WIRTSCHAFTS UNIVERSITÄT WIEN VIENNA UNIVERSITY OF ECONOMICS AND BUSINESS

BACHELOR'S THESIS

Managing an SQLite database with ooRexx and JDBC using BSF4ooRexx

submitted by

Cornelia Hofstetter

intended degree

Bachelor of Science (WU), BSc (WU)

Student ID number:	**01550814**
Degree programme:	Wirtschafts- und Sozialwissenschaften
Supervisor:	ao.Univ.Prof. Dr. Rony G. Flatscher

Vienna, July 2025

Declaration of Originality

I hereby declare that:

1. I have written this Bachelor's thesis myself, independently and without the aid of unfair or unauthorized resources. Whenever content has been taken directly or indirectly from other sources, this has been indicated and the source referenced.
2. This Bachelor's Thesis has not been previously presented as an examination paper in this or any other form in Austria or abroad.
3. This Bachelor's Thesis is identical with the thesis assessed by the examiner.

22.07.2025 _____ _____

Date Signature

Abstract

BSF4ooRexx, which is a bridge between ooRexx and Java, enables the usage of all Java capabilities such as the JDBC driver to connect to databases. The objective of this bachelor thesis is to demonstrate the abilities of ooRexx and BSF4ooRexx specifically to manage a SQLite database by invoking SQL statements using the JDBC driver. This will be done in the form of short nutshell examples to showcase the simplicity accompanied by an explanation of the SQL and ooRexx commands. Additionally, this thesis aims to serve as a complete installation guide for the software components necessary to run the nutshell programs on a windows operating system.

Table of Contents

List of Figures

List of Codes

1 Introduction

This chapter introduces the programming languages used in this bachelor thesis which are ooRexx and SQL as well as Java through BSF4ooRexx. First, we will take a closer look at ooRexx, its history and syntax. Then the focus will be on BSF4ooRexx, the bridge to the widely used programming language Java. Lastly the relational database engine SQLite as well as the JDBC Driver and the DB Browser will be presented in more detail.

1.1 ooRexx

ooRexx which is short for Open Object Rexx is a high-level, object-oriented programming language that works on all operating systems and is maintained to this day. Because of its cross-platform interoperability ensured by its compliancy with the "Information Technology – Programming Language REXX" ANSI X3.274-1996 standard programs from its predecessor "classic" Rexx work under ooRexx as well. (Wikipedia, 2025-a)

1.1.1 History

As Smalltalk became the main programming language at IBM in 1988 the project Oryx led by Simon C. Nash was tasked with merging the "classic" Rexx language with the object model of Smalltalk. As a result, Object Rexx was developed and presented in 1992. (EDM2, 2019)

The discontinuation of Object Rexx at IBM led to the transfer of the source code and licensing rights to the non-profit Rexx Language Association (RexxLA) in 2004. ooRexx was released in 2005 as free and open-source software and has been continuously improved ever since, ooRexx 5.2.0 being the newest version released in 2025. (Wikipedia, 2025-a)

1.1.2 Language

Rexx is an easy to learn programming language, that was designed with the general user in mind in contrast to languages that require more advanced programming knowledge such as C or Fortran. (Cowlishaw, 1984)

A focus lies on readability which is implemented in a way that Rexx code reads almost like normal text. There are several design choices for the purpose of improving readability such as the support for mixed upper- and lower-case letters, usage of blanks in the most readable way and the omission of punctuation. (Cowlishaw, 1984)

In an attempt to make the programming language as user-friendly as possible particular value was placed on high predictability of features as well as consistency without making it too restrictive. (Cowlishaw, 1984)

As opposed to the popular strong typing languages, Rexx supports natural data typing which means all data are defined in the same form and only checked depending on their usage. (Cowlishaw, 1984) Also variables do not need to be declared.

In accordance with the emphasis on Rexx being “human centric” it was a design choice to keep the language small with few key words to make it easier to learn and remember. (Flatscher, 2013)

1.2 BSF4ooRexx

BSF4ooRexx was developed based on the Bean Scripting Framework which was created by IBM to allow the use of scripting in Java code. It acts as a bridge between ooRexx and Java that allows for communication in both directions. For Java objects to be created in ooRexx the BSF.CLS package is required. (Wikipedia, 2024)

1.3 SQLite

SQLite was started in 2000 and is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. (SQLite, 2025) The entire database including definitions, tables, indices and data is stored as a single file that can be accessed by more than one process at the same time. To make this possible the file is locked during writing. (Wikipedia, 2025-b)

SQLite can be operated without the use of a database management system or a database administrator. (Wikipedia, 2025-b) In the process of developing the nutshell examples for this thesis the command-line tool as well as the DB Browser for SQLite were used to navigate and manipulate an SQLite database.

1.3.1 JDBC Driver

Via the Java Database Connectivity API any kind of tabular data as in this case a SQLite-database can be accessed by Java applications. The JDBC driver is necessary to connect to the database, send queries and update statements to the database and retrieve and process the results received as a response from the database. (Oracle, 2024)

1.3.2 DB Browser

The DB Browser for SQLite (DB4S) is a high quality, visual, open source tool designed to create, search and edit SQLite or SQLCipher database files. (SQLitebrowser, 2025) It works on all operating systems and provides a graphical interface that allows for the management of tables and data. Additionally, the execution of SQL queries is supported in order to define and manipulate SQLite databases. (SQLitebrowser, 2025)

2 Required Software

The following programs are needed to run the nutshell programs from chapter 4. For each software component a quick installation guide is provided including screenshots for every step. The installation steps mentioned are only valid for the Windows operating system, but there are versions of all the programs available for Linux and Apple. The full download links are given at the end of the thesis.

2.1 ooRexx

ooRexx is available for all operating systems at sourceforge.net where older versions can be downloaded as well. For this paper the following version was used: ooRexx 5.0.0-12583.windows.x86_64.exe.

Figure 1: Download link ooRexx

After downloading it is important to right click on the file in the download folder and accept the security warning to avoid an error message when trying to execute the file.

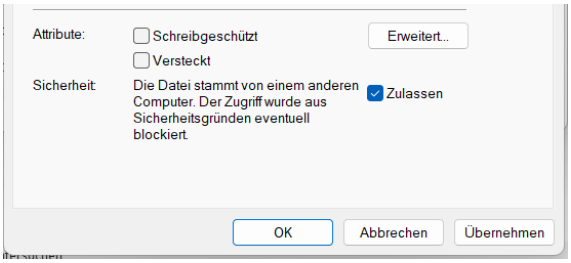

Figure 2: Unblocking file

Then simply follow the steps of the installation wizard as shown in the following screenshots.

Figure 3: Installation process ooRexx

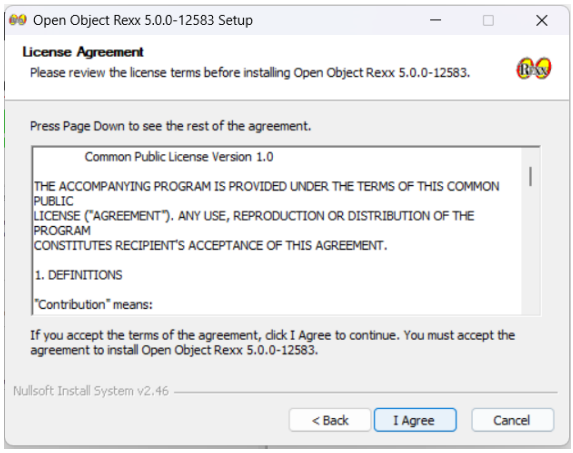

Figure 3 (continued): Installation process ooRexx

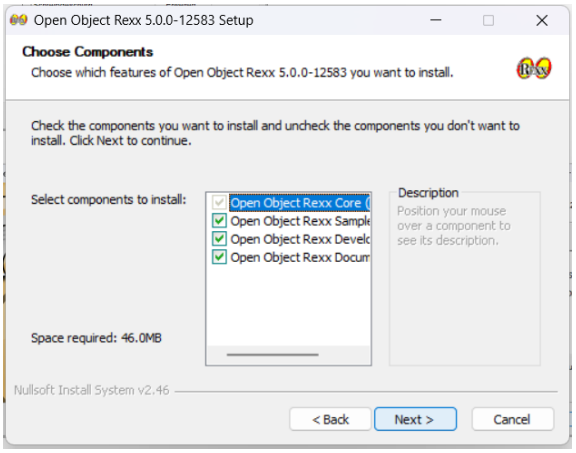

Figure 3 (continued): Installation process ooRexx

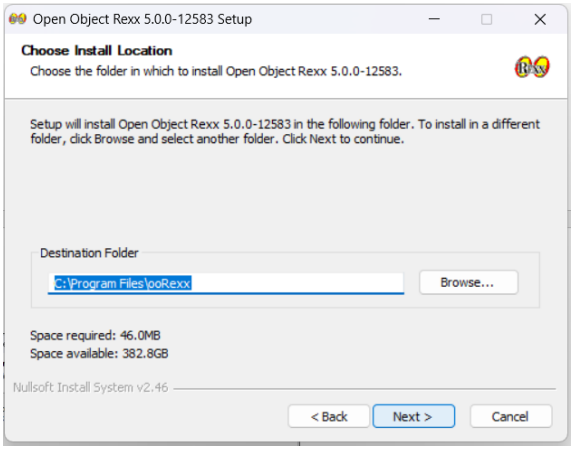

Figure 3 (continued): Installation process ooRexx

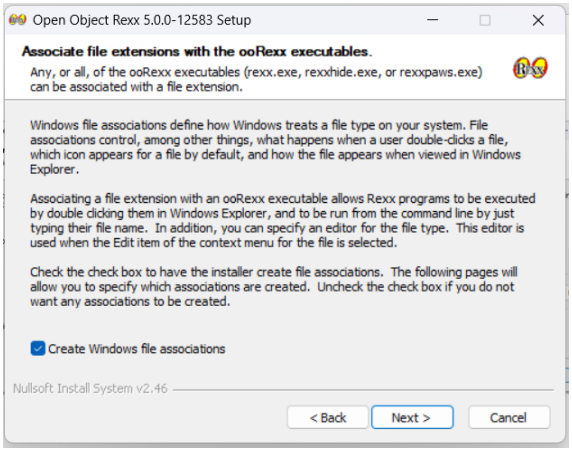

Figure 3 (continued): Installation process ooRexx

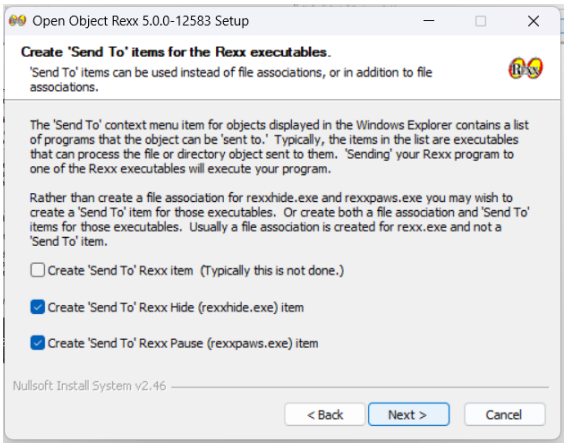

Figure 3 (continued): Installation process ooRexx

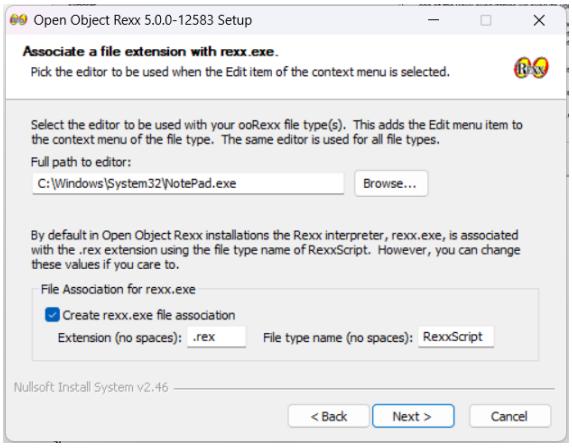

Figure 3 (continued): Installation process ooRexx

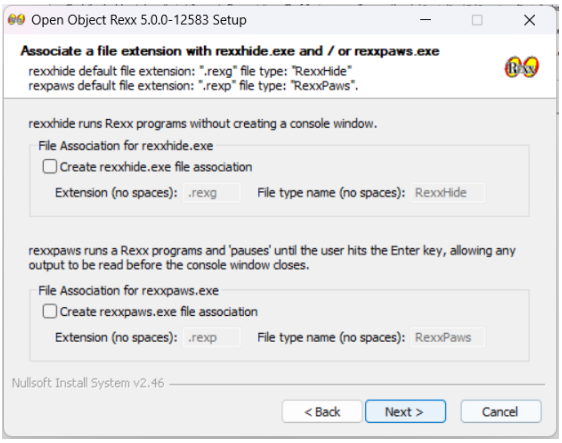

Figure 3 (continued): Installation process ooRexx

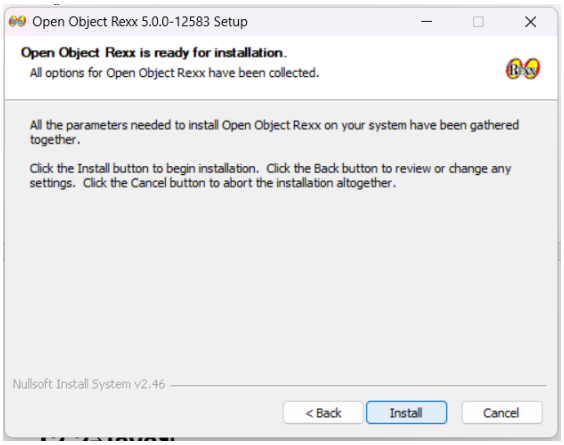

Figure 3 (continued): Installation process ooRexx

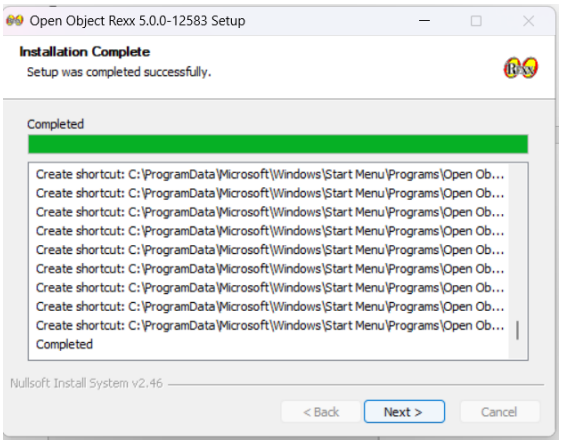

Figure 3 (continued): Installation process ooRexx

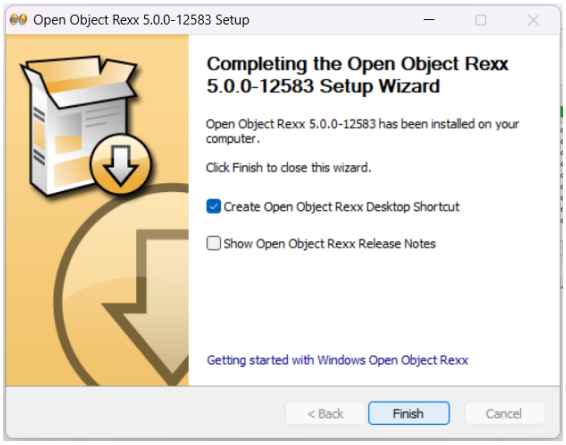

Figure 3 (continued): Installation process ooRexx

2.2 Java

Java can be retrieved from different websites such as azul.com and java.com. In this case Liberica Full JDK 24.0.1+11 x86 64 for Windows from bell-sw.com was installed, but older versions are sufficient as well to run the nutshell programs in chapter 4.

Figure 4: Download link Java

Just follow the installation client. Nothing needs to be changed.

