UNIVERSITÄT KAISERSLAUTERN

Lehrstuhl für
Datenbanken und Informationssysteme
Prof. Dr. Theo Härder

Objekt-relationale Datenbanken

Thema: Überblick über den (objekt-relationalen) SQL:1999 Standard

Referent: Prof. Dr. Theo Härder

Betreuer: Dipl. Informatiker Wolfgang Mahnke

Bearbeiter: Christian Gorius

Matrikel-Nummer: 336520

Adresse: Kurt-Schumacherstrasse 46
67663 Kaiserslautern

Heimatadresse: Wintringerstrasse 72
66271 Kleinblittersdorf

Eingereicht am: 07. Februar 2001

Inhaltsverzeichnis

1 Einleitung 3
2 Neuerungen in SQL:1999 3
3 Das Typsystem von SQL:1999 5
3.1 BOOLEAN 5
3.2 BLOB 5
3.3 CLOB 6
3.4 ROW TYPE 7
3.5 Reference Type 7
3.6 ARRAY 8
3.7 DISTINCT 8
3.8 Benutzerdefinierte Datentypen 8
3.8.1 Attribute in einem benutzerdefinierten Datentyp 9
3.8.2 Initialisierung von ADTs 10
3.8.3 Zugriff auf Attribute innerhalb eines ADTs 10
3.8.4 Methoden in Abstrakten Datentypen 11
3.8.5 Untertypen und Vererbung 12
4 Benutzerdefinierte Funktionen (PSM) 12
4.1 SQL Client Module 13
4.2 SQL Server Module 14
4.3 SQL-invoked Routinen 14
4.4 SQL Routinen 15
4.5 Externe Routinen 16
4.6 SQL Prozeduale Sprachelemente 17
4.6.1 Blockanweisungen 17
4.6.2 Lokale Variablen 18
4.6.3 Zuweisungen 18
4.6.4 IF Anweisung 18
4.6.5 CASE Anweisung 18
4.6.6 FOR Anweisung 19
4.6.7 LEAVE Anweisung 19
4.6.8 LOOP Anweisung 19
4.6.9 WHILE Anweisung 19
4.6.10 REPEAT Anweisung 20
4.6.11 Condition Handling 20
5 Zusammenfassung 21

1 Einleitung

Das objektrelationale Datenbankmodell basiert sowohl auf dem relationalen und wird vom objektorientierten Datenbankmodell erweitert. Es extrahiert die Vorteile beider Ansätze und versucht diese in einem Modell zu vereinigen. Relationen bilden im objektrelationalen Datenbankmodell die Grundlage, d.h. sie stellen die einzige Möglichkeit dar, um Daten oder Objekte dauerhaft in der Datenbank zu speichern. Der Hauptunterschied besteht darin, wie die Tabellen aufgebaut sind: Im objektrelationalen Datenbankmodell können auch Objekte (und andere neue Datentypen) in den Tabellen verwaltet werden. Es werden nun eine neue Syntax und Semantik benötigt, um diese Objekte abzufragen oder ihre Methoden aufzurufen. 1999 wurde von American National Standard for Information Technology der SQL:1999 Standard verabschiedet, in dem die Syntax und Semantik der neuen Möglichkeiten beschrieben werden [ANSI99a][ANSI99b]. SQL:1999 ist eine aufwärtsgerichtete Erweiterung von SQL-92. Dies impliziert, dass eine SQL-92 Datenbank und deren Anwendungen ohne Änderungen auch in einer SQL:1999 Umgebung fehlerfrei betrieben werden können. SQL:1999 revolutioniert die Konstruktion und Benutzung von Datenbanken. Wird der komplette Standard realisiert (über 1600 Seiten), so eröffnet es den Anwendern und Entwicklern von Datenbanken viele neue Möglichkeiten.
Fast alle großen Hersteller von relationalen Datenbanksystemen haben unterdessen objektorientierte Erweiterungen ihrer Produkte auf den Markt gebracht.
In dieser Arbeit werden die neuen Möglichkeiten im Hinblick auf das Typsystem und auf benutzerdefinierte Funktionen dargestellt und erörtert. Dabei bezieht sich die Literaturangabe, wenn nicht im Text anders gekennzeichnet, auf [Fo99].

2 Neuerungen in SQL:1999

Die meisten Änderungen betreffen den „Part 2: Foundation“ des Standards. Hier werden die Trigger, Roles, benutzerdefinierte Typen, Subtables, das Typsystem und andere Sprachelemente beschrieben.

SQL:1999 beinhaltet Rekursion, was SQL zu einer effizienteren Sprache werden lässt. Mit Hilfe der Rekursion können verschiedene Suchstrategien verwirklicht werden, wie z.B. Tiefen- oder Breitensuche.
Weiter wurden eine Vielzahl von neuen build-in Datentypen und Operationen hinzugefügt, wobei der signifikanteste der benutzerdefinierte Datentyp ist. Dieser und andere neue Datentypen werden in Kapitel 3 erörtert.
SQL:1999 call-level interface (SQL/CLI) ist ein Mittel, welches dem Aufruf eines SQL Codes von einem Anwendungsprogramm unterstützt. Es stellt damit den Zugang für ein nicht kompatibles Programm zu einer SQL Datenbank dar. CLI unterstützt eine funktionale Schnittstelle zu der Datenbank. CLI unterstützt Anwendungen beim Interagieren mit der Datenbank durch das Benutzen eines CLI call Handler. (vergl. Abbildung 2.1).

