from SQL to XLS

documentation

Denis Servalli 1014900

Fabio Terzi 1015282

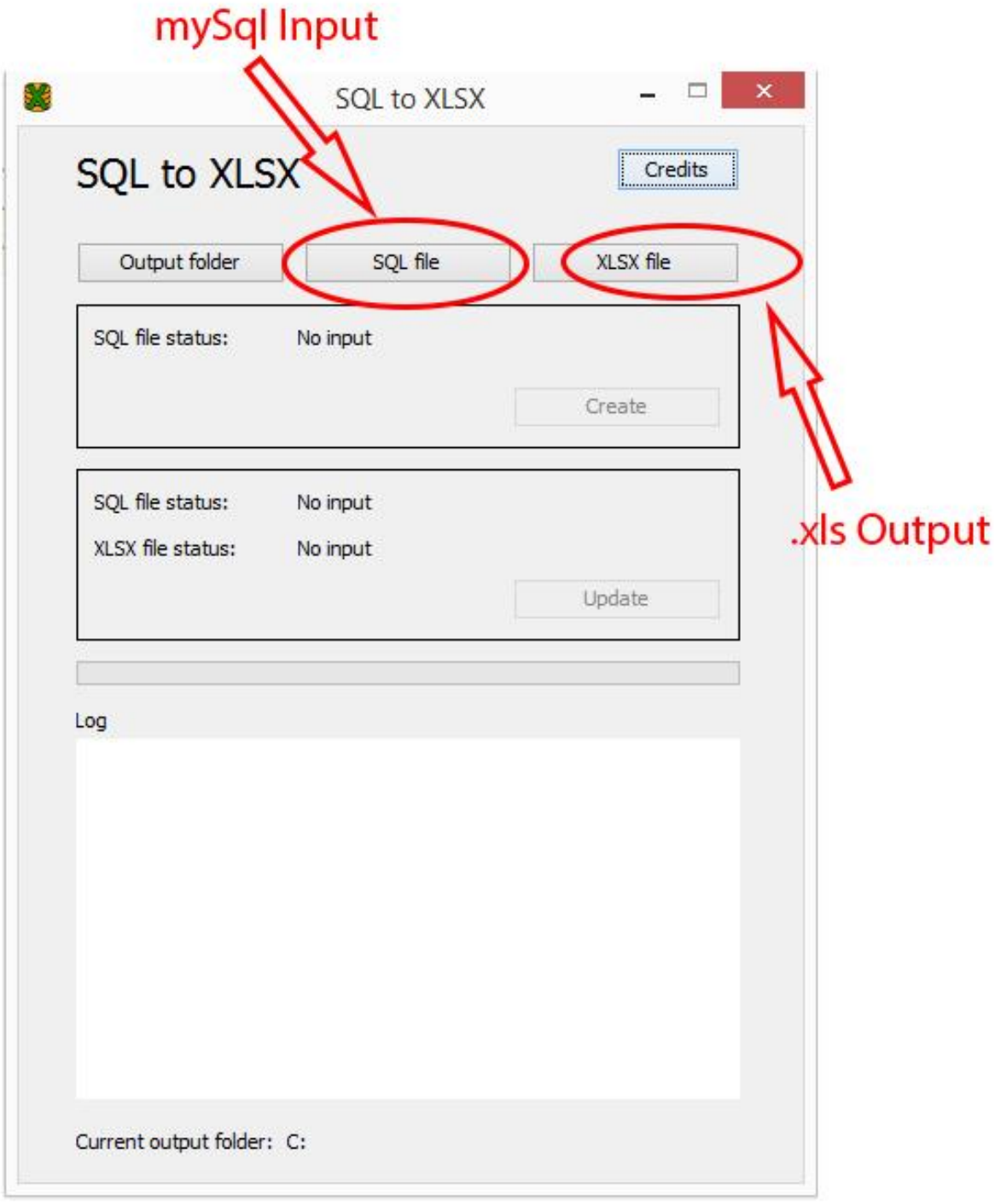
Matteo Gambirasio 1014578

Matteo Zambelli 1014593

Summary

INTRODUCTION:

The software that we have developed is related to the project of the course of languages and compilers. With this program you can take as input a mySql file (only CREATE, UPDATE and INSERT query), and we give an .xls file with the table that you want to create or modify.



CHAPTER 1 (Lexer):

1.1 Update Lexer:

This is our fragment and tokens for the update lexer:

```
fragment LETTER:('a'..'z' | 'A'..'Z');
fragment DIGIT:('0'..'9');
fragment AND_KEY: 'AND';
fragment OR_KEY: 'OR';

UPDATE_KEY : 'UPDATE';

TABLE_KEY: 'TABLE';

SET_KEY : 'SET';

WHERE_KEY: 'WHERE' ;

MULTIPLE_KEY: (AND_KEY | OR_KEY)*;
COMMA: ',';

END_STATEMENT: ';'  ;

ASSIGNMENT: '='  ;

PUNT: (',' | '.' | '!' | '?' | ';');

IDENTIFIER_TABLE_FIELD:'`' LETTER (LETTER | DIGIT | '_')* '`';

IDENTIFIER_VALUE_STRING:'\'' (LETTER | DIGIT | '_' | ' ' | PUNT )+ '\'';

IDENTIFIER_VALUE_NUMBER:DIGIT+;

WS: ('\r' | '\n' | '\t' | ' ') {$channel=HIDDEN;};
```

The lexer for the UPDATE command should identify the basic fields created between LEXER TOKENS which UPDATE , TABLE , SET, WHERE , then assigning these different values captured by the query . Thanks to the fields ' fields ' and ' targets ' let's go get the list of the various fields to be changed (fields) and the conditions to be awarded after the WHERE clause (targets) . Both the method ' fieldList ' the method ' targetRow ' then return strings containing the exact values captured by the query.
Once captured the structure , the fields and the target of the query , you must call on our UpdateHandler the update () method , which is required to run the control line by line . The reset () method is called to perform a reset instead of local variables used .

```
@members{
UpdateHandler uh=new UpdateHandler();
}

@init{}
     :
     (
     updateLine=UPDATE_KEY
     TABLE_KEY
     tableName=IDENTIFIER_TABLE_FIELD {uh.addTable(tableName.getText());}
     SET_KEY
     fields=fieldList {uh.addFields(tableName.getText(),fields);}
     WHERE_KEY
     targets=targetRow {uh.addTargets(tableName.getText(),targets);}
     END_STATEMENT
     {
      uh.update(updateLine.getLine());
      uh.reset();
     }
     )*
     {
        uh.printTableNames();
        uh.writeFile();
     }
     ;

fieldList returns[String list]
                              :
                              {String fieldName; String valueName = ""; String
returnString = "";}
                              (
                              fn=IDENTIFIER_TABLE_FIELD {fieldName=fn.getText()+'-';}
                              ASSIGNMENT
                              (vn=(IDENTIFIER_VALUE_STRING | IDENTIFIER_VALUE_NUMBER)
{valueName=vn.getText();})
                              COMMA* {returnString+=fieldName+valueName+'|';}
                              )+
                              {list=returnString;}
                              ;

targetRow returns[String list]
                              :
                              {String fieldName; String valueName = ""; String
returnString = "";}
                              (
                              fn=IDENTIFIER_TABLE_FIELD {fieldName=fn.getText()+'-';}
                              ASSIGNMENT
```

```
                                (vn=(IDENTIFIER_VALUE_STRING | IDENTIFIER_VALUE_NUMBER)
{valueName=vn.getText();})
                                mul=MULTIPLE_KEY* {
                                                    if(mul != null)

returnString+=fieldName+valueName+'['+mul.getText()+'|';
                                                    else
                                                    returnString+=fieldName+valueName+'|';
                                                    }
                                )*
                                {list=returnString;}
                                ;
```