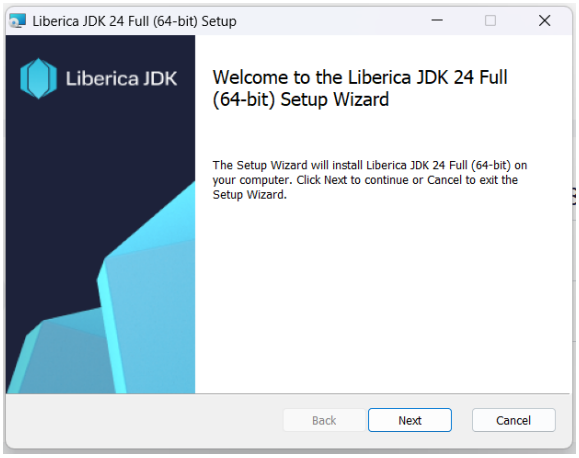

Figure 5: Installation process Java

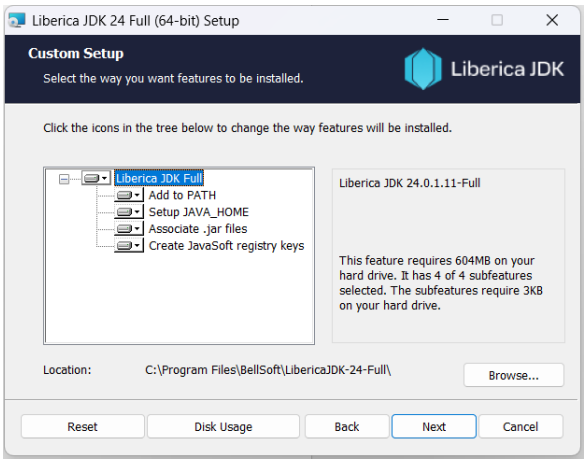

Figure 5 (continued): Installation process Java

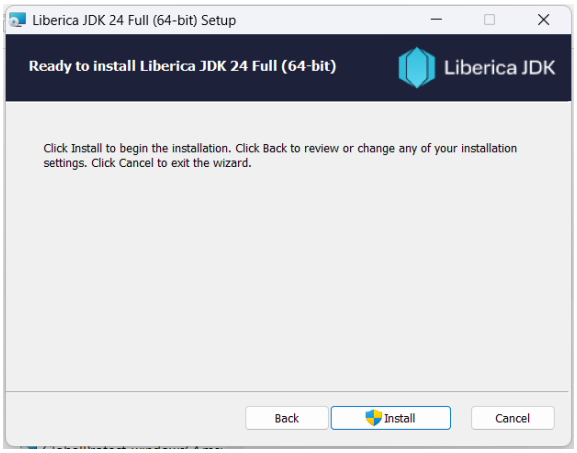

Figure 5 (continued): Installation process Java

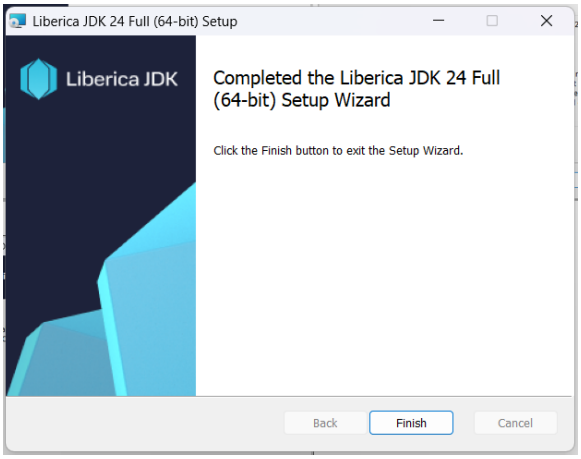

Figure 5 (continued): Installation process Java

2.3 BSF4ooRexx

After the successful installation of ooRexx and Java the “bridge” BSF4ooRexx can be installed. The download link of BSF4ooRexx_install_v850-20240707-refresh is available on sourceforge.net as well as older versions.

Figure 6: Download link BSF4ooRexx

As with the ooRexx file the downloaded .zip-file needs to be “unblocked” before unpacking.

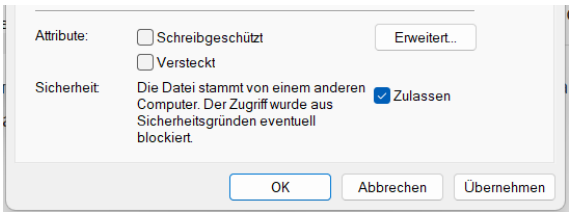

Figure 7: Unblocking file

In the unpacked BSF4ooRexx_install_v850-20240707-refresh folder go to “install”, then select the right operating system and click on install.cmd.

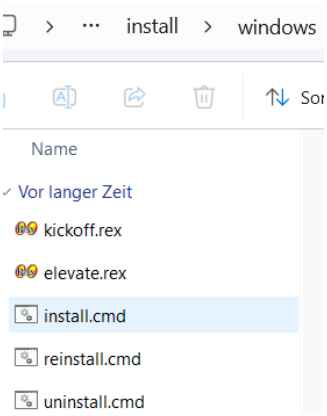

Figure 8: Installation file BSF4ooRexx

Then a command window opens to install the program. If neither Libre Office nor Open Office is installed there will be a warning message. In that case press Enter to continue the installation process.

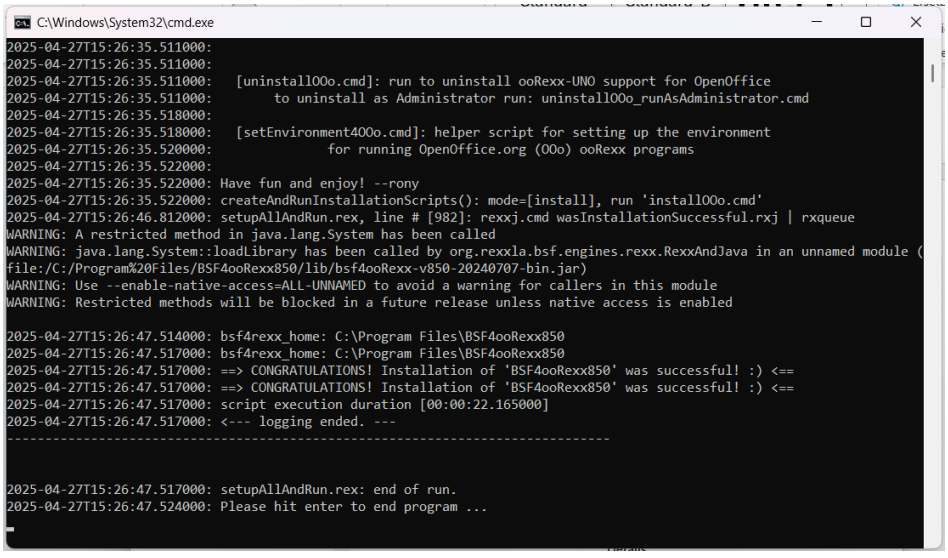

Figure 9: Installation process BSF4ooRexx

After the installation process you should be able to run Rexx programs using the Bean Scripting Framework via the shortcut GUI RexxTry Program or in the command line.

2.4 IntelliJ

Figure 10: Download link IntelliJ

As there is a ooRexx plugin available for the integrated development environment IntelliJ it was used for developing and running the nutshell programs. Apart from the fee-based version for professional development there is a community edition that is completely free and sufficient in this case. Both can be downloaded from the jetbrains.com download section. Then just execute the .exe file and follow the installation wizard.

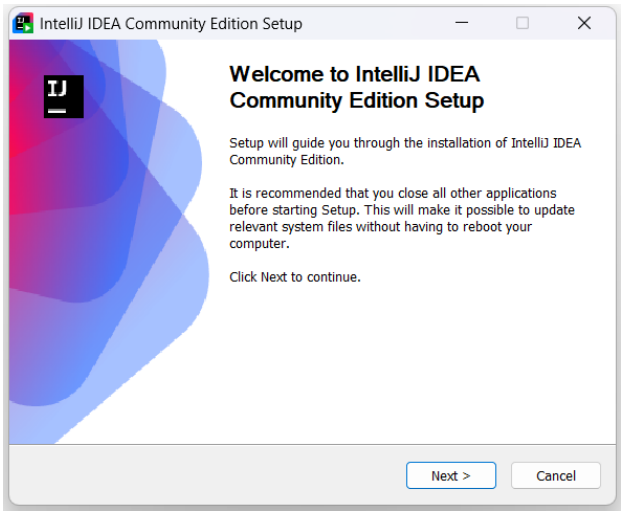

Figure 11: Installation process IntelliJ

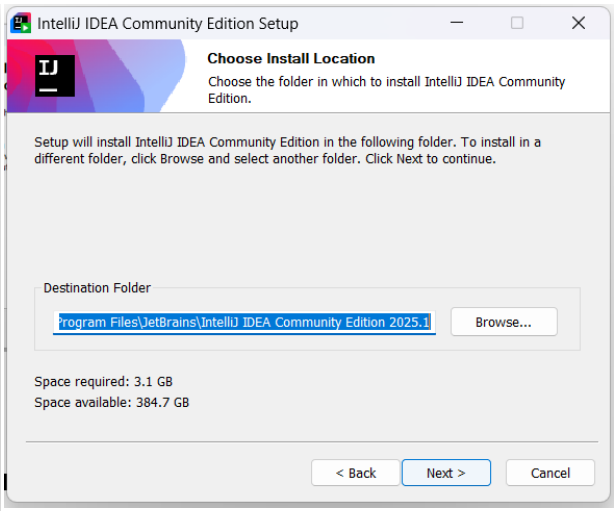

Figure 11 (continued): Installation process IntelliJ

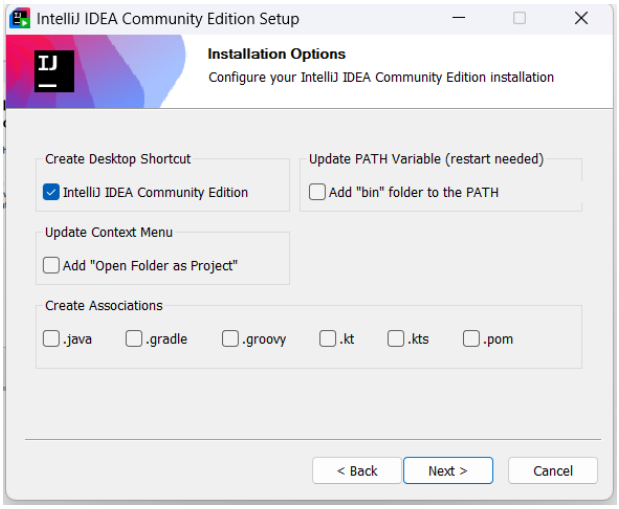

Figure 11 (continued): Installation process IntelliJ

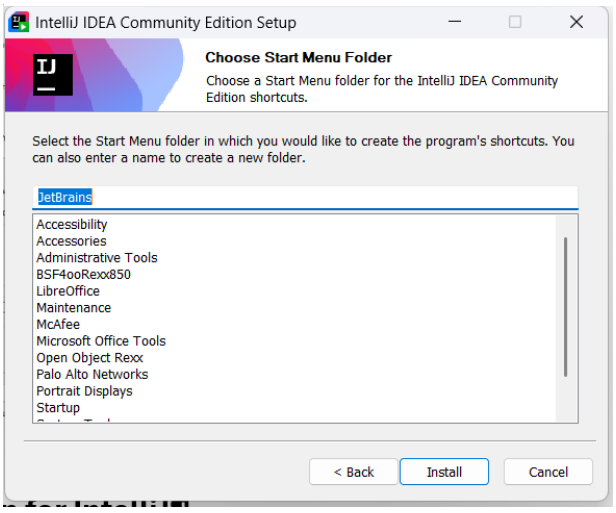

Figure 11 (continued): Installation process IntelliJ

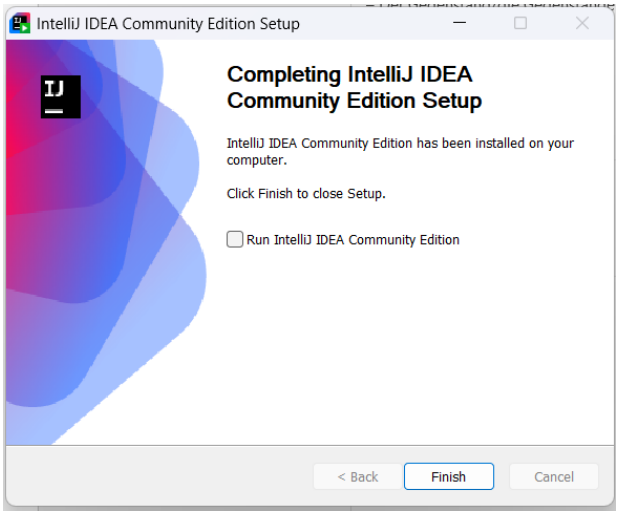

Figure 11 (continued): Installation process IntelliJ

2.5 ooRexx Plugin for IntelliJ

In order to program in ooRexx the ooRexx plugin for IntelliJ is required. It can be retrieved from sourceforge.net. For this paper the version ooRexxPlugin-2.5.0-GA was used.

ooRexxPlugin-2.5.0-GA.zip

Figure 12: Download link ooRexx Plugin

It is important to notice that the downloaded .zip file does not need to be unpacked, but only installed in IntelliJ as follows: On the startpage of IntelliJ go to “Settings”, then “Plugins” and in the drop-down menu select “Install Plugin from Disk”. Choose the downloaded .zip file and you should be able to use ooRexx as programming language in IntelliJ.

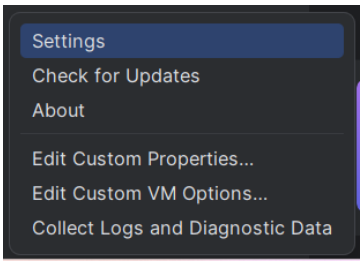

Figure 13: Configuration in IntelliJ

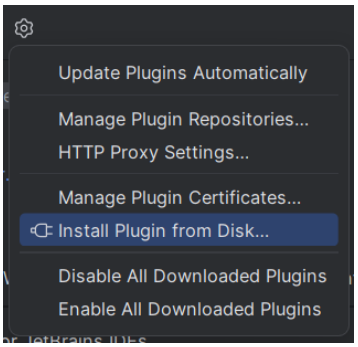

Figure 13 (continued): Configuration in IntelliJ

2.6 JDBC Driver

The JDBC driver necessary for the SQLite database connection can be downloaded from github.com.

sqlite-jdbc-3.49.1.0.jar

Figure 14: Download link JDBC Driver

In order to be found by Java the downloaded .jar-file needs to be put in the class path. This can be done by putting the file in the lib directory of the program folder BSF4ooRexx850 as it is part of the class path.

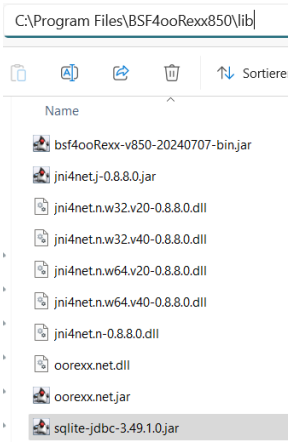

Figure 15: Path for .jar-file

The .jar-file could also be placed in the following newly created directory: "%USERPROFILE%\BSF4ooRexx\lib" (Windows) or "$HOME/BSF4ooRexx/lib". Another option is to check and adapt the current environment variables. To do this type “sysdm.cpl” in the search bar, go to “Advanced” and then select “Environment variables”. There select the line “CLASSPATH” and click on the “Edit” icon. Then either add a path via “New” or search for an existing folder.

2.7 SQLite

SQLite can be retrieved from the download page of sqlite.org, https://www.sqlite.org/download.html. As it is the case with the ooRexx programs the .zip file needs to be “unblocked” before unpacking.

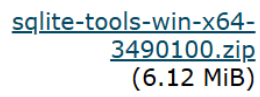

Figure 16: Download link SQLite

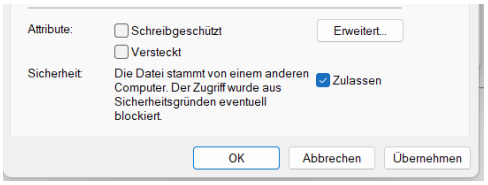

Figure 17: Unblock file

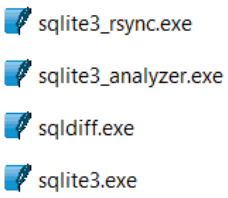

Figure 18: SQLite command line tools

2.8 DB Browser

The DB Browser for SQLite, which allows for a more graphical interface for managing SQLite databases can be downloaded from https://sqlitebrowser.org/dl/. There are two options the standard installer and a .zip version without installer. Here the “Standard installer for 64-bit Windows” was selected.

- DB Browser for SQLite - Standard installer for 64-bit Windows

Figure 19: Download link DB Browser

Then execute the downloaded file and just follow the installation wizard according to the screenshots below.

