Kapitel 5
SQL und Java

Folien zum Datenbankpraktikum
Wintersemester 2012/13 LMU München

Übersicht

Überblick über Java in Oracle

- Auf SQL-Daten kann in Java über *JDBC* (und SQLJ) zugegriffen werden.
- *Java Stored Procedures*: Es ist möglich, Java-Programme von PL/SQL aus und PL/SQL-Prozeduren von Java aus zu starten.
- Entwicklung verteilter Anwendungen mit *Object Request Brokern (ORB/CORBA)* und der Unterstützung durch *Enterprise Java Beans (EJB)* (wird hier nicht besprochen).
- Realisierung *dynamischer HTML-Seiten* durch *Servlets* und *Java Server Pages* möglich
- Als *Persistent Storage* über *Object-Relational-Mapping (ORM)*, z.B. mit Hibernate.

RDBMS und Java

Software-Entwicklung in Java wird in Oracle über folgende Schnittstellen unterstützt:

- ***Java Database Connectivity (JDBC)***: Klassenbibliothek zum komfortablen dynamischen Zugriff auf eine (objekt-)relationale Datenbank aus Java heraus.
- ***Embedded SQL in Java (SQLJ)***: Erlaubt die direkte Integration von *statischem SQL* in Java ohne Benutzung einer Klassenbibliothek. Setzt meist auf JDBC auf und kann mit JDBC kombiniert werden.

Datenbank-Zugriffe sind ***clientseitig*** (Java Applikationen und Applets) und ***serverseitig*** (*Java Stored Procedures*) möglich. Für letztere bietet der Oracle Server eine Laufzeitumgebung (*JServer*) und persistente Verwaltung an, ähnlich zu den Stored Procedures in PL/SQL.

Standards

- Version Oracle 11g unterstützt Java ab Version 1.5 und JDBC 4.0
- Version Oracle 10g unterstützt Java ab Version 1.4 und JDBC 3.0
- Version Oracle 9i unterstützt Java ab Version 1.3 und JDBC 2.0

Übersicht

JDBC: Aufbau einer Datenbank-Verbindung (*clientseitig)*

- Anmelden der benötigten Packages:

  ```
  import java.sql.*; // JDBC-Package
  ```

  Die Packages `oracle.jdbc.driver.*` und `oracle.sql.*` können außerdem für die Verwendung von Erweiterungen "importiert" werden. Dazu `$ORACLE_HOME/jdbc/lib/ojdbc14.jar` in den `CLASSPATH` legen.

- JDBC-Treiber laden:

  Oracle bietet zwei clientseitige JDBC-Treiber an: ***OCI*** und ***Thin***. Diese müssen einmalig im Programm registriert werden*:*

  ```
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
  ```

- Verbindung öffnen (***OCI-Client***):

  Wenn ein ***OCI-Client*** (*Oracle Call Interface*) installiert ist, sollte man den schnellen *JDBC-OCI-Treiber* benutzen. Da dieser plattformabhängig ist, funktioniert er allerdings nicht bei Applets.

  ```
  Connection conn = DriverManager.getConnection
  ("jdbc:oracle:oci:@flores.dbs.ifi.lmu.de:1521:dbprakt","user","password");
  ```

- Verbindung öffnen (***Thin-Client***):

  Der ***JDBC-Thin-Treiber*** hingegen ist plattformunabhängig. Eine Verbindung folgendermaßen aufgebaut:

  - *Aus einer Applikation:*

```
Connection conn = DriverManager.getConnection
("jdbc:oracle:thin:@flores.dbs.ifi.lmu.de:1521:dbprakt","user","password");
```

  - *Aus einem signierten Applet:*

```
import netscape.security.*;
   ...
PrivilegeManager.enablePrivilege("UniversalConnect");
Connection conn = DriverManager.getConnection
                    ("jdbc:oracle:thin:user/password@
                    flores.dbs.ifi.lmu.de:1521:dbprakt");

   ...
PrivilegeManager.revertPrivilege("UniversalConnect");
```

***Applet Security**: Aufgrund der 'Host of origin'-Politik des Java Sicherheitsmodells dürfen unsignierte Applets nur auf den WWW-Server zugreifen von dem sie heruntergeladen worden sind.*

JDBC: Aufbau einer Datenbank-Verbindung (*serverseitig*)

Über den *Oracle JServer* können Java Methoden auch im Datenbankserver gespeichert und ausgeführt werden (Java Stored Procedures). Für den serverseitigen JDBC-Treiber und Verbindungsaufbau sind einige Besonderheiten zu beachten.

```
Connection conn =
   DriverManager.getConnection("jdbc:default:connection");
```

Siehe dazu: Oracle Documentation Library, JDBC Developer's Guide and Reference, Kap. 1, Abschnitt: Server-Side Basics.