1.2 Create and Insert Lexer:

This is our fragment and tokens for the update lexer:

```
fragment LETTER: ('A'..'Z')|('a'..'z');

fragment DIGIT: ('0'..'9');

WS:
('\r' | '\n' | '\t' | ' ') {$channel=HIDDEN;};

SPECIAL_CHAR: '_';

APEX_VALUE: '\'';

FIELD_VALUE: APEX_VALUE (LETTER|DIGIT|SPECIAL_CHAR)+ APEX_VALUE;

NUMBER: DIGIT+ ;

CREATE: 'CREATE' ;

TABLE: 'TABLE';

IF : 'IF';

NOT: 'NOT';

EXISTS: 'EXISTS';

INSERT: 'INSERT';

INTO: 'INTO';

APEX_TABLE: '`';

PRIMARY: 'PRIMARY';

UNIQUE: 'UNIQUE';

KEY: 'KEY';

VALUES: 'VALUES';
```

```
VARCHAR: 'varchar' LP DIGIT+ RP ;

LP: '(';

RP: ')';

INT: 'int' ?(LP DIGIT+ RP);

DATE: 'date' ;

DATETIME: 'datetime';

DOUBLE: 'double';

COMMA: ',';

END_STATEMENT: ';';

NAME: APEX_TABLE (LETTER|DIGIT|SPECIAL_CHAR)+ APEX_TABLE;
```

This lexer handles the CREATE and INSERT query from in mySql file. The CREATE lexer works like that: we've create a Table object, which is initialized whit the table name. The token "type" capture the type of data that will be entered in the table (we only managed int, varchar, date and datetime). Now we create the Column object with the type that we've just took. Is now being verified if the column is a primary key, in this case is added to the list of primary key of the table. Finally is checked if the table is not already present (we can't have two tables with the same name), so if everything is ok, the table will be inserted into the CreateInsertHandler, the subject who managed the parser.

For the INSERT lexer is a little bit different: first of all we verify if the table the you want to modify exist, we take it from the file and modify it. Than the table will be inserted into the CreateInsertHandler.

```
@members {
  CreateInsertHandler ih = new CreateInsertHandler();
  ArrayList<Table> tables = new ArrayList<Table>();
  Table tmpTable;
  Column tmpColumn;
  int index;
  ArrayList<String> columnNames = new ArrayList<String>();
  int i = 0;

}


@init{}:
(
  (
```

```
        rowNumber = CREATE TABLE ?(IF NOT EXISTS) tableName = NAME {tmpTable = new
Table(tableName.getText());} LP
        (( columnName = NAME type = (INT|VARCHAR|DATE|DATETIME)
              {
                    tmpColumn = new
Column(columnName.getText(),CreateInsertHandler.getColumnType(type.getText()));
                    tmpTable.addColumn(tmpColumn);
              }
              COMMA )+)
              ?(PRIMARY KEY LP primaryKey = NAME
{tmpTable.setPrimaryKey(primaryKey.getText());} (COMMA primaryKey = NAME)*
{tmpTable.setPrimaryKey(primaryKey.getText());} RP)
              RP END_STATEMENT
              {
                    if(!ih.tableAlreadyExists(tmpTable.getName(), tables,
rowNumber.getLine()))
                    {
                        tables.add(tmpTable);
                        System.out.println(rowNumber.getLine());
                    }
              }
         )
        |
        (
        {columnNames.clear();}
         rowNumber = INSERT INTO tableTarget = NAME
         {
           index = ih.getIndexByName(tableTarget.getText(),tables,rowNumber.getLine());
           tmpTable = tables.get(index);
         }
         LP ( columnName = NAME {columnNames.add(columnName.getText());}COMMA)*
columnName = NAME {columnNames.add(columnName.getText());} RP
         {
           if(!tmpTable.checkColumns(columnNames,rowNumber.getLine()))
           {
             ih.addError("Invalid column name(s) at line " +  rowNumber.getLine(),
rowNumber.getLine());
           }
         }
         VALUES
         ({i = 0;}LP (value = FIELD_VALUE
{tmpTable.insertValueInTable(columnNames.get(i++),value.getText());} COMMA)* value =
FIELD_VALUE {tmpTable.insertValueInTable(columnNames.get(i++),value.getText());} RP
COMMA)*
         {i = 0;}LP (value = FIELD_VALUE
{tmpTable.insertValueInTable(columnNames.get(i++),value.getText());} COMMA)* value =
FIELD_VALUE {tmpTable.insertValueInTable(columnNames.get(i++),value.getText());} RP
END_STATEMENT
        )
)*
{
       for (Table t : tables)
       {
         System.out.println(t.toString());
       }
       System.out.println(ih.getErrorLog());
       ih.createXls(tables);
}
;
```

CHAPTER 2 (Parser and Handler):

2.1 Parser:

Our parser managed the stream of token from the MySql file, this token will be managed by the handler object, who will be run the logical part of the operation.

