3 Database Programming: Accessing SQL from other languages

Learning Goals:

- Understand the importance and fundamental concepts of connecting an application program with a database system, especially the concepts of static and dynamic database programming
- be able to implement a Java program that accesses data in a relational DBMS.

SQL can be used interactively, by using ad-hoc queries entered directly by the user to a prompt that can be started in a shell (for instance by typing mysql to start the MySQL prompt or SQLPLUS to start the Oracle prompt). Such interactive queries can also be run in batch mode by executing through the interactive interface the contents of a file containing SQL statements.

Generally, applications accessing a database are programmed in some general-purpose programming language such as C++, Java, Cobol, or others. Developing database applications, accessing databases from these host languages is called "database programming".

The following steps are usually performed in database programming:
1. Open a connection to the database with account name and password.
2. Execute SQL queries.
3. Close the connection to the database.

There are different approaches how to make use of SQL from other programming languages. Some of the more well-known ones are introduced here.

Since object-oriented programming languages have become so popular, the question how to match the world of objects on the one side to the world of relations and tuples on the other side.
This led to the development of object-relational mapping techniques (ORM).
ORM methods are not explored here because they are discussed in the class Middleware Technologies.

3.1 Embedded SQL

The ISO standard specifies support for Ada, C, Cobol, Fortran, Mumps, Pascal, and PL/1.
MySQL supports for instance C, C++, Perl, Python, Tcl, and Eiffel.

Connecting to the database is for instance done like this:

```
CONNECT TO <server name> AS <connection name>
AUTHORIZATION <username, password>;
```

Closing the connection is done by:

```
DISCONNECT <connection name>;
```

Embedding the SQL statements in a host language means that inside the routines of the programming language, SQL statements are used. They are usually preceded by a prefix like EXEC SQL so that a preprocessor or precompiler can find the SQL parts in the program and replace them by function calls defined in a database interface library. The resulting program can afterwards be treated by the normal compiler of the host language.

The application program and the SQL statements can share variables, so that both can refer to the same values. These shared variables can for instance in C be declared in a special section where the set of variable declarations is framed by the statements EXEC SQL BEGIN DECLARE SECTION and EXEC SQL END DECLARE SECTION. The C variable names can then be used inside the SQL statements and must be preceeded by a colon (:) to differentiate them from other SQL identifiers.

Besides the transferral of SQL result values into shared variables in the program, the database processing gives information to the application program via one of two communication variables (SQLCODE and SQLSTATE; the latter was defined later in the SQL standard). After the execution of SQL statements, the DBMS provides a value for the communication variable, from which the application program can recognize whether any errors or exceptions occurred, for example something went wrong, or no more data is available.

Example (C syntax) of the variable declaration part:

```
int loop;
EXEC SQL BEGIN DECLARE SECTION;
varchar sName [20];
int matNr;
int SQLCODE;
EXEC SQL END DECLARE SECTION;
```

Here the first line is outside of the shared variable declaration. The variable `loop` is a normal C variable that is not shared. Lines 2 and 6 contain SQL statements (they are preceeded with the prefix EXEC SQL). The lines 3-5 are normal C statements again, but the variables declared here ("host variables") will be understood by SQL as well. They are shared. The database system's host language API makes sure that a specified set of data types of the host language are matched to database data types.

Example of a small C routine using embedded SQL:
(using the variables declared above).

```
loop = 1;
while (loop) {
   prompt ("Enter matnumber: ", matNr);
```

```
    EXEC SQL
         SELECT sName, address
         INTO :sName, :address
         FROM Student
         WHERE matNr = :matNr;
    if (SQLCODE == 0) printf (matNr, sName, address)
         else printf ("The number ", matNr, " does not exist!");
    prompt ("For more matnumbers type 1, to exit 0: ", loop);
}
```

In the above example, each query execution delivers just one tuple, which makes the transferral of the values into the shared variables simple.
If the SQL query delivers multiple tuples, a cursor must be defined to browse through the tuples. The cursor is a way of keeping track of the tuples while inspecting a set of them. After definition of a cursor, the command FETCH can be used to fetch the values in the current tuple and move the cursor to the next tuple in the set.

