte Typen als Subtypen haben
  - Tabelle mit Typbindung (typed table) kann Subtabellen haben

- **Eigenschaften**
  - Subtabellen erben Attribute, Constraints, Trigger usw. von der Supertabelle
  - Anfragen erhalten eine höhere Ausdrucksmächtigkeit
  - Anfragen auf der Supertabelle arbeiten auch auf den Subtabellen

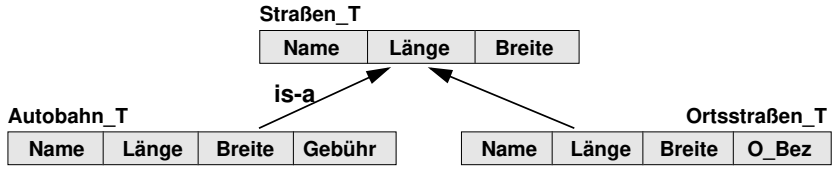
- **Aspekte der Zugriffskontrolle**
  - Anfragen, die Subtabellen referenzieren, benötigen SELECT-Privileg
  - SELECT-Privileg mit WITH HIERARCHY OPTION auf Supertabelle gewährt Zugriff auf alle zugehörigen Subtabellen

Typ- und Tabellenhierarchie - Beispiele (1)

- **Strukturierter Typ Straßen_T**
  CREATE TYPE Straßen_T **AS** (
  Name **CHAR** (40),
  Länge **DECIMAL** (9,2),
  Breite **DECIMAL** (5,2)) **NOT FINAL . . .**;

- **Subtypen**
  CREATE TYPE Autobahn_T **UNDER** Straßen_T(Gebühr Money) **NOT FINAL ...**;
  CREATE TYPE Ortsstraßen_T **UNDER** Straßen_T(O_Bez Orte) **NOT FINAL ...**;

- **Typhierarchie**



- **Tabellenhierarchie**
  - **CREATE TABLE** Straßen *// Supertabelle Straßen*
    OF Straßen_T (**PRIMARY KEY** Name, . . .);
  - **CREATE TABLE** Autobahnen *// Subtabelle Autobahnen*
    OF Autobahn_T **UNDER** Straßen;
  - **CREATE TABLE** Ortsstraßen *// Subtabelle Ortsstraßen*
    OF Ortsstraßen_T **UNDER** Straßen;

- **Erzeugen von Privatstraßen (vom Typ Ortsstraßen_T)?**

Wie sieht die Tabellenhierarchie aus?

Typ- und Tabellenhierarchie - Beispiele (2)

- **Einfügen von Zeilen in Straßen, Autobahnen, Ortsstraßen**

  INSERT INTO Ortsstraßen **VALUES** (‘Mozartstr’, 3.25, 8.75, ‘München’)
  INSERT INTO Autobahnen **VALUES** (‘A8’, 564.50, 20.10, US_Dollar(10))
  INSERT INTO Straßen **VALUES** (‘Schillerweg’, 7.75, 5.00)

- **Wie werden die Zeilen gespeichert?**

  Hausklassenmethode

  Straßen (**INSTANTIABLE**))

OID	Name	Länge	Breite
O21	Schillerweg	7.75	5.00

  Autobahnen

OID	Name	Länge	Breite	Gebühr
O08	A6	324.00	18.20	<D_Mark> 20
O71	A8	564.50	20.10	<US_Dollar> 10

  Ortsstraßen

OID	Name	Länge	Breite	O_Bez
O12	Schillerstr	2.50	7.50	Köln
O12	Mozartstr	3.25	8.75	München

- **Anfrage:** Automatische Berücksichtigung der ganzen Tabellenhierarchie

  Suche alle Straßen mit einer Breite größer als 5,50m.