2.2 Update Handler:

For convenience we have created the class UpdateStructure , which goes to instantiate four ArrayList : ' rowValues ' and ' ColumnValues ' to record the number of rows and columns , ' FieldValues ' to identify the field values field and finally ' clausolaType ' to differentiate between aND and OR clauses .

```
class UpdateStructure
{
	ArrayList<ArrayList<Integer>> rowValues;
	ArrayList<ArrayList<Integer>> columnValues;
	ArrayList<ArrayList<String>> fieldValues;
	ArrayList<ArrayList<String>> clausolaType;

	UpdateStructure()
	{
		rowValues = new ArrayList<ArrayList<Integer>>();
		columnValues = new ArrayList<ArrayList<Integer>>();
		fieldValues = new ArrayList<ArrayList<String>>();
		clausolaType = new ArrayList<ArrayList<String>>();
	}

}
```

In the class UpdateHandler go instead to create four Hashtable : ' FieldNames ' and ' targetNames ' for the management of the fields ' fields ' and ' targets ' , ' errorsTable ' for the management of semantic errors and finally ' updateTable ' representing the ' hashtable of what ' must be replaced. Finally creates two files, one for management of file input and output , a variable of type ' Workbook ' for the management of Excel spreadsheets and a string ' targetTable ' .

```
public class UpdateHandler {

	Hashtable<String,String> fieldNames;
	Hashtable<String,String> targetsNames;

	String targetTable;
	File inputXlsx;
	File outputXlsx;
	Workbook workbook;

	//gestione errori semantici (es, non trova la tabella o il campo)
	Hashtable<Integer,String> errorsTable;
```

```
//hashtable di quello che andrà sostituito
Hashtable<String, UpdateStructure> updateTable;
```

Below we analyze the behavior of our class expounding the ' usefulness of the various methods created :

private static WritableCellFormat getCellFormat(Colour colour, Pattern pattern) **throws** WriteException : This method has a utility function , is used to create and set the size and font of the cells .

private void updateStructure(ArrayList<Integer> rv, ArrayList<Integer> cv, ArrayList<String> fv,ArrayList<String> ct, String tableName) : With this method we are going to upgrade our facility .

public void addTable(String tableName) : Adds table name.

public void addFields(String tableName, String fields) : Adds field 'field' to table.

public void addTargets(String tableName, String targets) : Adds field 'target' to table.

public void printTableNames() : Print table names.

public void reset() : Reset of local variables used .

public void update(**int** lineNumber): This method is the heart of our Update. First, clean up the sheet of quotes typical SQL . Then initialize the array support to contain the pair of column index and the value that it should be replaced . Then we obtain the first row (row 0 , header) : we obtain the number of columns in the spreadsheet , scroll down the list of targets , we are looking in the header if any of the columns is called as the field under examination to obtain finally the list of fields to be replaced and the relative position of the columns . Let's go then to capture the kind of clause (OR or AND) .

public void writeFile() : In this method we are going to write the file , changing the fields analyzed and recorded in the update () method . We read the old workbook and check if the cell is to be changed or not. Before doing so , however, you must run the check on the type of clause : if we are in the presence of the OR clause , the change is carried out without problems . If we are dealing with the AND clause , the change is made only if the line number is repeated as many times as the number of clauses . Finally , since the file is regenerated , let's actually write the label created or that pre - existing (if it is not to be changed).

2.3 Create and insert Handler:

The CreateInsertHandler object took from create-insert lexer the Tables and put it into an excel (.xls) file thanks to the jExel API.

This is our class fields, who will be initialized with the CreateInsertHandler builder.

```
ArrayList<String> tableNames;
Hashtable<Integer, String> errorLog;
File output = new File("out2.xls");
```

We need this due to the jExel API:

```
WritableWorkbook workbook;
ArrayList<WritableSheet> sheetList;
```

addTable method is used to add the table name to the array list of table:

```
public void addTable(String tablename)
```

getColumnType return if input column is INT , VARCHAR, DATE or DATETIME:

```
public static ColumnType getColumnType(String in)
```

printTableNames return all table names in array tableNames:

```
public void printTableNames()
```

getColumnLength return, only for INT and VARCHAR type, the length specified from the user:

```
public static int getColumnLength(String in)
```

tableAlreadyExists return true if the table (name) already exists in our list of table (table). If is true, we save the error line and put it into the error log:

```
public boolean tableAlreadyExists(String name, ArrayList<Table> table,int line)
```

getIndexByName return where the table (name) is on the table list (list), if table (name) doesn't exist, the method save line error into error log:

```
public int getIndexByName(String name, ArrayList<Table> list, int line )
```

addError is used in many method to add an error into the hashtable errorLog:

```
public void addError(String msg, Integer line)
```

getErrorLog return the hashtable errorLog:

```
public Hashtable<Integer, String> getErrorLog()
```

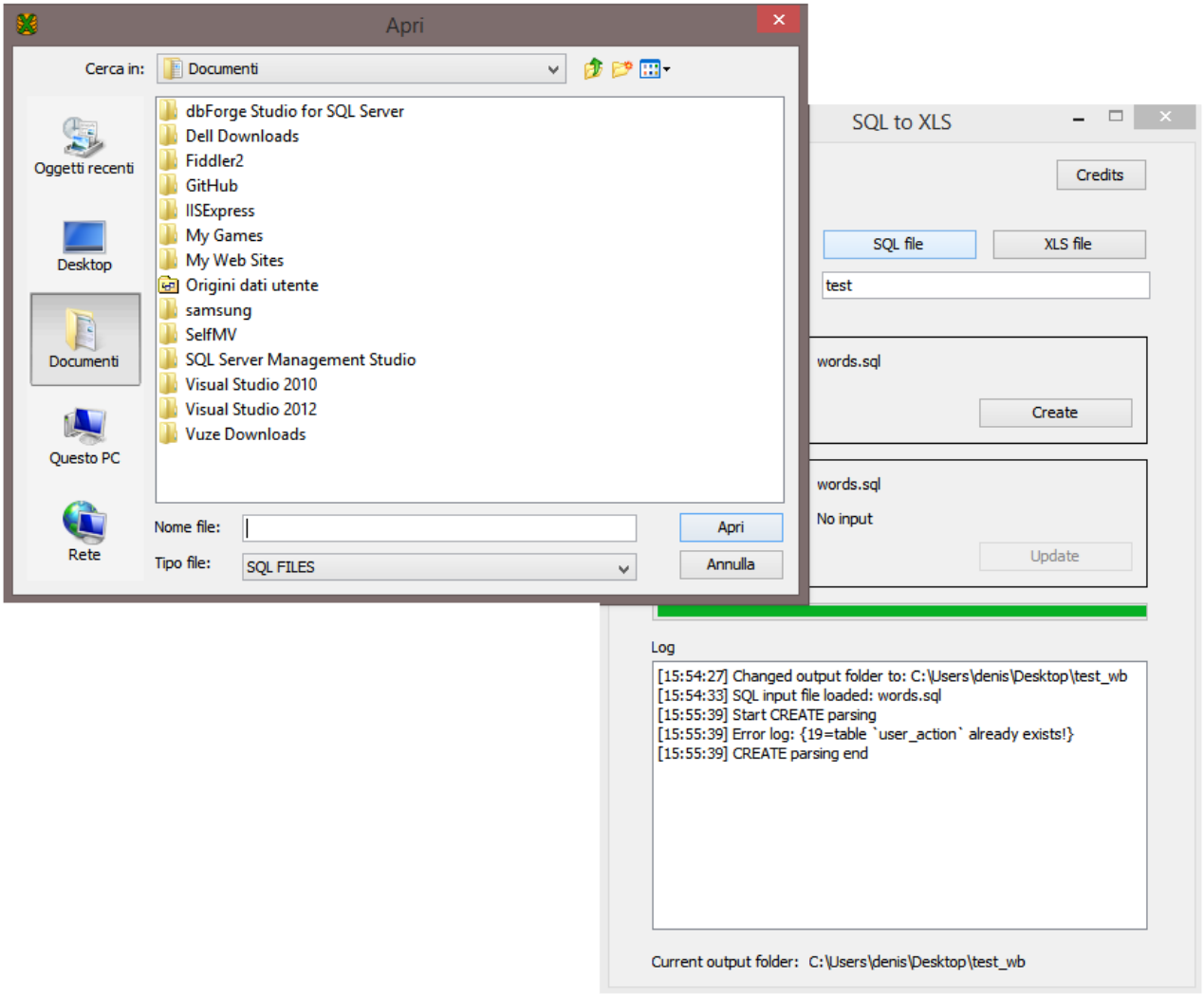
createXls take the arraylist of table and put it into an xls file:

```
public void createXls(ArrayList<Table> tables)
	{
		Label tmp;
		ArrayList<String> values;
		for(Table t : tables)
		{
			WritableSheet tmpSheet = workbook.createSheet(t.getName(),
tables.indexOf(t));
			for(Column c : t.getColumns())
			{
				tmp = new Label(t.getColumns().indexOf(c), 0, c.getName());

				try {
					tmpSheet.addCell(tmp);
				} catch (WriteException e) {
					// TODO Auto-generated catch block
					e.printStackTrace();
				}

				values = c.getValues();
				for(int i = 0 ; i < values.size() ; i++)
				{
					System.out.println("riga: " + t.getColumns().indexOf(c)
+ " colonna: " +  (i + 1 ) + " valore : " + values.get(i));
					tmp = new Label(t.getColumns().indexOf(c), i + 1,
values.get(i));

					try {
						tmpSheet.addCell(tmp);
					} catch (WriteException e) {
						// TODO Auto-generated catch block
						e.printStackTrace();
					}
				}

			}
		}
		try {
			workbook.write();
			workbook.close();
		} catch (WriteException | IOException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
```