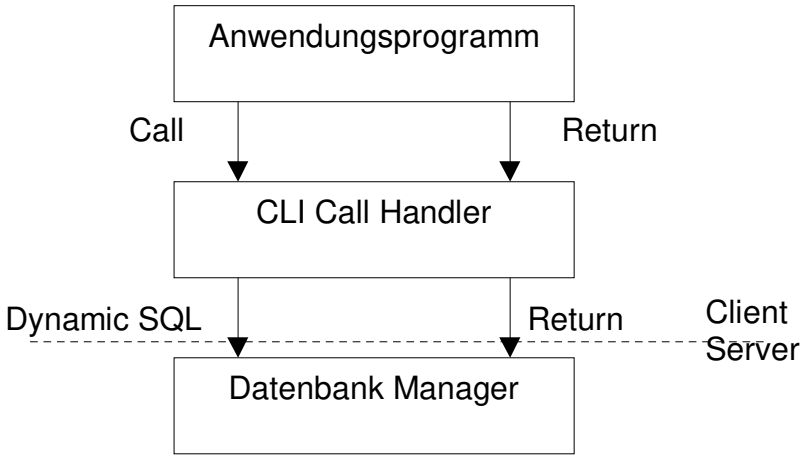

Abbildung 2.1: Call-Level Interface (CLI)

Der CLI call Handler empfängt als Input CLI „Handles“, welche er in SQL Kommandos konvertiert. Eine CLI-unterstützte Anwendung initialisiert einen SQL Befehl, indem zuerst ein CLI Umgebungshandler erstellt wird. Dann muss die Anwendung einen Connect Handler erstellen, welcher die Verbindung zu der Zieldatenbank kontrolliert. Nach dem Ausführen der SQL Anweisungen gibt die CLI die Handler wieder frei.
Eine weitere Neuerung innerhalb von SQL:1999 stellen die PSM (persistent stored modules) dar. Diese PSM erweitern SQL um einige Möglichkeiten, die in Kapitel 4 erörtert werden.
SQL Bindings spezifizieren die Beziehung und Operationen zwischen der SQL:1999 Sprache und einer anwendungsspezifischen Anwendungssprache. SQL Bindings

sind dazu da, dynamisch aufgerufenen, in einer Hostsprache eingebundenen SQL Code zu unterstützen. Die gewöhnliche Form der Bindings ist die EXEC SQL Anweisung. Es existieren Bindings Anweisungen zur Erstellung von Tabellen, zur Manipulation von Daten, zur Ausführung von SQL Exceptions und zur Prüfung und Auswertung von Constrains. Bindings sind für die Sprachen c, c++, Basic, ADA und COBOL definiert.

3 Das Typsystem von SQL:1999

Der Standard von SQL-2 umfasst zur Definition von Tabellen einige atomare Datentypen wie z.B. exakte numerische Werte, genäherte numerische Werte, Zeichenketten, Bit Strings, Datums- und Zeitangaben und Intervalle [Pa99]. Diese Datentypen werden auch in SQL:1999 unterstützt und um eingebaute Datentypen wie BLOB, CLOB, BOOLEAN und ARRAY ergänzt. Hinzu kommen noch benutzerdefinierte Datentypen, wie Row Type, Abstrakter Datentyp (ADT) und Reference Type. Diese neuen Datentypen werden im folgenden erläutert.

3.1 BOOLEAN

Dieser Datentyp ist eine Erweiterung von SQL:1999 gegenüber SQL-2. Dieser BOOLEAN-Datentyp unterscheidet sich von den logischen Operatoren in SQL-2. Dieser Datentyp besitzt die Werte `true` oder `false`.

3.2 BLOB

In SQL:1999 wurden weitere Datentypen eingeführt wie z.B. der Datentyp BLOB (large binary object). Der BLOB-Datentyp ist für binäre Daten vorgesehen. Man kann ihn wie folgt verwenden:

```
CREATE TABLE Angestellte (
        Anr     INTEGER,
        Name    CHARACTER VARYING (20),
        ...
        Bild    BLOB (10M)
)
```

Die Tabelle Angestellte besitzt ein Attribut Bild vom Typ BLOB. Dieses wird in der Datenbank als ein großes binäres Objekt gespeichert. Einige Operationen unterstützen die Manipulation und das Management solcher Objekte. Ein BLOB kann

durch ein anderes BLOB ersetzt werden; sie können verknüpft werden oder ihre Länge kann abgefragt werden. Weiter stehen die TRIM (Schnitt) Funktion und die OVERLAY Funktion zur Verfügung. Die Operatoren EQUAL und NOT EQUAL können angewendet werden. In einigen Fällen wird der UNION ALL-Operator unterstützt. Operatoren wie GREATER THAN, LESS THAN, GROUP BY, ORDER BY und PARTIAL UNION können nicht benutzt werden. Des weiteren dürfen in ihrer Definition PRIMARY KEY, FOREIGN KEY oder UNIQUE KEY nicht verwendet werden. Auch als Verbundattribute können sie nicht eingesetzt werden. BLOBs werden von der Datenbank manipuliert, nicht von der Anwendungsseite. Dies vermeidet viele unnötige Datenbewegungen ebenso wie anwendungsspezifisches Puffern der Daten, was notwendig wäre, wenn eine spezielle Anwendung viele BLOBs benötigt. Um ein BLOB-Objekt in der Datenbank zu lokalisieren wird ein LOB-Schlüssel (Locator) benötigt. Dieser kann von der Anwendung gehalten werden und ermöglicht einen schnellen Zugriff auf das Objekt. Wird der Locator nicht mehr benötigt, so kann er wieder freigegeben werden.

3.3 CLOB

Der Datentyp CLOB (Character Large Objects) ist dem BLOB-Datentyp sehr ähnlich. CLOBs sind geeignet, große Text Daten zu verwalten.

```
CREATE TABLE Angestellte (
        Anr     INTEGER,
        Name    CHARACTER VARYING (20),
        ...
        Lebenslauf      CLOB(50k)
)
```

Genau wie bei den BLOBs verwaltet und speichert die Datenbank die CLOBs für die Anwendung. Es gelten die gleichen Einschränkungen wie bei den BLOBs. Funktionen zur Teilzeichenfolge, Positions- oder Längenbestimmung werden unterstützt. Ähnlich wie bei den BLOBs werden bei CLOBs LOB-Schlüssel benutzt, um die Position innerhalb der Datenbank auszumachen. Der LOB-Schlüssel wird von der Datenbankseite benutzt, um CLOBs für die Benutzer-Anwendung zu lokalisieren und zu manipulieren. Wie bei den BLOBs wird dies gemacht, um die Zugriffskosten und die Datenbewegung zwischen Anwendung und Datenbank möglichst gering zu halten. Der LOB-Schlüssel erlaubt der Anwendung stückweisen Zugriff auf die CLOBs. Der LOB-Schlüssel kann von der Anwendung gehalten werden und, wenn er nicht mehr benötigt wird, wieder freigegeben werden.