Disadvantage of using embedded SQL:

- The queries must be programmed as part of the application. Whenever a query changes or a new query must be added, the program must be recompiled and debugged.

=> This is sometimes called *static database programming*. It is typically used in situations where at the time of the application specifications, the queries are known and unlikely to change during the lifetime of this system version. An example could be a software to be used at a students' registration office, where the same queries are performed all the time.

Advantage:

- An advantage is that if the queries are known and fixed when the application is programmed, their syntax can be checked to make sure that no errors will occur at runtime.
- Also, the application programming is relatively easy because it is known which and how many variables will be returned as query results and of which type they are. The use of shared variables is straightforward.

3.2 Dynamic SQL

In some situations, static programming is not appropriate because the queries that will be asked are not known beforehand but will be generated only at runtime of the application. In this case, a way is needed to pass a SQL query to the application, for instance as a string of characters, which is then passed to the DBMS that checks the syntax and processes it.

Example of an update query:

```
EXEC SQL BEGIN DECLARE SECTION;
varchar sqlupdatestring [256];
```

```
EXEC SQL END DECLARE SECTION;
...
prompt ("Type in your change request: ", sqlupdatestring);
EXEC SQL EXECUTE IMMEDIATE :sqlupdatestring;
...
```

If the SQL query must be executed several times (in a loop for instance) but can be only entered once (the user does not want to retype the query for every pass through the loop), one can take the EXECUTE IMMEDIATE statement apart. This requires however an additional variable:

```
EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring;
EXEC SQL EXECUTE sqlcommand;
```

In this example, an update query was used, which is simple because it does not return results. The handling of returned values is more complex and done with similar strategies as in ODBC or SQL/CLI.

Advantage:
- Greater flexibility than embedded SQL: Queries can be generated at runtime.

Disadvantages:
- Syntax checking of the queries can only be done at runtime.
- Programming is more difficult because the form of query results is not known when programming the application.
- Not standardized as ODBC, JDBC, or SQL/CLI.

3.3 Using function calls (ODBC, SQL/CLI, JDBC)

A dynamic way to access data in a database using SQL is using an application programming interface (API) of a library of functions that can access the database.
It has the same relative advantages and disadvantages as dynamic SQL versus (static) embedded SQL.

- ODBC (Open Database Connectivity) is a standardized API developed by Microsoft that provides a common interface for accessing heterogeneous SQL databases, based on SQL as the standard for accessing data. ODBC database drivers are available for most DBMSs.
- SQL/CLI (SQL Call Level Interface) was published in its first version as Microsoft's ODBC and was then further developed.
- JDBC is an API to access SQL databases from Java code. It was developed by Sun and modeled after ODBC since ODBC does not allow accessing an SQL database from Java code. There is also a JDBC-ODBC bridge, enabling calls to an ODBC driver from JDBC.
- Some DBMS vendors provide a proprietary API with functions to be used in the code.

In all cases, it is necessary to install the appropriate drivers before one can use the interface. The drivers are specific for each DBMS of the different vendors, but the APIs are standardized, so that one can use the same application program with different DBMSs (provided that one has installed all the necessary drivers).

3.3.1 JDBC

JDBC comes in the form of a package of classes called java.sql.*. They can be downloaded from the Web, for instance from Oracle. As in ODBC and SQL/CLI, the same Java program can access via JDBC different databases on different machines.