CHAPTER 3 (Use cases):

Now we will show some examples of usage from the “from SQL to xls” software.

In this screen we can see the selection fields of input and output file.



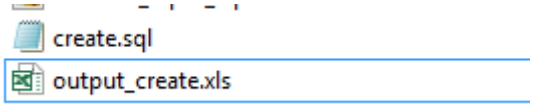
3.1 usage example (create):

This is an example of use of our software, starting from an sql file, we will generate an excel file with the table you want to create and fill

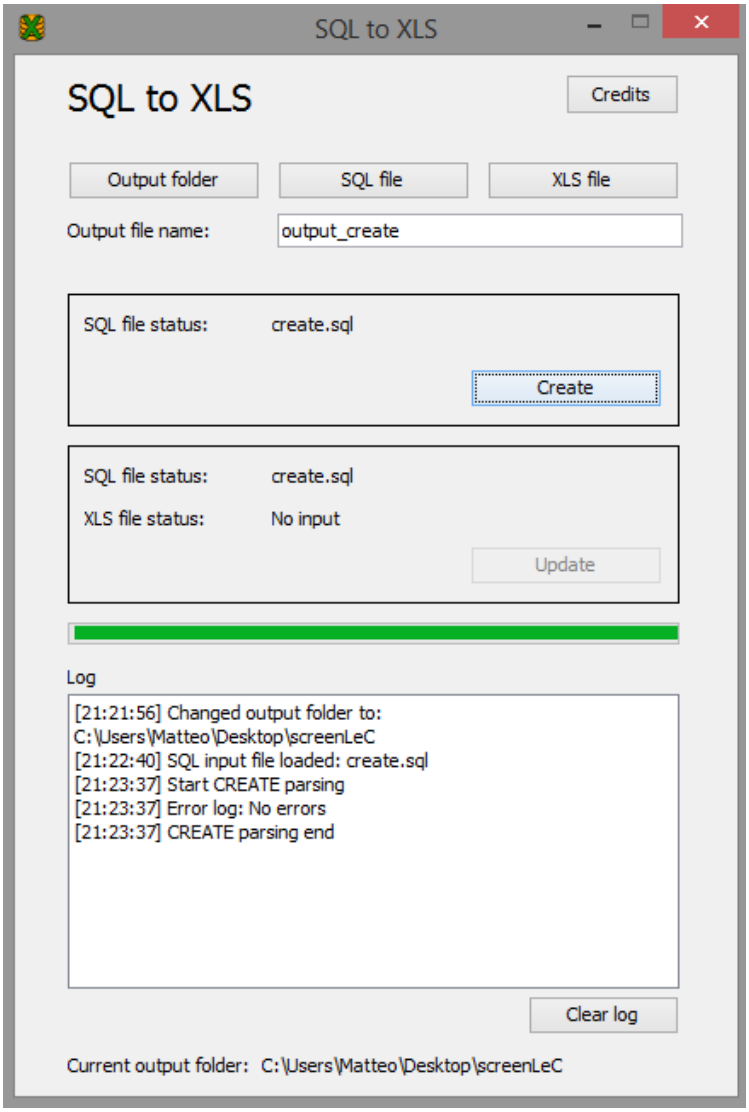
1. From an SQL file:

```
CREATE TABLE IF NOT EXISTS `admin_assert` (
  `assert_id` int(10),
  `assert_type` varchar(20),
  `assert_data` datetime,
  PRIMARY KEY (`assert_id`,`assert_type`)
);

CREATE TABLE IF NOT EXISTS `user_action` (
  `assert_id` int(10),
  `assert_type` varchar(20),
  `assert_data` datetime,
  PRIMARY KEY (`assert_id`)
);
CREATE TABLE IF NOT EXISTS `user_update` (
  `assert_id` int(10),
  `assert_type` varchar(20),
  `assert_data` datetime,
  `assert_data2` datetime,
  PRIMARY KEY (`assert_id`)
);

INSERT INTO `user_update` (`assert_id`, `assert_type`, `assert_data`, `assert_data2`) VALUES
('1', 'fabio', '1', '20150325'),
('2', 'gambi', '2', '20150325'),
('3', 'zambe', '3', '20150325');
```

