Seminararbeit zum Thema:

OR-Anwendungsentwicklung

Robert Klensch
r_klensc@informatik.uni-kl.de

Arbeitsgruppe Datenbanken und Informationssysteme

Seminar Anwendungsentwicklung
Im Wintersemester 2000/2001

Inhaltsverzeichnis

1 Einleitung 3

2 Call Level Interfaces 3

2.1 Open Database Connectivity 4

2.2 Java Database Connectivity 4

3 Beispieldatenbank 5

4 Anwendung von JDBC und ODBC auf die Beispieldatenbank 6

4.1 ODBC 6

4.1.1 Aufbau einer Verbindung zum DBMS 7

4.2 JDBC 7

4.2.1 Verbindungsaufbau zum Datenbankserver 7

4.2.2 Senden einer SQL-Anweisung an den Datenbankserver 8

4.2.3 Extrahieren der Daten aus dem ResultSet 8

5 SQLJ 9

5.1 Part 0: Embedded SQL in Java 9

5.1.1 Verbindungskontext 10

5.1.2 Ausführungskontext 11

5.1.3 Abspeichern in Java Variablen 11

5.1.4 Aufruf von Funktionen und Prozeduren 11

5.1.5 Iteratoren 12

5.1.6 Mehrere ResultSets 12

5.1.7 Umwandlungen zwischen Iterator und ResultSet 13

5.2 Part 1: SQL Routines using the Java Programming Language 13

5.2.1 Funktionen 13

5.2.2 Prozeduren 15

5.3 Part 2: SQL Types Using the Java Programming Language 16

5.3.1 Verwenden von eigenen Methoden 17

5.3.2 SELF AS RESULT 17

5.3.3 Statische Variablen und Methoden 17

5.3.4 Sortieren 18

5.3.5 Untertypen 18

6 Zusammenfassung 19

7 Ausblick 19

8 Literaturverzeichnis 20

1 Einleitung

Objektorientierte Programmiersprachen, wie Java oder C++, erlangen zunehmende Bedeutung. Aber das Konzept der objektorientierten Vorgehensweise nimmt auch in anderen Bereichen wie den Datenbanken Einzug. Objektorientierte Datenbanken bieten eine gute Basis, um auch den Ansatz der Objektorientierung zu verwenden. Leider bieten diese Systeme noch keine einheitliche Sprache, wie SQL bei relationalen Datenbanke. Objektrelationale Datenbanken sind relationale Datenbanken, die um einige Merkmale (Vererbung, Klassen, Typen und Funktionen) von objektorientierten Datenbanken erweitert wurden.
Im Rahmen dieser Ausarbeitung werden nun einige Techniken vorgestellt, die dem Anwendungsentwickler den Umgang mit objektrelationalen Datenbanken erleichtert. In Kapitel 2 wird ein kurzer Überblick über Call Level Interfaces gegeben. Kapitel 3 stellt eine Beispieldatenbank vor, mit deren Hilfe dann in Kapitel 4 der Zusammenhang von SQL3 und CLI vorgestellt wird. Dabei werden die wichtigsten Neuerungen anhand von Beispielen demonstriert.
Das Kapitel 5 widmet sich SQLJ. Hier werden die drei Teile des SQLJ-Standards kurz angeschnitten und die wichtigsten Merkmalen anhand von kleinen Beispielen demonstriert.

2 Call Level Interfaces

Call Level Interfaces (CLI) sind funktionale Interfaces zwischen Anwendungsprogramm und Datenbankserver (Abb.1). Die Anwendung schickt SQL-Anweisungen nicht direkt an den DBS-Server. Der CLI Call Handler wandelt die SQL-Anweisung dann auf DBMS-spezifische Anweisungen um und führt sie aus. Das Resultat wird dann vom CLI Call Handler an die Anwendung ausgegeben.

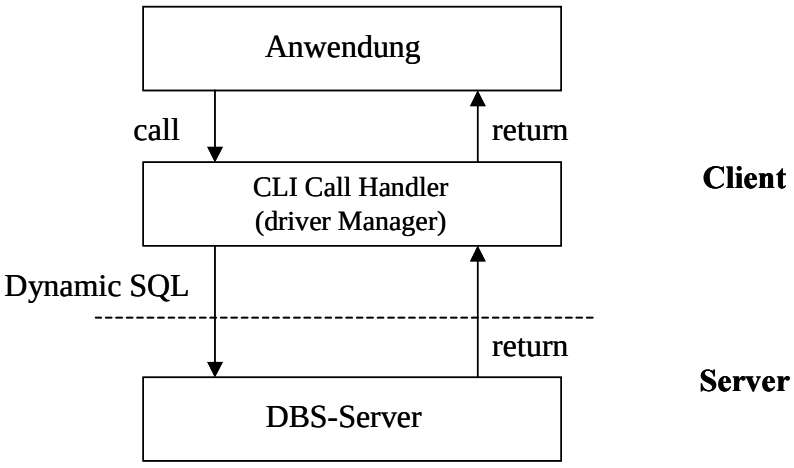

Abb.1.Zugriff auf DBS-Server über das CLI [Hä99]

SQL/CLI wurde 1995 von der ISO verabschiedet und dient als Erweiterung zum SQL2 Standard.
SQL/CLI baut größtenteils auf dem ODBC-Standard von Microsoft auf, der um Vererbung, Polymorphismus etc. an den neuen SQL3 Standard angepasst wurde. Die momentan meistbenutzten CLI sind ODBC von Microsoft sowie JDBC von Javasoft.

Vorteile:(gegenüber embedded SQL)

Anweisungen können während der Laufzeit des Programmes an den Datenbankserver geschickt werden, so dass die SQL-Anweisungen nicht an die Wirtssprache gebunden sind wie beim embedded SQL. Damit fällt die Precompilierung weg, was für kommerzielle Firmen den Vorteil bringt, dass sie den Quellcode für ihre Anwendung nicht offenlegen müssen, während bei embedded SQL für unterschiedliche Datenbankserver auch unterschiedliche Compiler benutzt werden müssen.

