ODBC/JDBC

Vortrag im Rahmen der Projektgruppe
Intelligente Datenbanken
Prof. Dr. Manthey
SS2003

Von

Margret Claaßen

Einleitung

SQL wurde speziell als Anfrage- und Manipulationssprache für relationale Datenbanken entwickelt. Zur Programmierung von Datenbankanwendungen ist jedoch zusätzlich die Verwendung einer algorithmisch mächtigeren Programmiersprache wie z. B. ***C***, ***C++*** oder ***Java*** nötig.

Für die Kopplung dieser konventionellen Programmiersprachen mit *SQL* werden in der Praxis besonders häufig die beiden folgenden Varianten eingesetzt:

- Verwendung einer **Call Level-Schnittstelle** (CLI – Call Level Interface) :
  Hierbei wird dem Programmierer eine Bibliothek von Prozeduren zur Verfügung gestellt, die aus der jeweiligen Programmiersprache heraus aufgerufen werden können und dann den Zugriff auf die Datenbank ermöglichen. Die *SQL*-Anweisungen werden dabei als Argumente der Prozeduraufrufe übergeben.

- **Einbettung** von *SQL* in die benutzte Programmiersprache :
  Die *SQL*-Anweisungen werden in den Quellcode der verwendeten Programmiersprache eingefügt und durch ein vorangestelltes Schlüsselwort (z.B. # SQL oder *exec* SQL) speziell gekennzeichnet. Bevor das Programm nun mit Hilfe des für die benutzte Programmiersprache üblichen Compilers übersetzt werden kann, muss es durch einen Präpcompiler (Precompiler) analysiert und vom *SQL*-Anteil separiert werden.

  Beispiel: Für die Einbettung von *SQL* in *Java* gibt es *Embedded SQL for Java* als Teil (Part 0) des Standards *SQLJ*.

Call-Level-Schnittstellen

Kommerzielle DBMS sind in der Regel mit mindestens einer Call-Level-Schnittstelle (prozedurale Programmierschnittstelle) ausgestattet, um die Programmierung von Anwendungen zu ermöglichen. Normalerweise wird für jede Kombination aus Datenbankmanagementsystem und unterstützter Programmiersprache eine eigene Call-Level-Schnittstelle benötigt.

Beispiel: Für die Kommunikation mit *C* bzw. *C++* gibt es von
- *Oracle* das *Oracle Call Interface (OCI)*
- *Sybase* die *CT-Bibliothek*
- *IBM DB2* das *Call Level Interface (DB2 CLI).*

Um zumindest auf Quelltextebene weitgehend portable Programme zu ermöglichen (d.h., dass das Programm auf unterschiedliche Datenbanken, die *SQL* verwenden, zugreifen kann; diese Eigenschaft wird auch **Interoperabilität** genannt), wurde von der *X/Open-Group* ***SQL/CLI*** als Standard für die prozeduralen Programmierschnittstellen definiert.
Sowohl ***ODBC*** als auch ***JDBC*** sind Call-Level-Schnittstellen, die der *SQL/CLI*-Spezifikation entsprechen.

ODBC

ODBC ist der Name einer von Microsoft erstellten, *SQL/CLI*-konformen API (Application Programming Interface). Meist wird *ODBC* als Abkürzung für ***Open Database Connectivity*** aufgefasst, ist aber streng genommen ein eingetragenes Warenzeichen.
ODBC stellt eine *C/C++*-Klassenbibliothek zur Verfügung, es lassen sich aber auch z.B. *Fortran* oder *Visual Basic* anbinden.
Fast alle DBMS-Hersteller unterstützen *ODBC*.
ODBC ist eine plattformübergreifende Lösung: obwohl ursprünglich für *Windows* konzipiert, existieren mittlerweile zahlreiche Portierungen auch für andere Betriebssysteme, insbesondere für *Unix*-Plattformen.

In seiner Funktionalität geht *ODBC* noch über die im *SQL/CLI*-Standard festgelegten Anforderungen hinaus: Anwendungsprogramme, die als Datenbankschnittstelle *ODBC* benutzen, sind nicht nur als Quellcode, sondern auch in compilierter Form portabel. Dadurch wird es möglich, erst zur Laufzeit des Programms zu entscheiden, auf welche Datenbank zugegriffen werden soll oder mit mehreren Datenbanken unterschiedlicher Hersteller gleichzeitig zu arbeiten.

ODBC erreicht dies, indem es das Anwendungsprogramm von dem Datenbanksystem durch Zwischenschalten eines Treibermanagers und DBMS-spezifischer Datenbanktreiber entkoppelt.