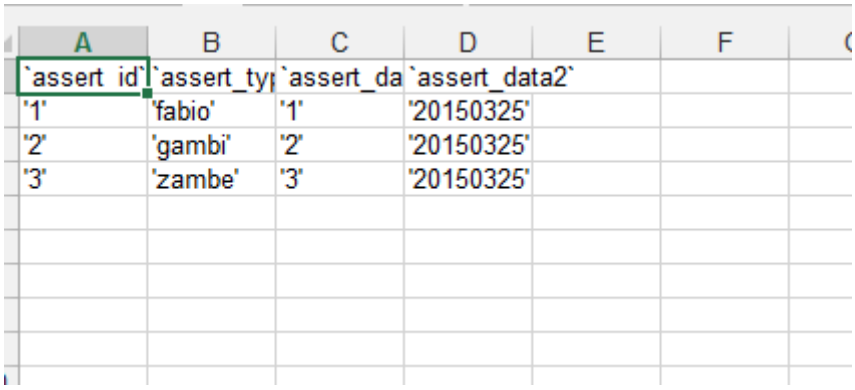
2. Choose eventually a specific name for the excel output file in the text field dedicated, like



3. Start the program, press button “Create”:



4. This is the output created:



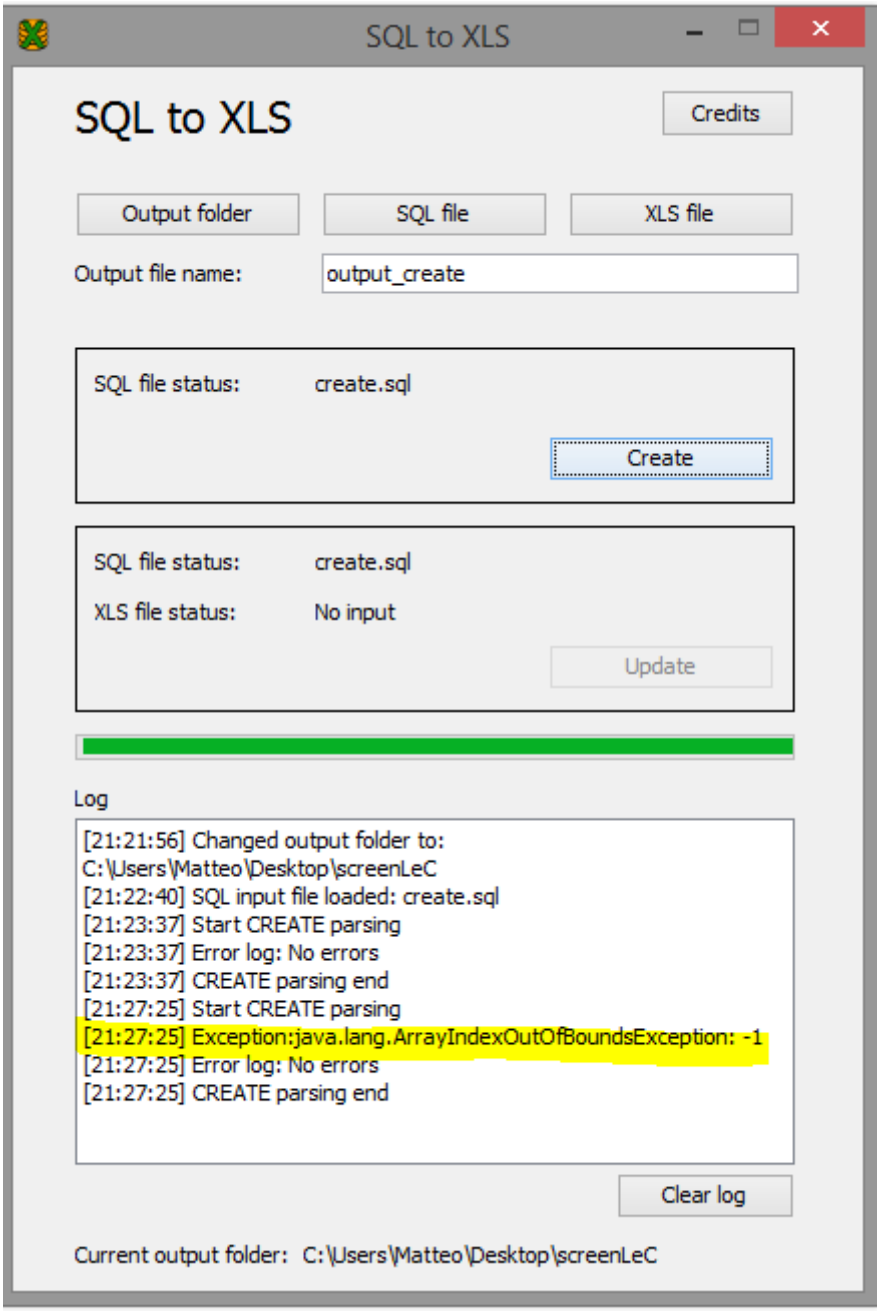
A	B	C	D	E	F
`assert_id`	`assert_typ	`assert_da	`assert_data2`		
'1'	'fabio'	'1'	'20150325'		
'2'	'gambi'	'2'	'20150325'		
'3'	'zambe'	'3'	'20150325'		

3.2 error example (insert):

1. Here we try to run a wrong query (the table doesn't exists):

```
INSERT INTO `user_update_wrong` (`assert_id`, `assert_type`, `assert_data`, `assert_data2`) VALUES
('1', 'fabio', '1', '20150325'),
('2', 'gambi', '2', '20150325'),
('3', 'zambe', '3', '20150325');
```