3.4 ROW TYPE

Der Row Type ist eine Folge von Attributnamen und stellt im wesentlichen eine verschachtelte Tabelle innerhalb einer Tabelle dar, was im Beispiel verdeutlicht wird:

```
CREATE TABLE Angestellte (
        Anr       INTEGER,
        Adresse  ROW  (
                        Straße  CHARACTER VARYING (20),
                        Stadt    CHARACTER VARYING (20),
                        Staat    CHARACTER (3)
                        )
        ...
)
```

Der Row Type kann eine Einheit von Typen in einer Tabelle oder ein Ergebnis einer Abfrage oder Sicht sein. Ein Row Type kann mit einem Namen versehen werden, indem er separat definiert wird. Er stellt sehr gutes Mittel dar, um die Informationserfassung zu verfeinern, indem er mehrere Ebenen der Verschachtelung zulässt.

3.5 Reference Type

Ein weiterer wichtiger Datentyp in SQL:1999 ist der Reference Type. Der Reference Type stellt ein Mittel dar, mit dem eine Instanz eines anderen Datentyps adressiert werden kann. Der Reference type kann als Weg aufgefasst werden, mit dem eine Datenposition lokalisiert werden kann. Er kann überall dort eingesetzt werden, wo auch andere Datentypen eingesetzt werden können:

```
CREATE ROW TYPE account_typ (
        account_nr       INTEGER,
        account_adresse  REF(adresse),
        ...
);
```

In diesem Beispiel ist *account_adresse* nicht der Zeilentyp, sondern es wird eine Referenz zu einer Instanz vom Typ *adresse* erzeugt. Der Reference type kann gerahmt werden, z.B. kann der Typ *account_typ* verwendet werden, um eine Tabelle zu erstellen, sogenannten Typed Tables:

```
CREATE TABLE  account  OF  account_typ
        PRIMARY KEY  account_nr,
        SCOPE  Anschrift
);
```

Die Repräsentation eines Reference Typs kann vom System (REF IF SYSTEM GENERATED) oder durch den Benutzer (REF USING <Typ>) generiert werden. Wird nichts spezifiziert, so wird die Repräsentation vom System generiert.

3.6 ARRAY

ARRAY-Datentypen spezifizieren ein Feld von eingebauten Datentypen oder benutzerdefinierten Datentypen. Allerdings sind keine verschachtelte Arrays möglich, auch kann nur eine Dimension verwendet werden. Es kann jedoch ein Array von benutzerdefinierten Datentypen erstellt werden, welcher selbst ein Array enthält. Im Gegensatz zu einem eindeutigen Datentyp hat der ARRAY-Typ mehrere Instanzen, z.B. hat ein Angestellter mehrere Telefonnummern:

```
CREATE TABLE Angestellte (
        Anr     INTEGER,
        ...
        TelNr   CHARACTER (10) ARRAY (3)
)
```

In diesem Beispiel kann ein Angestellter bis zu drei TelNr. besitzen, welche in einem Array in der Datenbank gespeichert werden. Der ARRAY-Datentyp ist ein Mittel, mit dem sich wiederholende Gruppen leicht in der Datenbank vereinigt werden können. Der Zugriff auf die einzelnen Positionen innerhalb eines Arrays erfolgt über eine Nummer, ähnlich dem Zugriff auf Felder in Pascal.

```
SELECT Anr, TelNr[1]  FROM Angestellte WHERE ...
```

Der Zugriff kann auch deklarativ erfolgen, z.B.:

```
SELECT r.id, a.name FROM reports r AS r, UNNEST (r.autor) AS a (name);
```

Für eine Array-Variable kann in einer Host-Variable ein Locator definiert werden:

```
SQL  TYPE IS  INTEGER ARRAY[3]  AS LOCATOR aloc;
```

3.7 DISTINCT

Der DISTINCT Datentyp basiert darauf, dass Datentypen, die logisch inkompatibel sind, einander zugewiesen werden können. Es können z.B. DISTINCT Typen für DM und Franc definiert werden, die beide einem DECIMAL (7,2) Typ entsprechen, aber diese logisch inkompatiblen Datentypen können dann nicht einander zugeordnet werden. Der DISTINCT Typ kann mit Hilfe der CAST Funktion in einer anderen Typ übertragen werden. Für einen DISTINCT Typ wird keine Vererbung unterstützt, d.h. er muss stets als FINAL spezifiziert werden.

3.8 Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen (auch abstrakte Datentypen genannt) sind eine der wichtigsten Neuerungen innerhalb des Typsystems von SQL:1999. Ein abstrakter Datentyp (kurz ADT) ermöglicht dem Benutzer, komplexe Strukturen zu definieren.

Das fundamentale Konzept, welches sich hinter den ADTs verbirgt, ist, dass diese Typen die Informationen und spezielles Verhalten zu diesen Informationen definieren. Eine ADT-Definition in SQL:1999 definiert die Inhalte des ADTs (die gespeicherte Information) und das Verhalten der Instanz des ADT.
Für jedes Attribut, welches innerhalb eines ADTs definiert wird, werden automatisch Schnittstellen-Funktion generiert, welche zur Wertabfrage und zum Setzen der Attributwerten dienen.
Die Vorteile der ADTs liegen darin, das der Kode mehrfach verwendet werden kann. Die Möglichkeit zur Überladung macht Anwendungsprogramme leichter; sind Programme von Änderungen betroffen, so sind die Änderungen isoliert und betreffen selten das gesamte Programm.

3.8.1 Attribute in einem benutzerdefinierten Datentyp

Attribute in einem abstrakten Datentyp besitzen einen Namen und einen Typ. Der Name muss innerhalb eines ADTs eindeutig sein und darf nicht einem in SQL:1999 reserviertes Wort entsprechen (z.B. FROM, SELECT). Ein Attribut in einem ADT kann von jedem beliebigen Typ sein, also auch vom Typ ROW, ARRAY, DISTINCT oder einem anderen ADT. So können verschiedene ADTs ineinander verschachtelt werden:

```
CREATE TYPE person  AS (
        name    VARCHAR (20),
        beruf   VARCHAR (20),
        anschrift  adresse,
        ...
) NOT FINAL;
```

Des weiteren wird zwischen gespeicherten (stored) und virtuellen Attributen unterschieden. Der Unterschied liegt darin, wie der Wert eines Attributes repräsentiert und festgelegt wird. Ein Attribut ist durch seinen Namen und Typ genau festgelegt. Der Datentyp kann ein eingebauter Datentyp (INTEGER, REAL..) oder ein ADT sein. Gespeicherte Attribute werden durch gespeicherte Daten verbunden mit einer ADT-Instanz repräsentiert. Jeder dieser Attributinstanzen ist mit einem möglichen Wert verbunden. Dies ist dem Klassenkonzept von C++ sehr ähnlich.
Im Gegensatz zu gespeicherten Attributen ist es bei virtuellen Attributen nicht notwendig, dass diese direkt gespeicherten Daten entsprechen. Virtuelle Attribute werden durch die Ausführung von benutzerdefinierten Funktionen abgeleitet.