Nachteile:(gegenüber embedded SQL)

- komplexere Programmierung

- kein "Precompilierungsvorteil" (also den Vorteil, dass man schon vor der Laufzeit des Programmes mögliche Strukturfehler, wie nicht vorhandene Tabellen, auf die zugegriffen wird, erkennt)

Der CLI-Standard umfasst über 40 Routinen, die sich in folgende Kategorien einteilen lassen:

- Ressourcen-Allokation und -Deallokation
- Ausführen von SQL-Befehlen
- Zugriff auf Diagnoseinformationen
- Transaktionsklammerung
- Informationsanforderung zur Implementierung

2.1 Open Database Connectivity

Open Database Connectivity (ODBC) wurde 1992 von der Firma Microsoft entwickelt. ODBC hat sich mittlerweile zum Quasi-Standard entwickelt und wird von den meisten Datenbankherstellern unterstützt, weshalb sich der SQL/CLI- Standard auch stark an ODBC orientierte.
ODBC ist eine Standard-Anwendungsschnittstelle, die den Zugriff auf Daten in relationalen und auch in nicht-relationalen DBMS ermöglicht. Des weiteren wird das Herunterladen und das anschließende dynamische Einbinden von Treibern ermöglicht, was das Zugreifen auf unterschiedliche DBMS ohne großen Aufwand erlaubt. Wie in Abb. 2, gezeigt verläuft der Zugriff über einen ODBC-Treiber unabhängig von dem DBMS. Mittlerweile wird ODBC nicht nur von den Microsoft-Betriebssystemen Windows 9x und Windows NT unterstützt, sondern auch von MacOS, OS/2, UNIX etc. Es wird von den meisten gängigen Programmiersprachen unterstützt. [Ge96]

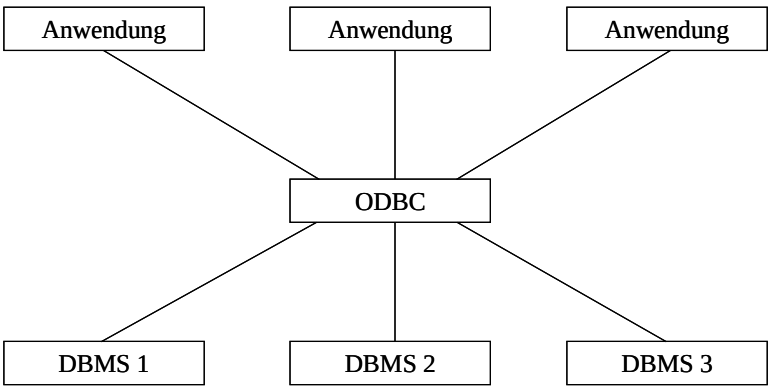

Abb.2. Zugriff auf mehrere DBMS über ODBC [Ge96]

2.2 Java Database Connectivity

Java Database Connectivity (JDBC) wurde von Javasoft entwickelt. Die momentane Version JDBC 2.0 wurde 1998 veröffentlicht. JDBC ist ähnlich aufgebaut wie ODBC und orientiert sich an dem SQL/CLI-Standard. JDBC ist optimal für Java angepasst und unterstützt auch Java-Exceptions.
Es gibt 4 mögliche Architekturen, die von JDBC-Treibern zum Zugriff auf DBMS benutzt werden (siehe Abb. 3):
Typ 1 - JDBC-ODBC Bridge
Hier werden von JDBC über ODBC Befehle an ein DBMS geschickt. Dafür benötigt man einen außerhalb von der Java-Umgebung installierten ODBC-Treiber.
Typ 2 - Native-API Partly Java Driver
Hier wird von JDBC über eine clientseitige DBMS-spezifische Anwendungsschnittstelle auf das DBMS zugegriffen.
Typ 3 - Net-Protocol All-Java Driver
Bei dieser Architektur werden die Nachrichten über ein vom DBMS unabhängiges Protokoll zum Server geschickt, wo sie dann von einem Prozess entgegengenommen und als Aufrufe an das DBMS gesendet werden. Hierbei kann auf Client-Seite vollständig auf Java zurückgegriffen werden.
Typ 4 - Native-Protocol All-Java Driver

Hierbei handelt es sich um eine ähnliche Architektur wie die des Typs 3, mit dem Unterschied, dass das Protokoll, über das die Nachrichten verschickt werden, DBMS-spezifisch ist, was sehr effizient ist. [ME00]

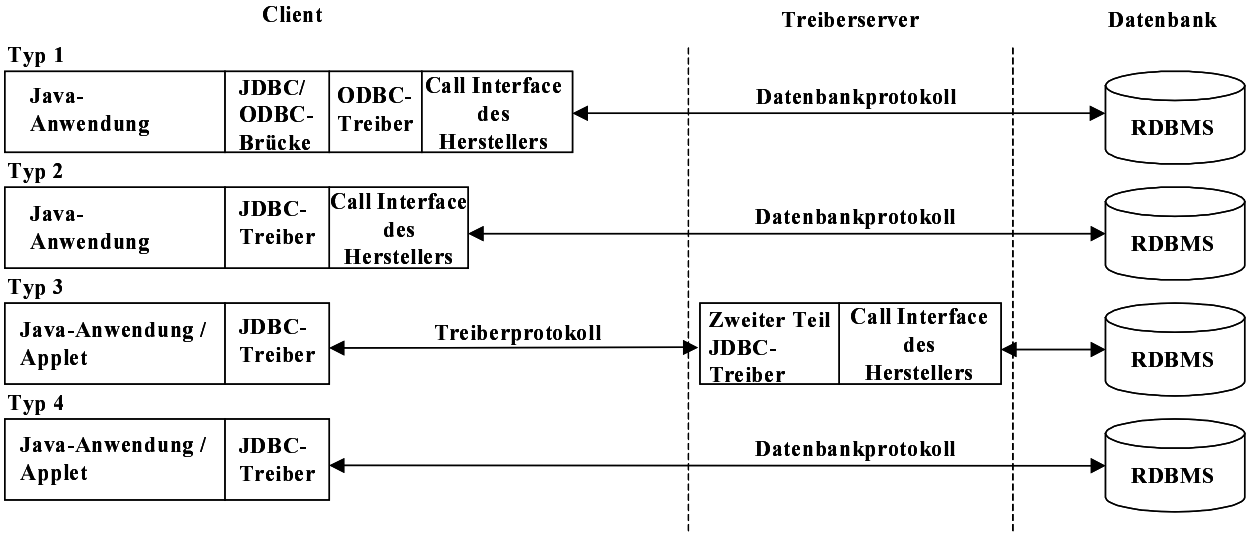

Abb.3 illustriert die vier JDBC-Treiber. [ME00]

3 Beispieldatenbank

Hier soll nun eine fiktive Datenbank kreiert werden, auf die später mit ODBC respektive mit JDBC zugegriffen werden soll. Das Beispiel dient hauptsächlich dazu, das Verhalten von JDBC und ODBC bezüglich der Datentypen ARRAY, DISTINCT TYPE, STRUCTERED TYPE, REF, BINARY LARGE OBJECTS, die teilweise aus dem neuen SQL3 Standard hervorgegangen sind, aufzuzeigen.
Die Datenbank verwendet all diese Datentypen und soll zum Abspeichern von CD-Informationen dienen, wobei hier folgende Attribute gelten:

```
Albumname    Varchar(50)
```

„Albumname“ bezeichnet den Namen eines Albums eines Musikers (der Einfachheit halber werden nur Musiker, keine Gruppen, betrachtet).

```
Titel ARRAY OF TitelStruktur
```

Eine TitelStruktur besteht aus einem Varchar(30), der den Titelnamen eines Liedes angibt, und einem Attribut Playtime vom Datentyp time, das die Spielzeit angibt . In der Tabelle gibt es zu jeder CD eine solche Menge von Titelstrukturen (eine CD enhält in der Regel mehrere Songs), dazu wird der ARRAY Datentyp verwendet.

```
Preis        DM
```

Für das Attribut Preis wird ein DISTINCT TYPE von double benutzt.

```
Coverbild    BLOB
```