Example:

```
import java.io.*;
import java.sql.*;

public class Student {
   ...
   public static void main (String[] args)
         throws SQLException, IOException {

      try {Class.forName("oracle.jdbc.driver.OracleDriver");
           /* forName returns the class or interface object
              with the name specified by the parameter string.
            This line here doesn't do anything with it, but just
            by referring to it, the class is loaded into the
            JVM => load the JDBC driver for Oracle */
      }
      catch (ClassNotFoundException e) {
         System.out.println("Driver could not be loaded.");
      }

      String user, password;
                  //username and password for database login
      String classNr;           //attribute of relation Takes
      int matNr, grade;         //attributes of relation Takes

      user = Reading.readString ("Type the db username: ");
             /* We assume the existence of a class Reading
                providing class methods for reading various data
                 types, here a String.
              */
      password = Reading.readString ("Type the password: ");

      Connection conn = DriverManager.getConnection
                        ("jdbc:oracle:oci8:" + user + "/"
                         + password);
```

```
        /* The class DiverManager handles all open
           connections. The function getConnection takes one
           String argument of the form
           jdbc:oracle:<drivertype>:<dbaccount>/<password>.
           The : are separators used in JDBC.
         */

      myMatNr = Reading.readInt ("Type the matrikel number: ");
              //the matNr of the student whose classes we want

      String query
                  = "select classNr, grade from Takes
                     where matNr = " + myMatNr;

      Statement stmt = conn.createStatement();
                // create a statement in the connection conn

      ResultSet result = stmt.executeQuery(query);
                /* execute the query for the created statement
                   and return the resulting set of tuples.
                   A result set is similar to a cursor in
                   embedded SQL or an iterator in SQLJ. */


      while (result.next()) {/*next() is null when no objects
                               are left. Otherwise goes to the
                               next result object. */
         classNr = result.getString(1);
                  /*read position 1 in the current result tuple
                     and return it as a String */
         grade = result.getInteger(2);
                  /*read position 2 in the current result tuple
                     and return it as an integer */
         System.out.println (classNr + ": " + grade);
      } //while
    conn.close(); //close the database connection
    } //main
} //Student
```

Note:

- For different types of queries, the methods executeQuery (for retrieval) and execute-Update (for insert, delete, and update). The method executeQuery returns an object of type ResultSet that provides methods to work with the result tuples. The method executeUpdate returns an integer value indicating the number of tuples that were affected.
- JDBC does not differentiate between queries that return one tuple and those that return several. The first is just a special case of the latter. => Simpler than other techniques.
- A JDBC tutorial can be found at http://download.oracle.com/javase/tutorial/jdbc/.

3.3.2 SQL/CLI (Call Level Interface)

SQL/CLI was developed as a standardization effort by the SQL Access Group. It is still part of the SQL standard, and very similar to ODBC. However, ODBC actually became a de facto standard and is used more often than CLI.
As in ODBC and JDBC, in SQL/CLI, SQL statements are passed to the database as character strings via parameters of function calls. Thus the queries can be flexibly generated at runtime. An example of SQL/CLI is shown here.

Example of SQL/CLI in C code:

```
#include sqlcli.h; // use the library of CLI functions

void printClassInfo() { // print to stdout all info about a particular class

   // variable declarations:
   SQLHSTMT statement;            //info about one SQL statement
   SQLHDBC connection;            //info about one particular connection
   SQLHENV environment;           //info about all connections
   /* SQLHDESC description; info about tuples, attributes, types,
      parameters. Not used in this example to keep it simpler
    */
   SQLRETURN ret1, ret2, ret3, ret4; /* return values of the CLI function
                                        calls. 0 means: call successful.
                                     */
   int room;
   char day[8], pName[20], classNr[8];
   int fetchlength1, fetchlength2, fetchlength3;


   ret1 = SQLAllocHAndle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
                         &environment);
                // CLI function. See explanation after the code example.

   if (!ret1) ret2 = SQLAllocHAndle(SQL_HANDLE_DBC, environment,
                                    &connection)
   else exit; /* If an environment was created successfully, we create
                 a connection in it. Otherwise, the function is terminated.
              */

   if (!ret2) ret3 = SQLConnect (connection, "dbs", SQL_NTS,
                                 "koch", SQL_NTS, "binky", SQL_NTS)
   else exit;
        /* If a connection was created successfully, establish it to the
           database server dbs for the user koch with the password binky.
           Otherwise exit the function.
           SQL_NTS: "null terminated string", makes SQL calculate the string
           length automatically instead of having to specify it.
         */

   if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, connection,
                                    &statement)
```

```
   else exit;
              /* If the connection was established successfully, create a
                 statement record in this connection.
                 Otherwise exit the function.
              */

   SQLPrepare (statement,
               "select room, day, pName from Class where classNr = ?",
               SQL_NTS);
        /* An SQL query is associated with the created statement record.
           The ? indicates a statement parameter whose value can be given
           at runtime, by binding it to a C variable.
         */

   prompt ("Enter a class number: ", classNr);
        /* assumes the existence of a function prompt that prompts the
           user with the specified text and reads into the variable the
           typed in reply.
         */

   SQLBindParameter (statement, 1, SQL_CHAR, &classNr, 8, &fetchlength1);
        /* The value of the C variable classNr is bound to ? on position 1
           in the statement string. It is of type Char and has a
           maximum of 8 characters. The actual length to be assigned
           is examined and assigned to the variable fetchlength1.
         */

   ret1 = SQLExecute(statement); //execute the SQL statement

   if (!ret1) {                /*if everything ok, bind the attributes
                                 of the query to the C variables */
      SQLBindCol(statement, 1, SQL_INT, &room, 2, &fetchlength1);
      SQLBindCol(statement, 2, SQL_CHAR, &day, 8, &fetchlength2);
      SQLBindCol(statement, 3, SQL_CHAR, &pName, 20, &fetchlength3);

      ret2 = SQLFetch (statement); /* Get the next tuple in the result,
                           here only one, and copy the attribute
                           values into the variables bound to them.
                           If no tuples are left, the returned value = 0 */
      if (!ret2) {     // If fetch was successful, output the result
         printf (classNr, room, day, pName)
         else printf ("Class number ", classNr, "does not exist.");
      } //end if
   } end if
} end printClassInfo()
```