  SELECT * **FROM** Straßen **WHERE** Breite > 5.50

- **Änderung:** Automatische Änderung in den korrespondierenden Super- und Subtabellen

  UPDATE Straßen **SET** Breite = 10.00 **WHERE** Name **LIKE** ‘Schiller%‘

  oder

  UPDATE Ortsstraßen **SET** Breite = 10.00 **WHERE** O_Bez = München

Konstruierte Typen: Tupeltyp

- **Ziel: Einfacher Umgang mit zusammengesetzten Werten**

- **Tupeltyp (*ROW Type*) als Datentyp von zusammengesetzten Attributen**
  - Zusammengesetztes Attribut kann in **einer** Spalte gespeichert werden
  - Tupel kann als **ein** Argument an Routinen und als Rückgabewert von Funktionen dienen
  - entspricht Record-Typen in Programmiersprachen

- **Tupeltyp** (früher: *unnamed row type*)
  - Definition von geschachtelten Tabellenstrukturen
  - spezielle Operationen:
    Konstruktor, Zuweisung, Vergleich von zusammengesetzten Werten

- **Beispiel:**

```
CREATE TYPE Straßen_T AS (                          // Strukturierter Typ
    Name          VARCHAR (40),
    Verwaltung    ROW (Bezeich VARCHAR (20),        // Tupeltyp
                       Stadt   VARCHAR (30)),
    Geometrie     Polygon,
    Referenzpunkt ROW ( Rechts  GK_Koordinate,      // Tupeltyp
                        Hoch    GK_Koordinate,
                        Höhe    NN_Höhe)) NOT FINAL . . .;
```

Referenztypen

- **Ziel: Verweise (Referenzen) auf andere Objekte**
- **Eigenschaften**
  - Referenzen repräsentieren Beziehungen zwischen Objekten
  - Ein Objekt kann von vielen anderen referenziert werden
  - Referenz verweist immer auf eine Zeile (Objekt) einer getypten Tabelle, nicht auf ein Objekt in einer Spalte
- **Verwendung**
  - Referenztypen können mit Zeilentypen kombiniert werden
  - Sie erleichtern die Modellierung von Beziehungen zwischen Typen
  - Wertebereich von Referenzen kann durch die SCOPE-Klausel eingeschränkt werden
  - Bei mehreren Tabellendefinitionen mit gleichem strukturierten Typ lassen sich für Referenzen unterschiedliche Gültigkeitsbereiche festgelegen
- **Beispiel**

```
CREATE TYPE Kunden_T AS (
    KNR         INTEGER,
    Name        CHAR (50),
    Anschrift   Adresse)
    NOT FINAL REF USING INTEGER;                    //Strukturierter Typ

CREATE TABLE Kunden OF Kunden_T (
    PRIMARY KEY KNR,
    REF IS kid USER GENERATED);     // selbst-referenzierendes Attribut

CREATE TABLE Privat_Kunden OF Kunden_T (
    PRIMARY KEY KNR,
    REF IS pid USER GENERATED);     // selbst-referenzierendes Attribut
```


Referenztypen (2)

- **Beispiel** (Fortsetzung)

```
CREATE TYPE Konto_T AS (
    Konto_Nr        INTEGER,
    Kunde           REF (Kunden_T),   // Verweis auf den zugehörigen Kunden
    Typ             CHAR (1),
    Eröffnet        DATE,
    Zinsrate        DOUBLE PRECISION,
    Kontostand      DOUBLE PRECISION
    ) NOT FINAL REF Konto_Nr;

CREATE TABLE Konto OF Konto_T (
    PRIMARY KEY Konto_Nr,
    REF IS koid DERIVED,
    Kunde WITH OPTIONS SCOPE Kunden);
```

Kunden

kid	KNR	Name	Anschrift
	...	...	Straße, Stadt, Land, PLZ

Privatkunden

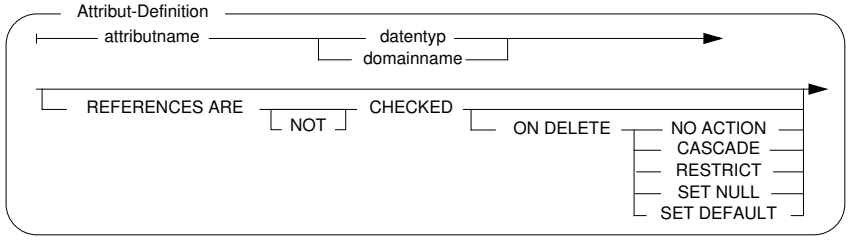
pid	KNR	Name	Anschrift
	...	...	Straße, Stadt, Land, PLZ

Konto

koid	Konto_Nr	Kunde	Typ	...	Kontostand
	...	...	...	...	...