ODBC-Architektur

Der **Treibermanager** (Driver Manager) und die **Treiber** (Driver) müssen der *ODBC*-API entsprechen. Die Treiber sind spezifisch für das jeweilige DBMS und werden vom DBMS-Hersteller mitgeliefert oder stammen von Drittanbietern. Die Aufgabe der Treiber ist es, *ODBC*-Aufrufe in Anweisungen zu übersetzen, die von dem jeweiligen DBMS verstanden werden. Außerdem muss der Treiber dann die Ergebnisse der Datenbankanfrage entgegennehmen und in eine für die verwendete Programmiersprache verständliche Form bringen.
Der Treibermanager regelt die Kommunikation zwischen Anwendung und Treibern. Er verwaltet die verschiedenen bei ihm registrierten Treiber und wählt den zum betreffenden DBMS passenden Treiber nach Aufforderung durch die Anwendung zur Laufzeit aus. Er implementiert alle Funktionen der *ODBC*-API, die meisten davon so, dass ein Funktionsaufruf an den (ausgewählten) Treiber weitergeleitet wird. Weil er für jeden geladenen Treiber eine Tabelle mit den Pointern auf jede einzelne Funktion bereithält, kann die Anwendung bei gleichzeitiger Angabe eines **Connection-Handles** (enthält Informationen zum Treiber und zur Datenquelle) eine Funktion im Treibermanager über ihren Namen aufrufen, statt über einen Pointer im Treiber.

Da eine Anwendung dadurch nicht mehr wissen muss, welches DBMS die gewünschte Datenbank verwaltet, führt *ODBC* sogenannte **Datenquellen** ein. Eine solche Datenquelle wird zuvor vom Nutzer konfiguriert (unter Windows z.B. über die Systemsteuerung), indem der Name der Datenquelle sowie das DBMS inklusive der notwendigen Verbindungsinformationen wie Servername, Netzwerkadresse usw. festgelegt werden. Beim Aufbau einer Verbindung zur Datenquelle ist neben Benutzername und Passwort dann nur noch der Name der Datenquelle anzugeben. *ODBC* wählt anhand der Konfigurationsinformationen den für die jeweilige Datenquelle geeigneten Treiber aus, lädt ihn und baut die Verbindung zur Datenbank auf.

JDBC

JDBC ist der Name einer von *Javasoft/Sun* entwickelten *SQL/CLI*-konformen API.
Als Teil der Standard-API von *Java* ist *JDBC die* Standardschnittstelle für den Zugriff auf *SQL*-Datenbanken aus einer in *Java* programmierten Anwendung.
JDBC wird meist mit ***Java Database Connectivity*** übersetzt, ist aber eigentlich ein geschützter Name.

JDBC wurde mit Erscheinen des *JDK 1.1* eingeführt, mittlerweile existiert es in der Version *3.0* als Bestandteil der *Java 2* Plattformen *J2SE 1.4* und *J2EE 1.4*. Es ist in zwei Pakete unterteilt:

- **`java.sql`**
  Dieses Paket enthält die grundlegenden Klassen und Schnittstellen.
- **`javax.sql`**
  Dieses Paket stellt erweiterte Funktionalitäten für die Arbeit mit Datenbanken bereit.

Das Paket `java.sql` wird auch als *JDBC Core API* bezeichnet und `javax.sql` als *JDBC Optional Package*, wobei letzteres in den vorhergehenden Versionen noch nicht Bestandteil der *J2SE (Java 2 Standardedition)* war.

JDBC genügt nicht nur demselben Standard wie *ODBC*, *SQL/CLI*, sondern orientiert sich auch in seiner sonstigen Konzeption eng an *ODBC*. So verwendet auch *JDBC* das Treiber/Treibermanager-Konzept. Wie bei *ODBC* erfordert der Wechsel des DBMS keine Änderungen am Quellcode oder erneutes Compilieren, was einer Anwendung den simultanen Zugriff auf unterschiedliche DBMSe ermöglicht.

JDBC ist eine **Low Level-API**. Das bedeutet, dass man sich um viele Details z.B. zum Verbindungsaufbau zur Datenbank nicht mehr selbst kümmern muss, da diese Funktionalität durch entsprechende Klassen und Methoden der API zur Verfügung gestellt wird. Der Anwendungsprogrammierer muss jedoch selbst die *SQL*-Befehle in Form von Strings erzeugen und die einzelnen Attribute eines Datensatzes einer Datenbankabfrage in Objekte verwandeln. Deshalb wird bei größeren Anwendungen meist eine zusätzliche Softwareschicht zur Datenaufbereitung oberhalb der *JDBC*-API implementiert, welche den Zugriff auf ein Datenbanksystem weiter abstrahiert und als **High Level-API** von den Anwendungen aufgerufen wird.

