SQLite 21

Database and programming

LILLEBAELT ACADEMY OF
PROFESSIONAL HIGHER EDUCATION

Author
Martin Grønholdt
mart80c7@edu.eal.dk

Sunday 14 May 2017

Table of Contents

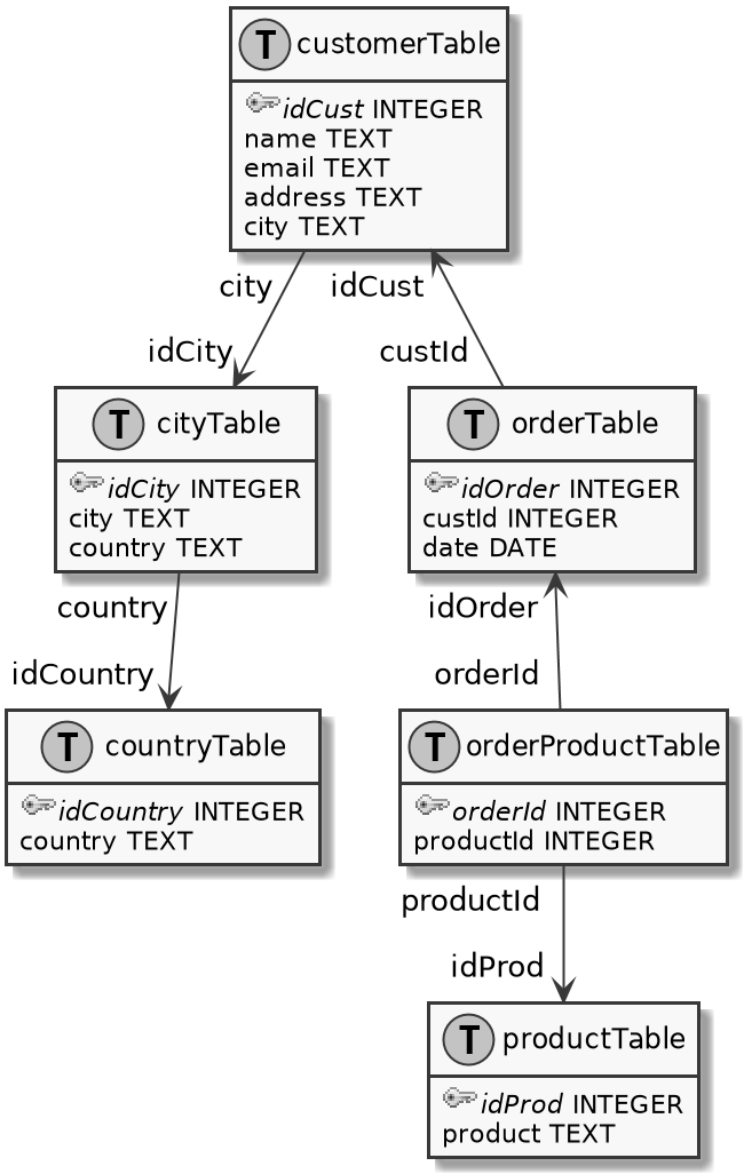
1. Introduction

This hand in is about using primary and foreign keys in SQLite databases.

All hand ins for this course are available on GitHub at:
https://github.com/deadbok/eal_programming

2. Database structure

From the description in the assignment the following database diagram has been deduced.



Database diagram and relations

In the diagram above each database table is represented by a box, the field names and data types are in the lower part of the box.

The primary key in each table are symbolised by a key icon next to the field name. The primary key is the field used to identify all records, in this case they all have the text “id” in their name. The primary key value for each record must be unique and not NULL.

Foreign keys create a relationship (a constraint) between the key in the current table, and a key in some other table. Using the “cityTable” above as an example, the line between this table and “countryTable” is an illustration of a foreign key. The labels on the line and the direction of the arrow, show that the field “country” in the table “cityTable” is referencing the field “idCountry” in the table “countryTable”.