3.8.2 Initialisierung von ADTs

Um eine Instanz einer ADT zu initialisieren, wird ein Konstruktor benötigt. Ein Konstruktor kann default-Werte besitzen, welche in jeder neuen Instanz, die kreiert wird, eingesetzt werden, oder er kann parametrisiert sein, was eine benutzerdefinierte Initialisierung verlangt. Der Defaultkonstruktor, der jedes Attribut mit seinem Defaultwert initialisiert, wird vom System automatisch generiert.

Man kann auch explizit einen Konstruktor definieren, dem einige Parameter übergeben werden, der dann mit der NEW Anweisung aufgerufen wird:

```
CREATE METHODE adresse
        ( s  VARCHAR(20),  d  VARCHAR(20),  f  CHAR(5) )   RETURNS adresse
        BEGIN
                SET     :self.straße = s;
                SET     :self.stadt    = d;
                SET     :self.staat    = f;
                RETURN  a;
        END;
```

Überläßt man die Initialisierung dem System, d.h. man verwendet den Default-Konstruktor, so werden alle Werte mit ihren entsprechenden Default-Werten gesetzt, z.B. NULL für CHAR und VARCHAR usw.

Die NEW Anweisung bewirkt, dass zuerst ein Konstruktor aufgerufen wird, bevor die angegebene Methode aufgerufen wird:

```
NEW  <Name>  <Parameterliste>
```

Ist der Datentyp als NOT INSTANTIABLE spezifiziert, so kann er nicht instanziiert werden (entspricht den abstrakten Klassen in OO-Programmiersprachen). Der Defaultwert dieser Klausel ist INSTANTIABLE.

3.8.3 Zugriff auf Attribute innerhalb eines ADTs

Um auf eine Instanz innerhalb einer ADT zuzugreifen, benötigt man die automatisch generierten oder benutzerdefinierten Schnittstellenfunktionen oder die Punktnotation.

Die erste Methode, um auf Attribute in einem ADT zuzugreifen, besteht darin, die automatisch generierten Funktionen zu verwenden. Dazu wird folgendes Beispiel betrachtet:

```
SELECT Wohnort.Straße() FROM  Mitarbeiter  WHERE ....
```

Die Punktnotation stellt eine einfache Möglichkeit dar, um auf Attribute zuzugreifen. Auf den Namen der ADT Variable folgt ein Punkt und der Name des Attributes, auf das zugegriffen werden soll. Die Syntax unterstützt auch mehrere Stufen der Verschachtelung.

3.8.4 Methoden in Abstrakten Datentypen

Es ist, wie schon erwähnt, mit Hilfe von ADTs möglich, den Zugriff auf einzelne Attribute zu kapseln und den Zugriff durch die automatisch generierten Schnittstellenfunktionen zu steuern. Diese Operatoren sind jedoch nicht alles, was die ADTs bereitstellen. Es ist möglich, weitere gekapselte Funktionen mit zusätzlichen Eigenschaften und Funktionalität zu definieren. Durch die Definition weiterer Funktionen kann die Funktionalität eines ADTs besser an die Bedürfnisse des Benutzers angepasst werden. Diese Funktionen, die an einen ADT gebunden sind, werden Methoden genannt. Sie unterscheiden sich von den üblichen Funktionen darin, dass sie in einem Typschema definiert werden müssen und sich der Aufruf dieser Methoden vom Aufruf von Funktionen unterscheidet. Im Beispiel des ADTs Mitarbeiter können z.B. Funktionen benötigt werden, die aus dem Lebenslauf eines Mitarbeiters ermitteln können, ob dieser Abitur oder eine abgeschlossene Ausbildung besitzt. Dieses wird wie folgt implementiert:

```
CREATE TYPE Mitarbeiter AS (
        Name VARCHAR (20),
        Wohnort adresse,
        Lebenslauf  CLOB (50K),
        METHODE  Abitur () RETURNS BOOLEAN;
) NOT FINAL;

CREATE METHODE  Abitur()  FOR  Mitarbeiter
BEGIN
....
END;
```

Diese neue Methode kann nun von den Anwendern benutzt werden. Wie in vielen modernen Programmiersprachen kann auch in SQL:1999 der Name der Methode überladen werden, d.h. es können Methoden gleichen Namens existieren, denen unterschiedliche Parameter übergeben werden. Daran wird erkannt, welche Implementation der Methode aufzurufen ist. Werden z.B. Methoden zur Umrechnung von verschiedenen Währungen in DM bereitgestellt, so kann ein Name für sämtliche Methoden verwendet werden.

Eine weitere Möglichkeit stellt die Überschreibung von Methoden dar. Wird eine Methode an einen Typ vererbt, so kann dieser die Funktionalität der geerbten Methode überschreiben:

```
CREATE TYPE  entlassene_Mitarbeiter  UNDER Mitarbeiter AS (
        ....
        OVERRIDING  METHODE  Abitur()  RETURNS BOOLEAN;
) NOT FINAL;
```

Um eine Methode aufzurufen, wird die Punktnotation benötigt.

3.8.5 Untertypen und Vererbung

Der ADT ist ein sehr vielseitiger Datentyp. Eine Definition eines ADT kann eine interessante Klausel beinhaltet, die Subtyp-Klausel. Diese Klausel erlaubt, dass ein ADT einen anderen ADT als Supertyp benutzt. Um einen ADT als Untertyp eines anderen ADT zu definieren wird das Schlüsselwort UNDER benötigt. Der abgeleitete Datentyp erbt die Struktur (Attribute) und das Verhalten (Methoden) des Datentyps, von dem er abgeleitet wurde. Als Beispiel wird der Typ *Mitarbeiter* betrachtet:

```
CREATE TYPE Mitarbeiter  AS (
        Name VARCHAR (20),
        Wohnort adresse,
) NOT FINAL;
```

Von diesem Typ können nun andere Typen abgeleitet werden:

```
CREATE TYPE  Führungskraft  UNDER  Mitarbeiter  AS (
        Titel     VARCHAR (5),
) NOT FINAL;
```

Durch die Ableitung eines ADTs von einem anderen können für diesen Typ weitere Attribute oder Methoden definiert werden. Wird ein Typ mit FINAL spezifiziert, so können keine weiteren Typen von diesem Datentyp abgeleitet werden.