Referenzen und Pfadausdrücke

- **Referenzen können nur auf „top-level rows“ verweisen**
  - Referenzwert ändert sich nicht, solange die entsprechende Zeile existiert
  - Referenzwerte werden nicht wiederbenutzt
  - Referenzwerte sind innerhalb der DB eindeutig
- **Es lassen sich „referentielle Aktionen” spezifizieren**



- **Referenzen können zur Spezifikation von Pfadausdrücken verwendet werden**

  ➥ Nur „scoped”-Referenzen können (mit Operator „->“) dereferenziert werden

```
SELECT   a.Konto_Nr, a.Kunde -> Name
FROM     Konto a
WHERE    a.Kunde -> Anschrift.Stadt = 'München'
         AND a.Kontostand > 1.000.000,00;
```

- **Autorisierung des Zugriffs wird nach dem SQL-Autorisierungsmodell überprüft**
  - Benutzer muß SELECT-Rechte auf den entsprechenden Attributen der Tabelle (Kunden) besitzen
  - Autorisierung kann zur Übersetzungszeit geprüft werden, falls eine SCOPE-Klausel spezifiziert wurde

Kollektionstypen

- **Warum Nutzung von Kollektionen?**
  - Modellierung von Attributen mit Kollektionswerten (*repeating groups*, *NF^2 tables*)
  - Wichtiger Baustein zur Modellierung: Oft Einsatz von Funktionen (prozeduraler Zugriff) erforderlich
  - häufig in Standard-Typ-Bibliotheken verwendet, z. B. SQL/MM (Fulltext, Spatial)
- **Kollektionstypen**
  - Array
  - Mengen
  - Listen (Sequenzen)
  - Multimengen (Bag)
- **Beispiel**

```
CREATE TABLE Stadtviertel (
   Name        CHAR (40) PRIMARY KEY,
   Geometrie   Polygon,
   Baublöcke   LIST (INTEGER),     // Nummern der enthaltenen Baublöcke
   Fläche      DECIMAL (10, 2),
   Straßen     SET (REF (Straßen_T)));          // Straßen im Stadtviertel
```

- **Wie geht man mit (großen) Kollektionen um?**
  - Laden, Ändern, Suchen, Transportieren?
  - Transformation von Kollektionen zu Tabellen sowie Anfragen darauf möglich

Beispiele - Kollektionstypen (1)

- **SQL99 unterstützt z. Z. nur ARRAY!**
  - z. Z. nur Definition „one-dimension, fixed-max-length“ möglich
  - Elementzugriff auf Array (Positionszugriff) und deklarative Anfragen auf Array-Elemente
  - Andere Operationen: Cardinality, Catenation, Vergleich, CAST, . . .

- **Tabelle Stadt mit ARRAY-wertigem Attribut S_Viertel**

CREATE TABLE Stadt (
Name **CHAR** (40) PRIMARY KEY,
S_Viertel **CHAR** (40) **ARRAY** [20],
Land **VARCHAR** (30));

- **Einfügen von Zeilen in Tabelle Stadt**

INSERT INTO Stadt (Name, S_Viertel, Land) **VALUES**
(‘Kaiserslautern’, **ARRAY** [‘Betzenberg’, ‘Uni-Wohngebiet’], ‘Rheinland-Pfalz’);

Stadt

Name	S_Viertel	Land
Mannheim	Käfertal	Baden-Württemberg
	Neckarau	
	Quadrate	
Ludwigshafen	BASF	Rheinland-Pfalz
	Fachhochschule	

Beispiele - Kollektionstypen (2)

- **Anfrage** (mit Positionszugriff)**:**
  Suche von allen Städten in Rheinland-Pfalz das erste aufgelistete Stadtviertel.