Table	**Primary key**	**Foreign keys.**
customerTable	idCust	idCust, city
cityTable	idCity	country
countryTable	idCountry	
orderTable	idOrder	idOrder
orderProductTable	orderId	orderId, productId
productTable	idProd	

Table of the primary and foreign keys in each database table.

3. Creating the tables.

```
$ sqlite3 order.db
SQLite version 3.17.0 2017-02-13 16:02:40
Enter ".help" for usage hints.
sqlite> CREATE TABLE productTable(
  ...>  product TEXT,
  ...>  idProd INTEGER PRIMARY KEY
  ...> );
sqlite> CREATE TABLE countryTable(
  ...>  idCountry INTEGER PRIMARY KEY,
  ...>  country TEXT
  ...> );
sqlite> CREATE TABLE cityTable(
  ...>  country TEXT,
  ...>  city TEXT,
  ...>  idCity INTEGER PRIMARY KEY,
  ...>  FOREIGN KEY(country) REFERENCES countryTable(idCountry)
  ...> );
sqlite> CREATE TABLE customerTable(
  ...>  address TEXT,
  ...>  email TEXT,
  ...>  idCust INTEGER PRIMARY KEY,
  ...>  name TEXT,
  ...>  city TEXT,
  ...>  FOREIGN KEY(city) REFERENCES cityTable(idCity)
```

```
  ...> );
sqlite> CREATE TABLE orderTable(
  ...>  idOrder INTEGER PRIMARY KEY,
  ...>  date DATE,
  ...>  custId INTEGER,
  ...>  FOREIGN KEY(custId) REFERENCES customerTable(idCust)
  ...> );
sqlite> CREATE TABLE orderProductTable(
  ...>  orderId INTEGER PRIMARY KEY,
  ...>  productId INTEGER,
  ...>  FOREIGN KEY(orderId) REFERENCES orderTable(idOrder),
  ...>  FOREIGN KEY(productId) REFERENCES productTable(idProd)
  ...> );
sqlite> .tables
cityTable          customerTable      orderTable
countryTable       orderProductTable  productTable
sqlite> .schema
CREATE TABLE productTable(
product TEXT,
idProd INTEGER PRIMARY KEY
);
CREATE TABLE countryTable(
idCountry INTEGER PRIMARY KEY,
country TEXT
);
CREATE TABLE cityTable(
country TEXT,
city TEXT,
idCity INTEGER PRIMARY KEY,
FOREIGN KEY(country) REFERENCES countryTable(idCountry)
);
CREATE TABLE customerTable(
address TEXT,
email TEXT,
idCust INTEGER PRIMARY KEY,
name TEXT,
city TEXT,
FOREIGN KEY(city) REFERENCES cityTable(idCity)
);
CREATE TABLE orderTable(
idOrder INTEGER PRIMARY KEY,
date DATE,
custId INTEGER,
FOREIGN KEY(custId) REFERENCES customerTable(idCust)
);
CREATE TABLE orderProductTable(
orderId INTEGER PRIMARY KEY,
productId INTEGER,
FOREIGN KEY(orderId) REFERENCES orderTable(idOrder),
FOREIGN KEY(productId) REFERENCES productTable(idProd)
);
```

Creating the tables in the SQLite CLI.

The above illustration shows actually creating the tables from the database diagram in the SQLite CLI. Lets focus on a few lines:

```
CREATE TABLE cityTable(
idCity INTEGER PRIMARY KEY,
city TEXT,
country TEXT,
FOREIGN KEY(country) REFERENCES countryTable(idCountry)
);
```

This creates the before mentioned “cityTable”. “idCity” is an example of a field that acts like the primary key. “country” is referencing “idCountry” in the table “countryTable”.

4. Inserting data

To enable SQLite to use foreign key constraints a command must be issued.

```
sqlite> PRAGMA foreign_keys = ON;
```

SQLite CLI command to enable foreign key constraints.