Treibertypen

Zur Realisierung von *JDBC*-Treibern stehen grundsätzlich die folgenden vier Möglichkeiten zur Verfügung:

- ***JDBC-ODBC-Bridge (Typ 1-Treiber)***
  Die *JDBC-ODBC-Bridge* ist Bestandteil des *JDK*. Sie sorgt dafür, dass alle *JDBC*-Aufrufe in *ODBC*-Aufrufe umgewandelt werden. Der Vorteil ist, dass kein separater *JDBC*-Treiber verfügbar sein muß, sondern sich bereits vorhandene *ODBC*-Treiber nutzen lassen (die es für alle gängigen DBMSe gibt). Allerdings bedingt die Verwendung der *JDBC-ODBC-Bridge* eine geringe Effizienz, da alle Funktionen doppelt durchlaufen werden. Zudem muss die Datenbank auf dem Client als *OBDC*-Datenquelle registriert sein, daher kann diese Lösung nicht von Applets über das Netz benutzt werden.

- ***Native API-Treiber (Typ 2-Treiber)***
  Dieser Treiber ist nur teilweise in *Java* geschrieben und greift deshalb auf eine in *C/C++* geschriebene Bibliothek des DBMS-Herstellers zu, die auf dem Client installiert sein muß. Er ist deshalb nicht für den Einsatz in Applets geeignet. Die *JDBC*-Aufrufe werden in DBMS-spezifische Anweisungen, also ein proprietäres Protokoll, umgesetzt. *Native API-Treiber* lassen sich schnell aus bereits bestehenden, z.B. in *C* geschriebenen Treiber herstellen.

- ***JDBC-Net-Treiber (Typ 3-Treiber)***
  Bei dieser Variante wird zur Kommunikation zwischen dem *JDBC*-Treiber und dem Datenbanksystem eine zusätzliche datenbankunabhängige Komponente (Middleware) eingeführt, die als Verteilinstanz wirkt und das DBMS-neutrale Protokoll in ein DBMS-spezifisches Protokoll übersetzt. Der *JDBC*-Treiber kann somit unabhängig vom konkreten DBMS gestaltet und vollständig in *Java* implementiert werden. *JDBC-Net-Treiber* stellen deshalb die flexibelste Lösung dar. Sie ermöglichen auch den Zugriff auf eine Datenbank aus einem Applet heraus.

- ***Native Protocol-Treiber (Typ 4-Treiber)***
  Diese Treiber übersetzen *JDBC*-Aufrufe direkt in das proprietäre Protokoll, ohne dazu das Datenbank-API zu nutzen, deshalb können sie ebenfalls komplett in *Java* geschrieben werden. Dadurch sind diese Treiber aber im Gegensatz zu den *JDBC-Net-Treibern* DBMS-spezifisch. Da bei dieser Treibervariante eine direkte Verbindung zwischen dem Client und dem Datenbankserver besteht, ist diese Art des Datenbankzugriffs besonders effizient. *Native Protocol-Treiber* werden in der Regel vom DBMS-Hersteller mitgeliefert, wenn das DBMS eine *JDBC*-Schnittstelle besitzt. Sie lassen sich in Verbindung mit Applets einsetzen.

Die beiden letztgenannten Treibertypen werden aus den genannten Gründen bevorzugt verwendet (wenn vorhanden), wohingegen die beiden ersten Treibertypen als Übergangslösungen bis zur Verfügbarkeit eines Treibers vom Typ *3* oder *4* anzusehen sind.

Der Aufbau der *JDBC*-Treiber ist in der *JDBC*-Treiber-API spezifiziert. Diese API ist nur für die Hersteller der Teiber von Bedeutung, Anwendungsentwickler arbeiten hauptsächlich mit der *JDBC*-API und somit unabhängig von der eigentlichen Implementierung der Treiber.

Wichtige Strukturen in JDBC

Die wichtigsten Klassen und Schnittstellen des Paketes *java.sql* sind:

- `java.sql.DriverManager`
  Bildet den Einstiegspunkt, da der Treibermanager Treiber registriert und Verbindungen zur Datenbank aufbaut.
- `java.sql.Connection:`
  Repräsentiert eine Datenbankverbindung.
- `java.sql.Statement:`
  Ermöglicht die Ausführung von SQL-Anweisungen über eine gegebene Verbindung.
- `java.sql.ResultSet:`
  Verwaltet die Ergebnisse einer Anfrage in Form einer Relation und unterstützt den Zugriff auf einzelne Spalten.

Zugriff auf eine Datenbank

Der Zugriff auf eine Datenbank aus einer Anwendung heraus läuft nach dem folgenden Schema ab:

1. Verbindungsaufbau zur Datenbank
2. Absetzen des SQL-Statements
3. Auswerten des Ergebnisses
4. Schließen des SQL-Statements
5. Schließen der Verbindung

Eine Anwendung kann gleichzeitig mehrere Verbindungen auch zu unterschiedlichen DBMSen aufbauen. Innerhalb jeder Verbindung kann dann eine Folge von *SQL*-Statements ausgeführt werden.