SELECT Name, S_Viertel [1] **AS** Stadtteil
FROM Stadt
WHERE Land = ‘Rheinland-Pfalz’;

Ergebnistabelle

Name	Stadtteil

- **Deklarative Anfragen auf Array-Elemente:**
  - implizite Umwandlung von Arrays in Tabellen
  - Element-Auswahl über Inhalt oder Position
  - Unnesting

- **Anfrage:**
  Suche von allen Städten in Rheinland-Pfalz alle aufgelisteten Stadtviertel.

SELECT s.Name, v.Stadtteil
FROM Stadt **AS** s, **UNNEST** (s.S_Viertel) **AS** v (Stadtteil)
// Umwandlung des Arrays in eine Tabelle
WHERE Land = ‘Rheinland-Pfalz’;

Ergebnistabelle

Name	Stadtteil

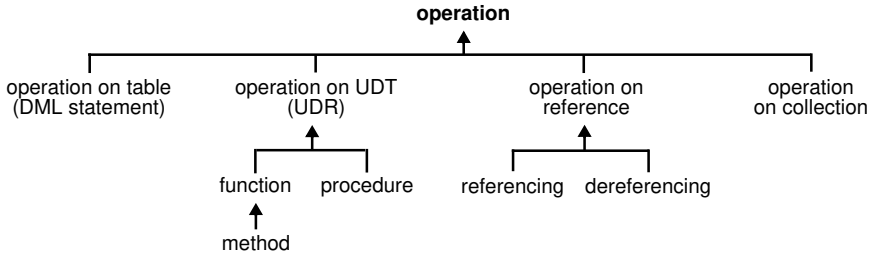
Von SQL aufrufbare Routinen

- **Überblick**
  - benannter, persistenter Code (*DB-stored*)
  - aufgerufen von SQL (*SQL-invoked*)
  - enthält einen Kopf (*Header*) und einen Rumpf (*Body*)

- **Klassifikation nach Sprache**
  - **SQL-Routine**
    - Header (Signatur) und Body spezifiziert in SQL
    - Routine-Body enthält eine einzige SQL-Anweisung inkl. einer zusammengesetzten PSM-Anweisung (*BEGIN . . . END*)
  - **Externe Routine**
    - Header (Signatur) spezifiziert in SQL
    - Body geschrieben in einer Wirtssprache
    - Body kann eingebettete SQL-Anweisungen enthalten

Von SQL aufrufbare Routinen (2)

- **Überblick**



- **Klassifikation nach Formen**
  - **Benutzerdefinierte Funktionen**
    - geben immer einzelne Werte als Ergebnis zurück
    - Überladbar, Typüberprüfung zur Übersetzungszeit
    - Auswahl über statische Parametertypen (statisches Binden)
  - **Benutzerdefinierte Prozeduren**
    - Aufgerufen durch eine CALL-Anweisung
    - Statisches Binden, kein Überladen
  - **Benutzerdefinierte Methoden**
    - sind Funktionen mit speziellen Aufrufkonventionen (p.JahresGehalt())
    - Überladbar, redefinierbar (überschreibbar)
    - „dynamic dispatch"; Laufzeit-Funktion von „SELF" bestimmt
    - Spätes Binden

SQL-Routinen

- **SQL-Prozeduren**
  - **Definitionsbeispiel**

```
CREATE PROCEDURE Kontoabfrage (IN KontoNr INT,
           OUT Betrag DECIMAL (15,2))
BEGIN
   SELECT      Kontostand INTO Betrag
      FROM     Konten
      WHERE    Kontonummer = KontoNr;
   IF Betrag < 100
      THEN     SIGNAL Niedriger_Kontostand
   END IF;
END
```

  - **Aufruf durch CALL-Anweisung**

```
CALL Kontoabfrage (4711, Betrag);
```

  - **Sind alle Arten von SQL-Anweisungen im Routine-Body erlaubt?**

- **Externe Prozeduren**

```
CREATE PROCEDURE Kontoabfrage (IN KontoNr INT,
           OUT Betrag DECIMAL (15,2))
LANGUAGE C
EXTERNAL NAME 'Konten/Abfrage_Prozedur'
```