Now to illustrate how foreign key constraints work, adding a row of data to the “orderTable” is going to fail because there are no valid “idCust” in the “customerTable” that corresponds to the one we are trying to insert in orderTable.

```
sqlite> INSERT INTO orderTable(idOrder, date, custId) VALUES(1,'2017-05-14',1);

Error: FOREIGN KEY constraint failed
sqlite> INSERT INTO orderProductTable(orderId, productId) VALUES(1, 1);

Error: FOREIGN KEY constraint failed
```

Failing to insert data because of a foreign key constraint.

As illustrated above this also fails for the “orderProductTable” for the same reason.

To put a data into the “orderProductTable” these constraints has to be fulfilled:

- “orderId” has to be in “idOrder” in the table “orderTable”
    - “custId” has to be in “idCust” in the table “customerTable”
        - “city” has to be in “idCity” in the table “cityTable”
            - “country” has to be in “idCountry” in the table “countryTable”
- “productId” has to be in “idProd” in the table “productTable”
    - “productId” has to be in “idProd” in the table “productTable”

This is like a dependency tree with the “orderProductTable” at the bottom.

If the tables are populated from the deepest dependencies (the leafs) like in the list below, everything should work out fine:

- INSERT INTO countryTable(idCountry, country) VALUES(3, 'Denmark');
- INSERT INTO cityTable(idCity, city, country) VALUES(1, 'Odense C', 3);
- INSERT INTO customerTable(idCust, name, email, address, city) VALUES(1, 'Per', 'pda@eal.dk', 'Mystreet 1', 1);
- INSERT INTO orderTable(idOrder, custId, date) VALUES(20140004, 1, '2015-06-06');
- INSERT INTO productTable(product, idProd) VALUES('HDMI cable', 10);
- INSERT INTO orderProductTable(orderId, productId) VALUES(20140004, 10);

The same in the SQLite CLI:

```
sqlite> INSERT INTO countryTable(idCountry, country) VALUES(3, 'Denmark');
sqlite> INSERT INTO cityTable(idCity, city, country) VALUES(1, 'Odense C', 3);
sqlite> INSERT INTO customerTable(idCust, name, email, address, city) VALUES(1, 'Per',
'pda@eal.dk', 'Mystreet 1', 1);
sqlite> INSERT INTO orderTable(idOrder, custId, date) VALUES(20140004, 1, '2015-06-06');
sqlite> INSERT INTO productTable(product, idProd) VALUES('HDMI cable', 10);
sqlite> INSERT INTO orderProductTable(orderId, productId) VALUES(20140004, 10);
```

Inserting the data with regards to the foreign key constraints.

As illustrated above no errors were reported when inserting data in, making sure the every foreign key existed in the referenced field.

```
sqlite> INSERT INTO orderProductTable(orderId, productId) VALUES(20140004, 10);
Error: UNIQUE constraint failed: orderProductTable.orderId
```

Inserting a row with an primary key that is already present.

Above is an example of hitting the primary key constraint, trying to insert data with an existing “orderId” into the “orderProductTable” is not allowed because the primary key needs to be unique.

As a final proof that everything worked, here is a listing of all tables at the SQLite CLI:

```
sqlite> select * from countryTable;
idCountry|country
3|Denmark
sqlite> select * from cityTable;
country|city|idCity
3|Odense C|1
sqlite> select * from customerTable;
address|email|idCust|name|city
Mystreet 1|pda@eal.dk|1|Per|1
sqlite> select * from orderTable;
idOrder|date|custId
20140004|2015-06-06|1
sqlite> select * from productTable;
product|idProd
HDMI cable|10
sqlite> select * from orderProductTable;
orderId|productId
20140004|10
```

Listing of all tables and rows after the insert commands.

As shown above all data was successfully added to the tables when honouring the constraints of each table.