Absetzen und Verarbeiten von SQL-Anfragen

- Instanziieren von Anweisungen:

  Auf einer offenen Verbindung **`conn`** werden Statement-Objekte instanziiert, um SQL-Anweisungen an das DBS zu senden. Statements können wiederverwendet werden:

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

- Anfragen absetzen:

  Über das Statement-Objekt **`stmt`** kann nun z.B. eine Anfrage abgesetzt werden:

```
ResultSet rset = stmt.executeQuery ("SELECT * FROM mytable");
```

- Ergebnisse verarbeiten:

  Abhängig vom Datentyp ***<Type>*** der Spalte ***<col_nr>*** können nun die Ergebnisse tupelweise aus dem *ResultSet* **`rset`** (Cursor!) ausgelesen werden:

```
while (rset.next())
     System.out.println (rset.get<Type>(<col_nr>));
```

- Objekte nach Gebrauch schließen (Reihenfolge beachten!):

```
rset.close();
stmt.close();
conn.close();
```

Eine kleine Anfrage:

```
import java.sql.*;
class Employee {
   public static void main (String args []) throws SQLException {
     DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
     Connection conn = DriverManager.getConnection
                          ("jdbc:oracle:oci:@dbprakt", "scott", "tiger");
     Statement stmt = conn.createStatement();
     ResultSet rset = stmt.executeQuery("select * from all_users");
     while (rset.next())
         System.out.println(rset.getString(1));
     rset.close();
     stmt.close();
     conn.close();
   }
}
```

Verarbeiten von *ResultSets*

- Navigation:

  Der Cursor kann mit der Methode **`next()`** des *ResultSets* geprüft und weitergesetzt werden. JDBC ab Version 2.0 erlaubt bereits eine weitgehend freie Positionierung des Cursors im *ResultSet*.

- Datentypen:

  Mit den **`get<Type>()`**-Methoden werden die Ergebnisse spaltenweise eingelesen. Die Java-Variablen müssen dabei zu den JDBC-Datentypen kompatibel sein. Die Spalten werden dabei entweder durch ihre Position (beginnend mit 1) oder durch ihren Namen (als String) identifiziert.

	INTEGER	FLOAT	DOUBLE	DECIMAL	CHAR	VARCHAR	DATE
getInt	●	●	●	●	●	●	
getFloat	●	●	●	●	●	●	
getDouble	●	●	●	●	●	●	
getBigDecimal	●	●	●	●	●	●	
getString	●	●	●	●	●	●	●
getDate					●	●	●

Kombinationen:
rot: empfohlen
schwarz: möglich

Verwendung von Platzhaltern:

- Platzhalter: entweder als `?` oder benannt als `:name` (SQLJ)
- Beispiel:

```
PreparedStatement pstmt =
    conn.prepareStatement("delete from test where col = ?");
pstmt.setString(1,"A");
pstmt.executeUpdate();
```

- Vorteile:
  - Weniger Stringoperationen und Parsing-Aufwand
  - Optimierung durch DBMS
  - Weniger Fehleranfällig (String-Serialisierung, Sonderzeichen!)
  - Weniger Sicherheitsprobleme (SQL Injection!)
- Verwenden bei:
  - Häufigen Statements (wegen Optimierung)
  - Variablen, unsicheren Werten (Fehler, Sicherheit)

Weitere SQL-Anweisungen

- DDL-Befehle:

```
stmt.executeUpdate("create table test(col varchar2(10))");
stmt.executeUpdate("drop table test");
```

- Einfügen, Ändern und Löschen von Daten:

```
stmt.executeUpdate("insert into test values ('A')");
stmt.executeUpdate("update test set col='B' where col='A'");
stmt.executeUpdate("delete from test where col = 'B'");
```

- Vorcompilierte Anweisungen:

```
PreparedStatement pstmt =
    conn.prepareStatement("delete from test where col = ?");
pstmt.setString(1,"A");
pstmt.executeUpdate();
```

- Aufrufen von Stored Procedures:

```
CallableStatement cstmt = conn.prepareCall("{ call proc(?) }");
cstmt.setInt(1,var);
cstmt.executeUpdate();
```

- Allgemeine SQL-Anweisungen:

  Wenn der Typ der SQL-Anweisung (`select`, `insert`, `create`, ...) erst zur Laufzeit bekannt ist, kann man die allgemeinere **`execute()`**−Methode verwenden:

```
String runtime_stmt;

...

if (stmt.execute(runtime_stmt)) {       // runtime_stmt war eine Anfrage
  rset = stmt.getResultSet();

  ...

}

else if (stmt.getUpdateCount() > 0) {
  // runtime_stmt war ein insert-, update- oder delete-Befehl

  ...

}

else { // runtime_stmt hat keinen Rueckgabewert (z.B. DDL)

  ...

}
```

Transaktionen

- Auto-Commit:

  Standardmäßig wird jede SQL-Anweisung, die über ein Statement abgesetzt wird, als eigenständige Transaktion betrachtet. Wenn mehrere DML-Anweisungen in einer Transaktion geblockt werden sollen, muss man den ***Auto-Commit-Modus*** abstellen:

```
conn.setAutoCommit(false);
```

  Auto-Commit kann reaktiviert werden mit:

```
conn.setAutoCommit(true);
```

- Commit Transaction:

```
conn.commit();
```

- Rollback Transaction:

```
conn.rollback();
```

Daneben gibt es eine Fülle von Erweiterungen, z.B. zur Verwaltung von *Batch Updates* oder des *Isolation Levels* einer Transaktion.