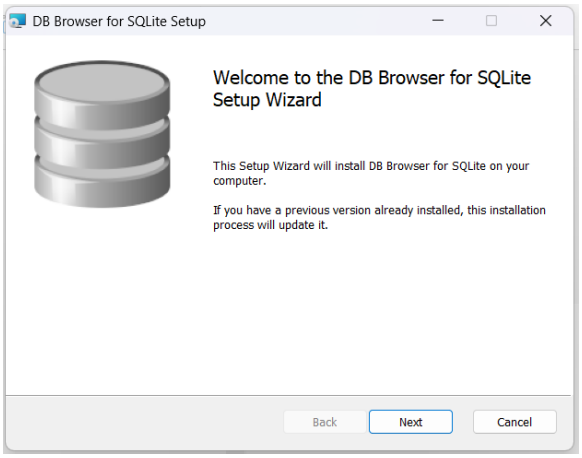

Figure 20: Installation process DB Browser

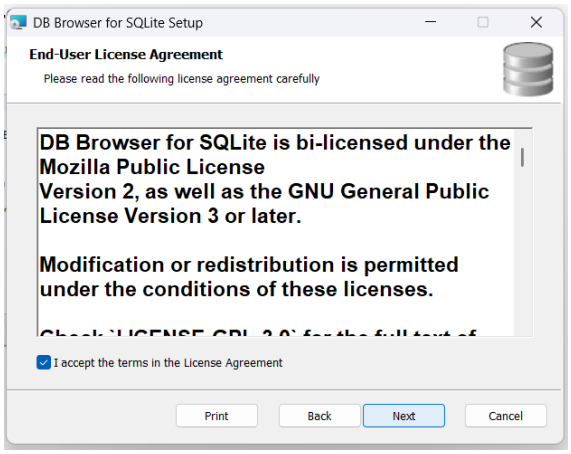

Figure 20 (continued): Installation process DB Browser

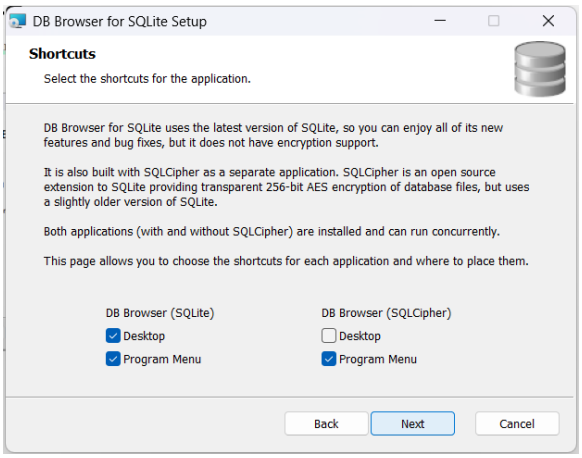

Figure 20 (continued): Installation process DB Browser

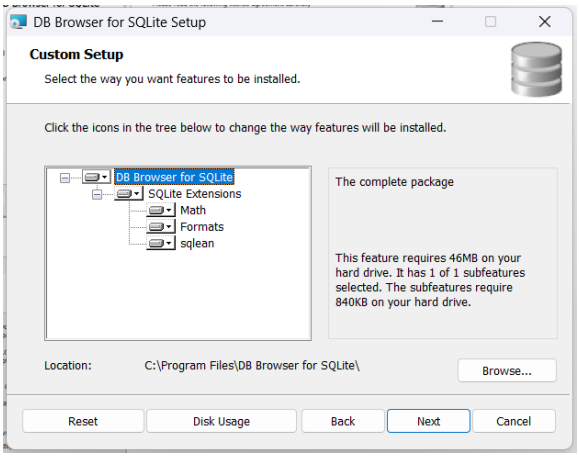

Figure 20 (continued): Installation process DB Browser

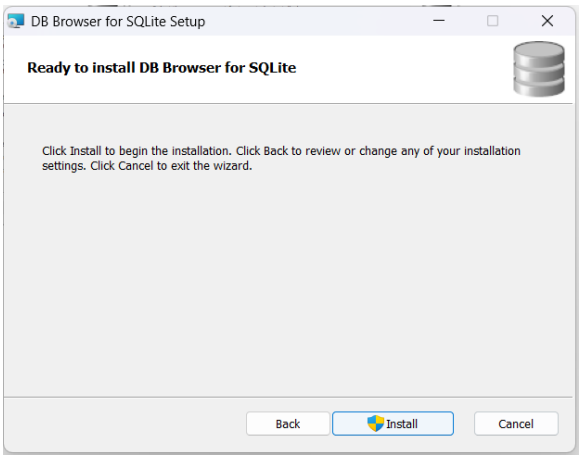

Figure 20 (continued): Installation process DB Browser

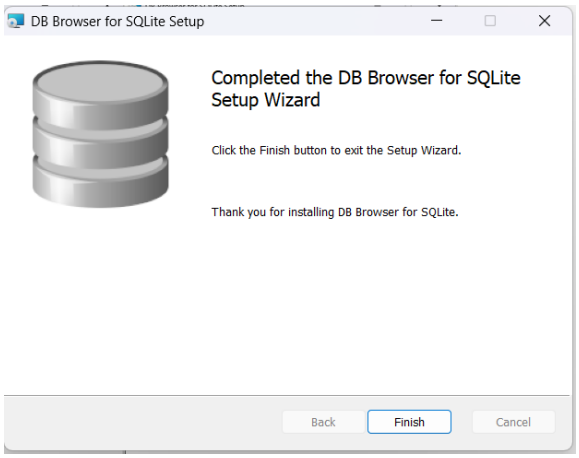

Figure 20 (continued): Installation process DB Browser

3 Database Theory

In this chapter some theoretical concepts concerning databases in general and the different types of database systems will be discussed. First a general definition will be given. Then the difference between relational and non-relational databases will be identified followed by information on database management systems and the query language SQL.

3.1 Definition

“A database is an organized collection of structured information, or data, typically stored electronically in a computer system.” (Oracle, 2020)

A database management system such as MySQL can be used to operate the database which results in a database system. A database usually consists of tables that store interconnected data in rows and columns. (Oracle, 2020)

3.2 Relational Database

Depending on the kind of usage different types of databases can be identified.

Relational databases are based on the relational model which provides a standard to unify the way how data is stored, namely in formally described tables. Before it took a lot of knowledge about the specific data base to retrieve data as there was no common way to structure databases. (Oracle, 2021)

Logical and physical data are stored separately. Changing the physical structure of the database does not affect the logical side such as tables, views and indexes. (Oracle, 2021)

A relational database consists of tables with columns and rows that each contain a unique instance of data. The tables are also in a relationship with each other. Typically, SQL is used to access and modify the data in relational databases. (Jatana et al., 2012)

3.2.1 OLTP

OLTP stands for Online Transaction Processing and is mostly used by relational databases as the data must always be consistent for fast-paced requirements such as the day-to-day operations in organizations like sales, accounting, manufacturing and so forth. (Harrington, 2016)

3.2.2 ACID

In contrast to non-relational databases relational databases are compliant with the ACID model that is used to check the reliability of databases. The properties of the ACID models are as follows:

- Atomicity means that a transaction is considered failed, if not every single part of the transaction was completed successfully.
- Consistency stands for a valid state of the database before and after a transaction.
- Isolation requires a serialised processing of transactions, so that they cannot affect each other when they are executed at the same time.
- Durability ensures that data stays in the same state and does not change after a finished transaction. (Jatana et al., 2012)

3.2.3 Normalization

The concept of normalization of databases aims to avoid data redundancy and thus problems with updates of the tables. Redundancy means that there is at least one item that occurs more than once in a database. Duplicates add potentially a lot of work to changes like “insert”, “update” or “delete” as they need to be executed on all the tables that include the same data. (Eessaar, 2016)

There are up to six normal forms, but usually only the criteria of the first three are met. The first normal form requires that a single cell does not hold more than one value, there must be at least a composite primary key, rows and columns are unique and for each column and row there can only be one value in the table.

For the second normal form the first normal form needs to be already established as a prerequisite and there cannot be any repeating groups. That means some values are dependent on a part of the composite primary key, which is not allowed in the second normal form. In order to solve this, the dependencies are shown in separate tables.

In third normal form it is not allowed that an attribute which is not part of the primary key is dependent on another non-prime attribute and the tables must be in second normal form as well. This connection needs to be placed in a separate table in order to comply with the third normal form. (FreeCodeCamp, 2022)

3.3 Non-Relational Database

Non-relational databases do not rely on related tables as related databases do, but have different ways to structure, store and retrieve data. While they do not necessarily provide the ACID properties as discussed before they eventually are consistent. They also do not possess a fixed schema, nor do they support the query language SQL. (Jatana et al., 2012)

So called NOSQL databases are primarily classified on how they store the data. (Jatana et al., 2012)

3.3.1 Key Value Store

The data is stored in pairs of a string key and the value, which can be of any type such as string, integer, array or an object. For this schema-less kind of data storage the requirement of a fixed data model goes away.

3.3.2 Document Store

Here a computer program with a storage structure that is called a document is responsible for managing data stored in a database. The data is encoded in a standard format such as XML, BSON, PDF or Microsoft office. Via queries or APIs documents corresponding to certain parameters can be retrieved quickly.

3.3.3 Graph Database

The data is represented via nodes, properties and edges. The nodes stand for entities like people or objects and possess further information represented by properties. Together with the edges, that show the relationship of the nodes the full picture of the database can be observed.

3.3.4 Column Oriented Database

In contrast to row-oriented databases column store databases provide data storage in columns which leads to the serialization of all data of one column. Adding new values for all rows of a column is more efficient as the other columns remain unchanged.

3.3.5 Object Oriented Database

Data is stored as objects in a database system. As it is the case in object-oriented programming inheritance and thus reusability is supported.

3.3.6 Grid and Cloud Database

This is a combination of grid and cloud computing in order to manage different databases with geographically distributed locations. Cloud computing helps with accessing remote hardware and storage resources.

3.3.7 XML Database

In this database system XML data is stored while XML is the main storage format.

3.3.8 Multidimensional Database

Here data is stored in a n-dimensional matrix. The precompilation and storage of relevant aggregates allows for interactive roll-ups and drill-downs.

3.3.9 Multivalue Database

A multivalue database possesses three dimension which are “field”, “value” which is a breakdown of “field” and “subvalue” which has in turn more detail on “value”. Advantages are the high flexibility of the database as well as the option of having calculated columns via small calculation programs.

3.3.10 Multimodel Database

Multimodel databases are a mixture of some of the database types that were already mentioned in order to maximize the advantages through combination. (Jatana et al., 2012)

3.3.11 OLAP

OLAP which is short for online analytical processing is mostly used by non-relational databases even though there are OLAP relational databases as well. OLAP provides for example the data on organizational performance in the right format and serves as the basis for high-level decisions. (Harrington, 2016)

3.4 Database Management Systems

A comprehensive software program that serves as an interface between databases and users is called a database management system (DBMS). On one hand it enables the users to retrieve, update and manage the information stored in the database, on the other hand administration including performance monitoring, backup and recovery is enabled. There are many popular database management systems such as My SQL, Microsoft Access and Oracle Database. (Oracle, 2020)

3.5 SQL

SQL which is short for Structured Query Language was developed at IBM in the 1970s and is the most used programming language to work with relational databases. It consists of a variety of statements to define, manipulate and query data. (Oracle, 2020)

The most common SQL commands can be divided into data definition language commands and data manipulation language commands where DDL commands deal with the creation or changes of the entire table and DML commands keep the data up to date. (Oracle, n.d.-a)

3.5.1 DDL Operations

The SQL statement “Create Table” creates an empty table that has a name and column names with a specified format for example “Text” for strings or “Integer” for numeric data. This command is mainly used at the beginning of building a database and not so much needed later when the management of the data inside the table is more relevant than the table itself.

“Alter Table” is used to change the structure of a table. That means a column is either added or removed, but it can also be used for changes of table constraints and column attributes. (Oracle, n.d.-a)

The command “Drop Table” removes the table entirely from the database. (Oracle, n.d.-a) If foreign key constraints are enabled in SQLite the data inside the table is deleted first to invoke any violations of foreign key restrictions. If that is the case the table is not dropped. (SQLite, 2024)

3.5.2 DML Operations

The “Insert” statement enables the addition of new rows to an existing table. The table can be empty or already filled with data before the “Insert”.

With the “Update” command the value of one or more existing fields in the table can be changed or added.

The “Delete” statement is used to remove one or more rows from a table.

The select statement does not change any data in the tables but only shows the content of one or more tables in a specified way. The rows that match the selection statement are called the “result set”. The selection can also be further specified by adding a “where” clause for only selecting rows that satisfy certain criteria. Via the “Join” operator data from two or more tables can be viewed combined. (Oracle, n.d.-a)

3.5.3 Transactions

In order to ensure data consistency as well as data concurrency, transactions, which are sets of one or more SQL statements are used. Transactions either leave the database in a consistent state with a “commit” or all changes are completely undone in a so-called “rollback”. Data concurrency means that more than one user should be able to access a database at the same time. There are different levels on which the data can be “locked” for other users while they are being manipulated. While a table lock locks the whole table if there are uncommitted transactions, a row lock locks all of the rows in a table and ensures that only one user can access the same row. (Oracle, n.d.-a)

4 Nutshell Examples

The explanation of the nutshell examples is split in five parts. First the concept of the database used in the programs is presented. Afterwards the establishment of the connection to the SQLite database using the JDBC driver is explained more thoroughly. Then the routines responsible for retrieving and formatting the resulting data output into readable tables, that are called in all the programs are described more precisely. Furthermore, the SQL commands as well as functions like prepared statements and rollback of a transaction are dealt with. Finally, a standalone program is introduced that explores the curl function in Rexx in combination with SQL statements.

4.1 Concept

The following chapters aim to describe the definition and manipulation of a simple database consisting of two tables with merely a few entries using only Rexx or BSF4ooRexx and SQL syntax. It is recommended to run the nutshell programs in the same order as the chapters to avoid error messages. At least the first one "zoo1createtable.rexx" is a prerequisite for the other nutshell examples.

After executing the nutshell programs but the last one, which would delete everything, the SQLite database "zoo.db" should look like this:

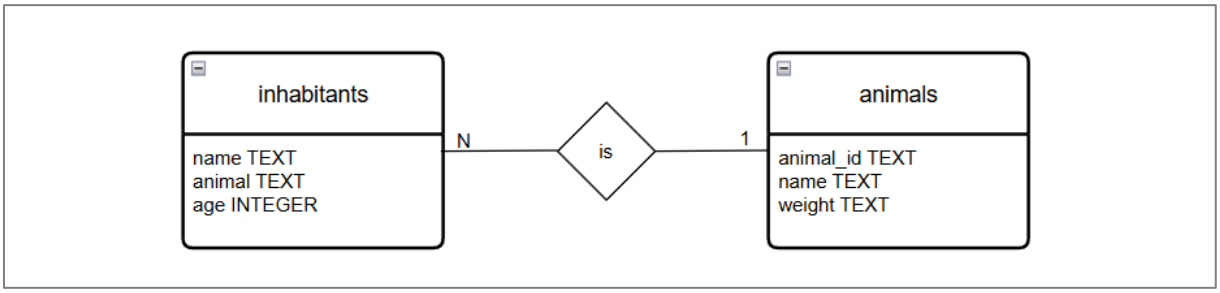

Figure 21: Entity Relationship Diagram of zoo.db

The table "inhabitants" includes the following data on animals that live in the zoo: "name" shows the first name of the animal, column "animal" holds the type of animal such as "lion" and age, which is added later via an "Alter Table" command. Then there is the second table "animals" which has information on animals in general. It possesses the columns "animal_id", which is the way the animal is named on the website of the zoo Vienna, "name" which is the animal type like "lion" and "weight". The relationship according to the entity-relationship model would be 1:n meaning one type of animal can be represented more than once in the zoo, but every zoo "inhabitant" can only be one type of animal. However, SQLite foreign key support was not activated so this relationship is not enforced by primary and foreign keys.

The visual representation of the tables will be structured as shown in figure 22 and 23:

ANIMALS

animal_id	name	weight
lowe	lion	250 kg
totenkopfaffe	monkey	1 kg
giraffe	giraffe	1.200 kg
burchell-zebra	zebra	300 kg
felsenpinguin	penguin	2,5 kg

Figure 22: Table "animals" format

INHABITANTS

name	animal	age
Nico	lion	The NIL object
Pingi	penguin	The NIL object

Figure 23: Table "inhabitants" format

4.2 Connection to Database

As a first step the connection to an SQLite database needs to be established. For this the folder for the database and the SQLite database file itself should be created if not already available. Afterwards the connection to the SQLite database must be configured resulting in the connection object “conn” that is used further for creating the statement objects in the nutshell programs. This happens in the routine “db_conn” that is part of the “zoodbtools.rexx” program that includes routines that apply to all the nutshell programs.

First the “getProperty” method from the System class in Java package java.lang is used to determine the home directory of the current user. Thanks to BSF4ooRexx it is possible to use Java classes like that. This should work for Windows as well as for Linux operating systems. Then a folder “Database” is created in the home directory via the Rexx utility function “sysMkDir” with the return code 0 for successful creation and 183 for the case, that the folder already exists. Subsequently the database file used for the nutshell programs “zoo.db” is added to the whole path which in combination with “jdbc:sqlite:” results in the url variable. Lastly the database connection object “conn “is retrieved from the “db_conn” routine.

```
4   /*Create directory for database */
5   homeDir=.java.lang.System~getProperty("user.home")
6   dbdir = "Database"
7   dbdirpath=homeDir ||"\"||dbdir
8   ret = sysMkDir(dbdirpath)
9   if ret = 183 then say "Database folder already there"
10  if ret = 0 then say "Database folder successfully created"
11
12  dbfile = "zoo.db"
13  dbfilepath = dbdirpath ||"\"||dbfile
14
15  /* Path to database */
16  url = "jdbc:sqlite:"||dbfilepath;
17
18  /* Pass url to db_conn routine */
19  conn = db_conn(url)
```

Code 1: Creation of database url

The routine “db_conn” enables the access to the SQLite database. First the JDBC driver class and the driver manager class are loaded. In the next step the JDBC driver is handed to the driver manager. Via the “getConnection” method the object “conn” for the database connection is created. This connection can be verified with a simple “if” statement that says “Connection successful” if the return code is not 0. If the connection object should be 0, the specific syntax error “No result object” is raised accompanied by the message “Connection to database failed”.

```
/* Establish connection to SQLite-database */
::ROUTINE db_conn public
    /* Uses url to database */
    USE ARG url
    /* Create new object, load JDBC-driver class */
    mydriver=.bsf~new('org.sqlite.JDBC')
    /* Load class DriverManager */
    man=bsf.loadClass("java.sql.DriverManager")
    /* Register driver as driver manager */
    man~registerDriver(mydriver)
    /* Returns connection object */
    conn=man~getConnection(url)

    if conn=0 then DO
        RAISE SYNTAX 91.900 additional "Connection to database failed"
    END
    say Connection successful

/* Return statement object */
RETURN conn
```

Code 2: Establish database connection

4.3 Routines

The routines which are described in the following chapters are called in all of the nutshell programs and therefore only once explained in more detail. They can be found in the "zoodbtools.rexx" program which is accessed by the nutshell programs. The purpose of the routine "showTable" is to select the table and display the current state in a readable way to verify the success of the SQL commands in the main program. For formatting the output, a method is needed to get the number of columns which is provided by the routine "getCols". Also, the names of the columns are coming from the routine "tableInfo". Both routines "getCols" and "tableInfo" showcase Rexx syntax, but could be replaced using metadata of the result set, which will be explained later.