4 Benutzerdefinierte Funktionen

Neben den abstrakten Datentypen sind die benutzerdefinierten Funktionen eine der wichtigsten Neuerungen innerhalb von SQL:1999. Innerhalb des Rahmens der SQL-Umgebung gibt es SQL-Kataloge und sog. Client Module. Ein SQL-Katalog besteht aus einer Sammlung von SQL-Schemata; eine Datenbank kann mehrere Kataloge enthalten. Jedes Schema hat einen Besitzer [BTW99]. Zum Beispiel kann eine Kaufhausdatenbank einen Katalog enthalten, der alle verfügbaren Artikel enthält und einen Katalog für alle Dienstleistungen. Ein Katalog besitzt eine Menge von Schemata, welche Tabellen und andere Elemente (ADTs, Triggers usw.) enthalten können.

Ein SQL-Modul ist ein wichtiges Schemaobjekt, welches Schnittstellen enthält sowie seine zugehörigen Prozeduren oder Funktionen, die die Schnittstellen implementieren. Das Schema, dem diese Module zugeordnet sind, hängt vom Benutzer ab, der diese Module erstellt hat, und von den eingebetteten Prozeduren und Funktionen. Prozeduren und Funktionen müssen nicht in dem SQL-Modul eingebettet sein; sie können auch als eigenständige Objekte in der Datenbank

existieren. Der Unterschied liegt darin, wie die Prozeduren und Funktionen verwaltet und genutzt werden und wie auf sie zugegriffen werden kann. Eine SQL-Umgebung kann auch Positionen beinhalten, welche außerhalb einer spezifizierten Datenbank existieren. Diese werden definiert und referenziert als Client Module (Client modules). Die Client Module können selbst eingebettete Prozeduren enthalten. Diese Datenbankobjekte sind dauerhaft in der Datenbank gespeichert, sind aber außerhalb der Domain einer bestimmten Datenbank instanziiert und mit einer Menge von verschiedenen Zugriffs- und Operationsqualitäten ausgestattet.

4.1 SQL Client Module

SQL Client modules existieren außerhalb des Datenbankkatalogs, d.h. sie sind auf einem Client der Datenbank definiert und existieren dort, nicht im Server. Client modules bestehen aus einer Vielzahl von SQL Strukturen; die primäre Struktur ist die eingebettete SQL Prozedur. Ein Beispiel einer Moduldeklaration ist:

```
MODULE Rechnungsprüfung
```

Damit ist ein Modul mit Namen *Rechnungsprüfung* deklariert worden. Diesem Teil folgt die Spezifikation der Sprache, welche zur Implementation benutzt werden soll, wie z.B. Ada, C, COBOL, FORTRAN, Pascal, PL/I Mumps. SQL wird als Sprache nicht unterstützt [ANSI99a]. Nach der Spezifikation der Sprache wird die Autorisation angegeben. Diese Autorisation beinhaltet, welche Benutzer oder Gruppen Rechte besitzen, auf dieses Modul zuzugreifen, und unter welchem Schema das Modul laufen darf. Wird kein Schema angegeben, so entspricht das Schema dem Schema, das durch den Benutzer impliziert wird [ANSI99b]. Weiter kann der Pfad angegeben werden, wo die externen Module in einem speziellen System zu finden sind:

```
MODULE Rechnungsprüfung
    LANGUAGE C
    AUTHORIZATION Manager
    PATH ORCL.Manager, ORCL.accounts
```

Die Module bestehen aus Prozeduren. Sie können auch temporäre Tabellen oder Cursor beinhalten. Dies sind typischerweise Prozeduren, welche von der Host-Sprache aufgerufen werden können. Die Prozeduren sind das Hauptelement in externen Client modules. Die Deklaration einer solchen Prozedur ist die gleiche wie bei einer internen SQL Prozedur. Das Schlüsselwort PROCEDURE zeigt den Anfang einer Prozedur an; die Deklaration muss auch einen SQL Status Parameter (SQLSTATE) enthalten. SQLSTATE ist ein fünfstelliger Kode, der genauere Auskunft darüber gibt, warum eine Prozedur nicht korrekt ausgeführt werden konnte. Eine

genaue Aufschlüsselung der Fehlerkodes findet man in [ANSI99a]. Die Deklaration folgt auf die Deklaration des Moduls:

```
PROCEDURE belasten (nummer INTEGER, last DECIMAL (10,2) );
        UPDATE Kunden_Konto
        SET Guthaben Guthaben - last
        WHERE Nr = nummer;
```

Diese Prozedur kann in das Modul eingebettet werden, indem die Moduldeklaration vor die Prozedurdeklaration geschrieben wird. Der Body der Prozedur kann alle zulässigen SQL-Anweisungen beinhalten. SQL Client Module, die in einer Clientumgebung gespeichert und spezifiziert werden, können von allen internen und externen Kodesegmenten aufgerufen werden, die die speziellen Rechte besitzen. Die Modul-Prozeduren werden durch einen Host-Language-Call aufgerufen, z.B.:

```
belasten ( &SQLSTATE, 5, 15.25 );
```

Auf diese Weise kann ein Programm leicht mit Hilfe der Client Elemente mit einer SQL Datenbank interagieren. Die Client Module stellen also einen Schnittstellen-mechanismus zu der SQL-Datenbank dar.

4.2 SQL Server Module

Server Module unterscheiden sich von den Client Module darin, dass sie im Server anstatt im Client existieren. Der Hauptunterschied liegt also darin, dass Server Module in der Domain einer Datenbank existieren und den Korrektheitskriterien und Konsistenzkriterien des Servers unterliegen. Diese Server Module sind Schemaobjekte; sie sind persistente Objekte und sind der Datenbankumgebung bekannt. Server Module können Prozeduren, Funktionen und temporäre Tabellen beinhalten.