Beispiel:

Das folgende Beispiel soll eine einfache Datenbankabfrage zeigen. Aus einer Datenbanktabelle `studenten` in der Datenbank mit dem Namen `unidatenbank` soll der Name und die Matrikelnummer aller Studenten angezeigt werden:

```
// Datei: JdbcTest.java

import java.sql.*;

public class JdbcTest {

    public static void main (String [] args) {

        Connection con = null;
        Statement stmt = null;
        ResultSet rs   = null;

        String driver  = "COM.ibm.db2.jdbc.net.DB2Driver"
        String url     = "jdbc:db2://pc1:6789/unidatenbank";
        String user    = "ich";
        String passwd  = "geheim";

        String query   = "SELECT matrikelnummer, name FROM studenten";

        String name;
        int matrikelnummer;

        try {
            // Treiber laden und Verbindung herstellen
            Class.forName(driver);
            con = DriverManager.getConnection(url, user, passwd);

                 // Absetzen eines SQL-Statements
                 stmt = con.createStatement();
            rs   = stmt.executeQuery (query);

                 // Auswerten des Ergebnisses
                 while (rs.next()) {
                      matrikelnummer = rs.getInt(1);
                      name           = rs.getString(2);
                      System.out.println(matrikelnummer + " " + name);
                 }

                 // Statement schliessen
                 stmt.close();

                 // Verbindung schliessen
```

```
                con.close();
        }

        catch (Exeption ex) {
                System.out.println("Exception: " + ex.getMessage());
            }
      }
}
```

Verbindungsaufbau zur Datenbank

Dieser verläuft in zwei Schritten: Zunächst wird ein geeigneter Treiber geladen, dann durch Erzeugen eines `Connection`-Objektes eine Verbindung hergestellt.
Für das Laden des Treibers gibt es verschiedene Möglichkeiten, wobei man insbesondere explizites und automatisches Laden unterscheiden kann:

1. Der Treiber (d. h. die Treiber-Klasse) wird über die Methode `forName` der Klasse `Class` explizit im *Java*-Programm geladen und ein Treiber-Objekt instanziiert, welches sich dann selbst beim *JDBC*-Treiber-Manager registriert, siehe obiges Beispiel:
   `Class.forName("COM.ibm.db2.jdbc.net.DB2Driver");`
   Hierbei ist `DB2Driver` der Name der Treiberklasse und `COM.ibm.db2.jdbc.net` bezeichnet den Pfad, unter dem die Treiberklasse zu finden ist.
   Diese Methode ist besonders einfach und wird deshalb oft verwendet.

2. Es wird explizit ein Objekt der Treiberklasse erzeugt und dieses mithilfe der statischen Methode `registerDriver()` der Klasse `DriverManager` explizit beim Treibermanager registriert.
   Beispiel:
   ```
   Driver treiber = new COM.ibm.db2.jdbc.net.DB2Driver();
   DriverManager.registerDriver(treiber);
   ```
   Der Vorteil dieser Methode ist, dass man schon zur Übersetzungszeit eine Fehlermeldung erhält, wenn der angegebene Treiber nicht vorhanden ist.

3. Der Treiber wird im *Java*-Programm in der `jdbc.drivers`-Systemeigenschaft (Property) abgelegt und dann beim Start vom Treibermanager automatisch geladen und registriert.
   Beispiel:
   `System.setProperty("jdbc.drivers", "COM.ibm.db2.jdbc.net.DB2Driver");`
   Sollen mehrere Treiber registriert werden, so müssen diese durch Doppelpunkte voneinander getrennt angegeben werden:
   ```
   System.setProperty("jdbc.drivers", "COM.ibm.db2.jdbc.net.DB2Driver :
   oracle.jdbc.OracleDriver");
   ```

4. Der Treiber wird beim Aufruf des *Java*-Interpreters auf der Kommandozeile als Option angegeben.
   Beispiel:
   `Java -Djdbc.drivers=COM.ibm.db2.jdbc.net.DB2Driver Jdbctest`
   Sollen mehrere Treiber registriert werden, so müssen sie durch Doppelpunkte voneinander getrennt angegeben werden.

Der Name der jeweiligen Treiberklasse muss der Dokumentation des DBMS-Herstellers entnommen werden.

Nach dem Laden des Treibers kann die Verbindung zur Datenbank aufgebaut werden, die durch ein `Connection`-Objekt als Implementierung der Schnittstelle `Connection` repräsentiert wird. Über eine geöffnete Verbindung können *SQL*-Anweisungen zur Datenbank gesendet, Transaktionsabläufe gesteuert und Informationen über die Datenbank abgefragt werden.
Zur Erzeugung eines `Connection`-Objekt stellt die Klasse `DriverManager` drei verschiedene Methoden `getConnection` bereit:

- `static Connection getConnection(String url)`
- `static Connection getConnection(String url, String user, String password)`
- `static Connection getConnection(String url, Properties info)`
  wobei `info` für treiberspezifische Informationen steht.