4.3.1 showTable

Within the routine "showTable" data from the table which is passed as an argument from the main program is selected and made visually readable by adding borders and separators. The object "conn" is required as well to create the statement object just like it is the case in the main programs. Then the names of the columns which are saved in the variable "Attribute.List" are retrieved from the subroutine "tableInfo" passing table and the statement object as arguments.

Afterwards the "select" query is executed using the "executeQuery" method of the statement object and thus creating the result set "rSet". As the length of the outer and inner borders made of "=" and "-" respectively depend on the number of columns, this information is retrieved from the subroutine "getCols" using the result set object "rSet". The "getCols" routine will be explained in more detail in the next chapter. Then the header variable is constructed out of a "|" at the beginning and between the names of the columns which are retrieved from the "Attribute.List" variable. This is done as many times as there are columns in the table that was passed as an argument to the routine. In this program the width of each column is 15.

```
1   ::ROUTINE showTable public
2       /* Use table and statement for sql select */
3       USE ARG table, conn
4       /* Create statement object */
5       statement=conn~createStatement
6
7       /* Get column names from routine tableInfo */
8       Attribute.List = tableInfo(table, statement)
9       /* Execute SELECT statement */
10      rSet = statement~executeQuery("SELECT * FROM" table ";")
11      /* Create outer and inner border, including spaces and separators */
12      border.top = "="~copies(15*getCols(rSet)+(getCols(rSet)-1)*3+4)
13      border.in = "-"~copies(15*getCols(rSet)+(getCols(rSet)-1)*3+4)
14      /* Create header separator */
15      header = "|"
```

Code 3: Select statement and format output

```
16      /* Get number of columns from routine getCols */
17      DO i=1 TO getCols(rSet)
18          /* Get column names from routine tableInfo */
19          header = header left(Attribute.List[i], 15) "|"
20      END
```

Code 3 (continued): Select statement and format output

To print the table name as well as the header including outer, inner borders and column names the Rexx “say” command is used. In the second step the data output is formatted in a readable way. This is done for every row in the result set as long as there is data for the “next” method of the “rset” object. As it was the case for the header, the data variable also starts with a “|” and then includes the field from the “getString” method of the “rSet” object. The width of each column is again 15 and every field ends with a “|” to create a vertical separating line in the output. At the end of every “do” loop the entire row is printed via the Rexx “say” command and at the end the outer border is printed to mark the end of the table. At the end of the routine no value is returned.

```
21      /* Print table name */
22      SAY table
23      /* Header output */
24      say border.top
25      say header
26      say border.in
27
28      /* Create data output */
29      DO WHILE rset~next
30          /* Create data separator */
31          data = "|"
32          /* Use column number from routine getCols */
33          DO i=1 TO getCols(rSet)
34              /* Format data output */
35              data = data left(rset~getString(i), 15) "|"
36          END
37          /*Data output */
38          say data
39      END
40      say border.top
41
42  RETURN
```

Code 3 (continued): Select statement and format output

4.3.2 getCols

In the routine “getCols” the number of columns is determined with Rexx functionalities using the result set as an argument. The Rexx command “signal on” is evoked when the program runs into an error. This is the case when the “do” block reaches an error because the “getString” method of the result set with a certain number does not exist. That means one column must be subtracted to receive the number of columns of the table in the result set which is what happens when the trapname “done” is reached. Finally, the routine returns the number of columns.

```
44  /* Get number of columns of table */
45  ::ROUTINE getCols public
46      /* Use resultset from select query */
47      USE ARG res
48      /* Stop when "done" is reached */
49      SIGNAL ON ANY NAME done
50      cols=0
51      /* Get string from result set until the column doesn't exist --> Error */
52      DO FOREVER
53          cols+=1
54          res~getString(cols)
55      END
56      done:
57  /* Return last successful column = number of columns */
58  return cols-1
```

Code 4: Get number of columns

4.3.3 tableInfo

The routine “tableInfo” stores the column names in an array using the table and the statement object as arguments. First the result set is created by executing the “PRAGMA table_xinfo” query. This is specific to SQLite and stores every column name in a new row. Then a new array is created namely “Attribute.List” which is then filled with the data in the result set “rSet” via the “getString” method as long as the “next” method of the result set is valid. At the end of the routine the array “Attribute.List” with the column names is returned.

```
60  /* Get column names */
61  ::ROUTINE tableInfo public
62      USE ARG table, statement
63      /* Get info from pragma table */
64      rSet=statement~executeQuery("Pragma table_xinfo(" table ");")
65      /* Array for column names */
66      Attribute.List=.Array~new
67
68      DO WHILE rSet~next()
69          /* Add name to array */
70          Attribute.List~append(rSet~getString(2))
71      END
72
73  RETURN Attribute.List
```

Code 5: Get column names

4.4 SQL Statements

In this chapter only the “SQL” parts of the nutshell programs are described in more detail.

4.4.1 CREATE TABLE

For the purposes of this thesis two tables, “animals” and “inhabitants”, should be created in the until now empty SQLite database “zoo.db”. They serve as the base to showcase different SQL commands in simple nutshell programs. After creating these tables, the success should be verified by selecting them followed by formatting and printing the result. To achieve this ooRexx and Java via the BSF4ooRexx bridge as well as the JDBC driver for database connection should be used. Creating the tables and performing a “select” command could be done alternatively by using the DB Browser or the SQLite3 command line. As the tables are necessary for the other nutshell programs to function, “zoo1createtable.rexx” needs to be run before any of the following programs.

As shown in figure 24, 25 and 26 the tables hold no data.

```
sqlite> .open zoo.db
sqlite> Create table if not exists animals ('animal_id' TEXT, 'name' TEXT, 'weight' TEXT);
sqlite> Create table if not exists inhabitants ('name' TEXT, 'animal' TEXT);
sqlite> select * from animals;
sqlite> select * from inhabitants;
sqlite>
```

Figure 24: SQLite command line tool - Create Table

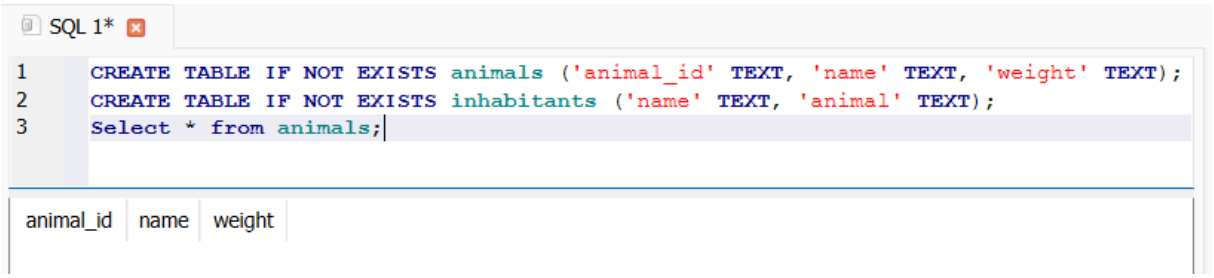

Figure 25: DB Browser - Create Table, Select “animals”

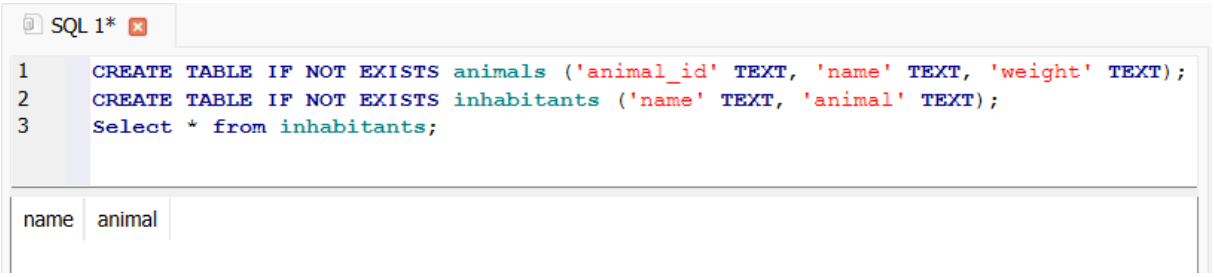

Figure 26: DB Browser - Create Table, Select "inhabitants"

First the statement object needs to be created with the “createStatement” method of the JDBC connection object “conn”. (Oracle, n.d.-b) To perform data defining language operations as in this case the “Create table” SQL command, the “executeUpdate” method of the before created “statement” object is used. The SQL command is provided as a string with the “-“ to go over more than one line according to Rexx syntax. Finally, the result set object “rSet” is checked if the update was successful. Because of the “if not exists” in the SQL command the result set is also 0, meaning successful, if the tables have already existed before the update. The code lines for the table creation are executed twice with different data as two tables are needed for the presentation of the “join” command.

Afterwards the routine “showTable” is called for both tables which includes a “select” command and the “design” of the output which has already been discussed in more detail. The connection object “conn” is passed, as it is necessary to create “statements” within the routine. Finally, the connection to the database is closed via the “close” method of the “conn” object. To have access to the routines “db_conn” and “showTable” the “zoodbtools.rexx” program is required as well as the “BSF.CLS” file to gain access to the Java functionalities. This is accomplished by using the “::REQUIRES” statement.

```
21  /* Creates statement object for SQL commands */
22  statement=conn~createStatement
23
24  /* Execute CREATE TABLE statement */
25  rSet = statement~executeUpdate("CREATE TABLE IF NOT EXISTS animals" -
26  "('animal_id' TEXT, 'name' TEXT, 'weight' TEXT);")
27  /* Output when successful (rSet = 0) */
28  if rSet = 0 then say "Table 'animals' successfully created"
29
30  rSet = statement~executeUpdate("CREATE TABLE IF NOT EXISTS inhabitants" -
31  "('name' TEXT, 'animal' TEXT);")
32  /* Output when successful (rSet = 0) */
33  if rSet = 0 then say "Table 'inhabitants' successfully created"
34
35  /* Call routine to show table graphically */
36  CALL showTable animals, conn
37  CALL showTable inhabitants, conn
38
39  /* Close database connection */
40  conn~close
41
42  ::REQUIRES zoodbtools.rexx
43  ::REQUIRES BSF.CLS
```

Code 6: Create Table statement

As shown in the following figure the output of the program “zoo1createtable.rexx” the folder “Database” did not exist in the home directory before, so it was newly created. Then the connection to the SQLite database was successfully established and two tables were created that show no data yet.

```
Database folder successfully created
CONNECTION SUCCESSFUL
Table 'animals' successfully created
Table 'inhabitants' successfully created
ANIMALS
=========================================================
| animal_id       | name            | weight          |
---------------------------------------------------------
=========================================================
INHABITANTS
====================================
| name            | animal          |
------------------------------------
====================================

Process finished with exit code 0
```

Figure 27: Result of Create Table statement

4.4.2 INSERT

After successfully creating two empty tables in the SQLite database “zoo.db” they need to be filled with data. In this case the table “animals” should hold different types of animals while the table “inhabitants” should include specific animals that live in the zoo. The lines in this chapter should show how to insert data into the until now empty tables with SQL statements by using ooRexx and Java with the help of BSF4ooRexx and the JDBC driver. Before and after the “Insert” the tables should be displayed graphically to prove that it has worked. Again, this could be done by using the DB Browser or via the SQLite3 command line tool.

```
sqlite> Insert into animals (animal_id, name, weight) values ('lowe', 'lion', '250 kg');
sqlite> Insert into animals (animal_id, name, weight) values ('totenkopfaffe', 'monkey', '1 kg');
sqlite> Insert into animals (animal_id, name, weight) values ('giraffe', 'giraffe', '1.200 kg');
sqlite> Insert into animals (animal_id, name, weight) values ('burchell-zebra', 'zebra', '300 kg');
sqlite> Insert into animals (animal_id, name, weight) values ('felsenpinguin', 'penguin', '2,5 kg');
sqlite> Insert into inhabitants (name, animal) values ('Nico', 'lion');
sqlite> Insert into inhabitants (name, animal) values ('Pingi', 'penguin');
sqlite> Select * from animals;
```

animal_id	name	weight
lowe	lion	250 kg
totenkopfaffe	monkey	1 kg
giraffe	giraffe	1.200 kg
burchell-zebra	zebra	300 kg
felsenpinguin	penguin	2,5 kg

```
sqlite> Select * from inhabitants;
```

name	animal
Nico	lion
Pingi	penguin

Figure 28: SQLite command line tool - Insert Into

```
1  Insert into animals (animal_id, name, weight) values ('lowe', 'lion', '250 kg');
2  Insert into animals (animal_id, name, weight) values ('totenkopfaffe', 'monkey', '1 kg');
3  Insert into animals (animal_id, name, weight) values ('giraffe', 'giraffe', '1.200 kg');
4  Insert into animals (animal_id, name, weight) values ('burchell-zebra', 'zebra', '300 kg');
5  Insert into animals (animal_id, name, weight) values ('felsenpinguin', 'penguin', '2,5 kg');
6  Select * from animals;
```

	animal_id	name	weight
1	lowe	lion	250 kg
2	totenkopfaffe	monkey	1 kg
3	giraffe	giraffe	1.200 kg
4	burchell-zebra	zebra	300 kg
5	felsenpinguin	penguin	2,5 kg

Figure 29: DB Browser - Insert Into, Select animals

```
1  Insert into inhabitants (name, animal) values ('Nico', 'lion');
2  Insert into inhabitants (name, animal) values ('Pingi', 'penguin');
3  Select * from inhabitants;
4
```

	name	animal
1	Nico	lion
2	Pingi	penguin

Figure 30: DB Browser - Insert Into, Select inhabitants

As an attempt to prevent SQL injection attacks prepared statements should be used instead of the regular statement object. The program “zoo1createtable.rexx” is a prerequisite for this code to work.