4.3 SQL-invoked Routinen

SQL-invoked Routinen sind entweder Elemente einer SQL Server Modul oder Client Modul Definition oder Elemente eines Datenbankschemas. Sie können Prozeduren, Funktionen oder Methoden darstellen. SQL-invoked Routinen, die Elemente eines Schemas darstellen, werden auch als Schema-level Routinen bezeichnet [ANSI99a] Eine SQL-invoked Routine kann in verschiedenen Programmiersprachen erstellt werden; zur Spezifikation der Sprache dient die LANGUAGE Klausel. Wird diese nicht angegeben, so wird die Sprache SQL impliziert. Eine SQL-invoked Routine, die in SQL geschrieben ist, nennt man SQL Routine; ist sie in einer anderen Sprache erstellt, so wird sie externe Routine genannt [ANSI99b].

4.4 SQL Routinen

Routinen in SQL:1999 sind Bestandteile der Datenbank Server Modulen. Sie stellen „first-class“ Schemaobjekte dar, was bedeutet, dass diese Objekte dauerhaft in der Datenbank gespeichert und verwaltet werden. Weiterhin kann ein Benutzer die Existenz dieses Objektes in der Datenbank sehen. Ihre Definition ist von der Datenbank erreichbar, eine äußere Interaktion zur Verwaltung und zum Speichern wird nicht benötigt. Die Hauptklasse der Routinen stellen Funktionen und Prozeduren dar. Die Neuerungen liegen vor allem darin, wie die Routinen in der SQL Umgebung oder von einem SQL Client verwaltet und zugegriffen werden. Eine Routine in SQL besteht aus drei Teilen, dem Namen, der Parameterliste und dem Body der Routine.
Eine SQL Prozedur stellt eine Routine dar, die mit der CALL Anweisung aufgerufen wird. Sie kann Input, Output Parameter besitzen oder Parameter, die sowohl Input als auch Output Parameter darstellen. Eine SQL Funktion ist eine Routine, die einen Wert zurückgibt. Sie kann nur Input Parameter besitzen. Eine Funktion kann auch als SQL Methode definiert werden. Bevor auf die genaue Syntax eingegangen wird, werden die Vor- und Nachteile von Routinen aufgezeigt.
Ein wichtiger Vorteil liegt in der Flexibilität; mit Routinen kann wesentlich mehr realisiert werden als nur mit Triggern und Constrains. Es können langsame, generierte SQL Anweisungen durch sorgfältig optimierte Routinen ersetzt werden, insbesondere durch externe Routinen, die nicht in SQL geschrieben sind. Durch den Einsatz von Routinen kann es vermieden werden, den gleichen SQL Code mehrmals schreiben zu müssen. Routinen können z.B. zur speziellen Privileg-Prüfung nützlich eingesetzt werden.
Nachteile von SQL Routinen liegen darin, dass sie nicht so vollständig sind wie eine Host-Sprache es ist. Externe Routinen haben den Nachteil, dass die Datentypen zwischen der Host-Sprache und den SQL Datentypen konvertiert werden müssen [Gu99].
Eine SQL Routine wird mit der CREATE PROCEDURE/ FUNCTION/ METHODE erzeugt. Die Beschreibung der Routine wird als Schemaobjekt gespeichert, deshalb muss sie allen Regeln, wie andere Schemaobjekte, genügen. Der Besitzer eines Schemas hat immer EXECUTE Rechte auf allen Routinen, die zu seinem Schema gehören. Der Body einer SQL Routine kann sämtliche SQL Anweisungen beinhalten, allerdings sind DDL Anweisungen, CONNECT oder DISCONNECT Anweisungen, dynamisches SQL oder COMMIT und ROLLBACK Anweisungen nicht zulässig. Die

RETURN Klausel ist nur in Verbindung mit CREATE FUNCTION/ METHODE erlaubt. Der Aufruf einer Prozedur erfolgt mit Hilfe der CALL Anweisung, Funktionen benötigen diese nicht.

4.5 Externe Routinen

Externe Routinen besitzen mehrere Subklauseln, von denen einige wichtige hier erörtert werden. Besitzt eine externe Routine einen Array als Parameter, oder ist der Rückgabewert einer Funktion ein Array, so wird dieser als Locator spezifiziert. Sobald die Routine aufgerufen wird, wird eine eindeutige implementationsabhängige Integer Locator Variable für jeden Input Locator Parameter generiert. Nach Transaktionsende wird der Locator wieder gelöscht. Ein HOLD LOCATOR kann allerdings bei Parametern von externen Routinen nicht definiert werden. Mit Hilfe der LANGUAGE Klausel kann die Host-Sprache spezifiziert werden, in der die Routine geschrieben wird, dabei muss es sich um eine der von SQL:1999 unterstützten Sprachen handeln. Die SQL Data Access Indication Klausel gibt an, welche Art von SQL Anweisungen in der Routine verwendet werden. Ist die Sprache z.B. mit LANGUAGE PASCAL angegeben, so kann die Routine dennoch SQL/CLI Funktionen oder eingebettetes SQL enthalten; deshalb muss dann die Sprache als CONTAINS SQL spezifiziert werden. MODIFIES SQL DATA gibt an, dass die Routine INSERT, UPDATE oder DELETE Anweisungen enthalten darf. Weiter stehen NO SQL und READS SQL DATA als Optionen zur Verfügung. Der Default-Wert ist CONTAINS SQL.
Die PARAMETER STYLE Klausel unterstützt zwei Optionen: SQL und GENERAL. Ist PARAMTER STYLE SQL spezifiziert, so besitzt jede Variable eine Indikator Variable. Dies ist nötig, um Null-Werte zu handhaben. Zusätzlich werden noch ein SQLSTATE Parameter und weitere Parameter übergeben, welche zusätzliche Informationen liefern. Ist PARAMTER STYLE GENERAL angegeben, so werden keine zusätzlichen Variablen benötigt, allerdings können keine Null-Werte unterstützt werden, d.h. sobald eine Variable einen Null-Wert besitzt, wird eine Ausnahme (Exception) geworfen. Handelt es sich um eine externe Funktion, die einen Array zurückgibt, so muss PARAMETER STYLE SQL angegeben oder impliziert werden. Die RETURN Klausel kann bei externen Funktionen durch eine CAST Funktion spezifiziert werden, die die Returnvariable in einen anderen Datentyp überführt.