Further explanations:

The function SQLAllocHandle(<handleType>, <handle1>, <handle2>) is used to create a record for environment, connection, statement, or description.

The meaning of the parameters is:

- handleType: this can assume the constant values SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC to indicate the created record is of the kind environment, database connection, SQL statement, or description, respectively.
- handle1: indicates in which container the new record is created. For a connection:

the environment to which it belongs; for a statement: which connection it belongs to; for a description: the statement to which it belongs; for an environment: the value is null because it is not part of any container. The null pointer is indicated by the constant SQL_NULL_HANDLE.
- handle2: the value of this is a pointer to the newly created record which has the type indicated in handleType.

In the above example, a single tuple was retrieved. For the retrieval of several tuples, the SQLFetch (statement) will be repeated in a while loop until no more tuples are available.
For an example with several tuples see for instance Elmasri/Navathe, 4th edition.

Also, in the above example, the SQL query was given as a string constant in the code, i.e. again in a static way. If the query is given as a string parameter that is input by the user, the handling becomes more complex because the number and types of the retrieved attributes is not known in advance. Also, the type of the query is not known in advance (select, update, delete, create_table, etc.). The description record must be used for this purpose.

All CLI functions return a 16-bit value of the data type SQLRETURN which is declared in the file sqlcli.h:
typedef SQLSMALLINT SQLRETURN;

In standard SQL there are only six possible SQLRETURN values:
-2: „invalid handle"
-1: „error".
0: „success".
1: "success with information".
99: "need data".
100: "no data".

3.4 SQLJ

Several companies (Oracle (and Sun Microsystems which was back then still separate from Oracle), IBM, Informix, Sybase, Compaq, Cloudscape) developed SQLJ as a way of embedding SQL code in a Java program. SQLJ statements are compiled to Java code that uses JDBC functions to interact with the database. Therefore a JDBC driver must be installed in order to use SQLJ. Today (2013) SQLJ is considered outdated because it cannot be connected with Object Relational Mapping frameworks (see later section in this chapter).
This section is still left in to show the idea which was actually good.

Opening and closing a connection can be done for instance like this:

```
import java.sql.*;
import java.io.*;
import sqlj.runtime.*;
import oracle.sqlj.runtime.*; //one of these classes
                              //is the class Oracle
```

```
...
// in some method:
DefaultContext mycontext = Oracle.getConnection
       ("<url name>", "<user name>", "<password>");
       // open a connection.
DefaultContext.setDefaultContext(mycontext);
... //perform the necessary queries
#sql [myContext] { commit }; //commit all changes done in the
                             // context myContext. Note: several
                            // contexts may be open concurrently
mycontext.close(); //close the connection
```

The parameter database url could be for instance:
`jdbc:oracle:thin:@noether.lida.fht-stuttgart.de:1521:orac`
-> use the JDBC Thin driver to connect to a database on the server noether.lida.fht-stuttgart.de through port 1521, where orac is the SID (system id, to identify which database) of the database to connect to.