To emphasize the difference that the update makes, the tables are displayed via the routine “showTable” to show their state before the update. The “insert” command is a data manipulation language operation. Therefore, prepared statements with variable bind parameters were used. (Oracle, n.d.-b) Instead of the classic “statement” object the “preparedStatement” object was created via the “prepareStatement” method of the “conn” object. The values in the SQL command are replaced with “?” and passed later with the “setString” method. Then the prepared statement gets executed via the “executeUpdate” method, just like the normal statement, but with empty brackets.

```
22    CALL showTable animals, conn
23    CALL showTable inhabitants, conn
24
25    /* Prepare object preparedStatement with variables */
26    preparedStatement = conn~prepareStatement("INSERT INTO animals" -
27    "(animal_id, name, weight) VALUES (?, ?, ?)")
28
29    /* Pass Values for the variables in the prepared statement */
30    preparedStatement~setString(1, "lowe")
31    preparedStatement~setString(2, "lion")
32    preparedStatement~setString(3, "250 kg")
33    /* Execute prepared Statement */
34    rSet = preparedStatement~executeUpdate()
35    /* Pass Values for the variables in the prepared statement */
36    preparedStatement~setString(1, "totenkopfaffe")
37    preparedStatement~setString(2, "monkey")
38    preparedStatement~setString(3, "1 kg")
39    /* Execute prepared Statement */
40    rSet = preparedStatement~executeUpdate()
41    /* Pass Values for the variables in the prepared statement */
42    preparedStatement~setString(1, "giraffe")
43    preparedStatement~setString(2, "giraffe")
44    preparedStatement~setString(3, "1.200 kg")
```

Code 7: Insert Into with prepared statement

The passing of variables with the “setString” method and the “executeUpdate” method afterwards is then repeated with different values. Then a new prepared statement is defined for the second table and with only two variables which are filled again with the “setString” method and “executeUpdate”. Afterwards the tables are displayed again via the “showTable” routine and the database connection is closed.

```
60      /* Prepare object preparedStatement with variables */
61      preparedStatement = conn~prepareStatement("INSERT INTO inhabitants" -
62      "(name, animal) VALUES (?, ?)")
63
64      /* Pass Values for the variables in the prepared statement */
65      preparedStatement~setString(1, "Nico")
66      preparedStatement~setString(2, "lion")
67      /* Execute prepared Statement */
68      rSet = preparedStatement~executeUpdate()
69
70      /* Pass Values for the variables in the prepared statement */
71      preparedStatement~setString(1, "Pingi")
72      preparedStatement~setString(2, "penguin")
73      /* Execute prepared Statement */
74      rSet = preparedStatement~executeUpdate()
75
76      /* Call routine to show table graphically */
77      SAY "Tables after"
78      CALL showTable animals, conn
79      CALL showTable inhabitants, conn
80
81      /* Close database connection */
82      conn~close
83
84      ::REQUIRES zoodbtools.rexx
85      ::REQUIRES BSF.CLS
```

Code 7 (continued): Insert Into with prepared statement

Figure 31 shows the empty tables before the “insert” statement.

```
Tables before
ANIMALS
=========================================================
| animal_id       | name            | weight          |
---------------------------------------------------------
=========================================================
INHABITANTS
=====================================
| name            | animal          |
-------------------------------------
=====================================
```

Figure 31: Result before Insert statement

To show the progress, the content of both tables is selected again via the “showTable” routine.

```
Tables after
ANIMALS
=========================================================
| animal_id       | name            | weight          |
---------------------------------------------------------
| lowe            | lion            | 250 kg          |
| totenkopfaffe   | monkey          | 1 kg            |
| giraffe         | giraffe         | 1.200 kg        |
| burchell-zebra  | zebra           | 300 kg          |
| felsenpinguin   | penguin         | 2,5 kg          |
=========================================================
INHABITANTS
=====================================
| name            | animal          |
-------------------------------------
| Nico            | lion            |
| Pingi           | penguin         |
=====================================
```

Figure 32: Result after Insert statement

4.4.3 ALTER TABLE

The goal of this program is to demonstrate the DDL operation “Alter Table” which in this case adds a column “age” to the existing table “inhabitants” in the SQLite database “zoo.db”. Here ooRexx and Java via BSF4ooRexx should be used together with the JDBC driver to enable database connection. Before and after the “Alter Table” command the table should be selected and the result printed in a readable way. As before DB Browser and the SQLite3 command line tool are other ways to achieve this. To be able to change the structure of the table, it must already be there, so it is crucial that the table “inhabitants” has been already created by the program “zoo1createtable.rexx”.

```
sqlite> Alter table inhabitants add column age integer;
sqlite> Select * from inhabitants;
```

name	animal	age
Nico	lion	
Pingi	penguin	

Figure 33: SQLite command line tool - Alter Table

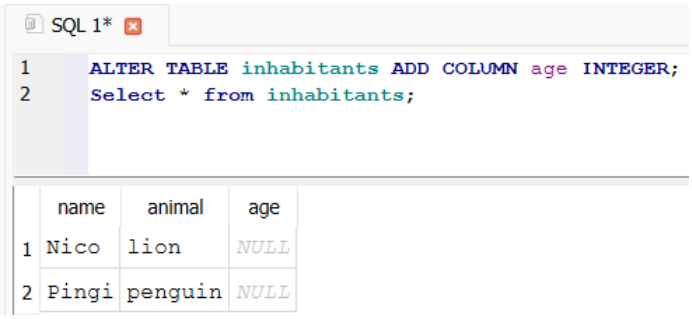

Figure 34: DB Browser - Alter Table

As this command does not have several variables a classic “statement” object is created via the “createStatement” method of the JDBC connection object “conn”. Again, the table is shown before any changes are implemented for comparison purposes. As before, the SQL command is provided as a string. Afterwards the contents of the table are selected again via the “showTable” routine to verify the result and finally the database connection is closed via the “close” method of the “conn” object.

```
20  /* Creates statement object for SQL commands */
21  statement=conn~createStatement
22
23  /* Show table before alter statement */
24  SAY "Table before"
25  CALL showTable inhabitants, conn
26
27  /* Execute ALTER TABLE statement */
28  rSet = statement~executeUpdate("ALTER TABLE inhabitants ADD COLUMN age INTEGER;")
```

Code 8: Alter Table statement

```
29
30      /* Show table after alter statement */
31      SAY "Table after"
32      CALL showTable inhabitants, conn
33
34      /* Close database connection */
35      conn~close
36
37      ::REQUIRES zoodbtools.rexx
38      ::REQUIRES BSF.CLS
```

Code 8 (continued): Alter Table statement

Figure 35 shows that the table “inhabitants” has a new column “age” after the update.

```
Table before
INHABITANTS
====================================
| name            | animal          |
------------------------------------
| Nico            | lion            |
| Pingi           | penguin         |
====================================
Table after
INHABITANTS
=========================================================
| name            | animal          | age             |
---------------------------------------------------------
| Nico            | lion            | The NIL object  |
| Pingi           | penguin         | The NIL object  |
=========================================================
```

Figure 35: Result after Alter Table

4.4.4 UPDATE TABLE

Now data inside the table “inhabitants” in the SQLite database “zoo.db” is supposed to be changed. A lion with the name Emma should be inserted into the table “inhabitants” followed by a change of the value in the column “animal” from “lion” to “zebra”. This program should insert data into a table and then change it with the SQL “update” command. Prepared Statements should be used as there are variables to avoid injection attacks. Before and after the update the table “inhabitants” should be shown to present the success of the update. DB Browser or the SQLite3 command line tool are valid options to process the insert and update commands. However, they should be implemented with ooRexx and Java via BSF4ooRexx and the JDBC driver for connecting to the database as already explained. As it is the case for the other programs, the nutshell example “zoo1createtable.rexx” needs to be run before running this program in order to create the required table “inhabitants”.

```
sqlite> Insert into inhabitants (name, animal) values ('Emma', 'lion');
sqlite> Select * from inhabitants;
```

name	animal
Nico	lion
Pingi	penguin
Emma	lion

```
sqlite> Update inhabitants set animal = 'zebra' where name = 'Emma';
sqlite> Select * from inhabitants;
```

name	animal
Nico	lion
Pingi	penguin
Emma	zebra

Figure 36: SQLite command line tool – Insert Into and Update Table

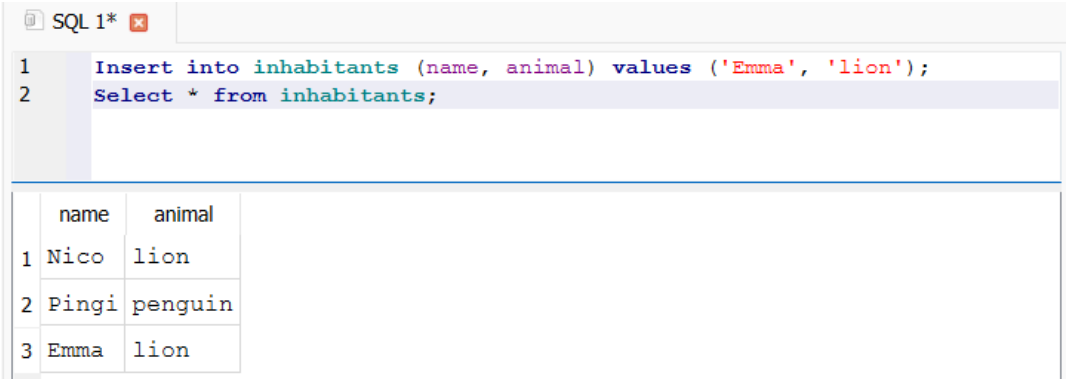

Figure 37: DB Browser – Table "inhabitants" before Update

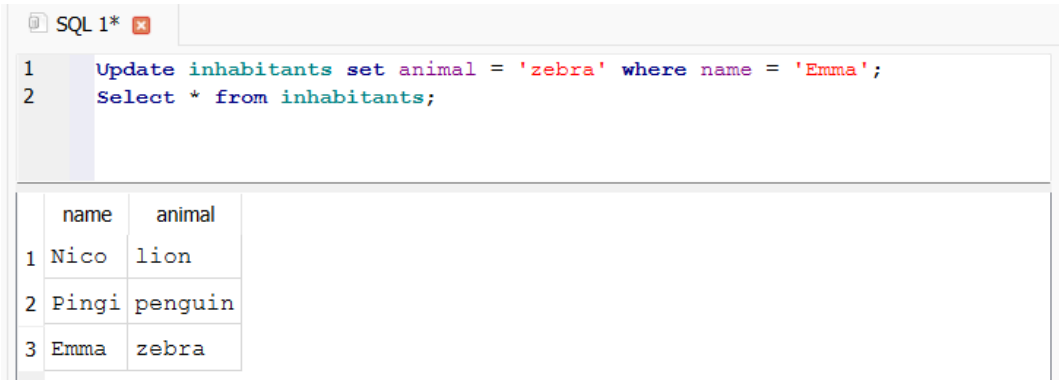

Figure 38: DB Browser - Table "inhabitants" after Update

First one set of data is inserted into the table “inhabitants” in the form of a prepared statement. For this the “preparedStatement” object is created with the “prepareStatement” method of the “conn” object. Then the values for the variables are given via the “setString” method and finally the “preparedStatement” object is executed via the “executeUpdate” method with empty brackets. Afterwards the contents of the table is selected in the “showTable” routine to show the data before the update.

```
20    /* Prepare object preparedStatement with variables */
21    preparedStatement = conn~prepareStatement("INSERT INTO inhabitants" -
22    "(name, animal) VALUES (?, ?)")
23
24    /* Pass Values for the variables in the prepared statement */
25    preparedStatement~setString(1, "Emma")
26    preparedStatement~setString(2, "lion")
27    /* Execute prepared Statement */
28    rSet = preparedStatement~executeUpdate()
29
30    /* Call routine to show table graphically */
31    SAY "Table before update"
32    CALL showTable inhabitants, conn
```

Code 9: Insert Into with prepared statement

The prepared statement for the SQL “update” follows the same logic as with the SQL “insert” command. A prepared statement object is created via the “prepareStatement” method, and the SQL “update” command is provided as a string except for the values for animal and the “where”-clause. These binding variables are given in the next step via the “setString” method of the “preparedStatement” object. Subsequently the prepared statement is executed. The routine “showTable” is called again, to verify the changes that have been made to the data. At the end of the program, the database connection is closed with the “close” method of the “conn” object.

```
34    /* Prepare object preparedStatement with variables */
35    preparedStatement = conn~prepareStatement("UPDATE inhabitants SET animal = ? WHERE name = ?;")
36
37    /* Pass Values for the variables in the prepared statement */
38    preparedStatement~setString(1, "zebra")
39    preparedStatement~setString(2, "Emma")
40    /* Execute prepared Statement */
41    rSet = preparedStatement~executeUpdate()
42
43    /* Call routine to show table graphically */
44    SAY "Table after update"
45    CALL showTable inhabitants, conn
46
47    /* Close database connection */
48    conn~close
49
50    ::REQUIRES zoodbtools.rexx
51    ::REQUIRES BSF.CLS
```

Code 10: Update with prepared statement

Figure 39 shows the table “inhabitants” after the insert, which is before the update and after the update. The field in the column “animal” was changed from “lion” to “zebra” where the animals name equals “Emma”.

```
Table before update
INHABITANTS
=====================================
| name            | animal          |
-------------------------------------
| Emma            | lion            |
=====================================
Table after update
INHABITANTS
=====================================
| name            | animal          |
-------------------------------------
| Emma            | zebra           |
=====================================
```

Figure 39: Result after Insert Into and Update

4.4.5 SELECT with JOIN

In contrast to the programs above, there should be no changes to the data or the tables in the SQLite database “zoo.db”. A table structure should be selected that contains the following information: name, animal and weight. The columns “name” and “animal” come from the table “inhabitants” while the column “weight should be retrieved from the table “animals”. The tables should be joined via the columns “animal” and “name” respectively. The following implementation is supposed to use ooRexx and Java via BSF4ooRexx and the JDBC driver to show the result of an SQL “join” command. The resulting table should be formatted and printed. As before, doing this with the DB Browser or the SQLite3 command line tool would be an option as well. To function, the programs “zoo1createtable.rexx” and “zoo2insert.rexx” must be run in this order to create the tables and fill them with the necessary data.

```
sqlite> SELECT inhabitants.name, inhabitants.animal, animals.weight FROM inhabitants LEFT
 JOIN animals ON animals.name = inhabitants.animal WHERE inhabitants.animal = 'lion';
```

name	animal	weight
Nico	lion	250 kg

Figure 40: SQLite command line tool - Select with Join

```
1   SELECT inhabitants.name, inhabitants.animal, animals.weight FROM inhabitants
2   LEFT JOIN animals ON animals.name = inhabitants.animal WHERE inhabitants.animal = 'lion';
```

	name	animal	weight
1	Nico	lion	250 kg

Figure 41: DB Browser - Select with Join

Initially, the statement object “statement” is created via the “createStatement” method of the database connection object “conn”. Now for the SQL “select” command the “executeQuery” method of the “statement” object is used. As a “join” operation is used data is selected from two tables to form a new table. In this case it is a left join which means that from the first table all data and from the second table only corresponding data is selected. The whole SQL command is entered as a string over three lines with the “-“ symbol following Rexx syntax.

Afterwards the routine “showJoin” is called with the result set as variable to format and show the result.

```
22  /* Creates statement object for SQL commands */
23  statement=conn~createStatement
24
25  /* Execute SELECT query with LEFT JOIN */
26  rSet = statement~executeQuery("SELECT inhabitants.name, inhabitants.animal," -
27  "animals.weight FROM inhabitants LEFT JOIN animals ON animals.name =" -
28  "inhabitants.animal WHERE inhabitants.animal = 'lion';")
29
30  /* Call routine to show table graphically */
31  SAY "Joined Table"
32  CALL showJoin rSet
```

Code 11: Select statement with Join

Then the outer border which should consist of “=” forming a double line and the inner border with a single line out of “-“ are defined as “border.top” and “border.in”. Also the titles of the columns should be separated by “|” which is declared as header. Then the headers “Field1” to “Field3” are created for the three columns of the newly joined table. Finally, the outer border, the header and the inner boarder are printed via “say” as stated in the Rexx syntax.

```
37  ::ROUTINE showJoin
38      /* Use result set for data output */
39      USE ARG rSet
40      /* Check if select successful */
41      IF rSet <> 0 THEN
42          DO
43              SAY "JOIN: inhabitants LEFT JOIN animals ON" -
44                  "animals.name = inhabitants.animal WHERE inhabitants.animal = lion"
45          END
46      ELSE SAY "Error"
47      /* Create outer and inner border, including spaces and separators */
48      border.top = "="~copies(15*3+(3-1)*3+4)
49      border.in = "-"~copies(15*3+(3-1)*3+4)
50      /* Create header separator */
51      header = "|"
```

Code 12: Format output for Select statement with Join

```
52      /* Iterate over 3 columns */
53      DO i=1 TO 3
54          header = header left("Field"||i, 15) "|"
55      END
56      /*Header output */
57      say border.top
58      say header
59      say border.in
```

Code 12 (continued): Format output for Select statement with Join

The “next” method is used to iterate through the result set until it is empty. (Oracle, n. d.) Then the data separator is defined as “|”. With the “getString” method of the “rset” object the data from each of the three columns is retrieved. At the end of the “do” block the variable “data” contains one row of the table. To close the table with the before defined outer border the variable “border.top” is printed again.

```
61      /* Create data output */
62      DO WHILE rset~next
63          /* Create data separator */
64          data = "|"
65          DO i=1 TO 3
66              /* Format data output */
67              data = data left(rset~getString(i), 15) "|"
68          END
69          /* Data output */
70          say data
71      END
72      say border.top
73
74  RETURN
```

Code 12 (continued): Format output for Select statement with Join

Finally the database connection is closed at the end of the program.

```
34  /* Close database connection */
35  conn~close
```

Code 12 (continued): Format output for Select statement with Join

These are the tables used for the “join” operation.

```
ANIMALS
=======================================================
| animal_id       | name            | weight          |
-------------------------------------------------------
| lowe            | lion            | 250 kg          |
| totenkopfaffe   | monkey          | 1 kg            |
| giraffe         | giraffe         | 1.200 kg        |
| burchell-zebra  | zebra           | 300 kg          |
| felsenpinguin   | penguin         | 2,5 kg          |
=======================================================
INHABITANTS
====================================
| name            | animal          |
------------------------------------
| Nico            | lion            |
| Pingi           | penguin         |
====================================
```

Figure 42: Tables used for Join operation

This is the result of the left join with a “where” clause.

```
JOIN: inhabitants LEFT JOIN animals ON animals.name = inhabitants.animal WHERE inhabitants.animal = lion
=====================================================
| Field1          | Field2          | Field3          |
-----------------------------------------------------
| Nico            | lion            | 250 kg          |
=====================================================
```