Die Null-Call Klausel ist für Routinen, die in einer Host-Sprache geschrieben wurden, da diese keine Null-Werte managen kann. Wird die Klausel mit RETURN NULL ON NULL INPUT angegeben, so ist die Routine eine „Null-Call“ Routine. Wird eine „Null-Call“ Routine aufgerufen und ist dabei irgendein Input Parameter mit dem Wert Null belegt, so gibt dieser direkt Null zurück. Wird die Option CALL ON NULL INPUT gewählt und der Aufruf der Funktion erfolgt mit einem Null-Wert bei einer beliebigen Variablen, so wird diese Funktion ausgeführt und folgt den Regeln für die Operationen mit Null-Werten (z.B. Vergleich eines Null-Wertes mit einem anderen Wert ergibt UNKNOWN) [Gu99].

4.6 SQL Prozeduale Sprachelemente

Durch die Erweiterung von SQL:1999 um einige prozeduale Sprachelemente, kann eine Applikation jetzt komplett in SQL geschrieben werden. Die Hauptelemente dieser Erweiterung umfassen Variablendeklaration, Blockanweisungen, IF, CASE, LOOP, WHILE, LEAVE, REPEAT Anweisungen und Condition Handling, die im folgenden kurz erläutert werden.

4.6.1 Blockanweisungen

Mit Hilfe der Blockanweisung können Gruppen von SQL Anweisungen in eine Einheit zusammengefasst werden. Um einen Block zu definieren, sind die Schlüsselwörter BEGIN und END notwendig. Blockanweisungen können als ATOMIC oder als NOT ATOMIC spezifiziert werden; wird nichts spezifiziert, so wird immer NOT ATOMIC impliziert. Ist der Block als ATOMIC spezifiziert und eine SQL Anweisung innerhalb des Blockes kann nicht ausgeführt werden, so wird der ganze Block nicht ausgeführt, es kommt aber nicht zu einem ROLLBACK der Transaktion. Wird bei NOT ATOMIC eine SQL Anweisung erfolgreich beendet und eine andere innerhalb des Blockes führt zu einem Fehler, dann bleibt die durchgeführte Anweisung bestehen und wird nicht zurückgesetzt. Variablen oder Cursor, die in einem Block definiert wurden, sind nur innerhalb des Blockes gültig. Blockanweisungen können ineinander verschachtelt sein. Wird in der inneren Blockanweisung eine Variable mit dem gleichen Namen wie in der äußeren Blockanweisung deklariert, so verdeckt die innere die äußere Variable [BTW99c].

4.6.2 Lokale Variablen

Funktionen, Prozeduren und Blöcke können lokale und globale Variablen in ihrer Struktur definieren. Diese können mit einem Defaultwert definiert und initialisiert werden und können einen Null-Wert beinhalten. Um eine Variable zu definieren wird das Schlüsselwort DECLARE verwendet, gefolgt vom Namen und Typ der Variablen und optional von einem Defaultwert:

```
DECLARE Hilfsvariable INTEGER DEFAULT 0;
```

4.6.3 Zuweisungen

Zuweisungen ermöglichen, dass das Ergebnis eines Ausdruckes einer Zielvariablen zugeordnet werden kann. Folgende Syntax kann verwendet werden [BTW99b]:

```
SET variablen_name = Ausdruck ;
oder
SELECT Straße INTO variablen_name FROM ...
```

4.6.4 IF Anweisung

Die IF Anweisung dient der Überprüfung von Bedingungen, wie dies auch in anderen Programmiersprachen der Fall ist. Diese Anweisung wird wie folgt benutzt:

```
IF Bedingung
  THEN SQL-Anweisung
  ELSEIF Bedingung
  THEN SQL-Anweisung
 ...
   ELSE SQL-Anweisung
END IF
```

Die ELSEIF und ELSE-Anweisungen sind optional.

4.6.5 CASE Anweisung

Mit der CASE Anweisung kann eine Anweisung oder ein Anweisungsblock, als Resultat einer Bedingung, z.B.:

```
CASE (SELECT status FROM Kunde WHERE ...)
   WHEN 'VIP' THEN ...
   WHEN 'Privat' THEN ...
   ELSE ...
END CASE;
```

Hierbei ist die Spezifikation der ELSE Anweisung optional. Wird kein ELSE spezifiziert und keine der Zweige wird mit wahr ausgewertet, so wird eine Ausnahme (Exception) geworfen [BTW99c].

4.6.6 FOR Anweisung

Die FOR Anweisung in SQL:1999 ist eine iterative Anweisung. Sie iteriert über die Elemente einer Ergebnistabelle. Die FOR Iteration muss mit einem Suchausdruck verbunden sein; für jede Zeile der Ergebnistabelle können dann SQL Anweisungen durchgeführt werden, z.B.:

```
DECLARE x INTEGER DEFAULT 0;
FOR L1 AS SELECT Gehalt FROM Angestellte DO
SET x = x + Gehalt;
END FOR;
```

Es ist nicht zulässig, dass der Body der FOR Anweisung eine LEAVE Anweisung enthält, die sich auf *L1* bezieht. Bei einer FOR Anweisung wird implizit ein Cursor zu Beginn der Ausführung geöffnet und automatisch am Ende wieder geschlossen. Es ist möglich, diesen Cursor mit einem Namen zu spezifizieren:

```
FOR L1 AS curs1 CURSOR FOR
SELECT * FROM Kunde WHERE ...
```

Allerdings dürfen innerhalb der FOR Anweisung keine OPEN, FETCH oder CLOSE Anweisungen, die sich aus *curs1* beziehen, enthalten sein [BTW99c].

4.6.7 LEAVE Anweisung

Die LEAVE Anweisung kann benutzt werden, um eine gekennzeichnete Ausführung zu beenden. Dies kann eine Blockanweisung oder eine Schleife sein. LEAVE ist allerdings nicht im Zusammenhang mit einer FOR Anwendung einsetzbar. Die Anwendung einer LEAVE Anweisung ist im Beispiel zur LOOP Anweisung dargestellt.