5. Database diagram and creation in this hand in.

The database diagram in this hand in was created by (ab)using PlantUML[1], PlantUML is a program that uses a simple language to represent a wide range of graphs. The database diagram in this hand in look like this in PlantUML language:

```
@startuml

skinparam monochrome true
hide methods
hide stereotypes
scale 2

!define table(x) class x << (T,#FFAAAA) >>
!define primary_key(x) <$primary><i>x</i>

sprite $primary [17x12/16z] bOqv3e1030CJRzPn9Fx_NWY7n4eqJ3TJs6OVa5pTpD-
5tl3YyFHG-4DsqAOnWgawWp0r0KGagDuGMYMJxbMrBxzLPJ_O0G00

table(customerTable) {
    primary_key(idCust) INTEGER
    name TEXT
    email TEXT
    address TEXT
    city TEXT
}

table(cityTable) {
    primary_key(idCity) INTEGER
    city TEXT
```

1 PlantUML - http://plantuml.com

```
    country TEXT
}

table(countryTable) {
    primary_key(idCountry) INTEGER
    country TEXT
}

table(orderTable) {
    primary_key(idOrder) INTEGER
    custId INTEGER
    date DATE
}

table(orderProductTable) {
    primary_key(orderId) INTEGER
    productId INTEGER
}

table(productTable) {
    primary_key(idProd) INTEGER
    product TEXT
}

customerTable "city  " --> "idCity" cityTable
cityTable "country" --> "idCountry" countryTable
customerTable "idCust " <-- "custId  " orderTable
orderTable "idOrder " <-- "orderId " orderProductTable
orderProductTable "productId " --> "idProd " productTable


@enduml
```

The database diagram in this hand in written in PlantUML language.

Since all data needed to create the tables are in this file, a python program is able to parse this and output the SQLite commands needed to create the tables.

5.1. dbdiy2sql.py

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# The above lines tell the shell to use python as interpreter when the
# script is called directly, and that this file uses utf-8 encoding,
# because of the country specific letter in my surname.
"""
Name: dbdia2sql.py
Author: Martin Bo Kristensen Grønholdt.
Version: 1.0.0 (2017-05-14)

Convert a database diagram written in a subset of PlantUML to SQLite syntax
that will create the actual tables and relations.
"""
import argparse
import re
```

```
# Program version.
__VERSION__ = '1.0.0'


class Table:
    """
    Parses a table from PlantUML file.
    """

    def __init__(self):
        """
        Constructor.
        """
        # No name.
        self.name = None
        # No fields
        self.fields = {}

    def parse(self, lines):
        """
        Parse a tables information from the PUML file.

        :param lines: The remaining lines of the PUML file.
        """
        # Use regular expressions to isolate the table name.
        exp = re.search('\s*table\((\w+)', lines[0])
        if exp:
            self.name = exp.group(1)

        # Go through the rest of the lines.
        for line in lines[1:]:
            # Isolate all words and the ending '}'.
            tokens = re.findall(r'[\w\}]+', line)
            # If there was anything to isolate.
            if len(tokens) > 0:
                if tokens[0] == 'primary_key':
                    # This is a primary key, add as such.
                    self.fields[tokens[1]] = {'name': tokens[1],
                                              'primary': True,
                                              'type': tokens[2]}
                elif '}' not in line:
                    # This is not a primary key, add it.
                    self.fields[tokens[0]] = {'name': tokens[0],
                                              'primary': False,
                                              'type': tokens[1]}
                else:
                    # Done.
                    break

    def sql(self):
        """
        Return the SQL command to create the table.

        :return: SQL command string.
        """
        # List of foreign keys.
        foreign = ''
        # SQL to create the table itself
        ret = 'CREATE TABLE {}(\n'.format(self.name)
```

```
        # Loop over the fields.
        for field in self.fields.values():
            # Add the field.
            ret += ' {0} {1}'.format(field['name'], field['type'])

            if field['primary']:
                # It is a primary key.
                ret += ' PRIMARY KEY'
            ret += ',\n'

            if 'reference' in field.keys():
                # It references another key in another table, add it to the
                # list.
                foreign += ' FOREIGN KEY ({0}) REFERENCES {1}({2})'.format(
                    field['name'], field['reference']['table'],
                    field['reference']['field'])
                foreign += ',\n'

        # Add all foreign keys to the end.
        ret += foreign
        return ret[0:-2] + '\n);'


class ForeignKey:
    """
    Parses a foreign key from PlantUML file.
    """

    def __init__(self):
        """
        Constructor.
        """
        # Tables and fields for the foreign key.
        self.source_table = None
        self.source_field = None
        self.target_table = None
        self.target_field = None

    def parse(self, lines):
        """
        Parse a foreign key relationship.