Das Coverbild wird als Binary Large Object abgespeichert.

```
Musiker      REF(KuenstlerTabelle)
```

Die Daten über den Musiker, von dem das Album ist. Datentyp dieses Attributs ist eine Referenz auf eine andere Tabelle, die die Attribute Name, Geburtsdatum und Nationalität enthält.
Die einzelnen zusammengesetzten Datentypen in SQL Notation:

Datentyp TitelStruktur

```
CREATE TYPE TitelStruktur AS (
      Titelname   VARCHAR(100),
      Spielzeit       TIME
)
NOT FINAL;
```

Datentyp DM

```
CREATE TYPE DM AS DOUBLE FINAL;
```

Datentyp KuenstlerType

```
CREATE TYPE KuenstlerType AS (
      Name        VARCHAR(50),
      Geburtsdatum      DATE,
      Nationalitaet          VARCHAR(50)
)
NOT FINAL
REF IS SYSTEM GENERATED;
```

Folgende Tabellen sind in der Datenbank:

Tabelle KuenstlerTabelle

```
CREATE TABLE KuenstlerTabelle OF KuenstlerType (
      REF IS aref SYSTEM GENERATED );
```

CDTabelle

```
CREATE TABLE CDTabelle (
      Albumname   VARCHAR(50) PRIMARY KEY,
      Titel       TitelStruktur ARRAY[20],
      Preis       DM,
      Coverbild   BLOB(1M),
      Musiker     REF(KuenstlerTabelle)
);
```

4 Anwendung von JDBC und ODBC auf die Beispieldatenbank

Im Folgenden wird nun über JDBC und ODBC auf die oben definierte Beispieldatenbank zugegriffen. Es werden jeweils der Verbindungsaufbau und anhand von Daten aus der Beispieldatenbank die Anbindung der neuen SQL3-Datentypen an die jeweilige Programmiersprache aufgezeigt. Für JDBC wird Java und für ODBC C als Programmiersprache verwendet.

4.1 ODBC

Im Folgenden sollen einige Datensätze aus der Beispieldatenbank über ODBC ausgelesen werden. Als Version von ODBC wird die Version 3.0 betrachtet:
Folgende Variablen werden dabei benutzt:

```
/* DSN */
      char * dsn;

/* Handles */
      SQLHDBC     hdbc;
      SQLHENV     henv;
```

```
      SQLSTMT     hstmt;

/* Andere Variablen */
      SQLRETURN rc = 0;
      SQLINTEGER cbPreis,cbAlbumname,cbBild;
      char* selectStmt = "SELECT Albumname,Titel,Preis,Coverbild,Musiker
      FROM CDTabelle WHERE Albumname='Fields of Gold'";
      char Albumname[100];
      double Preis;
```

4.1.1 Aufbau einer Verbindung zum DBMS

```
/* dsn für den Verbindungsaufbau */
      dsn = "odbc_demo";
/* Environment-Handler anbinden */
      rc = SQLAllocHandle (SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
/* ODBC Version wird auf 3.0. gesetzt */
      rc = SQLSetEnvAttr(henv,SQL_ATTR_ODBC_VERSION,
      (SQLPOINTER)SQL_OV_ODBC3,0);
/* Verbindungs-Handler anbinden. */
      rc = SQLAllocHandle (SQL_HANDLE_DBC, henv, &hdbc);
/* Verbindung aufbauen. */
      rc = SQLConnect (hdbc, dsn, SQL_NTS, "", SQL_NTS, "", SQL_NTS);
/* Statement-Handle anbinden. */
      rc = SQLAllocHandle (SQL_HANDLE_STMT, hdbc, &hstmt );
/* Ausführen der SQL-Anfrage */
      rc = SQLExecDirect (hstmt, selectStmt, SQL_NTS);
/* Anbinden der Spalten des ResultSets an Variablen. */
      rc = SQLBindCol (hstmt, 1, SQL_C_CHAR, Albumname, 100,
      &cbAlbumname);
      rc = SQLBindCol (hstmt, 3, SQL_C_DOUBLE, &Preis, 5, &cbPreis);
/* Daten auswählen und ausgeben. */
      while (1)
      {
       rc = SQLFetch (hstmt);
       if (rc = SQL_NO_DATA_FOUND)
             break;
       else
             /* Fehlerbehandlung */
      }
```

4.2 JDBC

Die benutzte Version von JDBC ist 2.0, da diese erst entstand nachdem der SQL3-Standard festgeschrieben wurde, und somit schon viele der neuen SQL3 Besonderheiten unterstützt.

4.2.1 Verbindungsaufbau zum Datenbankserver

```
      Connect con = null;
```

Über das con-Object, das Instanz von der Klasse Connect und eine Art Referenz zur Datenbank ist, kann später auf die Datenbank zugegriffen werden.

```
      String URL = "jdbc:sybase:TDS:localhost:2244";
```

Im String URL ist das Unterprotokoll sowie Adresse, Port, ... gespeichert. Dieser String sollte immer mit jdbc: beginnen, dann folgt das Unterprotokoll und ein vom Unterprotokoll abhängiger Teil.

```
      try {
            Class.forName("com.sybase.jdbc.SybDriver");
```

```
}
catch (ClassNotFoundException cnf) {
      System.out.println(cnf.getMessage());
 return;
}
```

"com.sybase.jdbc.SybDriver" ist der Treibername. Dieser wird später bei der Instanzierung des Driver-Managers benutzt, der dann über jede der im Treibernamen angegebenen Klassen versucht, eine Verbindung mit der Datenbank aufzubauen.

```
try {
      con = DriverManager.getConnection (URL,"user","password");
}
catch (SQLException sql) {
      System.out.println(sql.getMessage());
      return;
}
```

Hier wird nun eine Connection-Instanz gebildet und con zugewiesen. Somit kann über con jetzt auf die Datenbank zugegriffen werden. Dabei wird dem DriverManager durch die getConnection Methode die Parameter URL, user, password mitgeteilt und dieser versucht nun, mit Hilfe dieser Parameter und den Treibern eine Verbindung aufzubauen.

4.2.2 Senden einer SQL-Anweisung an den Datenbankserver

Nachdem die Verbindung aufgebaut ist, soll eine Anweisung an den Datenbankserver geschickt werden.

```
Statement stmt = con.createStatement();
```

stmt wird nun ein Statementobjekt zugewiesen. Über dieses können dann SQL-Anweisungen verschickt werden.

```
String stmtSource = "SELECT Albumname, Titel, Preis, Coverbild,
Musiker
      FROM CDTabelle
      WHERE Albumname = 'Fields of Gold' ";
ResultSet rs = stmt.executeQuery(stmtSource);
```

executeQuery übergibt den String stmtSource an stmt. Über stmt wird er an die Datenbank geschickt und dort ausgeführt. Das Resultat wird als Instanz vom Datentyp ResultSet zurückliefert und rs zugewiesen.