4.6.8 LOOP Anweisung

Die LOOP Anweisung ermöglicht dem Programmierer Schleifen zu realisieren. Beendet wird diese Schleife durch eine LEAVE Anweisung:

```
DECLARE x INTEGER DEFAULT 0;
L1: LOOP
IF x>100 THEN LEAVE L1;
SET x = x + 1;
END LOOP;
```

4.6.9 WHILE Anweisung

Die WHILE Anweisung dient dazu, Gruppen von Anweisungen zu wiederholen, bis eine bestimmte Bedingung erfüllt wird:

```
DECLARE x INTEGER DEFAULT 0;
WHILE x<100 DO
```

```
SET  x = x + 1;
END WHILE;
```

4.6.10 REPEAT Anweisung

Auch die REPEAT Anweisung dient zur wiederholten Ausführung von Anweisungen, allerdings benötigt die REPEAT Anweisung eine End-Bedingung. Die Struktur einer solchen Anweisung ist:

```
REPEAT
        SQL Anweisungen
        UNTIL  Bedingung
END  REPEAT;
```

4.6.11 Condition Handling

Der Condition Handling Mechanismus bietet die Möglichkeit, auf bestimmte Bedingungen, die während der Ausführung eines Programmes auftreten können, zu reagieren. Die Grundidee liegt darin, den Routinen zu erlauben, Systemverhalten wie UNDO, REDO, CONTINUE, EXIT oder ähnliches, zu definieren. Das Grundgerüst ist im folgenden Beispiel aufgezeigt:

```
DECLARE  deklarationen
BEGIN
        SQL Anweisungen
EXEPTION
        WHEN  Exeption_name  OR  Exeption_name
        THEN
                SQL-Code
END;
```

Um ein Condition Handling zu konstruieren, wird ein Mechanismus benötigt, der das Auftreten einer Bedingung anzeigt. Eine Condition kann implizit durch das System oder durch die Benutzung von SIGNAL oder RESIGNAL geworfen werden. Die SIGMAL Anweisung wirft eine Ausnahme für eine getestete Bedingung (SIGNAL falsche_daten). Die Grundform der Ausnahmesignale besteht darin, zu testen, ob eine Bedingung erfüllt ist und dann die Steuerung an den Handler zu geben, um auf die Bedingung reagieren zu können:

```
DECLARE Exeption_name  EXEPTION ... ;
...
IF Bedingung THEN
        SIGNAL Exeption_name
ELSE  RESIGNAL  Exeption_name
END IF;
```

Durch die Deklaration wird die Ausnahme definiert. die SIGNAL Anweisung versucht, dass eine bestimmte Ausnahme geworfen wird. Wenn der Condition Handler die signalisierte Ausnahme nicht bearbeiten kann, dann wird durch die RESIGNAL

Anweisung die Ausnahme erneut geworfen und an den nächst höheren Handler übergeben.
Anweisungen, die in einem Condition Handler definiert werden, können sämtliche SQL Anweisungen beinhalten:

```
BEGIN
DECLARE niedrige_Bilanz CONDITION;
DECLARE CONTINUE HANDLER FOR niedrige_Bilanz
BEGIN
...
END;
...
END
```

Ein Condition Handler wird automatisch ausgeführt, wenn die Bedingung irgendwann während der Ausführung innerhalb des Blockes auftritt. Ein Condition Handler kann als CONTINUE, EXIT oder UNDO spezifiziert werden, wobei UNDO nur innerhalb von ATOMIC Blöcken verwendet werden darf.
Der EXIT Handler erfüllt die Handler Anweisungen. Danach wird die Anwendung außerhalb des Blockes, wo die Exception aufgetreten ist, fortgesetzt.
Der UNDO Handler vollzieht die definierte Handler Anweisungen und setzt alle Änderungen, die in dem Anweisungsblock, der die Exception verursacht hat, vollzogen wurden, zurück. Dies beinhaltet das Zurücksetzen sämtlicher Trigger, die durch den Anwendungsblock aktiviert wurden. Danach wird die Anwendung außerhalb des Blockes, welcher die Ausnahme verursacht hat, fortgesetzt.
Der CONTINUE Handler führt die definierten Handler Anweisungen aus und kehrt an die Stelle im Programm zurück, wo die Ausnahme entstanden ist.

5 Zusammenfassung

Mit dem SQL:1999 Standard wurde ein großer Sprung gemacht, in Hinblick darauf, welche Möglichkeiten einem Datenbankdesigner oder Anwender zur Verfügung stehen. Durch die Erweiterung des Typsystems kann eine Datenbank der zu modellierenden Miniwelt besser angepasst werden. Der ADT stellt hierbei die fundamentalste Neuerung dar.
Durch die Erweiterung der Abfragesprache SQL um einige prozeduale Elemente können nun Funktionen komplett in SQL geschrieben werden. Mit Hilfe der SQL

invoked Routinen sind Anwendungsprogramme leichter zu erstellen, da diese auf die SQL invoked Routinen zugreifen können.
Allerdings sind einige Ansätze des SQL:1999 Standards noch verbesserungsfähig. So wird z.B. in SQL:1999 keine Mehrfachvererbung unterstützt. Des weiteren wird noch keine Datenkapselung innerhalb der abstrakten Datentypen unterstützt. Das Typsystem hat einige wichtige Datentypen hinzugewonnen, ein Datentyp und Operationen für Multimedia Daten sind jedoch noch nicht vorhanden. Der Array Typ stellt den einzigen Collection Typ dar, weitere Typen wie z.B. SET wären von Vorteil. Zudem unterstützt der Array Typ nur eine Dimension und lässt keine Verschachtelung von Arrays zu.
Die Begriffswahl ist in SQL:1999 zum Teil irreführend. So werden im Teil 4 des Standards unter dem Begriff „Persitent stored modules“ sämtliche prozeduale Spracherweiterungen innerhalb SQL zusammengefasst, ohne dass ein erkennbarer Bezug zu den Modulen besteht.
Insgesamt stellt der SQL:1999 Standard ein sehr umfangreiches Werk dar, in dem zahlreiche nützliche Neuerungen beschrieben werden.

Literaturverzeichnis

[ANSI99a] American National Standard for Information Technology: Database Languages SQL Part 2: Foundation, 1999

[ANSI99b] American National Standard for Information Technology: Database Languages SQL Part 4: PSM, 1999

[BTW99] Nelson Mattos, Peter Pistor, Stefan Dessloch: Tutorium anläßlich der 8. Fachtagung BTW `99: Ein Überblick über SQL3 und SQLJ. Freiburg, 1999

[Fo99] Paul Fortier: Implementing the SQL Foundation Standard. McGraw Hill, 1999

[GU99] Peter Gulutzum, Trudy Pelzer: SQL-99 Complete, Really. 1999

[Pa99] Wolfgang Panny, Alfred Taudes: Einführung in den Sprachkern von SQL-99. Springer, 2000