Figure 43: Result of Select statement with Join

4.4.6 SELECT

While in the other nutshell programs the routines “getCols” and “tableInfo” were used to gain information on the number of columns and the column names, this example showcases another method to get this type of information. Therefore, only the columns “name” and “weight” from the SQLlite “zoo.db” database table “animals” should be selected, formatted and printed. The select command would work with DB Browser or SQLite3 command line tools with no problem, but the point of this nutshell program is to show the use of meta data from the result set. So, this example should be implemented with ooRexx and Java via BSF4ooRexx and the JDBC driver for database connectivity. As in the previous example the programs zoo1createtable.rexx and zoo2insert.rexx need to be run first, before running this program to have data, that can be selected.

```
sqlite> SELECT name, weight FROM animals;
```

name	weight
lion	250 kg
monkey	1 kg
giraffe	1.200 kg
zebra	300 kg
penguin	2,5 kg

Figure 44: SQLite command line tool - Select

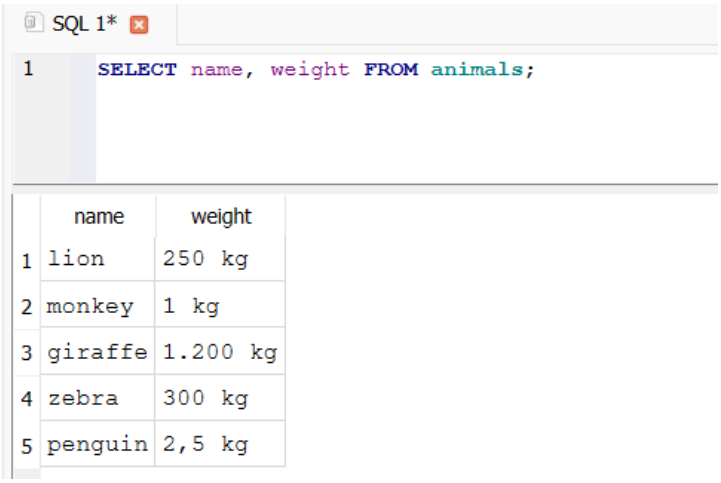

Figure 45: DB Browser - Select

First, the statement object is created via the “createStatement” method of the connection object. Then the “Select” command that selects the columns “name” and “weight” from the table “animals” is executed with the “executeQuery” method. The table name is printed as well as information on the output and the “showSelect” routine is called.

```
20  /* Creates statement object for SQL commands */
21  statement=conn~createStatement
22
23  rSet = statement~executeQuery("SELECT name, weight FROM animals;")
24
25  /* Call routine to show table graphically */
26  SAY "ANIMALS: Result of Select"
27  CALL showSelect rSet
```

Code 13: Select statement

The routine “showSelect” uses the result set “rSet” as an argument to derive the meta data via the “getMetaData()” method from it. The newly created object “rSetMeta” includes information on the number of columns via the method “getColumnCount()”. The creation of the outer and inner border is the same as in the “showTable” routine except the variable columncount is used instead of the “getCols” routine. The header starts with an “|” followed by the column name via “getColumnName” and a “|”. That is repeated as long as there are columns. Finally the outer border, the header and the inner border are printed via the Rexx command “say”.

```
::ROUTINE showSelect public
    /* Use table and statement for sql select */
    USE ARG rSet

    rSetMeta = rSet~getMetaData()
    columncount = rSetMeta~getColumnCount()

    /* Create outer and inner border, including spaces and separators */
    border.top = "="~copies(15*columncount+(columncount-1)*3+4)
    border.in = "-"~copies(15*columncount+(columncount-1)*3+4)
    /* Create header separator */
    header = "|"
    /* Get number of columns from routine getCols */
    DO i=1 TO columncount
        /* Get column names from routine tableInfo */
        header = header left(rSetMeta~getColumnName(i), 15) "|"
    END
    /* Header output */
    say border.top
    say header
    say border.in
```

Code 14: Get metadata and format output

The “do” block is again the same as in the “showTable” routine with the exception that the variable “columncount” is used instead of the routine “getCols”. As the header the data output starts with an “|” as well, followed by the value from the result set via the “getString” method and ends with a “|”. This gets repeated for every column. Afterwards the created “data” string is printed as well as the outer border string. At the end of the program the database connection is closed via the “close” method of the “conn” object.

```
    /* Create data output */
    DO WHILE rset~next
        /* Create data separator */
        data = "|"
        /* Use column number from routine getCols */
        DO i=1 TO columncount
            /* Format data output */
            data = data left(rset~getString(i), 15) "|"
        END
        /*Data output */
        say data
    END
    say border.top

RETURN
```

Code 14 (continued): Get metadata and format output

```
/* Close database connection */
conn~close
```

Code 15: Close database connection

Figure 46 shows the result of the “Select” command. Only the values of the columns “name” and “weight” are displayed.

```
ANIMALS: Result of Select
====================================
| name            | weight          |
------------------------------------
| lion            | 250 kg          |
| monkey          | 1 kg            |
| giraffe         | 1.200 kg        |
| zebra           | 300 kg          |
| penguin         | 2,5 kg          |
====================================
```

Figure 46: Result of Select statement

4.4.7 DELETE FROM with ROLLBACK

After having created and manipulated the tables “animals” and “inhabitants” in the SQLite “zoo.db” database, the data of both tables are now supposed to be deleted, but the deletion should be reversed with eventually no changes occurring. This should be implemented with the “rollback” command whose effect should be visualized by selecting, formatting and printing the tables before and after the rollback. This is also doable in the DB Browser or the SQLite3 command line tool. However, in this thesis ooRexx and Java with the help of BSF4ooRexx and the JDBC Driver for connecting to the database should be used. The programs “zoo1createtable.rexx” and “zoo2insert.rexx” are prerequisites for this code to function properly.

```
sqlite> Select * from animals;
```

animal_id	name	weight
lowe	lion	250 kg
totenkopfaffe	monkey	1 kg
giraffe	giraffe	1.200 kg
burchell-zebra	zebra	300 kg
felsenpinguin	penguin	2,5 kg

```
sqlite> Select * from inhabitants;
```

name	animal
Nico	lion
Pingi	penguin

```
sqlite> begin transaction;
sqlite> delete from animals;
sqlite> delete from inhabitants;
sqlite> Select * from animals;
sqlite> Select * from inhabitants;
sqlite> rollback;
```

Figure 47: SQLite command line tool – Select tables, Delete and Rollback

```
sqlite> select * from animals;
```

animal_id	name	weight
lowe	lion	250 kg
totenkopfaffe	monkey	1 kg
giraffe	giraffe	1.200 kg
burchell-zebra	zebra	300 kg
felsenpinguin	penguin	2,5 kg

```
sqlite> select * from inhabitants;
```

name	animal
Nico	lion
Pingi	penguin

Figure 48: SQLite command line tool - Select tables after Rollback

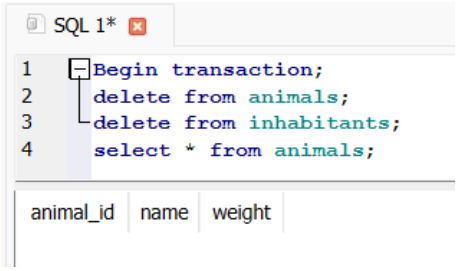

Figure 49: DB Browser – Delete, Select "animals"

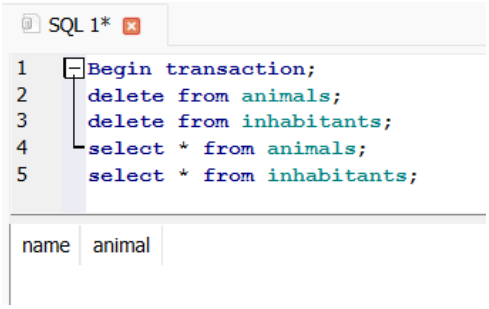

Figure 50: DB Browser – Delete, Select "inhabitants"

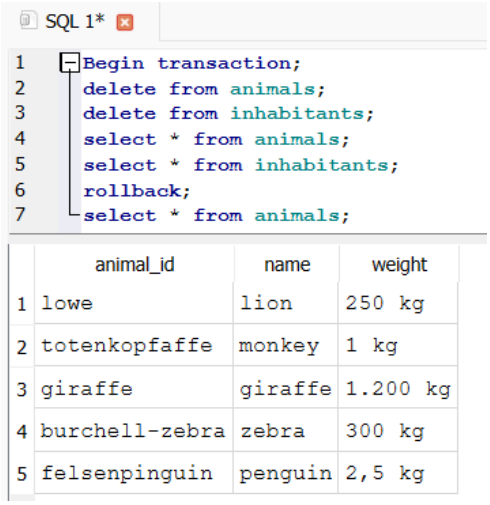

Figure 51: DB Browser – Select "animals" again after Rollback

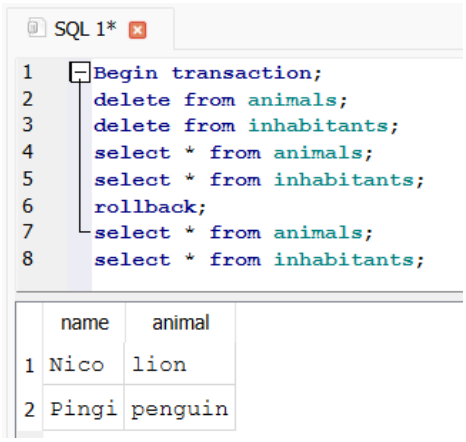

Figure 52: DB Browser – Select "inhabitants" again after Rollback

Firstly, the statement object "statement" is created as before using the "createStatement" method of the database connection object "conn". Then a transaction is started with the "begin transaction" command in SQL followed by the "delete" SQL statements for both tables. Subsequently the "showTable" routine is called to show the content of the tables, which should be empty as all data has been deleted. Then the SQL operation "rollback" is executed which reverses the effects of the "delete" SQL commands. Alternatively, the SQL statement "Commit" would cause all changes to be saved. To prove that the rollback worked, the "showTable" routine is called a second time

and should show entries in both tables. At the end of the program the database connection is closed again with the “close” method of the “conn” object.

```
/* Creates statement object for SQL commands */
statement=conn~createStatement

/* Start of transaction */
rSet = statement~executeUpdate("BEGIN TRANSACTION;")

/* Execute DELETE statement */
rSet = statement~executeUpdate("DELETE FROM animals;")

rSet = statement~executeUpdate("DELETE FROM inhabitants;")

SAY "Tables before"
CALL showTable animals, conn
CALL showTable inhabitants, conn

/* End of transaction */
rSet = statement~executeUpdate("ROLLBACK;")
say "ROLLBACK"

/* Call routine to show table graphically */
SAY "Tables after rollback"
CALL showTable animals, conn
CALL showTable inhabitants, conn

/* Close database connection */
conn~close

::REQUIRES zoodbtools.rexx
::REQUIRES BSF.CLS
```

Code 16: Delete From statement with Rollback

As shown in figure 53 the tables are empty after the execution of the “delete” command, but contain data after the rollback as illustrated in figure 54.

```
Tables before
ANIMALS
=========================================================
| animal_id       | name            | weight          |
---------------------------------------------------------
=========================================================
INHABITANTS
=====================================
| name            | animal          |
-------------------------------------
=====================================
```

Figure 53: Result of Delete From statement before rollback

```
ROLLBACK
Tables after rollback
ANIMALS
=======================================================
| animal_id       | name            | weight          |
-------------------------------------------------------
| lowe            | lion            | 250 kg          |
| totenkopfaffe   | monkey          | 1 kg            |
| giraffe         | giraffe         | 1.200 kg        |
| burchell-zebra  | zebra           | 300 kg          |
| felsenpinguin   | penguin         | 2,5 kg          |
=======================================================
INHABITANTS
====================================
| name            | animal          |
------------------------------------
| Nico            | lion            |
| Pingi           | penguin         |
====================================
```

Figure 54: Result of Delete From statement after rollback

4.4.8 DROP TABLE

Finally, the tables “inhabitants” as well as “animals” of the SQLite database “zoo.db” should be deleted entirely. This ooRexx program with the help of Java via BSF4ooRexx and the JDBC driver for the database connection should delete the entire tables “animals” and “inhabitants”. The tables should be shown in a formatted way before being dropped. If that was successful information should be given. Dropping a table via the SQL “Drop” command in the SQLite3 command line or just pressing “Delete table” in the DB Browser are alternative options. In order to work, the program “zoo1createtable.rexx” needs to be run before running this nutshell example.

```
sqlite> drop table animals;
sqlite> drop table inhabitants;
sqlite> select * from animals;
Parse error: no such table: animals
sqlite> select * from inhabitants;
Parse error: no such table: inhabitants
sqlite>
```

Figure 55: SQLite command line tool - Drop Table

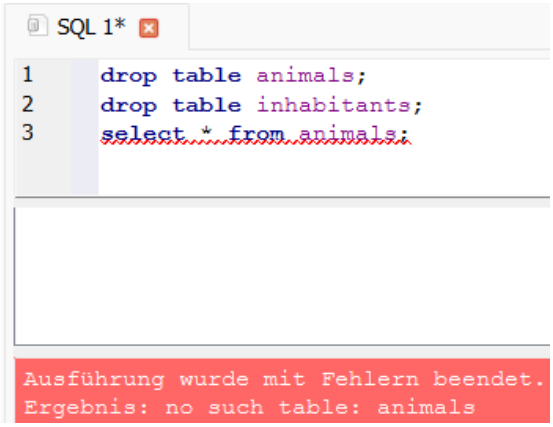

Figure 56: DB Browser – Drop Table, Select "animals"

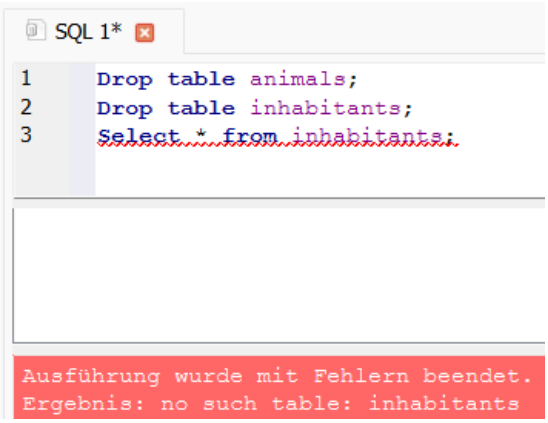

Figure 57: DB Browser - Drop Table, Select "inhabitants"

It starts with the creation of the statement object "statement" using the "createStatement" method of the database connection object "conn". Then the data of the tables are selected via the "showTable" routine to show the tables before making any changes. Then the result set object "rSet" is created via the "executeUpdate" method of the "statement" object and the SQL "drop table" command is given as a string in brackets. For this SQL command the result set should be 0 if the execution was successful. This is checked with an "if" clause. Finally, the database connection is closed again via the "close" method.

```
21    /* Creates statement object for SQL commands */
22    statement=conn~createStatement
23
24    /* Call routine to show table graphically */
25    SAY "Tables before"
26    CALL showTable animals, conn
27    CALL showTable inhabitants, conn
28
29    /* Execute DROP statement */
30    rSet = statement~executeUpdate("DROP TABLE animals ;")
31    if rSet = 0 then say "Table 'animals' successfully dropped"
32
33    rSet = statement~executeUpdate("DROP TABLE inhabitants;")
34    if rSet = 0 then say "Table 'inhabitants' successfully dropped"
35
36    /* Close database connection */
37    conn~close
38
39
40    ::REQUIRES zoodbtools.rexx
41    ::REQUIRES BSF.CLS
```

Code 17: Drop Table statement

As the figure below shows the tables “animals” and “inhabitants” exist and contain data before they are dropped.

```
Tables before
ANIMALS
=======================================================
| animal_id       | name            | weight          |
-------------------------------------------------------
| lowe            | lion            | 250 kg          |
| totenkopfaffe   | monkey          | 1 kg            |
| giraffe         | giraffe         | 1.200 kg        |
| burchell-zebra  | zebra           | 300 kg          |
| felsenpinguin   | penguin         | 2,5 kg          |
=======================================================
INHABITANTS
====================================
| name            | animal          |
------------------------------------
| Nico            | lion            |
| Pingi           | penguin         |
====================================
Table 'animals' successfully dropped
Table 'inhabitants' successfully dropped
```

Figure 58: Result of Drop Table statement

4.5 CURL

This program works on its own and explores the “curl” command in combination with SQL commands as well as the transaction/rollback function. First the table “animals” should be created in the SQLite database “zoo.db”. Then a list of animals should be created, that is inserted into the table as well as the animals weight that should be retrieved from the website of the zoo Vienna. This should be implemented by using the “curl” command to get the HTML-code of the website of the zoo Vienna. Curl is a shell command in Windows and Linux that is given to the shell via the “ADDRESS SYSTEM” statement. Then the output should be narrowed further down to the information on the animals weight by using the “parse” command. After the insert as well as after the rollback the table should be selected, formatted in a readable way and printed to view the progress. All of this should be implemented using ooRexx and Java via BSF4ooRexx with JDBC driver for connecting to the SQLite database. Just the SQL commands could be executed with the SQLite command line tool or the DB Browser as well.

```
sqlite> CREATE TABLE IF NOT EXISTS animals ('animal_id' TEXT, 'name' TEXT, 'weight' TEXT);
sqlite> begin transaction;
sqlite> Insert into animals (animal_id, weight) values ('lowe', '250 kg');
sqlite> Insert into animals (animal_id, weight) values ('totenkopfaffe', '1 kg');
sqlite> Insert into animals (animal_id, weight) values ('giraffe', '1.200 kg');
sqlite> Insert into animals (animal_id, weight) values ('burchell-zebra', '300 kg');
sqlite> Insert into animals (animal_id, weight) values ('felsenpinguin', '2,5 kg');
sqlite> Select * from animals;
```

animal_id	name	weight
lowe		250 kg
totenkopfaffe		1 kg
giraffe		1.200 kg
burchell-zebra		300 kg
felsenpinguin		2,5 kg

```
sqlite> rollback;
sqlite> Select * from animals;
sqlite> drop table animals;
sqlite> Select * from animals;
Parse error: no such table: animals
```

Figure 59: SQLite command line tool - Create Table, Insert Into, Rollback, Drop Table

```
1   CREATE TABLE IF NOT EXISTS animals ('animal_id' TEXT, 'name' TEXT, 'weight' TEXT);
```

Figure 60: DB Browser - Create Table “animals”

```
1  BEGIN TRANSACTION;
2    Insert into animals (animal_id, weight) values ('lowe', '250 kg');
3    Insert into animals (animal_id, weight) values ('totenkopfaffe', '1 kg');
4    Insert into animals (animal_id, weight) values ('giraffe', '1.200 kg');
5    Insert into animals (animal_id, weight) values ('burchell-zebra', '300 kg');
6    Insert into animals (animal_id, weight) values ('felsenpinguin', '2,5 kg');
7    Select * from animals;
```

	animal_id	name	weight
1	lowe	NULL	250 kg
2	totenkopfaffe	NULL	1 kg
3	giraffe	NULL	1.200 kg
4	burchell-zebra	NULL	300 kg
5	felsenpinguin	NULL	2,5 kg

Figure 61: Insert Into, Select table "animals"

Figure 62: Rollback, Select "animals"

Figure 63: Drop Table, Select "animals"

It starts with the creation of the statement object via the “createStatement” method of the connection object. Afterwards the table animals is created if it does not exist already with the column names “animal_id”, “name” and “weight”. They are all type “text”.