Alle Methoden erwarten einen **URL** (Uniform Resource Locator) als Parameter.
Die *JDBC*-URLs haben folgende Form:

```
jdbc:<subprotocol>:<subname>
```

wobei `<subprotocol>` das Unterprotokoll kennzeichnet.

Beispiele:
`db2` für *DB2* von *IBM*
`odbc` für die *ODBC*-Brücke
`oracle:oci8` oder `oracle:thin` für *Oracle*

`<subname>` ist der Identifikator für die Datenbank, das Format von `<subname>` hängt vom verwendeten Unterprotokoll ab.

Beispiele:
```
jdbc:odbc:mydb
jdbc:db2:mydb
jdbc:db2://myhost.domain.de:6790/mydb
jdbc:oracle:oci8:@server:1521:mydb
```

Beim Aufruf der Methode `getConnection` sucht der Treibermanager unter den geladenen Treibern einen, der die angegebene URL akzeptiert. Mir dem ersten gefundenen Treiber wird dann die Verbindung zur Datenbank hergestellt. Dabei sucht er zuerst in der Systemvariablen `jdbc.drivers`, danach prüft er, ob ein Treiber explizit geladen wurde.

Eine Applikation kann mehrere Verbindungen zu einer oder zu verschiedenen Datenbanken öffnen.

Absetzen eines SQL-Statements

In *JDBC* wird jede *SQL*-Anweisung durch ein Statement-Objekt gekapselt. Dieses Objekt sendet die Anweisung zur Datenbank, liefert das Ergebnis zurück und verarbeitet die der Anweisung übergebenen Parameter. Es gibt drei Formen von Statements:

- `java.sql.Statement` als die grundlegende Schnittstelle ermöglicht die Verarbeitung einfacher Anweisungen ohne Parameter.
- `java.sql.PreparedStatement` leitet sich von `java.sql.Statement` ab und kapselt eine vorkompilierte Anweisung. Ein `PreparedStatement`-Objekt wird besonders dann eingesetzt, wenn eine Anweisung mehrfach mit unterschiedlichen `IN`-Parametern ausgeführt werden soll.
- `java.sql.CallableStatement` leitet sich von `java.sql.PreparedStatement` ab und ermöglicht den Aufruf von gespeicherten Prozeduren.

Zunächst muss eine Instanz der Schnittstelle `Statement` erzeugt werden.
Da die Statement-Objekte abhängig vom benutzten *JDBC*-Treiber sind, werden sie nicht mit dem `new`-Operator, sondern durch Aufruf folgender Methoden des `Connection`-Objektes erzeugt:

- `Statement createStatement()`
- `PreparedStatement prepareStatement(String sql)`
- `CallableStatement prepareCall(String sql)`

Bei Anweisungen vom Typ `PreparedStatement` und `CallableStatement` muß der *SQL*-Code schon bei der Instanziierung der Statement-Objekte angegeben werden, wohingegen bei einfachen Anweisungen (d.h. vom Typ `Statement`) der *SQL*-String erst bei der Ausführung (mittels `executeXXX`) übergeben wird.

Für eine geöffnete Verbindung können mehrere Statements erzeugt werden, die maximale Anzahl hängt vom verwendeten DBMS ab.

Nun soll die Anweisung ausgeführt werden.

Die Schnittstelle `Statement` stellt dazu drei Methoden zur Verfügung:

- `ResultSet executeQuery(String sql)`
- `int execteUpdate(String sql)`
- `boolean execute(String sql)`

Mit der Methode `executeQuery` wird eine SELECT-Anweisung ausgeführt. Diese Methode liefert ein `ResultSet`-Objekt mit dem Anfrageergebnis.

Beispiel:
```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery ("SELECT matrikelnummer, name FROM studenten");
```

DDL- und DML-Anweisungen, d.h. CREATE TABLE bzw. INSERT, UPDATE, DELETE, werden über die Methode executeUpdate der Datenbank übermittelt. Bei DML-Anweisungen entspricht der Rückgabewert der Anzahl der manipulierten Tupel, bei DDL-Anweisungen ist er immer 0.

Beispiel:
```
Statement stmt = con.createStatement();
int numberOfRows = stmt.executeUpdate("DELETE FROM studenten WHERE semesterzahl
= 20");
```
Die Methode `execute` kann zum Ausführen aller Arten von Statements benutzt werden, wird aber meist nur dann eingesetzt, wenn die Anweisung mehrere Ergebnisse gleichzeitig liefert, z.B. bei Verwendung gespeicherter Prozeduren.