4.2.3 Extrahieren der Daten aus dem ResultSet

Der so erhaltene ResultSet rs enthält nun die Reihen aus der Datenbank, die der Anfrage genügen, in unserem Beispiel nur eine Reihe. Des Weiteren wollen wir nun eine solche Reihe aufspalten, die darin enthaltenen Informationen extrahieren und sie in Java-Datentypen speichern.

```
while (rs.next()) {
      JAlbumname = rs.getString("Albumname");
```

Die next()-Methode liefert true zurück, falls es noch weitere Resultate gibt, andernfalls false. rs.getString("Albumname") liefert einen String zurück, der in unserem ResultSet unter der Spalte mit dem Namen "Albumname" gespeichert ist. Dieser entspricht dem Spaltennamen der Datenbank. Die getString Methode erwartet, dass die Daten im ResultSet mit dem String Datentyp in Java korrespondieren. Im Falle der Beispieldatenbank entspricht der SQL-Datentyp VARCHAR dem String-Datentyp. Eine andere Möglichkeit, die getString Methode zu benutzen, wäre mit einem Integer-Parameter als Eingabe, der der Spaltennummer in der Datenbank enspricht.

```
Array JTitel = rs.getArray(2);
```

Hiermit wird das SQL-Array aus Spalte 2 mit dem korrespondierenden Array-Datentyp in Java verbunden.

```
Struct[] structArray = (Struct[]) JTitel.getArray();

for (int i=0;i < structArray.length;i++) {
      Object[] Attribute = structArray[i].getAttributes();
      for (int j=0; j<Attribute.length;j++) {
            System.out.println(Attribute[j]);
      }
}
```

Mit dieser for-Schleife wird das StructArray durchlaufen. Auf jedes seiner Elemente vom Typ Struct wird die Methode getAttributes() angewendet, die ein Object[] zurückliefert, das alle Attribute des Structs enthält, also hier die Attribute Titelname und Spielzeit. Diese Attribute sind in einem Array, das wir mit der zweiten for-Schleife durchlaufen. Hier ist Vorsicht geboten, da wir die Werte ausgeben, ohne die Datentypen der Attribute zu kennen. Bei der Ausgabe wird jedoch die toString Methode aufgerufen, falls vorhanden, oder eine standard-toString-Methode, was unter Umständen zu Problemen führen kann. In unserem Beispiel gibt es hier aber kein Problem, da wir korrespondierende Datentypen zu den SQL-Datentypen in Java haben. Hier hätte man statt einer Ausgabe auch die Werte auf Java-Datentypen casten und in Variablen abspeichern können.

```
Blob Coverbild = rs.getBlob(4);
long CoverbildLaenge = Coverbild.length();
```

Mit getBlob (Spaltennummer oder Spaltenname) kann man das Blob aus dem ResultSet extrahieren und in einem Java-Blob-Datentyp abspeichern. Das Blob kann dann weiterverarbeitet werden.

```
      double JPreis = rs.getDouble("Preis");
      Ref JRef = rs.getRef("Musiker");
}
```

Die Reference-Typen sind etwas unhandlicher, da hier in der Datenbank eine Referenz auf einen structured-Typ angelegt wird, über die man innerhalb von Java nicht auf die Attribute zugreifen kann. Man muss umständlich eine weitere SQL-Anweisung an die Datenbank schicken und über diesen Weg die Daten erhalten.

5 SQLJ

Als Literaturquelle diente für das ganze Kapitel [ME00].SQLJ wurde von namhaften Herstellern wie Informix, IBM, Microsoft etc. als Standard für SQL mit Javaprogrammen definiert. Im Standard sind die unterschiedlichen Arten beschrieben, mit denen man SQL in Javaprogramme einbetten kann, wie „stored procedures“ mit Java implementiert werden können und wie SQL-Datentypen durch Java-Klassen als Datentypen ergänzt werden. Der SQL-Standard wurde dementsprechend in drei Teile unterteilt.

5.1 Part 0: Embedded SQL in Java

In diesem Teil wird auf die Einbettung von SQL in Java als Embedded-SQL eingegangen. Ein Generator, der SQLJ-Translator, erzeugt aus den Quelltexten ein übersetzbares Programm. Dieses kann dann ganz normal mit dem Javacompiler kompiliert werden kann. Abb.4 zeigt den ganzen Ablauf. Der SQLJ-Translator kann, falls eine Verbindung mit dem DBMS besteht, eine Semantikprüfung durchführen. Dabei wird geprüft, ob die Befehle zum Schema der Datenbank passen. Bei erfolgreicher Komprimierung wird ein normales Java-Programm und einige profile-Dateien generiert. Diese profile-Dateien (Profiles) enthalten die Verbindungskontexte sowie eine vollständige Beschreibung der SQL-

Anweisungen. Ein Profile enthält mehrere Java-Instanzen. Die Profiles können mit herstellerspezifischen „customizer“ nachträglich an ein bestimmtes DBMS angepasst werden, so dass eigene, von der JDBC-API unabhängige, Funktionsaufrufe verwendet werden können. Mit Hilfe von Profiles können so Programme optimal an ein DBMS angepasst werden und Leistungssteigerungen erreicht werden. Dabei bietet SQLJ eine ganze Reihe von Vorteilen gegenüber JDBC :

- kurze und prägnante Schreibweise
- komfortables Programmieren von Datenbankanfragen
- Syntaxprüfung zur Übersetzungszeit
- Semantikprüfung (Vergleiche mit bestehenden Datenbankschemata) zur Übersetzungszeit Anpassbarkeit an ein Zieldatenbanksystem, ohne den Quellcode zu ändern.

Ein großer Nachteil jedoch ist, dass das eingebettete SQL statisch ist. Die SQL-Befehle stehen schon zur Übersetzungszeit fest.
Obwohl SQLJ Part 0 auch ohne JDBC auskommt, besteht die Möglichkeit, beide zu kombinieren und damit gleichzeitig SQL-Anweisungen und JDBC-Anweisungen in einem Programm zu verwenden.

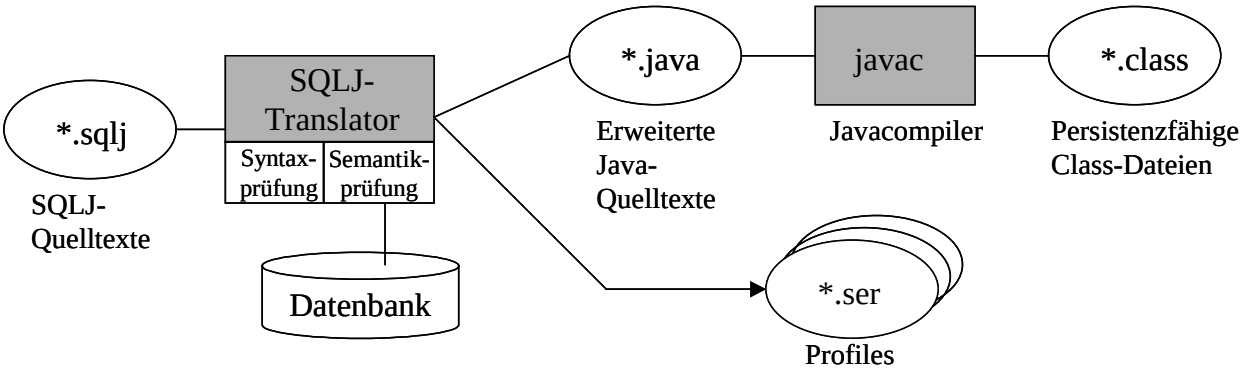

Abb.4. SQLJ-Translator [ZB00]