Fehlerbehandlung

- Exceptions:

  JDBC macht Fehlersituationen des DBMS über Java Exceptions sichtbar:

```
try {
   // Code, der eine SQL-Exception erzeugt
} catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
}
```

- Warnings:

  Instanzen von *Connection*, *Statement* und *ResultSet* können SQL-Warnmeldungen erzeugen. Diese werden mit der Methode **`getWarning()`** sichtbar:

```
SQLWarning warn = stmt.getWarnings();
   if (warn != null) {
        System.out.println("SQLWarning: " + warn.getMessage());
    }
```

  Warnungen unterbrechen den normalen Programmlauf nicht. Sie machen auf mögliche Problemfälle aufmerksam (z.B. `DataTruncation`).

Übersicht

Embedded SQL in Java: *SQLJ*

- Entwicklungs- und Laufzeitumgebung zur Einbettung statischer SQL-Befehle in Java
- Komponenten von ***SQLJ***:
  - ***SQLJ Translator****:* Präcompiler, der *SQLJ*-Code (*.sqlj) in Java Quellcode (*.java) umwandelt und anschließend automatisch in Java Bytecode compiliert (*.class).
    Aufruf: `sqlj <prog>.sqlj`
  - ***SQLJ Runtime****:* JDBC-basierende Laufzeitumgebung für kompilierte (*.class) SQLJ-Programme. Die *SQLJ Runtime Packages* werden automatisch zur Laufzeit benutzt.
    Aufruf: `java <prog>`
- Installation von ***SQLJ***:

  Sämtliche *SQLJ*-Klassen (100% Java-Code) befinden sich in der Bibliothek `$ORACLE_HOME/sqlj/lib/translator.zip` (in den `CLASSPATH` legen).

Ein kleines Beispiel:

```
import java.sql.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

#sql iterator UserIter(String, Long, Date);

public class SqlJ {
    public static void main(String argv[]) throws SqljException
    {
        DefaultContext cx1 =
          Oracle.getConnection("jdbc:oracle:oci:@dbprakt“, “user",“pass", true);
        DefaultContext.setDefaultContext(cx1);
```

```
    UserIter iter;
    String username = null;
    Long user_id = null;

    #sql iter = { SELECT username, user_id FROM all_users };
    #sql {FETCH :iter INTO :username, :user_id};

    // retrieve and display the result from the SELECT statement
    while (!iter.endFetch()) {
        //TODO do something with the current tuple
        #sql {FETCH :iter INTO :username, :user_id};
    }

    iter.close();
    db.disconnect();
  }
}
```

Übersicht

Java Stored Procedures: Eigenschaften

- Die *Oracle Java Virtual Machine* ist eigene JVM im DB-Server. Komponenten wie *Garbage Collection* und *Class Loader* sind auf die Serverumgebung abgestimmt.
- Prinzipiell sind alle Java-Methoden aufrufbar (Ausnahme: GUI-Programme). D.h. *Java Stored Procedures* als Functions, Procedures, Trigger, PL/SQL-Unterprogramme und Packages.
- Objektrelationale User-Defined Datatypes (siehe Kapitel 6) können ebenfalls unter Java angesprochen werden. Die geschieht über explizites Mapping oder über das Interface ***Struct***, das ein Standard-Mapping bereitstellt.
- Java-Klassen können von PL/SQL und SQL aus angesprochen werden. Hierzu müssen die Klassen über sogenannte ***Call Specs*** im Data-Dictionary publiziert werden.

- Beispiel: Erstellen einer Java Stored Procedure

```
SQL> create or replace and compile java source named USERMANAGER as
import java.sql.*;
public class UserManager {
public static String sampleuser() throws SQLException {
    DriverManager.registerDriver(
           new oracle.jdbc.driver.OracleDriver() );
    Connection conn =
           DriverManager.getConnection("jdbc:default:connection");
    Statement statement = conn.createStatement();
    ResultSet results = statement.executeQuery("select * from all_users”);
    if( results.next() ) {
       return results.getString(1);
    }
    return new String("no user yet“);
    }
}
/
```

- Für Methoden die in Oracle direkt aufgerufen werden sollen werden ***Call Specs*** angelegt. Dies ist nur für Top-Level-Methoden erforderlich. Klassen die von anderen Klassen verwendet werden brauchen nicht publiziert werden.

```
SQL> create or replace function SAMPLEUSER return VARCHAR2 as
2 language java name 'UserManager.sampleuser () return java.lang.String';
3 /
Function created.
```

- Aufruf der Methode

```
SQL> variable myString varchar2[20];
SQL> exec :myString := SAMPLEUSER();
Call completed.
SQL> print myString;
MYSTRING
------------------------------------
USER1
```