        :param lines: The remaining lines of the PUML file.
        """
        # Tokenise by words.
        tokens = re.findall(r'[\w]+', lines[0])
        # Get the direction of the reference.
        dir = re.findall(r'<|>', lines[0])

        # Set the tables and fields according to the direction.
        if (dir[0] == '>'):
            self.source_table = tokens[0]
            self.source_field = tokens[1]
            self.target_table = tokens[3]
            self.target_field = tokens[2]
        elif (dir[0] == '<'):
            self.source_table = tokens[3]
            self.source_field = tokens[2]
```

```
            self.target_table = tokens[0]
            self.target_field = tokens[1]
        else:
            print('Error parsing foreign key: {}'.format(lines[0]))


def lineNormalise(line):
    """
    Utility function convert string to a known format by:

     * Stripping any white spaces at the beginning.
     * Converting to lower case.

    :param line: The string to process.
    :return: The processed string.
    """
    # Strip initial white spaces and lower case the string.
    return (line.lstrip().lower())


def isTable(line):
    """
    Tell if a PlantUML table definition is starting at this line.

    :param line: The line to check.
    :return: True if there is a table definition is starting at this line.
    """
    # Make the string easier to parse.
    line_stripped = lineNormalise(line)

    # Return value.
    ret = False

    # If the line starts with the word table, we have a table definition!
    if line_stripped.startswith('table'):
        ret = True

    # Tell the horrible truth that this code could not find a table.
    return ret


def isForeignKey(line):
    """
    Tell if a PlantUML foreign key definition is at this line.

    :param line: The line to check.
    :return: True if there is a foreign key definition is at this line.
    """
    # Make the string easier to parse.
    line_stripped = lineNormalise(line)

    # Return value.
    ret = False

    # Split into tokens by space.
    tokens = line.split(' ')

    # Parse the tokens
    for token in tokens:
```

```
        # Match direction tokens, these should only appear in foreign key
        # lines.
        if ('<' in token) or ('>' in token):
            ret = True
            # Got it, get out.
            break

    return ret


class PUMLReader:
    """
    Class to read, parse, and convert tables from a PlantUML file into SQL
    commands to create them in the database.
    """
    keywords = (
        '@startuml', 'skinparam', 'scale', '!', 'hide methods',
        'hide stereotypes',
        'sprite', '@enduml')

    def __init__(self):
        """
        Constructor.
        """
        # All tables en up here.
        self.tables = {}

    def parse(self, lines):
        """
        Parse all lines of a PlantUML file.

        :param lines: The lines in the PlantUML file.
        """
        # Keep count of the current line number.
        i = 0

        # Array of foreign keys.
        fks = []

        # Loop through all lines.
        for i in range(0, len(lines)):
            # Used to OK parsing of the line.
            skip = False

            # Look for keywords at the beginning of the line.
            for keyword in PUMLReader.keywords:
                if lines[i].startswith(keyword):
                    # Found one, do not parse.
                    skip = True

            # Only parse lines that has no keywords.
            if not skip:
                if isTable(lines[i]):
                    # There was a table at that line, parse it.
                    table = Table()
                    table.parse(lines[i:])
                    # Add it.
                    self.tables[table.name] = table
                elif isForeignKey(lines[i]):
```

```
                    # There was a foreign key at that line, parse it.
                    fk = ForeignKey()
                    fk.parse(lines[i:])
                    # Add it
                    fks.append(fk)

        # Add all foreign keys to the tables.
        for name, table in self.tables.items():
            for fk in fks:
                # Find the right table to add the foreign key.
                if fk.source_table == name:
                    table.fields[fk.source_field]['reference'] = {
                        'table': fk.target_table, 'field': fk.target_field}

    def sql(self):
        """
        Return the SQL command to create the tables.