5.1.1 Verbindungskontext

```
try {
      Class.forName("com.sybase.jdbc.SybDriver");
      String URL = "jdbc:sybase:TDS:localhost:2244";
```

Wie bei JDBC werden auch hier die JDBC Treiber in gleicher Weise eingebunden.

```
DefaultContext defCtxt =
      new DefaultContext(URL,"user","password",false);
DefaultContext.setDefaultContext(defCtxt);
```

Das Verbindungskontext-Objekt dient zum Managen aller relevanten Informationen bezüglich der Datenbank. Hiermit werden alle SQL-Anweisungen mit einer Datenbankverbindung verknüpft. Die Verbindungskontext-Objekte können, wie nachfolgend angegeben, auch explizit definiert werden:

```
/*    #sql context ContextName;
      ...
      ContextName ContextNameInstanz = new
ContextName(URL,"user","password",false);
      ...
      #sql [ContextNameInstanz] {
                  <SQL-Anweisung>
      } */
```

Mit Hilfe dieses expliziten Typs hat man die Möglichkeit, mehrere Verbindungen zu einer oder zu unterschiedlichen Datenbanken aufzubauen und sie über den Namen zu identifizieren. Dies ermöglicht es auch, die SQL-Anweisungen während der Übersetzungszeit zu überprüfen.

```
#sql { DELETE
         FROM CDTabelle
         WHERE    Albumname='Fields of Gold'
       };
```

Nach dem #sql kommen embedded SQL-Anweisung, wobei hier Groß- und Kleinschreibung für die Spaltennamen, Tabellennamen,... keine Rolle spielen (im Gegensatz zu den Variablennamen in Java).

```
}
catch(SQLException sqlex) {
      System.out.println(sqlex.getMessage());
}
catch (Exception ex) {
      System.out.println(ex.getMessage());
}
```

Man bemerkt den Unterschied zu anderen Sprachen mit embedded SQL, wo normalerweise Fehlermeldungen über die Variable SQLSTATE übermittelt werden, während dies hier über eine ganz normale SQL-Exception behandelt wird.

5.1.2 Ausführungskontext

Neben den Verbindungskontext-Objekten gibt es noch die Ausführungskontext-Objekte, die weitere Optionen bereitstellen, wie:

- Definieren von Timeouts, nach dem die Anweisungen gestoppt und eine Exception geworfen wird,
- maximale Anzahl von Reihen, die nach der Ausführung der Anweisung zurückgeliefert werden sollen,
- maximale Feldgröße eines Attributes,
- Informationen über die von der SQL-Anweisung manipulierten Reihen.

5.1.3 Abspeichern in Java Variablen

Java-Programmvariablen können auch in SQL-Anweisungen benutzt werden. Im nachfolgenden Beispiel wird die Java-Variable JAlbumname in der SQL-Anweisung benutzt, indem das Resultat der „SELECT“-Anweisung dieser zugeordnet wird. Werte von Variablen (zum Beispiel von „i“) können auch als Eingabe verwendet werden.
Beispiel:

```
String JAlbumname;
int i=35;
#sql { SELECT   Albumname
       INTO     :JAlbumname
       FROM   CDTabelle
       WHERE      Preis = :i
};
```

5.1.4 Aufruf von Funktionen und Prozeduren

```
Datentyp1 inVariable = WERT;
Datentyp2 outVariable;
# sql { CALL prozedur ( :in (inVariable),:out (outVariable))
};
```

Datentyp steht allgemein für einen beliebigen Datentyp, WERT für einen Datentyp-spezifischen Wert. Die Prozedur wird einfach durch CALL aufgerufen und mit :in werden Eingabevariablen und mit :out Ausgabevariablen definiert .

```
Datentyp3 funktionsWert;
#sql funktionsWert = {VALUES (funktion (:in (inVariable)))};
```

VALUES liefert den Funktionswert zurück, der dann in funktionsWert gespeichert wird (Datentyp3 sollte hier kompatibel sein mit dem von der Funktion zurückgelieferten Wert).

5.1.5 Iteratoren

Iteratoren können mit Cursorn in SQL verglichen werden. Es gibt zwei verschiedene Iteratorentypen, die nicht vermischt werden dürfen.
1. Named iterator

Beim „named iterator“ werden die Spalten des Resultats über den Namen referenziert
Beispiel:

```
#sql iterator CdIterator (String Albumname, double Preis);
...
CdIterator cditerator1;
#sql cditerator1 = { SELECT Albumname,Preis
                     FROM CDTabelle
                     WHERE Musiker->name ='Sting' };
...
while (cditerator1.next()) {
      System.out.println(cditerator1.Albumname()+"kostet :
                          "+cditerator1.Preis()+"DM");
}
cditerator1.close();
```

2. Positioned iterator

Beim „positioned iterator” werden bei der Iteratordeklaration nur die Datentypen angegeben. Die Anbindung an Variablen erfolgt dann durch „FETCH Iteratorname INTO :Variablenname_1 ... :Variablenname_n“, wobei die Reihenfolge der Datentypen der Variablen (Variablenname_1 ... Variablenname_n) sich mit der Iteratordeklaration decken muss.
Beispiel:

```
#sql iterator CdIterator2(String,double);
....
CdIterator cditerator2;
String JAlbumname;
Double JPreis;
#sql cditerator2 = { SELECT Albumname,Preis
                     FROM CDTabelle
                     WHERE Musiker->name ='Sting' };
while(true) {
            #sql {FETCH :cditerator2 INTO :JAlbumname,JPreis};
            if (cditerator2.endFetch()) break;
            System.out.println(Albumname+"kostet :"+JPreis+"DM");
}
cditerator2.close();
```

5.1.6 Mehrere ResultSets

In SQLJ Part 0 ist die gleichzeitige Behandlung von mehreren ResultSets nicht vorgesehen, deshalb muss man auf JDBC ausweichen.

```
ExecutionContext execCont = new ExecutionContext();
#sql [execCont] {CALL Prozedur() };
ResultSet rs;
while ((rs = execCont.getNextResultSet()) = null) {
      while(rs.next()) {
            System.out.println(rs.getString(1));
      }
}
```

5.1.7 Umwandlungen zwischen Iterator und ResultSet

Da in SQLJ Part 0 die Möglichkeit gegeben ist, JDBC zu verwenden, muss es möglich sein, einen Iterator in einen ResultSet sowie einen ResultSet in einen Iterator zu verwandeln.
Beispiel für einen Iterator, der aus einem ResultSet hervorgeht:

```
#sql iterator CdIterator(String Albumname);
...
String rsQuery = "SELECT Albumname FROM CDTabelle
                  WHERE Preis < 35"
DefaultContext defCtxt = new
      DefaultContext(URL,"user","password",false);
PreparedStatement ps =
      defCtxt.getConnection.prepareStatement(rsQuery);
ResultSet rs = ps.executeQuery();
CdIterator cditerator;
#sql cditerator = {CAST :rs};
while(cditerator.next()){
      System.out.println("Albumname: "+cditerator.Albumname());
}
```