Auswerten des Ergebnisses

Die Methode `executeUpdate`, die für UPDATE-, DELETE-, INSERT- oder CREATE TABLE-Befehle verwendet wird, liefert einen int-Wert zurück, hier ist also die Auswertung des Ergebnisses gar kein Problem.
Als Ergebnis einer *SQL*-Anfrage, also eines SELECT-Befehls mittels der Methode `executeQuery`, liefert das DBMS eine Menge von Tupeln, die als Relation aufgefasst werden können und in *JDBC* durch ein `ResultSet`-Objekt repräsentiert werden. Das Problem ist, dass *Java* als imperative Programmiersprache Tupel als grundlegende Datenstruktur verwendet, es kann nicht wie *SQL* auf einer Relation als Ganzes, also mengenorientiert, arbeiten. Diesen Gegensatz in den Datenstrukturkonzepten der beiden zu koppelnden Programmiersprachen nennt man auch **Impedance Mismatch**.
Zur Lösung dieses Problems führt *SQL* führt **Cursor** ein, d.h. Iteratoren, über die sich auf einzelne Tupel der Relation zugreifen lässt.

Beispiel:
```
DECLARE Crsr CURSOR FOR
        SELECT name, matrikelnummer
        FROM studenten
```

Der Zugriff auf die einzelnen Tupel erfolgt dann über die FETCH-Anweisung des Cursors.
In *JDBC* ist aber die explizite Nutzung (und Deklarierung) des Cursors nicht nötig, da das Interface `ResultSet` einen internen Cursor verwaltet, der über die Methode `next` zugänglich ist.

- `boolean next()`

Zu Beginn ist der Cursor vor dem ersten Tupel positioniert, sodass schon vor dem ersten Zugriff `next` aufgerufen werden muß. Der Rückgabewert ist `true`, wenn es noch weitere Tupel in der Relation gibt und `false`, wenn die Ergebnisrelation entweder leer war oder vollständig durchlaufen wurde.

Nachdem der interne Cursor positioniert ist, kann auf die einzelnen Attribute des Tupels zugegriffen werden. Hierfür sind zwei Gruppen von `getXXX`-Methoden definiert: Methoden zum Zugriff über den Spaltenindex und über den Spaltennamen:

1. für den Zugriff über den Spaltenindex, z.B.
   - `int getInt(int colIndex)`
   - `String getString(int colIndex)`
   - `double getDouble(int colIndex)` usw.

   wobei die Zählung bei 1 beginnt und sich auf die Liste der Attribute in der SELECT-Anweisung bezieht

2. für den Zugriff über den Spaltennamen, z.B.
   - `int getInt(String colName)`
   - `String getString(String colName)`
   - `double getDouble(String colName)` usw.

Man muß also durch Auswahl der passenden Methode selbst für die Umwandlung von einem *SQL*-Datentyp in einen passenden *Java*-Datentyp sorgen. Dazu kann die folgende Tabelle herangezogen werden:

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	BOOLEAN	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	DATALINK	STRUCT	JAVA_OBJECT
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x	x													
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x	x													
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x	x													
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x	x													
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x	x													
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x	x													
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x	x													
getBoolean	x	x	x	x	x	x	x	x	x	X	X	x	x	x													
getString	x	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x					x		
getBytes															X	X	x										
getDate												x	x	x				X		x							
getTime												x	x	x					X	x							
getTimestamp												x	x	x				x	x	X							
getAsciiStream												x	x	X	x	x	x										
getBinaryStream															x	x	X										
getCharacterStream												x	x	X	x	x	x										
getClob																					X						
getBlob																						X					
getArray																							X				
getRef																								X			
getURL																									X		
getObject	x	x	x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X

TABLE B-6 Use of `ResultSet` getter Methods to Retrieve JDBC Data Types

Dabei kennzeichnet ein großes X, dass die entsprechende Methode bervorzugt zu benutzen ist.
Bei den Datentypen im Kopf der Tabelle handelt es sich strenggenommen nicht um *SQL*-Datentypen, sondern um Konstanten der Klasse `java.sql.Types` (= *JDBC*-Typen). Diese wurden in *JDBC* eingeführt, um die Inkompatibilitäten zu überwinden, die dadurch entstehen, dass einige *SQL*-Typen von den DBMS-Herstellern teilweise unterschiedlich bezeichnet werden. *JDBC*-Typen werden in *JDBC* überall dort verwendet, wo *SQ*L-Typen anzugeben sind. Die Umsetzung in die DBMS-spezifischen Datentypen erfolgt – wenn notwendig – durch den Treiber.

Beispiel zur Anwendung der sogenannten “getter”-Methoden:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT name, matrikelnummer FROM studenten");
while (rs.next()) {
    int matrikelnummer = rs.getInt(2);
```

```
    String name = rs.getString(1);
}
```

bzw.