Then a new array called “animalslist” is created and filled with values, namely animals that are presented on the zoo Vienna website, with the “makeArray” method. Subsequently a transaction is started via the SQL command “Begin transaction”. The “insert into” SQL statement is placed inside a loop over all the entries in the array “animalslist” to fill the table with the value in the “animal” variable. The weight of the respective animal is determined with the help of the “animalWeight” routine that uses the argument “animal”. Then the loop ends.

```
20  /* Creates statement object for SQL commands */
21  statement=conn~createStatement
22
23  /* Execute CREATE TABLE statement */
24  rSet = statement~executeUpdate("CREATE TABLE IF NOT EXISTS animals" -
25  "('animal_id' TEXT, 'name' TEXT, 'weight' TEXT);")
26  /* Define new array */
27  animalslist = .Array~new
28  /* Fill array with values */
29  animalslist = ('lowe', 'totenkopfaffe', 'giraffe', 'burchell-zebra', 'felsenpinguin')~makeArray
30  /* "Start of Transaction" */
31  rSet = statement~executeUpdate("BEGIN TRANSACTION;")
32  /* Iterate over array */
33  loop animal over animalslist
34      /* Execute INSERT statement, get weight and unit from animalWeight routine */
35      rSet = statement~executeUpdate("INSERT INTO animals" -
36      "(animal_id, weight) VALUES ('"animal||"','"animalWeight(animal)"');")
37  end
```

Code 18: Create Table and Insert Into with loop

In the routine “animalWeight” the weight of the respective animal in the argument “animal” is retrieved from the “zoovienna” website in real time. First the variable “command” is constructed out of “curl”, the url of the zoo and the “animal” variable in lowercase. Then an array to store the output is created as well as one to store error messages. Then the “ADDRESS” keyword statement is executed with before defined variables. To extract only the small information needed which is the weight of the animal in this case the Rexx “parse” command is used with a pattern before and after the information of interest. As the weight is not always given in the same format, it is checked with an “if” clause that changes the variables if the first value is numeric. To verify the result, it is printed via the “say” command. The double “|” serves to eliminate any spaces.

Finally, the routine returns the variables weight and unit of the respective animal.

```
56  /* Get animal weight */
57  ::ROUTINE animalWeight
58      USE ARG animal
59      /* Curl zoo website */
60      command="curl https://www.zoovienna.at/tiere/saeugetiere/"||animal~lower||"/"
61      /* Create array to store output */
62      outArr=.array~new
63      /* Create array for error messages */
64      errArr=.array~new
65      /* Give command to operating system */
66      ADDRESS SYSTEM command WITH OUTPUT USING (outArr) ERROR USING (errArr)
67      /* Parse only weight */
68      parse VAR outArr '"animal-fact-list__headline">Gewicht</span><span>' bis weight unit "</span>"
69      /* Check format and adapt */
70      if bis~datatype(numeric) then do
71          unit = weight
72          weight = bis
73      end
74      say animal||": " weight unit
75  /* Return weight and unit of animal */
76  RETURN weight unit
77
78  ::REQUIRES zoodbtools.rexx
79  ::REQUIRES BSF.CLS
```

Code 19: Parse animal weight from website

After inserting data into the table the routine “showTable” is called to show the state of the table after defining and manipulating the data. Then the SQL command “Rollback” is executed and the “showTable” routine is called again to check if the table is empty again. Afterwards the table is dropped via the “drop table” SQL statement that gets executed with the “executeUpdate” method of the “statement” object. If the result set is 0 then the table was successfully dropped. At the end of the program the connection is closed via the “close” method of the connection object and the “exit” instruction finally ends it.

```
39    /* Call routine to check if inserts were successful */
40    CALL showTable animals, statement
41    /* Rollback, changes are not saved */
42    rSet = statement~executeUpdate("ROLLBACK;")
43    SAY rollback
44    /* Check if inserts are gone */
45    CALL showTable animals, statement
46    /* Delete table, rSet should be 0 */
47    rSet = statement~executeUpdate("DROP TABLE animals;")
48    IF rSet = 0 THEN SAY table dropped
49
50    /* Close database connection */
51    conn~close
52    /* End program */
53    exit
54    /* End main program */
```

Code 20: Rollback and Drop Table

As visible in the figure below the information was curled and then parsed correctly from the website of the zoo Vienna. The table was filled with data until the rollback set the contents back to zero. Finally, the table was dropped with a result set of 0 and therefore "Table dropped" was printed.

```
lowe:  250 kg
totenkopfaffe:  1 kg
giraffe:  1.200 kg
burchell-zebra:  300 kg
felsenpinguin:  2,5 kg
ANIMALS
=========================================================
| animal_id       | name            | weight          |
---------------------------------------------------------
| lowe            | The NIL object  | 250 kg          |
| totenkopfaffe   | The NIL object  | 1 kg            |
| giraffe         | The NIL object  | 1.200 kg        |
| burchell-zebra  | The NIL object  | 300 kg          |
| felsenpinguin   | The NIL object  | 2,5 kg          |
=========================================================
ROLLBACK
ANIMALS
=========================================================
| animal_id       | name            | weight          |
---------------------------------------------------------
=========================================================
TABLE DROPPED
```

Figure 64: Result of Insert Into, rollback and Drop Table

5 Round-up and Outlook

First an overview of the programming languages that were used for several nutshell programs was given. Afterwards the information on how to install the necessary software components was provided. Subsequently the difference between relational and non-relational databases was shown along with a more detailed explanation of the two types of databases. Finally, the combination of the programming languages ooRexx, Java via the bridge BSF4ooRexx and SQL was demonstrated in some nutshell examples.

While the aim of this thesis was only to show the basic functionalities of SQLite, further work could be about designing a more complex database taking integrity constraints into account. For this, SQLite foreign key support would need to be activated. With a bigger database looking into indexing could be interesting to speed up database queries. Another field to explore would be triggers, a tool to automate the response of the database to certain events such as “Insert” or “Update”.

Appendix

A1. CREATE TABLE

Listing 21 shows the complete code for the program zoo1createtable.rexx.

```
/* This program creates the tables "animals" and "inhabitants"
in the SQLite Database "zoo.db" */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database folder already there"
if ret = 0 then say "Database folder successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;

/* Pass url to db_conn routine */
conn = db_conn(url)

/* Creates statement object for SQL commands */
statement=conn~createStatement

/* Execute CREATE TABLE statement */
rSet = statement~executeUpdate("CREATE TABLE IF NOT EXISTS animals" -
"('animal_id' TEXT, 'name' TEXT, 'weight' TEXT);")
/* Output when successful (rSet = 0) */
if rSet = 0 then say "Table 'animals' successfully created"

```

Code 21: zoo1createtable.rexx

```
rSet = statement~executeUpdate("CREATE TABLE IF NOT EXISTS inhabitants" -
"('name' TEXT, 'animal' TEXT);")
/* Output when successful (rSet = 0) */
if rSet = 0 then say "Table 'inhabitants' successfully created"

/* Call routine to show table graphically */
CALL showTable animals, conn
CALL showTable inhabitants, conn

/* Close database connection */
conn~close

::REQUIRES zoodbtools.rexx
::REQUIRES BSF.CLS
```

Code 21 (continued): zoo1createtable.rexx

A2. INSERT

Listing 22 shows the complete code for the program zoo2insert.rexx.

```
/* This program inserts data into the table "animals".
It is required to execute zoocreatetable before running this program. */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database file already there"
if ret = 0 then say "Database file successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;
/* Pass url to db_conn routine */
conn = db_conn(url)

/* Show table before insert statement */
SAY "Tables before"
CALL showTable animals, conn
CALL showTable inhabitants, conn

/* Prepare object preparedStatement with variables */
preparedStatement = conn~prepareStatement("INSERT INTO animals" -
"(animal_id, name, weight) VALUES (?, ?, ?)")

```

Code 22: zoo2insert.rexx

```
/* Pass Values for the variables in the prepared statement */
preparedStatement~setString(1, "lowe")
preparedStatement~setString(2, "lion")
preparedStatement~setString(3, "250 kg")
/* Execute prepared Statement */
rSet = preparedStatement~executeUpdate()
/* Pass Values for the variables in the prepared statement */
preparedStatement~setString(1, "totenkopfaffe")
preparedStatement~setString(2, "monkey")
preparedStatement~setString(3, "1 kg")
/* Execute prepared Statement */
rSet = preparedStatement~executeUpdate()
/* Pass Values for the variables in the prepared statement */
preparedStatement~setString(1, "giraffe")
preparedStatement~setString(2, "giraffe")
preparedStatement~setString(3, "1.200 kg")
```

Code 22 (continued): zoo2insert.rexx

```
45    /* Execute prepared Statement */
46    rSet = preparedStatement~executeUpdate()
47    /* Pass Values for the variables in the prepared statement */
48    preparedStatement~setString(1, "burchell-zebra")
49    preparedStatement~setString(2, "zebra")
50    preparedStatement~setString(3, "300 kg")
51    /* Execute prepared Statement */
52    rSet = preparedStatement~executeUpdate()
53    /* Pass Values for the variables in the prepared statement */
54    preparedStatement~setString(1, "felsenpinguin")
55    preparedStatement~setString(2, "penguin")
56    preparedStatement~setString(3, "2,5 kg")
57    /* Execute prepared Statement */
58    rSet = preparedStatement~executeUpdate()
59
60    /* Prepare object preparedStatement with variables */
61    preparedStatement = conn~prepareStatement("INSERT INTO inhabitants" -
62    "(name, animal) VALUES (?, ?)")
63
64    /* Pass Values for the variables in the prepared statement */
65    preparedStatement~setString(1, "Nico")
66    preparedStatement~setString(2, "lion")
67    /* Execute prepared Statement */
68    rSet = preparedStatement~executeUpdate()
69
70    /* Pass Values for the variables in the prepared statement */
71    preparedStatement~setString(1, "Pingi")
72    preparedStatement~setString(2, "penguin")
73    /* Execute prepared Statement */
74    rSet = preparedStatement~executeUpdate()
```

Code 22 (continued): zoo2insert.rexx

```
75
76    /* Call routine to show table graphically */
77    SAY "Tables after"
78    CALL showTable animals, conn
79    CALL showTable inhabitants, conn
80
81    /* Close database connection */
82    conn~close
83
84    ::REQUIRES zoodbtools.rexx
85    ::REQUIRES BSF.CLS
```

Code 22 (continued): zoo2insert.rexx

A3. ALTER TABLE

Listing 23 shows the complete code for the program zoo3altertable.rexx.

```
/* This program adds a new column "age" to the table "inhabitants".
It is required to execute zoocreatetable before running this program. */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database file already there"
if ret = 0 then say "Database file successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;
/* Pass url to db_conn routine */
conn = db_conn(url)

/* Creates statement object for SQL commands */
statement=conn~createStatement

/* Show table before alter statement */
SAY "Table before"
CALL showTable inhabitants, conn

/* Execute ALTER TABLE statement */
rSet = statement~executeUpdate("ALTER TABLE inhabitants ADD COLUMN age INTEGER;")

```

Code 23: zoo3altertable.rexx

```
/* Show table after alter statement */
SAY "Table after"
CALL showTable inhabitants, conn

/* Close database connection */
conn~close

::REQUIRES zoodbtools.rexx
::REQUIRES BSF.CLS
```

Code 23 (continued): zoo3altertable.rexx

A4. UPDATE TABLE

Listing 24 shows the complete code for the program zoo4update.rexx.

```
/* This program inserts data into table "inhabitants" and changes a field.
It is required to execute zoocreatetable before running this program. */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database file already there"
if ret = 0 then say "Database file successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;
/* Pass url to db_conn routine */
conn = db_conn(url)

/* Prepare object preparedStatement with variables */
preparedStatement = conn~prepareStatement("INSERT INTO inhabitants" -
"(name, animal) VALUES (?, ?)")

/* Pass Values for the variables in the prepared statement */
preparedStatement~setString(1, "Emma")
preparedStatement~setString(2, "lion")
/* Execute prepared Statement */
rSet = preparedStatement~executeUpdate()

```

Code 24: zoo4update.rexx

```
/* Call routine to show table graphically */
SAY "Table before update"
CALL showTable inhabitants, conn

/* Prepare object preparedStatement with variables */
preparedStatement = conn~prepareStatement("UPDATE inhabitants SET animal = ? WHERE name = ?;")

/* Pass Values for the variables in the prepared statement */
preparedStatement~setString(1, "zebra")
preparedStatement~setString(2, "Emma")
/* Execute prepared Statement */
rSet = preparedStatement~executeUpdate()

/* Call routine to show table graphically */
SAY "Table after update"
CALL showTable inhabitants, conn

/* Close database connection */
conn~close

::REQUIRES zoodbtools.rexx
::REQUIRES BSF.CLS
```

Code 24 (continued): zoo4update.rexx

A5. SELECT with JOIN

Listing 25 shows the complete code for the program zoo5selectjoin.rexx.

```
/* This program joins data from table "animals" and "inhabitants".
It is required to execute the following programs before running this program:
1) zoo1createtable
2) zoo2insert */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database file already there"
if ret = 0 then say "Database file successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;
/* Pass url to db_conn routine */
conn = db_conn(url)

/* Creates statement object for SQL commands */
statement=conn~createStatement

/* Execute SELECT query with LEFT JOIN */
rSet = statement~executeQuery("SELECT inhabitants.name, inhabitants.animal," -
"animals.weight FROM inhabitants LEFT JOIN animals ON animals.name =" -
"inhabitants.animal WHERE inhabitants.animal = 'lion';")

```

Code 25: zoo5selectjoin.rexx

```
/* Call routine to show table graphically */
SAY "Joined Table"
CALL showJoin rSet

/* Close database connection */
conn~close
```

Code 25 (continued): zoo5selectjoin.rexx

```
::ROUTINE showJoin
    /* Use result set for data output */
    USE ARG rSet
    /* Check if select successful */
    IF rSet <> 0 THEN
        DO
            SAY "JOIN: inhabitants LEFT JOIN animals ON" -
                "animals.name = inhabitants.animal WHERE inhabitants.animal = lion"
        END
    ELSE SAY "Error"
```

Code 25 (continued): zoo5selectjoin.rexx

```
47      /* Create outer and inner border, including spaces and separators */
48      border.top = "="~copies(15*3+(3-1)*3+4)
49      border.in = "-"~copies(15*3+(3-1)*3+4)
50      /* Create header separator */
51      header = "|"
52      /* Iterate over 3 columns */
53      DO i=1 TO 3
54          header = header left("Field"||i, 15) "|"
55      END
56      /*Header output */
57      say border.top
58      say header
59      say border.in
60
61      /* Create data output */
62      DO WHILE rset~next
63          /* Create data separator */
64          data = "|"
65          DO i=1 TO 3
66              /* Format data output */
67              data = data left(rset~getString(i), 15) "|"
68          END
69          /* Data output */
70          say data
71      END
72      say border.top
73
74  RETURN
```