SQL statements in SQLJ are preceded by the prefix #sql for identification by the pre-processor.
An INTO clause is used for transferring the SQL result values to the Java variables (similar as in embedded SQL).
The shared variables are preceded by colons (:) as in embedded SQL. However, they do not need to be declared in a way different from other Java variables.

Example:
(part of a Java method)

```
String classNr, day, pName;
int room;

classNr = Reading.readString ("Type the class number: ");
    /* We assume the existence of a class Reading providing
       class methods for reading various data types,
       here a String.
     */
try {#sql { select day, room, pName
            into :day, :room, :pName
            from Class where classNr = :classNr};
    }
catch (SQLException e) {
       System.out.println (e.getMessage());
       e.printStackTrace();
}
System.out.println ("The class " + classNr + "is given by " +
     pName + "and takes place on " + day + " in room " + room);
```

In this example, it was known that only one tuple would be returned.
For queries that retrieve more tuples, an iterator is defined, a concept comparable to

the cursor in embedded SQL. There are two types of iterators in SQLJ: named iterators (use attribute names and types) and positional iterators (use only attribute types). We consider only an example with a named iterator here:

List the information about all teaching assistants for a class entered by the user:

```
String classNr;
int matNr;
double hours, taSalary;

classNr = Reading.readString ("Type the class number: ");
    /* We assume the existence of a class Reading providing
       class methods for reading various data types,
       here a String.
     */

System.out.println ("Teaching assistants for " + classNr);

#sql iterator Tas(int matNr, double hours, double taSalaray);
        /* declare new iterator with name Tas and the specified
           attributes that must match the ones in the query! */
Tas t = null;   //instantiate new iterator object and initialize
#sql t = {select matNr, hours, taSalary
          from TA
          where classNr = :classNr};

while (t.next()) {
   /* the method next() iterates to the next tuple in the
      query result. If no more tuples are left, the value
      null is returned and the loop is exited.
    */
   System.out.println ("Matnr: " + t.matNr + "hours: " + t.hours
                       "salary: " + t.taSalary);
} //while
t.close(); //close the iterator
```

The positional iterator is somewhat more similar to embedded SQL in that it uses a FETCH operation to transfer the retrieved values INTO the program variables. See literature.

Advantage of SQLJ versus JDBC:
SQLJ is easier to use (fewer lines of code and easier to understand).

So far, SQLJ has only been supported by Oracle, IBM and Sybase.

3.5 Proprietary APIs

Before the development of ODBC and SQL/CLI, some database vendors provided a proprietary API. The problem with this was that if the same application was supposed to be used for different DBMSs, it had to be rewritten according to each DBMS's API and run through the respective preprocessor again. This defied interoperability.

3.6 Object-relational Mapping (ORM)

In order to better bridge the gap between object-oriented classes in applications and non-object-oriented databases, object-relational mapping frameworks have been developed.
They provide APIs with methods to access the database without actually using SQL (this is hidden in lower layers of the frameworks). Hence, the relational database appears to have an object-oriented structure.

There are many ORM frameworks (sometimes also called persistence frameworks), commercial as well as open-source.

A first approach in this direction was published in 2001 by Sun, IBM, and Apple under the name Java Data Objects (JDO). Since 2006, JDO is being further developed by the Apache Software Foundation under the name Apache DB Project.

Popular frameworks today are for instance
- Hibernate (open-source, for Java)
- ORMLite (open source, for Java)
- Oracle TopLink (for Java)
- Doctrine (open-source, for PHP)
- SQLAlchemy (open-source, for Python)

EJB (Enterprise JavaBeans) also contains ORM functionality.
As part of EJB, a new Java API was developed:
Java Persistance API (JPA) (a Java API, defined in the package javax.persistence; there are open-source as well as commercial implementations of this API).

ORM is available also for other programming languages, like C++, C#, or Smalltalk.

The topic of ORM is somewhat removed from direct database technology. In order to use it, it is necessary to make use of general software development technologies, entity-relationship modelling, and transaction concepts.
The topic is typically categorized under middleware technologies.