```
while (rs.next()) {
    int matrikelnummer = rs.getInt("matrikelnummer");
    String name = rs.getString("name");
}
```

Ein weiteres Beispiel für den Fall, dass eine SELECT-Anweisung nur einen Wert liefert:

```
Statement stmt = con.createStatement();
ResultSet rs   = stmt.executeQuery("SELECT AVG(semester) FROM studenten");
rs.next();
double semesterdurchschnitt = rs.getDouble(1);
```

PreparedStatement

Die Schnittstelle `PreparedStatement` definiert eine spezielle Version eines Statements, bei der der String mit der *SQL*-Anweisung schon beim Erzeugen des `PreparedStatement`-Objektes zum DBMS gesendet und dort kompiliert wird. Dabei werden die **variablen Parameter (`IN`-Parameter)** des *SQL*-Befehls bei der Definition des Statements nur durch Platzhalter in Form eines "?" markiert und erst bei der Ausführung durch konkrete Werte ersetzt. Hierzu gibt es in `PreparedStatement` verschiedene `setXXX()`- Methoden, z.B.:

- `void setBoolean(int paramIndex, boolean b)`
- `void setString(int paramIndex, String s)`
- `void setInt(int paramIndex, int i)`
- `void setFloat(int paramIndex, float f)`

Zusätzlich zum aktuellen Wert des Parameters muss auch noch sein Index angegeben werden (dieser kennzeichnet die Reihenfolge der Fragezeichen).
Durch die jeweilige Methode wird der *Java*-Datentyp dann in einen passenden *JDBC*-Typ, d.h. eine Konstante der Klasse `java.sql.Types` umgewandelt entsprechend der folgenden Tabelle:

Java Type	JDBC Type
String	CHAR, VARCHAR, or LONGVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT or BOOLEAN
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	BINARY, VARBINARY, or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
Clob	CLOB
Blob	BLOB
Array	ARRAY
Struct	STRUCT
Ref	REF
java.net.URL	DATALINK
Java class	JAVA_OBJECT

TABLE B-2 Standard Mapping from Java Types to JDBC Types

Wenn nötig, übersetzt der Treiber den *JDBC*-Typ in den DBMS-spezifischen *SQL*-Typ.

Als zusätzliche Methode benötigt man zum Setzen von Null-Werten die Methode

```
void setNull(int paramIndex, int jdbcType)
```

wobei außer dem Index noch der *JDBC*-Typ angegeben werden muss, der dem *SQL*-Typ entspricht.

Die von der Schnittstelle `Statement` geerbten `executeXXX`- Methoden sind überschrieben worden, weil ihnen kein *SQL*-String mehr übergeben wird:

- `ResultSet executeQuery()`
- `int execteUpdate()`
- `boolean execute()`

Beispiel:
Es sollen drei Datensätze in die Tabelle `studenten` eingefügt werden:

```
String[] name        = {Müller, Meier, Schmidt},
int[] matrikelnummer = {123456, 345678, 567890};
int[]semesterzahl    = {2, 10, 20};

PreparedStatement stmt = con.prepareStatement("INSERT INTO studenten VALUES (?,
?, ?)");

for (int i = 0; i < name.length; i++) {
     stmt.setString(1, name[i]);
     stmt.setInt(2, matrikelnummer[i]);
     stmt.setInt(3, semesterzahl[i]);
     stmt.executeUpdate();
}
```

Anweisungen der Form `PreparedStatement` werden vorteilhaft dann eingesetzt, wenn die Anweisung mehrmals mit unterschiedlichen `IN`-Parametern ausgeführt werden soll. Der Vorteil liegt sowohl in einer kürzeren Laufzeit, da nur einmal compiliert und optimiert werden muss als auch in einer besseren Handhabung der Parameter, denn diese müssen nicht mehr als Strings in den *SQL*-Ausdruck eingebaut werden, sondern können, wie bereits gesehen, in Form der ursprünglichen *Java*-Datentypen übergeben werden.

Man kann die Performanz noch verbessern, indem man die einzelnen Anweisungen zu einer **Gruppe (batch)** zusammenfasst und gemeinsam ausführt:

```
String[] name        = {"Müller", "Meier", "Schmidt"},
int[] matrikelnummer = {123456, 345678, 567890};
int[]semesterzahl    = {2, 10, 20};
PreparedStatement stmt = con.prepareStatement("INSERT INTO studenten VALUES (?,
?, ?)");