Beispiel für einen ResultSet, der aus einem Iterator hervorgeht:

```
#sql iterator CdIterator(String Albumname);
...
CdIterator cditerator;
#sql cditerator= {SELECT Albumname FROM CDTabelle
                  WHERE Preis < 35};
ResultSet rs =cditerator.getResultSet();
while(rs.next()){
System.out.println("Albumname: "+rs.getString(1));
}
```

5.2 Part 1: SQL Routines using the Java Programming Language

Dieser Teil des SQLJ-Standards wurde 1999 verabschiedet und sieht die Benutzung von Java Code in DBMS vor. Man kann also sogenannte "Stored Procedures" komplett in Java spezifizieren und sie dann als Jar-Archiv (.jar) in die Datenbank integrieren, wo sie SQL-Anweisungen innerhalb des DBMS benutzen können und von außerhalb aufgerufen werden können. Semantisch gesehen gibt es keine Unterschiede zwischen herkömmlichen und durch Java Methoden gebildete Prozeduren und Funktionen, sie werden gleichermaßen aufgerufen, so dass man keinen Unterschied merkt.
Voraussetzung für die Benutzung von Jar-Archiv Dateien in dem DBMS ist eine Java Virtual Machine (JVM).

5.2.1 Funktionen

Hier eine Funktion an einem Beispiel:
Es soll eine Funktion in die Beispieldatenbank integriert werden, die einen gegebenen Preis von DM in Dollar umwandelt.

```
public static double DMinDollar(double DM) {
      return(DM/2,17);
}
```

In SQL:

```
CREATE FUNCTION SQLDMinDollar (DOUBLE) RETURN DOUBLE
EXTERNAL NAME 'CDTabelle_jar:CDTabelle.DMinDollar(double)'
LANGUAGE JAVA
PARAMETER STYLE JAVA
```

Um jetzt diese Funktion aufzurufen, genügt eine SQL-Anweisung wie zum Beispiel:

```
SELECT       Albumname,SQLDMinDollar(Preis)
FROM  CDTabelle
WHERE Musiker->Name='Sting'
```

Diese Anweisung liefert alle Sting-Alben inklusive Preisangabe in Dollar zurück.
Auch hier müssen die von der Methode zurückgelieferten Werte auf SQL-Datentypen abbildbar sein, Methoden vom Typ void sind als Funktionen nicht zulässig.

Java Methoden, die SQL-Anweisungen ausführen

Java-Methoden, die SQL-Anweisungen ausführen, müssen zuerst eine Verbindung mit dem DBMS aufbauen. Dies kann wahlweise über JDBC oder über SQLJ Part 0 geschehen. Wie bei JDBC muss auch hier eine URL des DBMS angegeben werden, dabei kennzeichnet "jdbc:default:connection" eine Standardverbindung, und der AUTOCOMMIT Parameter ist standardmäßig auf false gesetzt.
Hierzu nun ein Beispiel:
Das nachfolgende Beispiel zeigt eine Funktion, die als Eingabe einen Künstlernamen erhält und als Resultat die Anzahl der Alben dieses Künstlers zurückliefert.

```
public static int ZaehleAlben (String Kuenstlername)
            throws SQLException {
      int Anzahl =0;
      String stmt = "SELECT COUNT(*) "
                  +" FROM CDTabelle"
                  +" WHERE Musiker->Name = "+"'"+Kuenstlername+"'";
      Connection con =
            DriverManager.getConnection("jdbc:default:connection");
```

Hier sieht man den Unterschied zwischen dem Verbindungsaufbau in einem "normalen" Programm und innerhalb des DBMS.

```
  ResultSet rs = con.createStatement().executeQuery(stmt);
  rs.next();
  Anzahl = rs.getInt(1);
  return Anzahl;
}
```

In SQL:

```
CREATE FUNCTION SQL_ZAEHLEALBEN(IN Kuenstlername VARCHAR(50))
RETURNS INTEGER
EXTERNAL NAME 'CDTabelle_jar:CDTabelle.ZaehleAlben'
LANGUAGE JAVA
PARAMETER STYLE JAVA
```

Bemerkung : Allgemein kann man Charakteristiken angeben. Wenn man keine Charakteristik angibt, wird standardmäßig „MODIFIES SQL DATA“ angenommen, was fast alle Lese- und Schreiboperati-

onen erlaubt. Daneben gibt es noch „NO SQL“ (überhaupt kein SQL), „CONTAINS SQL“ (erlaubt das Ausführen von SQL-Statements, die nicht auf die Datenbank zugreifen) und „READS SQL DATA“ (nur Leseanweisungen wie select und fetching von Daten über einen Cursor). Die Charakteristiken dienen zur Optimierung.

5.2.2 Prozeduren

Prozeduren können in SQL einen "OUT" Parameter oder "INOUT" Parameter benutzen, der die Werte mit "call-by-reference" übergibt. Da Java jedoch weder "call-by-reference" noch Pointer unterstützt, muss dies auf eine ganz spezielle Art gelöst werden. Man übergibt die Variable als Array von dem Datentyp, den man übergeben will, und speichert den Wert im ersten Element (Index 0) ab.
Hierzu ein Beispiel:

```
public static void DMinDollar (double DM,double[] Dollar) {
      Dollar[0] = DM / 2.17;
}
```

In SQL würde diese Methode dann folgendermaßen integriert werden:

```
CREATE PROCEDURE DMinDollar (IN DOUBLE, OUT DOUBLE)
EXTERNAL NAME 'CDTabelle_jar:CDTabelle.DMinDollar '
LANGUAGE JAVA
PARAMETER STYLE JAVA
```

ResultSets in Prozeduren

Wie schon bei den Funktionen gesehen, können die Methoden auch SQL-Anweisungen ausführen und dabei ein ResultSet zurück bekommen. Wie diese in Prozeduren behandelt werden, sehen wir in nachfolgendem Beispiel:
Das Beispiel umfasst eine Prozedur, die alle Alben eines bestimmten Künstlers zurückliefert.

```
public static void Alben (String Name,ResultSet[] rs)
            throws SQLException {
      String stmt = "SELECT Albumname"
                  +"FROM CDTabelle"
                  +"WHERE Musiker->Name = "+"'"+Name+"'";
      Connection con =
            DriverManager.getConnection("jdbc:default:connection");
      rs[0] = con.createStatement().executeQuery(stmt);
}
```

SQL:

```
CREATE PROCEDURE SQL_Alben (IN NAME VARCHAR(20))
READS SQL DATA
DYNAMIC RESULT SETS 1
EXTERNAL NAME 'CDTabelle_jar:CDTabelle.Alben(String,ResultSet[]) '
LANGUAGE JAVA
PARAMETER STYLE JAVA
```

Es können auch mehrere ResultSets zurückgeliefert werden.