SQL-Routinen (2)

- **SQL-Funktionen**
  - **Definitionsbeispiel**

```
CREATE FUNCTION Kontoabfrage (KontoNr INT)
RETURNS DECIMAL (15,2)
   BEGIN
      DECLARE     Betrag DECIMAL (15,2);
      SELECT      Kontostand INTO Betrag
         FROM     Konten
         WHERE    Kontonummer = KontoNr;
      IF Betrag < 100
         THEN SIGNAL Niedriger_Kontostand
      END IF;
      RETURN Betrag
   END
```

  - **Funktionsaufruf als Teil eines Ausdrucks**

```
SELECT Kontonummer, Kontoabfrage (KontoNr)
FROM Konten
```

  - **Ausnahmen?**

- **Externe Funktionen**

```
CREATE FUNCTION Kontoabfrage (KontoNr INT)
RETURNS DECIMAL (15,2)
LANGUAGE C
EXTERNAL NAME 'DBA/Konten/Abfrage'
```

Zusammenfassung

- **Objekt-Relationale Erweiterungen sind zentral**
  - Erweiterbares Typsystem
    - Benutzerdefinierte Typen (UDT) beschreiben die Anwendungsdaten
    - Benutzerdefinierte Routinen (UDR) definieren ein Verhalten für die Anwendungsdaten
  - Regeln und Trigger
  - Große Objekte (LOB) bis GByte werden unterstützt

- **Diese Erweiterungen erlauben mächtigere SQL-Anfragen**
  - einfachere und bessere Anwendungsentwicklung und -optimierung
  - einfachere, schnellere und mächtigere SQL-Anfragen
  - bessere Entscheidungsunterstützung, mächtigere Anfragegeneratoren

- **Trigger/Constraints erlauben**
  - Verbesserung der Datenintegrität
  - bessere Modellierung der Anwendungssemantik
  - Implementierung von Anwendungsregeln („Geschäftsregeln“)

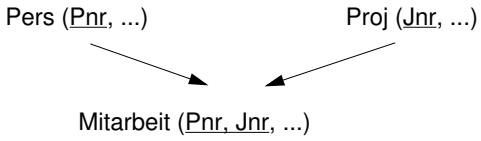
- **Ziel: offene Architektur für SQL-Klassenbibliotheken (z. B. SQL/MM)**

➥ erlaubt Anwendern die Integration von Funktionalität von externen Anbietern

- **Wettbewerber**
  - IBM: DB2 Universal Database V7, mit Parallelität verfügbar
  - Informix Dynamic Server 2000
  - Oracle 8i
  - Microsoft SQL Server 7.5, Sybase Adaptive Server
  - CA Associates (OpenIngres ++?), Software AG (Adabas C ++?)

Zusammenfassung (2)

- **SQL:1999 wurde hier nur teilweise vorgestellt!**
  - SQL:1999 ist vollständig aufwärtskompatibel zu SQL2
  - Mächtigkeit einer modernen Programmiersprache (SQL/PSM)
  - Allgemeine Tabellenausdrücke, Rekursion
  - **Verbesserte Orthogonalität der Anfragesprache**



Anfrage: Finde die Angestellten (PNR), die in allen Projekten mitarbeiten (mit ausschließlicher Hilfe mengentheoretischer Operationen).

```
(SELECT DISTINCT Pnr FROM Pers)
MINUS
(SELECT DISTINCT Pnr
FROM   (
          (SELECT  Pnr, Jnr
          FROM       (SELECT DISTINCT Pnr FROM Pers),
                     (SELECT DISTINCT Jnr FROM Proj)
          )
        MINUS
          ( SELECT DISTINCT Pnr, Jnr FROM Mitarbeit )
        )
)
```

- In der FROM-Klausel sind neben Basistabellen (und Views) auch durch ein geschachteltes SELECT berechnete Tabellen zulässig.
- Es werden zwei einspaltige (Pnr bzw. Jnr) berechnete Tabellen genutzt.

```
FROM   (SELECT DISTINCT Pnr FROM Pers),
       (SELECT DISTINCT Jnr FROM Proj)
```