      for (int i = 0; i < name.length; i++) {
                stmt.setString(1, name[i]);
                stmt.setInt(2, matrikelnummer[i]);
                stmt.setInt(3, semesterzahl[i]);
                stmt.addBatch();
      }
      stmt.executeBatch();
      stmt.close();
```

CallableStatement

Die Schnittstelle `CallableStatement` wird zum Aufruf von **gespeicherten Prozeduren (Stored Procedures)** eingesetzt. Dazu muss die Prozedur zuerst in der Datenbank definiert werden. Wie dies geschieht und ob das Konzept der gespeicherten Prozeduren überhaupt unterstützt wird, hängt von dem jeweiligen DBMS ab.
Der Effekt von gespeicherten Prozeduren ist, dass ein Teil der Anwendungslogik vom Client bzw. dem Applikationsserver zum Datenbankserver verlagert wird und dadurch viel weniger Daten über das Netz transportiert werden müssen.

Metadaten

Metadaten sind diejenigen Informationen, die über die in den Relationen gespeicherten Daten hinausgehen.
JDBC stellt drei Schnittstellen zur Verfügung, um die Metadaten darstellen zu können:

- `java.sql.DatabaseMetaData`
  Stellt Informationen über die Datenbank und das DBMS bereit. So kann z.B. zur Laufzeit das Datenbankschema ermittlet oder die unterstützten Datenbankkonzepte (wie *SQL*-Dialekt oder implementierte Operationen) abgefragt werden. Letzteres ist insbesondere für die Entwicklung DBMS-neutraler Anwendungen von Bedeutung.

- `java.sql.ResultSetMetaData`
  Beschreibt die Struktur eines spezifischen `ResultSet`-Objektes.

- `java.sql.ParameterMetaData`
  Liefert Informationen über Typ und Eigenschaften von Parametern in einem vorliegenden `PreparedStatement`-Objekt.

Beispiel zu `java.sql.ResultSetMetadata`:
Ein Benutzer kann zur Laufzeit Anfragen an die Datenbank richten. Der Entwickler will für diese Interaktionen eine Bildschirmmaske bereitstellen. Dazu benötigt er Informationen zum Relationenschema der von der Datenbank in Form eines `ResultSet`-Objektes zurückgelieferten Ergebnisrelation:

Beispiel:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM studenten");

ResultSetMetaData rsmeta = rset.getMetaData();
int numCols = rsmeta.getColumnCount();

for (int i = 1; i <= numCols; i++) {
      int ct = rsmeta.getColumnType(i);       // JDBC-Typ
      String cn = rsmeta.getColumnName(i);
      String ctn = rsmeta.getColumnTypeName(i);
      System.out.println("Spalte Nr" + i + ": "
            + cn + " of type "
            + ctn + "(JDBC-Typ: " + ct + ")");
}
```

Fazit

ODBC und *JDBC* sind Programmierschnittstellen, die *dynamisches SQL* unterstützen, d.h. *SQL*-Anweisungen werden zur Laufzeit zusammengestellt und als Zeichenkette zum Datenbankserver gesendet. Sie sind auf (fast) allen Plattformen zu verwenden und gewähren Zugriff auf alle gängigen DBMSe, insbesondere auch auf verschiedene DBMSe gleichzeitig. Sie sind somit flexibel und vielseitig einsetzbar.

Bibliographie

Date, C. und Darwen, H.: *SQL- Der Standard - SQL/92 mit den Erweiterungen CLI und PSM.* Addison-Wesley-Longman, Bonn; Reading, Mass., 1998.

Dehnhardt, W.: *Java und Datenbanken – Anwendungsprogrammierung mit JDBC, Servlets und JSP.* Carl Hanser Verlag, München, Wien, 2003.

Goll, J., Weiß, C. und Müller, F.: *Java als erste Programmiersprache – Vom Einsteiger zum Profi.* Teubner, Stuttgart,Leipzig,Wiesbaden, 2001.

Heuer, A. und Saake, G.: *Datenbanken: Konzepte und Sprachen.* mitp-Verlag, Bonn, 2000.

Heuer, A., Saake, G. und Sattler, K.: *Datenbanken - kompakt.* mitp-Verlag, Bonn, 2001.

Horstmann, C. und Cornell, G.: *Core Java: Band 2-Expertenwissen.* Markt+Technik Verlag, München, 2000.

Kemper, A. und Eickler, A.: *Datenbanksysteme – Eine Einführung.* Oldenbourg, München, Wien, 2001.

Petkovic, D. und Brüderl, M.: *Java in Datenbanksystemen – JDBC, SQLJ, Java DB-Systeme und – Objekte.* Addison-Wesley Verlag, München, 2002.

Saake, G. und Sattler, K.: *Datenbanken & Java – JDBC,SQLJ,ODMG und JDO.* dpunkt.verlag, Heidelberg, 2003.

http://java.sun.com/products/jdbc/

http://msdn.microsoft.com/library, -> Data Access -> Reference -> Microsoft Open Database Connectivity -> ODBC Programmer's Reference

http://java.rrzn.uni-hannover.de/jdbc/

http://web.f4.fhtw-berlin.de/hartwig/JDBC/jdbc.html

http://www.syssoft.uni-trier.de/systemsoftware/Download/Seminare/Middleware/

http://www.hta-be.bfh.ch/~schmd/jdbc