Nullwerte

Wird dem primitiven Datentyp „int“ der Wert „null“ zugewiesen, so gibt es eine Exception. Um dies zu vermeiden, gibt es zwei Möglichkeiten:

```
1) CREATE FUNCTION Funk1 (INTEGER) RETURNS VARCHAR(10)
EXTERNAL NAME 'CDTabelle_jar:CDTabelle.JavaFunk1(Integer)'
      ...
```

Neben dem primitiven Datentypen int gibt es noch die Klasse Integer. Ihrer Instanz kann der Wert null zugewiesen werden, ohne dabei eine Exception zu erhalten. Jeder Wert, der ungleich null ist, kann dann durch den Methodenaufruf „intValue()“ auf einen primitiven int Wert zurückgeführt werden. Der Vorteil besteht darin, dass man den Integer-Wert auf „null“ prüfen kann und nur bei Werten ungleich „null“ eine Umwandlung nach int durchführt oder eine Ausnahmebehandlung macht.

```
2)CREATE FUNCTION Funk2 (INTEGER) RETURNS VARCHAR(10)
  RETURNS NULL ON NULL INPUT
  EXTERNAL NAME 'CDTabelle_jar:CDTabelle.JavaFunk2'
  ...
```

Wird Funk2 mit einem Nullwert aufgerufen, so kommt ein Nullwert zurück, ohne dass dabei die Methode aufgerufen wird. Hier kann der Parameter von JavaFunk2 als int deklariert werden, da nur Werte ungleich „null“ an die Methode übergeben werden und so keine Exception auftritt.

5.3 Part 2: SQL Types Using the Java Programming Language

Der wesentliche Aspekt dieses Teiles des SQLJ-Standards ist das Abbilden von Java-Klassen auf SQL-Datentypen, so dass man direkt Java-Objekte in dem DBMS speichern kann. In serverseitigem Java können so beliebige Objekte als Parameter verwendet werden.
Hierfür sollten die Klassen in einem Jar-Archiv File in die Datenbank eingebunden werden. Im Beispiel könnte diese Datei in der Datenbank CDTabelle_jar heißen. Weiter müssen die Klassen das Interface Serializable oder java.sql.SQLDATA implementieren oder beide. Alle in der Klasse verwendeten Klassenmember müssen als "public" deklariert werden.
Ein Beispiel:
Als Beispiel soll nun die Beispieldatenbank um einen weiteren Datentyp erweitert werden, der in dem Jar-File in der Datenbank als Klasse Kommentar abgelegt ist und der dazu dient, eine Kritik zu einem Album abzulegen.

```
package CDTabelle;
import java.util.*;
import java.io.*;
import java.sql.*;

public class Kommentar implements java.io.Serializable {
      public String Name;
      public String Kommentar;

      public Kommentar(String Name,String Kommentar) {
            this.Name = Name;
            this.Kommentar = Kommentar;
      }
}
```

Um diesen neuen Datentyp nun in SQL zu integrieren, muss Folgendes getan werden:

```
CREATE TYPE SQL_Kommentar
      EXTERNAL NAME 'CDTabelle_jar:CDTabelle.Kommentar '
      LANGUAGE JAVA
      ( Name            VARCHAR(20) EXTERNAL NAME 'Name'
        Kommentar VARCHAR(100) EXTERNAL NAME 'Kommentar'

      METHOD SQL_Kommentar(v1 VARCHAR(20), v2 VARCHAR(100))
            RETURNS SQL_Kommentar
            EXTERNAL NAME 'Kommentar'
      )
```

Die CDTabelle in der Datenbank wird nun um ein weiteres Attribut Kommentar dieses neuen Datentyps erweitert.
Um nun einen Wert einzufügen, als Beispiel zum Album "Fields of Gold", muss Folgendes getan werden:

```
INSERT INTO CDTabelle(Kommentar) VALUES (NEW
SQL_Kommentar('MR.X','sehr gutes Album')) WHERE Albumname = 'Fields
of Gold';
```

5.3.1 Verwenden von eigenen Methoden

Ein Beispiel:

```
public class Kommentar2 implements java.io.Serializable {
      ...
      public String toString() {
return( "Der Kommentar von"+Name+"zu diesem Album :"+Kommentar);
      }
}
```

In SQL würde der Datentyp nun so aussehen:

```
CREATE TYPE SQL_Kommentar
      EXTERNAL NAME 'CDTabelle_jar:CDTabelle.Kommentar2'
      LANGUAGE JAVA
      ( ...

      METHOD SQLtoString()
            RETURNS VARCHAR(200)
            EXTERNAL NAME 'toString'
      )
```

Die Methode könnte folgendermaßen aufgerufen werden:

```
SELECT      Kommentar.SQLtoString
FROM CDTabelle
WHERE Albumname = 'Fields of Gold'
```

Resultat: "Der Kommentar von MR.X zu diesem Album: sehr gutes Album"

5.3.2 SELF AS RESULT

Wenn man auf Objekten Veränderungen durchführt und die Methode das veränderte Objekt zurückliefert, dann geht die Referenz auf das Objekt verloren und das Objekt wird nach kurzer Zeit von der JVM gelöscht, damit wäre die Änderung nicht wirksam. Um nun das Objekt an denjenigen zurückzuschicken, der die Methode aufgerufen hat, muss das Schlüsselwort "SELF AS RESULT" angegeben werden.

5.3.3 Statische Variablen und Methoden

Hierzu ein Beispiel:

```
public class Kommentar implements java.io.Serializable {
...
      public static String Irgendwas (Integer Zahl) {
            String Resultat;
            ...
            return (Resultat);
      }
}
```

In SQL sähe dies folgendermaßen aus:

```
CREATE TYPE SQL_Kommentar
      EXTERNAL NAME 'CDTabelle_jar:CDTabelle.Kommentar'
      LANGUAGE JAVA
      (     ...
            STATIC METHOD SQL_Irgendwas(INTEGER)
            RETURNS VARCHAR(20)
            EXTERNAL NAME 'Irgendwas'
      )
```

Diese statische Methode könnte man in SQL nun wie folgt einsetzen

SET irgendwas = SQL_Kommentar::SQL_Irgendwas(12).
Um auf statische Variablen zuzugreifen, zum Beispiel auf die Variable: public final static int preis = 35, müsste man in SQL Folgendes tun:

```
CREATE TYPE SQL_Kommentar
EXTERNAL NAME 'CDTabelle_jar:CDTabelle.Kommentar'
      LANGUAGE JAVA
      ( ...
        STATIC METHOD SQLPreis()
        RETURNS INTEGER
        EXTERNAL VARIABLE NAME 'preis'
      )
```