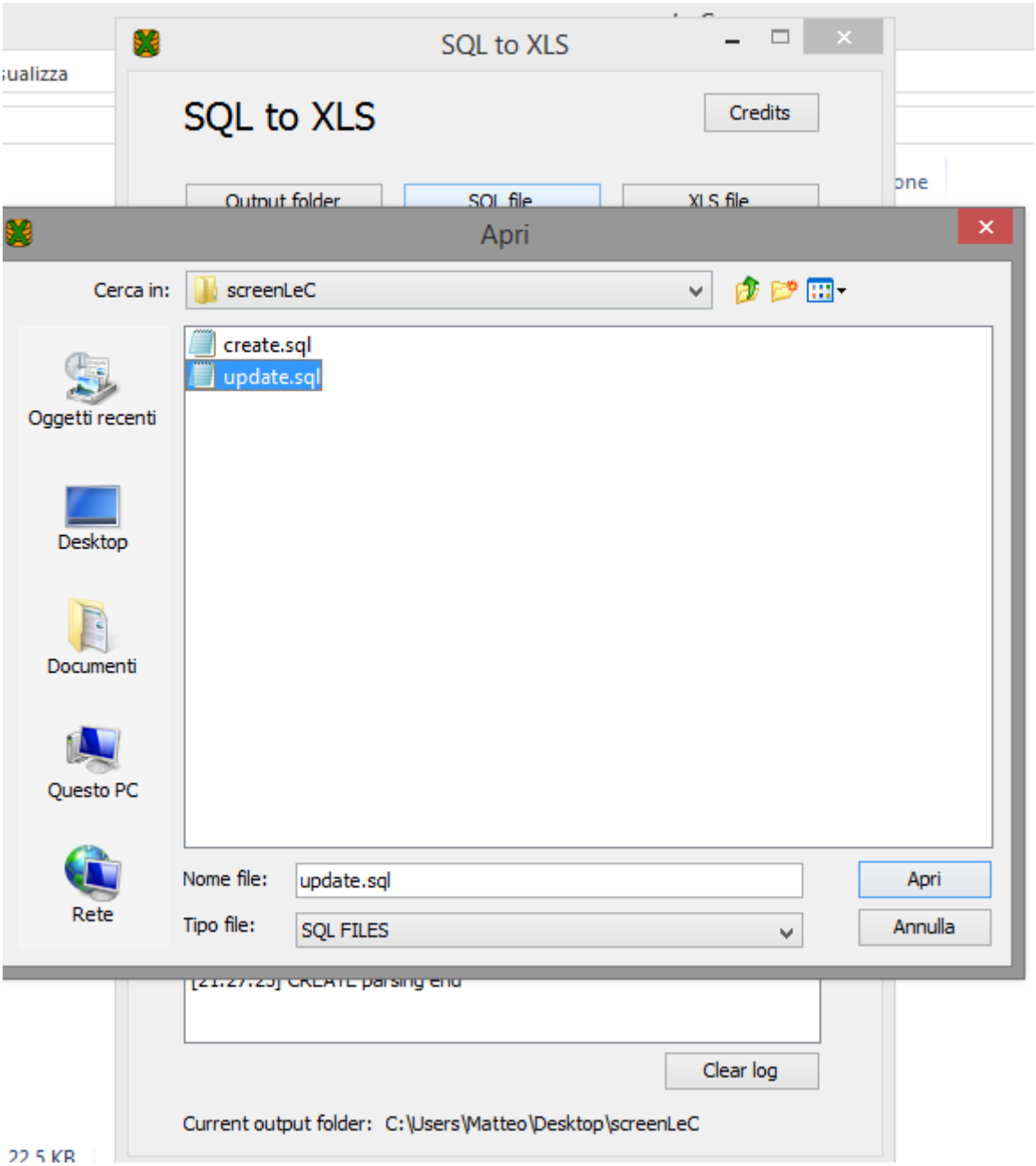
2. The software will catch an exception:



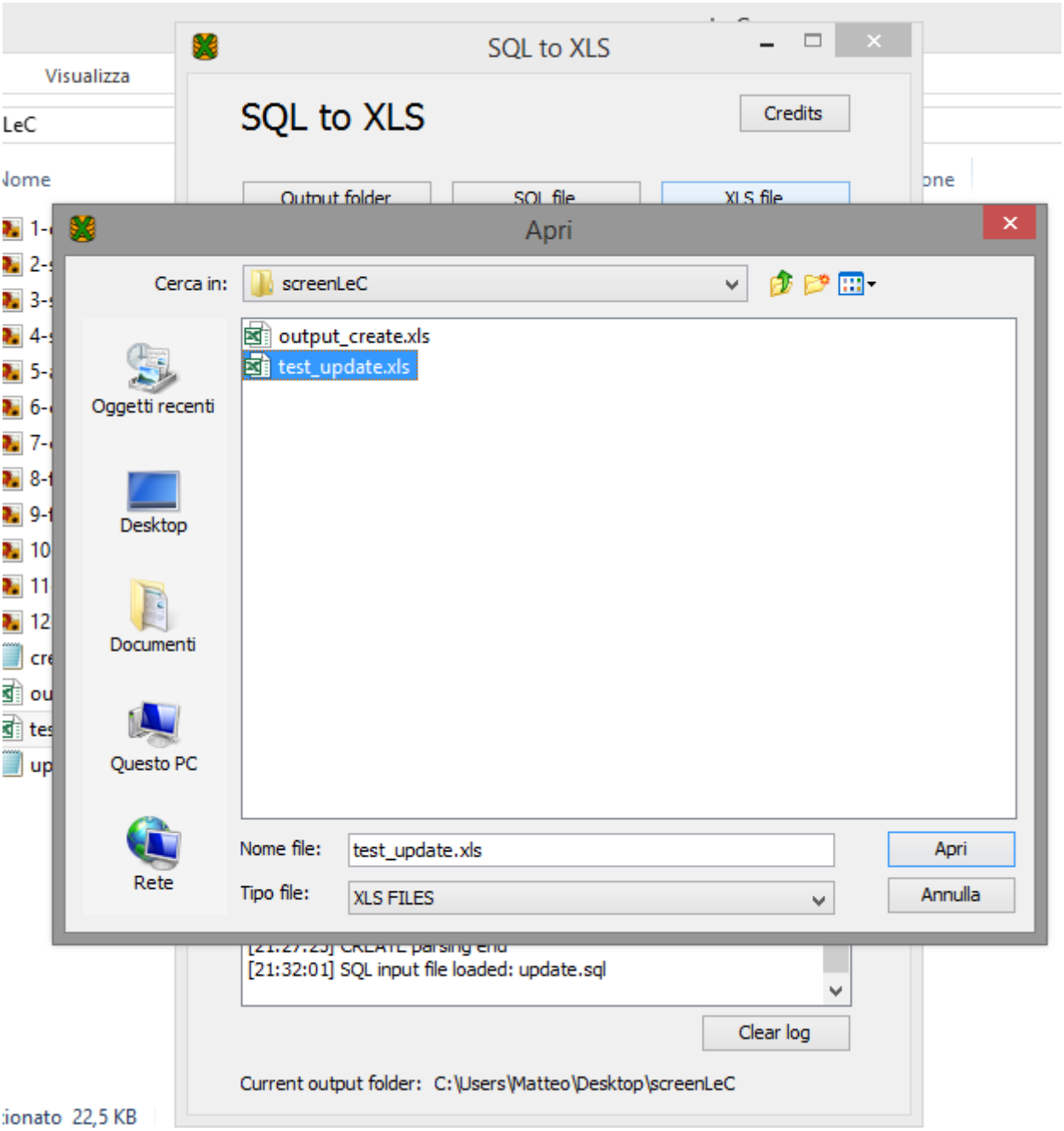
3.3 usage example (update):

This is an example of update:

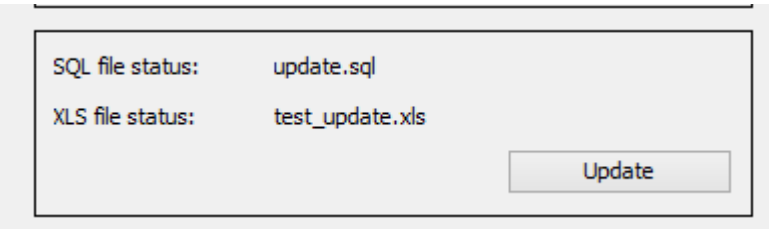
1. From an sql file:



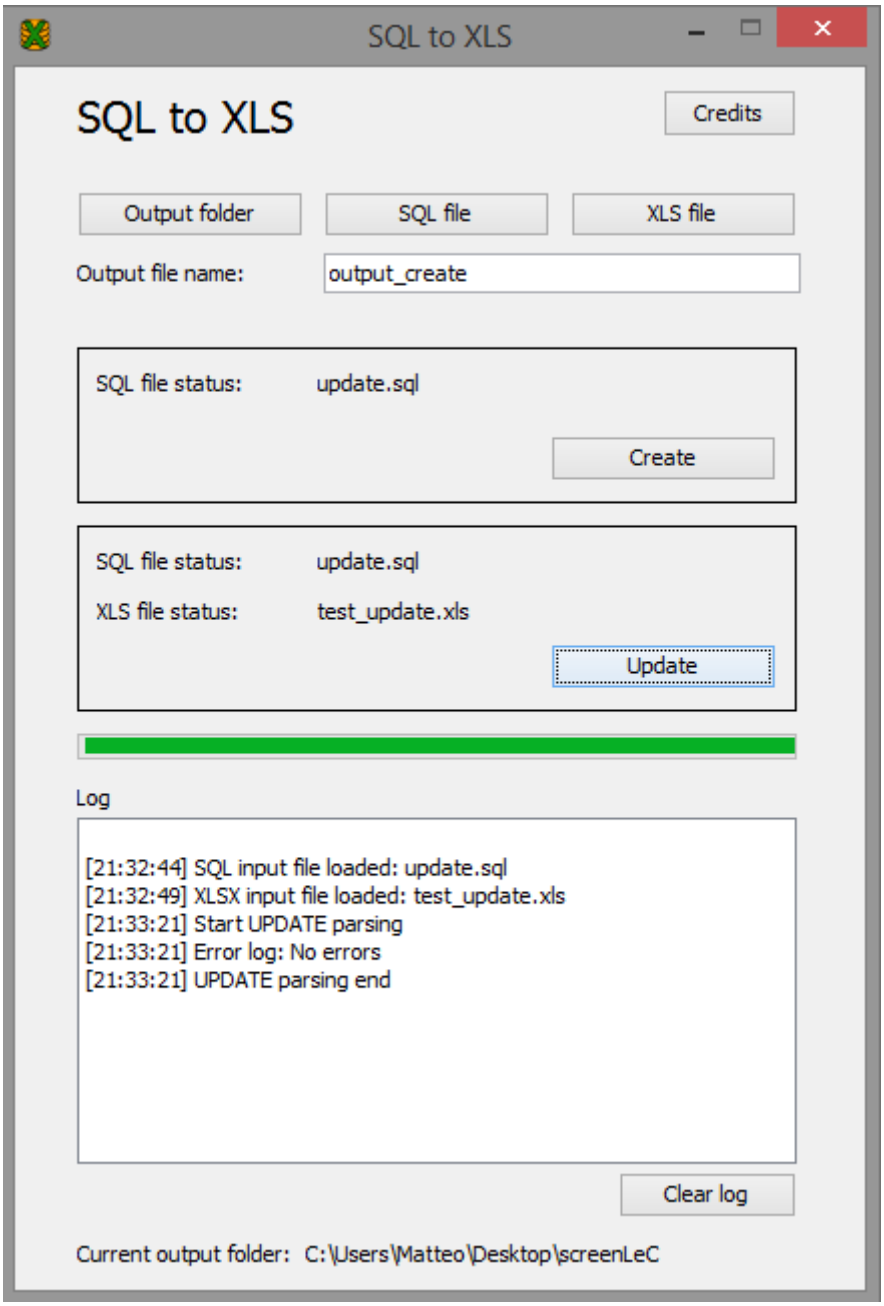
2. Select the .xls file that contains the tables and tuplas to analyse and eventually modify in accord with the .sql file :



3. The software enable the update button with the file specified previously:



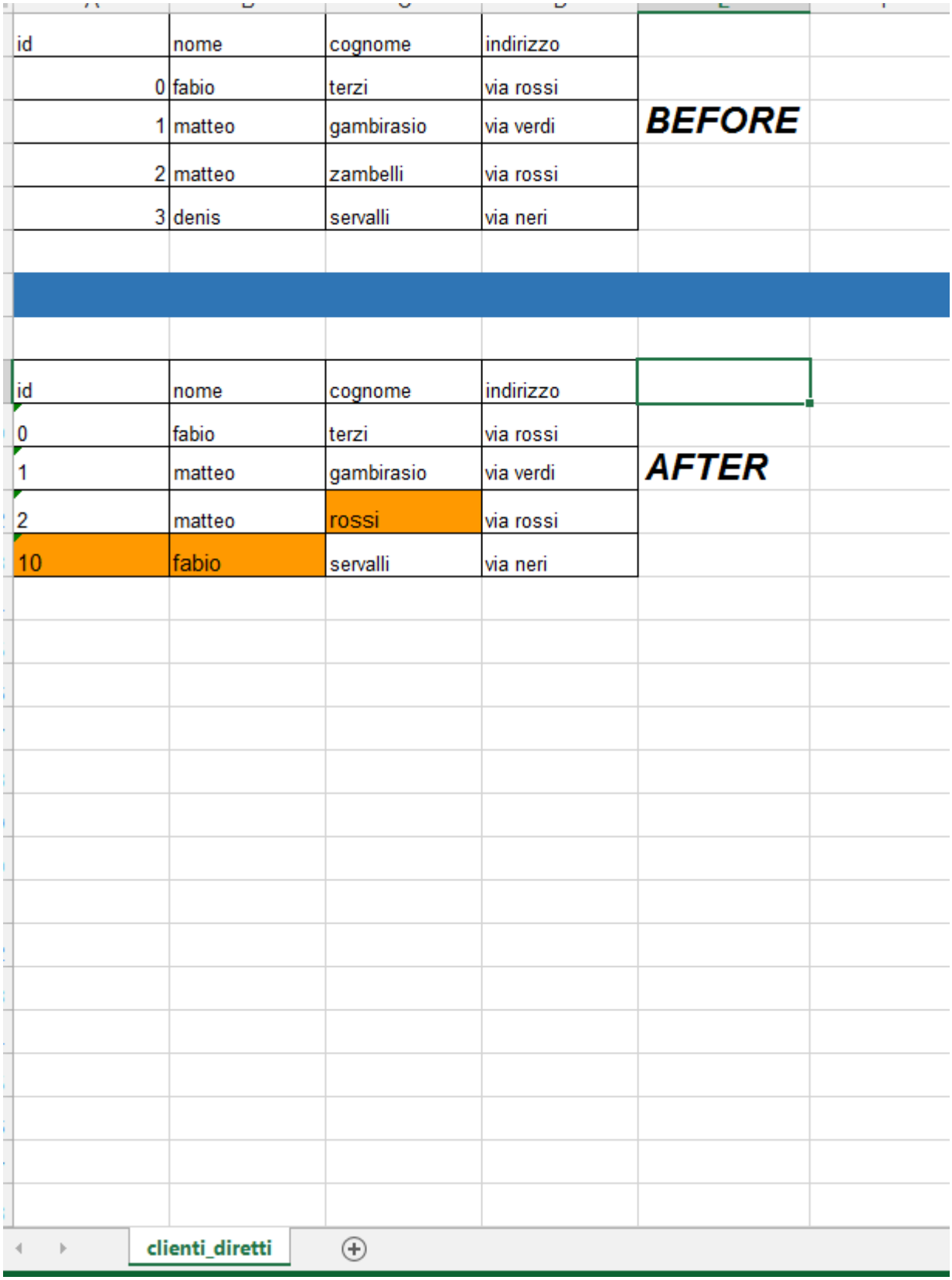
4. Press on “update” button:



This is the query from we start:

```
UPDATE TABLE `clienti_diretti` SET `id` = 10, `nome` = 'fabio' WHERE `nome` = 'denis';
UPDATE TABLE `clienti_diretti` SET `cognome` = 'rossi' WHERE `nome` = 'matteo' AND `indirizzo` = 'via rossi';
```

Here we can see the .xls file before and after the update:



id	nome	cognome	indirizzo
0	fabio	terzi	via rossi
1	matteo	gambirasio	via verdi
2	matteo	zambelli	via rossi
3	denis	servalli	via neri

BEFORE

id	nome	cognome	indirizzo
0	fabio	terzi	via rossi
1	matteo	gambirasio	via verdi
2	matteo	rossi	via rossi
10	fabio	servalli	via neri

AFTER

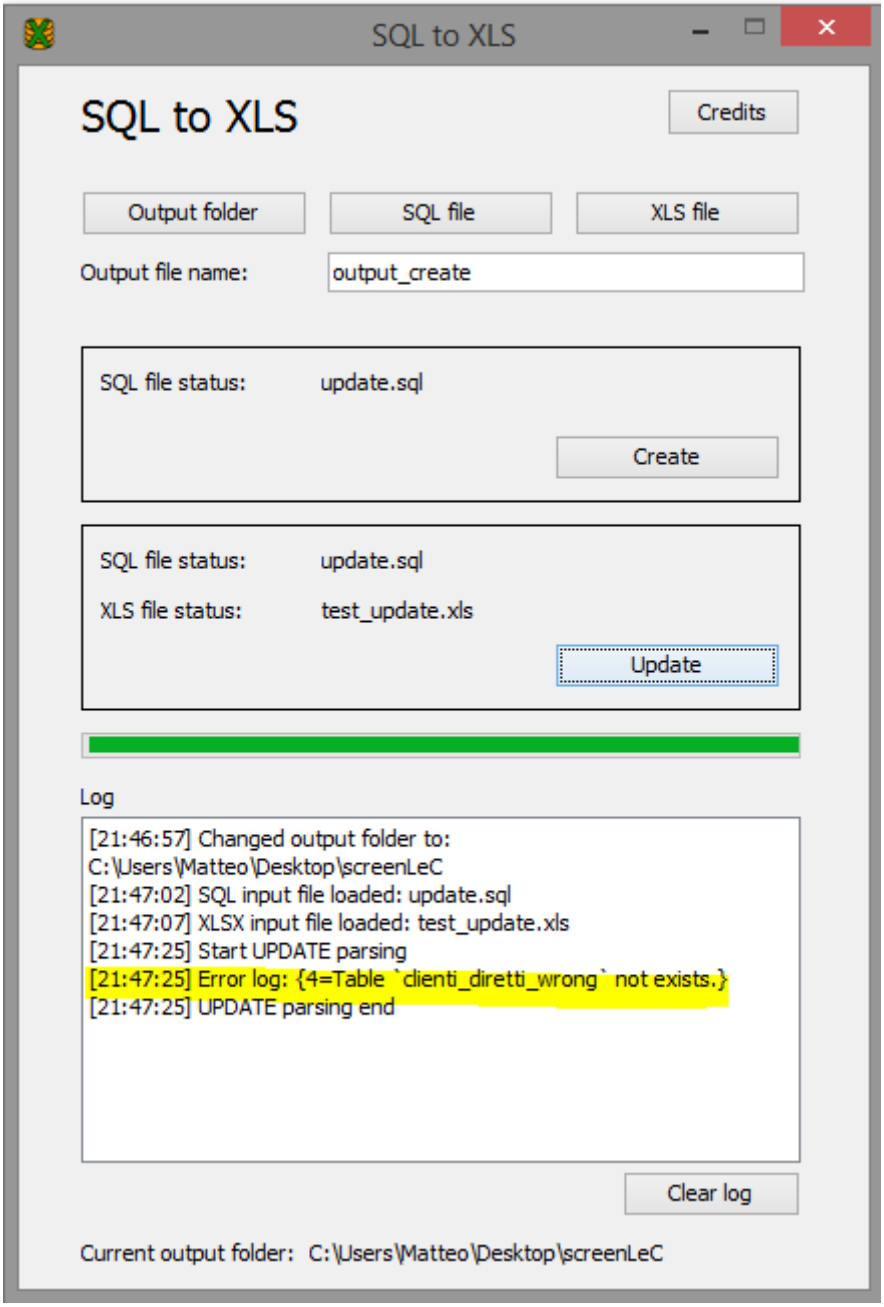
clienti_diretti

3.4 error example (update):

1. Here we try to run a wrong update query, adding this line to the same file we have used before:

```
UPDATE TABLE `clienti_diretti_wrong` SET `id` = 10, `nome` = 'fabio' WHERE `nome` = 'denis';
```

2. The software will show this error:



CHAPTER 4 (Conclusion):

4.1 IDE & tools:

This is the list of tools, IDE and software versioning that we have used to develop this software:

IDE: eclipse;

Tools: WindowBuilder and Antlr;

Software versioning: SVN;

4.2 conclusion:

In this paper we presented our work about two different kind of lexers, in accord to the different syntax for queries UPDATE and CREATE/INSERT.

We have focussed on this queries because they rappresent best practise for restore data, after for example a database backup. With the traditional format on MySQL backup and this software you will be alble to reproduce the whole database in .xls format. We have left out queries like SELECT because the aim of the project was only to reproduce in a more user friendly way the sql syntax, and not to manipulate or search data.

In this way you will be able to read the content in a simple graphical layout instead of the traditional and more complex sql format.

The source of our implementation is available for download on GitHub:

https://github.com/matteozambelli/sql2xls