Code 25 (continued): zoo5selectjoin.rexx

A6. SELECT

Listing 26 shows the complete code for the program zoo6selectcolumns.rexx.

```
/* This program selects data from the table "animals".
It is required to execute the following programs before running this program:
1) zoo1createtable
2) zoo2insert */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database folder already there"
if ret = 0 then say "Database folder successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;

/* Pass url to db_conn routine */
conn = db_conn(url)

/* Creates statement object for SQL commands */
statement=conn~createStatement

rSet = statement~executeQuery("SELECT name, weight FROM animals;")

/* Call routine to show table graphically */
SAY "ANIMALS: Result of Select"
CALL showSelect rSet
```

Code 26: zoo6selectcolumns.rexx

```

/* Close database connection */
conn~close
```

Code 26 (continued): zoo6selectcolumns.rexx

```
::ROUTINE showSelect public
    /* Use table and statement for sql select */
    USE ARG rSet

    rSetMeta = rSet~getMetaData()
    columncount = rSetMeta~getColumnCount()

    /* Create outer and inner border, including spaces and separators */
    border.top = "="~copies(15*columncount+(columncount-1)*3+4)
    border.in = "-"~copies(15*columncount+(columncount-1)*3+4)
    /* Create header separator */
    header = "|"
    /* Get number of columns from routine getCols */
    DO i=1 TO columncount
        /* Get column names from routine tableInfo */
        header = header left(rSetMeta~getColumnName(i), 15) "|"
    END
    /* Header output */
    say border.top
    say header
    say border.in

```

Code 26 (continued): zoo6selectcolumns.rexx

```
    /* Create data output */
    DO WHILE rset~next
        /* Create data separator */
        data = "|"
        /* Use column number from routine getCols */
        DO i=1 TO columncount
            /* Format data output */
            data = data left(rset~getString(i), 15) "|"
        END
        /*Data output */
        say data
    END
    say border.top

RETURN
```

Code 26 (continued): zoo6selectcolumns.rexx

A7. DELETE FROM with ROLLBACK

Listing 27 shows the complete code for the program zoo7deletefrom.rexx.

```
/* This program deletes all data from "animals" and inhabitants,
but changes are followed by a "rollback"
It is required to execute the following programs before running this program:
1) zoo1createtable
2) zoo2insert */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database file already there"
if ret = 0 then say "Database file successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;

/* Pass url to db_conn routine */
conn = db_conn(url)

/* Creates statement object for SQL commands */
statement=conn~createStatement

/* Start of transaction */
rSet = statement~executeUpdate("BEGIN TRANSACTION;")

```

Code 27: zoo7deletefrom.rexx

```
/* Execute DELETE statement */
rSet = statement~executeUpdate("DELETE FROM animals;")

rSet = statement~executeUpdate("DELETE FROM inhabitants;")

SAY "Tables before"
CALL showTable animals, conn
CALL showTable inhabitants, conn

/* End of transaction */
rSet = statement~executeUpdate("ROLLBACK;")
say "ROLLBACK"

/* Call routine to show table graphically */
SAY "Tables after rollback"
CALL showTable animals, conn
CALL showTable inhabitants, conn

/* Close database connection */
conn~close

::REQUIRES zoodbtools.rexx
::REQUIRES BSF.CLS
```

Code 27 (continued): zoo7deletefrom.rexx

A8. DROP TABLE

Listing 28 shows the complete code for the program zoo8droptable.rexx.

```
/* This program deletes the tables "animals" and "inhabitants"
It is required to execute zoocreatetable before running this program. */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database file already there"
if ret = 0 then say "Database file successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;

/* Pass url to db_conn routine */
conn = db_conn(url)

/* Creates statement object for SQL commands */
statement=conn~createStatement

/* Call routine to show table graphically */
SAY "Tables before"
CALL showTable animals, conn
CALL showTable inhabitants, conn

```

Code 28: zoo8droptable.rexx

```
/* Execute DROP statement */
rSet = statement~executeUpdate("DROP TABLE animals ;")
if rSet = 0 then say "Table 'animals' successfully dropped"

rSet = statement~executeUpdate("DROP TABLE inhabitants;")
if rSet = 0 then say "Table 'inhabitants' successfully dropped"

/* Close database connection */
conn~close


::REQUIRES zoodbtools.rexx
::REQUIRES BSF.CLS
```

Code 28 (continued): zoo8droptable

A9. CURL

Listing 29 shows the complete code for the program curlzoo.rexx.

```
/* Main program */

/*Create directory for database */
homeDir=.java.lang.System~getProperty("user.home")
dbdir = "Database"
dbdirpath=homeDir ||"\"||dbdir
ret = sysMkDir(dbdirpath)
if ret = 183 then say "Database folder already there"
if ret = 0 then say "Database folder successfully created"

dbfile = "zoo.db"
dbfilepath = dbdirpath ||"\"||dbfile

/* Path to database */
url = "jdbc:sqlite:"||dbfilepath;

/* Pass url to db_conn routine */
conn = db_conn(url)

/* Creates statement object for SQL commands */
statement=conn~createStatement

/* Execute CREATE TABLE statement */
rSet = statement~executeUpdate("CREATE TABLE IF NOT EXISTS animals" -
"('animal_id' TEXT, 'name' TEXT, 'weight' TEXT);")
/* Define new array */
animalslist = .Array~new
/* Fill array with values */
animalslist = ('lowe', 'totenkopfaffe', 'giraffe', 'burchell-zebra', 'felsenpinguin')~makeArray
```

Code 29: curlzoo.rexx

```
/* "Start of Transaction" */
rSet = statement~executeUpdate("BEGIN TRANSACTION;")
/* Iterate over array */
loop animal over animalslist
    /* Execute INSERT statement, get weight and unit from animalWeight routine */
    rSet = statement~executeUpdate("INSERT INTO animals" -
    "(animal_id, weight) VALUES ('"animal||"','"animalWeight(animal)"');")
end

/* Call routine to check if inserts were successful */
CALL showTable animals, statement
/* Rollback, changes are not saved */
rSet = statement~executeUpdate("ROLLBACK;")
SAY rollback
```

Code 29 (continued): curlzoo.rexx

```
/* Check if inserts are gone */
CALL showTable animals, statement
/* Delete table, rSet should be 0 */
rSet = statement~executeUpdate("DROP TABLE animals;")
IF rSet = 0 THEN SAY table dropped

/* Close database connection */
conn~close
/* End program */
exit
/* End main program */
```

Code 29 (continued): curlzoo.rexx

```
/* Get animal weight */
::ROUTINE animalWeight
    USE ARG animal
    /* Curl zoo website */
    command="curl https://www.zoovienna.at/tiere/saeugetiere/"||animal~lower||"/"
    /* Create array to store output */
    outArr=.array~new
    /* Create array for error messages */
    errArr=.array~new
    /* Give command to operating system */
    ADDRESS SYSTEM command WITH OUTPUT USING (outArr) ERROR USING (errArr)
    /* Parse only weight */
    parse VAR outArr '"animal-fact-list__headline">Gewicht</span><span>' bis weight unit "</span>"
    /* Check format and adapt */
    if bis~datatype(numeric) then do
        unit = weight
        weight = bis
    end
    say animal||": " weight unit
/* Return weight and unit of animal */
RETURN weight unit

::REQUIRES zoodbtools.rexx
::REQUIRES BSF.CLS
```

Code 29 (continued): curlzoo.rexx

A10. Routine – db_conn

Listing 30 shows the complete code for the routine db_conn.

```
/* Establish connection to SQLite-database */
::ROUTINE db_conn public
    /* Uses url to database */
    USE ARG url
    /* Create new object, load JDBC-driver class */
    mydriver=.bsf~new('org.sqlite.JDBC')
    /* Load class DriverManager */
    man=bsf.loadClass("java.sql.DriverManager")
    /* Register driver as driver manager */
    man~registerDriver(mydriver)
    /* Returns connection object */
    conn=man~getConnection(url)

    if conn=0 then DO
        RAISE SYNTAX 91.900 additional "Connection to database failed"
    END
    say Connection successful

/* Return statement object */
RETURN conn
```

Code 30: db_conn routine

A11. Routine – showTable

Listing 31 shows the complete code for the routine showTable.

```
::ROUTINE showTable public
    /* Use table and statement for sql select */
    USE ARG table, conn
    /* Create statement object */
    statement=conn~createStatement

    /* Get column names from routine tableInfo */
    Attribute.List = tableInfo(table, statement)
    /* Execute SELECT statement */
    rSet = statement~executeQuery("SELECT * FROM" table ";")
    /* Create outer and inner border, including spaces and separators */
    border.top = "="~copies(15*getCols(rSet)+(getCols(rSet)-1)*3+4)
    border.in = "-"~copies(15*getCols(rSet)+(getCols(rSet)-1)*3+4)
    /* Create header separator */
    header = "|"
    /* Get number of columns from routine getCols */
    DO i=1 TO getCols(rSet)
        /* Get column names from routine tableInfo */
        header = header left(Attribute.List[i], 15) "|"
    END
    /* Print table name */
    SAY table
    /* Header output */
    say border.top
    say header
    say border.in
```

Code 31: showTable routine

```

    /* Create data output */
    DO WHILE rset~next
        /* Create data separator */
        data = "|"
        /* Use column number from routine getCols */
        DO i=1 TO getCols(rSet)
            /* Format data output */
            data = data left(rset~getString(i), 15) "|"
        END
        /*Data output */
        say data
    END
    say border.top

RETURN
```

Code 31 (continued): showTable routine

A12. Routine – getCols

Listing 32 shows the complete code for the routine getCols.

```
/* Get number of columns of table */
::ROUTINE getCols public
    /* Use resultset from select query */
    USE ARG res
    /* Stop when "done" is reached */
    SIGNAL ON ANY NAME done
    cols=0
    /* Get string from result set until the column doesn't exist --> Error */
    DO FOREVER
        cols+=1
        res~getString(cols)
    END
    done:
/* Return last successful column = number of columns */
return cols-1
```

Code 32: getCols routine

A13. Routine – tableInfo

Listing 33 shows the complete code for the routine tableInfo.

```
/* Get column names */
::ROUTINE tableInfo public
    USE ARG table, statement
    /* Get info from pragma table */
    rSet=statement~executeQuery("Pragma table_xinfo(" table ");")
    /* Array for column names */
    Attribute.List=.Array~new

    DO WHILE rSet~next()
        /* Add name to array */
        Attribute.List~append(rSet~getString(2))
    END

RETURN Attribute.List
```

Code 33: tableInfo routine

References

Cowlishaw, M. F. (1984). The design of the REXX language. *IBM Systems Journal*, Vol 23, No 4, 1984. Retrieved May 7, 2025, from https://dl.acm.org/doi/pdf/10.1145/24686.24687

EDM2. (2019). IBM Object REXX. Retrieved May 7, 2025, from https://www.edm2.com/index.php/IBM_Object_REXX

Eessaar, E. (2016). Database Normalization Theory and The Theory of Normalized Systems: Finding a Common Ground. *Baltic J. Modern Computing,* Vol. 4, No. 1, 5-33. Retrieved June 8, 2025, from https://www.researchgate.net/profile/Erki-Eessaar/publication/297731569_The_Database_Normalization_Theory_and_the_Theory_of_Normalized_Systems_Finding_a_Common_Ground/links/56e18d9508ae40dc0abf50a1/The-Database-Normalization-Theory-and-the-Theory-of-Normalized-Systems-Finding-a-Common-Ground.pdf

Flatscher, R. G. (2013). Introduction to Rexx and ooRexx: From Rexx to Open Object Rexx (ooRexx) (1. ed.). Facultas Verlags- und Buchhandels AG.

FreeCodeCamp. (2022). Database Normalization – Normal Forms 1nf 2nf 3nf Table Examples. Retrieved June 8, 2025, from https://www.freecodecamp.org/news/database-normalization-1nf-2nf-3nf-table-examples/

Harrington, J. L. (2016). Relational Database Design and Implementation. *Elsevier Inc*. Retrieved May 14, 2025, from https://books.google.at/books?hl=de&lr=&id=yQgfCgAAQBAJ&oi=fnd&pg=PP1&dq=database+design&ots=qQFxlYVA2v&sig=iGewcrNn5SkuZqpywZ76_eDai80&redir_esc=y#v=onepage&q=database%20design&f=false

Jatana, N., Puri, S., Ahuja, M., Kathuria, I., Gosain, D. (2012). A Survey and Comparison of Relational and Non-Relational Database. *International Journal of Engineering Research & Technology (IJERT)*, 2278-0181, Vol. 1 Issue 6. Retrieved May 14, 2025, from https://d1wqtxts1xzle7.cloudfront.net/76957411/a-survey-and-comparison-of-relational-and-non-relational-database-libre.pdf?1640094061=&response-content-disposition=inline%3B+filename%3DA_Survey_and_Comparison_of_Relational_an.pdf&Expires=1747240350&Signature=YuYTP2lM6JKHOUIHrqnUhRdkY~byEfyrMOYQWhPxLtdHa12WmFmlfmAVTlaghV6J0iG6~aeQPTL5RQr3nRLEpVzjwI0zMuPXrZaJgWaJ5lpGAaydeVkXR9fsnGbEtNJChW9KhXOQ3RfiQqH0nk-

g6L393ukAzdwhKa967M-QNZJi1SlX3t7nOiIYm5WWIMwyxgY~kXjapRveNlftNEIGT8f19eQ6Bu4025vHbZvd4SSjsqwqdTrzfmqjgl7DjZy32sRfLmzBimMZeGQUtHMNvPCRGO8KRoS-CwFe7zLsBrndIBXLp11SGBtvR37lUTXSUT-AJ2Vw9WUqVbuxnhpSDg__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

Oracle. (n.d.-a). A Relational Database Overview. Retrieved June 1, 2025, from https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html

Oracle. (n.d.-b). JDBC-getting started. Retrieved May 22, 2025, from https://docs.oracle.com/en/database/oracle/oracle-database/23/jjdbc/JDBC-getting-started.html#GUID-44000A82-7B5E-49A9-BB87-8A3A794E8F3A

Oracle. (2020). What Is a Database?. Retrieved May 14, 2025, from https://www.oracle.com/uk/database/what-is-database/

Oracle. (2021). What Is a Relational Database? (RDBMS)?. Retrieved May 14, 2025, from https://www.oracle.com/uk/database/what-is-a-relational-database/

Oracle. (2024). JDBC Introduction. Retrieved May 7, 2025, from https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html

SQLite. (2024). SQLite Foreign Key Support. Retrieved June 2, 2025, from https://www.sqlite.org/foreignkeys.html

SQLite. (2025). About SQLite. Retrieved May 7, 2025, from https://www.sqlite.org/about.html

SQLitebrowser. (2025). DB Browser for SQLite. Retrieved May 7, 2025, from https://sqlitebrowser.org/

Wikipedia. (2024). Bean Scripting Framework. Retrieved May 7, 2025, from https://en.wikipedia.org/wiki/Bean_Scripting_Framework

Wikipedia. (2025-a). Object Rexx. Retrieved May 7, 2025, from https://en.wikipedia.org/wiki/Object_REXX

Wikipedia. (2025-b). SQLite. Retrieved May 7, 2025, from https://en.wikipedia.org/wiki/SQLite

Download Links

Java	https://bell-sw.com/pages/downloads/#jdk-24 (retrieved 27.04.2025)
ooRexx	https://sourceforge.net/projects/oorexx/files/oorexx/5.1.0beta/ (retrieved 27.04.2025)
BSF4ooRexx	https://sourceforge.net/projects/bsf4oorexx/files/GA/BSF4ooRexx-850.20240304-GA/ (retrieved 27.04.2025)
IntelliJ	https://www.jetbrains.com/idea/download/?section=windows (retrieved 27.04.2025)
Plugin IntelliJ	https://sourceforge.net/projects/bsf4oorexx/files/Sandbox/aseik/ooRexxIDEA/GA/2.5.0/ (retrieved 27.04.2025)
JDBC Driver	https://github.com/xerial/sqlite-jdbc/releases (retrieved 27.04.2025)
SQLite	https://www.sqlite.org/download.html (retrieved 27.04.2025)
DB Browser	https://sqlitebrowser.org/dl/ (retrieved 27.04.2025)

List of aids for seminar paper/thesis

Title of paper/thesis: Managing an SQLite Database with ooRexx and JDBC using BSF4ooRexx

Author(s):

Last name(s), first name(s), student ID number(s):
Hofstetter, Cornelia, 01550814

Aids/tools used	Type(s) of use	Affected areas/chapters	Documentation
ChatGPT	ChatGPT was used to describe ooRexx code	4.2 Connection to Database p. 20 4.5 Curl p. 50	https://chatgpt.com/share/685172d2-81cc-8000-9d0c-b95f714d26d0
ChatGPT	ChatGPT was used to gain general information on a topic	Abstract	https://chatgpt.com/share/685172d2-81cc-8000-9d0c-b95f714d26d0

I hereby declare that I have listed all the aids I have used in the list above. If no aids have been used, it is also indicated in the list (to be listed under "Aids/tools used: none").

22.07.2025 Cornelia Hofstetter

Date Signature(s)