5.3.4 Sortieren

Ein Java Datentyp kann keine der SQL-Sortierroutinen wie GROUP BY, ORDER BY oder DISTINCT benutzen. Nach dem SQL3-Standard sind neue Routinen (EQUALS ONLY,ORDER FULL) eingeführt worden, die es ermöglichen, diese Datentypen zu sortieren.
Als Beispiel für EQUALS ONLY könnte man die toString Methode als Sortierkriterium benutzen.

```
CREATE ORDERING FOR SQL_Kommentar
      EQUALS ONLY BY MAP WITH METHOD SQLtoString
```

Dies bedeutet, dass die SQLtoString Methode das Kriterium für die Sortierung bestimmt. Bei relativen Vergleichen müsste die Methode zwei Parameter haben und einen INTEGER-Wert als Resultat zurückliefern, der -1,0,1 ist, je nachdem, ob das Resultat kleiner, gleich oder größer ist.
ORDER FULL hingegen benutzt die Methode compareTo(Object) von java.lang.Comparable als Kriterium für die Sortierung aus. Unter anderem ermöglicht ORDER FULL, in den SQL-Anweisungen ORDER BY zu benutzen.

```
CREATE ORDERING FOR SQL_Kommentar
      ORDER FULL BY RELATIV WITH COMPARABLE INTERFACE
```

5.3.5 Untertypen

Java ermöglicht es, Klassen zu erweitern, hierzu gibt es das Schlüsselwort "extends". In SQL kann dies auch angewendet werden.
Zum Beispiel:

```
public class Auswertung extends Kommentar
            implements java.io.Serializable {
      public int rating;

      public Auswertung(String Name,String Kommentar, int rating) {
            super.(Name,Kommentar);
            this.rating = rating;
```

```
        }
    }
```

In SQL würde dies folgendermaßen umgesetzt werden:

```
CREATE TYPE SQL_Auswertung UNDER SQL_Kommentar
      EXTERNAL NAME 'CDTabelle_jar:CDTabelle.Kommentar'
      LANGUAGE JAVA
      ( rating INTEGER

METHOD SQL_Auswertung(v1 VARCHAR(20) ,v2 VARCHAR(100), v3 INTEGER)
            RETURNS SQL_Auswertung
            EXTERNAL NAME= 'Auswertung'
      )
```

6 Zusammenfassung

In Kapitel zwei wurden die Call Level Interfaces angeschnitten, diese funtionalen Interfaces erlauben es, durch Routinenaufrufe, Befehle an die Datenbank zu schicken. Die zwei populärsten Vertreter der CLI, Open Database Connectivity (ODBC) von Microsoft und Java Database Connectivity (JDBC) von JavaSoft wurden kurz vorgestellt.
Im dritten Kapitel wurde eine Beispieldatenbank angegeben, die das Abspeichern von CD-Alben ermöglicht. Eine Vielzahl der verwendeten Attributdatentypen sind aus dem SQL3 hervorgegangen, wie zum Beispiel large objects (LOB) oder user-defined types (UDT).
Das vierte Kapitel befasste sich mit dem Zugriff von ODBC und JDBC auf die Beispieldatenbank. Es wurde jeweils der Verbindungsaufbau realisiert und eine Anfrage an die Datenbank geschickt. Die Daten aus dem Resultat wurden später in einer Programmiersprache verarbeitet. Es wurde gezeigt, falls möglich, wie man die einzelnen Attribute aus dem Resultset extrahiert und in Variablen der verwendeten Programmiersprache abspeichert. Hierbei wurde sichtbar, dass ODBC 3.0 viele der neuen Datentypen nicht oder nur schlecht unterstützt. Auf andere Technologien wie ADO oder OLE /DB wurde hier nicht eingegangen. Diese Technologien sind Weiterentwicklungen und bieten besseren Einsatz in objektrelationalen Datenbanken als dies zum Beispiel ODBC 3.0 tut, welches nur für relationale Datenbanken gedacht ist.
Kapitel 5 befasste sich mit SQLJ, das Anwendungsprogrammierern unter anderem erlaubt, embedded SQL in Java zu verwenden. SQLJ besteht aus drei Teilen. Der erste Teil befasste sich, wie eben erwähnt, mit der Einbettung von SQL in Java. Das Einbetten von Java ist in anderen Programmiersprachen, wie C oder COBOL, schon lange möglich. Es wurde anhand von Beispielen einmal kurz die einzelnen Möglichkeiten, die dieser Teil bietet, vorgestellt. Der zweite Teil des Standards befasst sich mit der Integration von Java-Methoden als Prozeduren oder Funktionen, den sogenannten stored procedures, in die Datenbank. Auch hier wurde anhand von Beispielen aufgezeigt, wie die Java-Methoden integriert werden und wie sie dann im DBMS aufgerufen werden können. Zur Ausführung solcher Java-Methoden ist eine Java Virtual Machine (JVM) Bedingung. Die so definierten stored-Procedures sind von denen in SQL nicht zu unterscheiden. Der dritte und letzte Teil von SQLJ ermöglicht die Definition von eigenen Datentypen. Diese Datentypen können vollständige Java-Klassen umfassen, inklusive der Konstruktoren. Wie schon bei den beiden anderen Teilen, wurde im Rahmen dieser Ausarbeitung ein Überblick gegeben und mit Beispielen illustriert.

7 Ausblick

Objektrelationale Datenbanken stehen noch am Anfang ihrer Entwicklung und bieten noch eine Vielzahl von Erweiterungsmöglichkeiten. Man könnte sich die Unterstützung weiterer Kollektionstypen vorstellen. In SQL3 ist nur der Kollektionstyp ARRAY bekannt. Java bietet jedoch noch eine Vielzahl weiterer Möglichkeiten, zum Beispiel ArrayList und Map. ArrayListen kann man auf einfache Weise erweitern, während ARRAY mit einer, zuvor definierten, festen Anzahl von Elementen definiert werden muss. Demzufolge wären ArrayListen eine sinnvolle Erweiterung zum ARRAY.

Eine Vielzahl von objektorientierten Ansätzen können für den Anwendungsentwickler bereits durch die hier vorgestellten Techniken abgedeckt werden (SQLJ und JDBC). Bessere Einsatzmöglichkeiten währen meines Erachtens nur noch durch objektorientierte Datenbanken zu realisieren, diese haben sich bis jetzt jedoch noch nicht durchgesetzt.

8 Literaturverzeichnis

[Ge95] Kyle Geiger: Inside ODBC: Der Entwicklerleitfaden zum Industriestandard für Datenbank-Schnittstellen, Microsoft Press, 1995, ISBN: 3-86063-359-7.

[Hä99] Theo Härder: Vorlesungsskript DBS I, WS 1999/2000, Universität Kaiserslautern, FB Informatik.

[ME00] Jim Melton, Andrew Eisenberg: Understanding SQL and Java together: A Guide to SQLJ, JDBC, and Related Technologies, Morgan Kaufmann Publishers, 2000, ISBN: 1-55860-562-2.

[ZB00] Jürgen Zimmermann, Gerd Beneken: Verteilte Komponenten und Datenbankanbindung: Mehrstufige Architekturen mit SQLJ und Enterprise JavaBeans 2.0, Addison-Wesley, 2000, ISBN: 3-8273-1552-2.