        :return: SQL command string.
        """
        # Return value.
        ret = ''

        # Variables for figuring out dependencies between tables.
        done = []
        dependencies = {}

        # Run through all tables.
        for table in self.tables.values():
            # Assume no references.
            reference = False

            # Check fields for references.
            for field in table.fields.values():
                if 'reference' in field.keys():
                    # Add the reference to the dependencies of the table.
                    if table.name not in dependencies.keys():
                        dependencies[table.name] = []
                    dependencies[table.name].append(
                        field['reference']['table'])
                    reference = True

            # If the table has no dependencies, just print it.
            if not reference:
                ret += '\n' + table.sql()
                done.append(table.name)

        # Solve dependencies.
        while (len(dependencies) > 0):
            # Run through all dependencies.
            for table, deplist in dependencies.items():
                # Check is some has been solved since the last run.
                for solved in done:
                    if solved in deplist:
                        # Bingo. Remove it.
                        deplist.remove(solved)
                # If there are no more dependencies
                if len(deplist) == 0:
                    # Add the SQL to the return value,
```

```
                    ret += '\n' + self.tables[table].sql()
                    # Add the table name to the solved list.
                    done.append(table)

            # Remove all tables that have had its dependencies solved.
            for solved in done:
                if solved in dependencies.keys():
                    del dependencies[solved]

        return ret


def parse_commandline():
    """
    Parse command line arguments.

    :return: A tuple with a list of files path to include, and a list of paths
             to exclude.
    """
    # Set up the arguments.
    parser = argparse.ArgumentParser(description='Count lines of Python ' +
                                                 'code in given paths.')
    parser.add_argument('infile', type=argparse.FileType('r'),
                        help='PlantUML file to read the database structure
from.')

    # Parse command line
    args = parser.parse_args()

    # Return the paths.
    return ((args.infile))


def main():
    """
    Program main entry point.
    """
    # Print welcome message
    print(
        'dbdia2sql.py v{} by Martin B. K. Grønholdt\n'.format(__VERSION__))
    # Parse the command line.
    puml_file = parse_commandline()

    # Instantiate the PUMLReader class and parse the file given on the command
    # line.
    reader = PUMLReader()
    reader.parse(puml_file.readlines())

    # Print the SQL.
    print(reader.sql())


# Run this when invoked directly
if __name__ == '__main__':
    main()
```

5.2. Result

Running the above program on the PlantUML database diagram file from this hand in gives the following output:

```
dbdia2sql.py v1.0.0 by Martin B. K. Grønholdt


CREATE TABLE countryTable(
idCountry INTEGER PRIMARY KEY,
country TEXT
);
CREATE TABLE productTable(
product TEXT,
idProd INTEGER PRIMARY KEY
);
CREATE TABLE cityTable(
city TEXT,
idCity INTEGER PRIMARY KEY,
country TEXT,
FOREIGN KEY(country) REFERENCES countryTable(idCountry)
);
CREATE TABLE customerTable(
idCust INTEGER PRIMARY KEY,
address TEXT,
email TEXT,
name TEXT,
city TEXT,
FOREIGN KEY(city) REFERENCES cityTable(idCity)
);
CREATE TABLE orderTable(
custId INTEGER,
idOrder INTEGER PRIMARY KEY,
date DATE,
FOREIGN KEY(custId) REFERENCES customerTable(idCust)
);
CREATE TABLE orderProductTable(
orderId INTEGER PRIMARY KEY,
productId INTEGER,
FOREIGN KEY(orderId) REFERENCES orderTable(idOrder),
FOREIGN KEY(productId) REFERENCES productTable(idProd)
);
```

The output of the PlantUML database diagram to SQL converter.

This method was successfully used to generate the SQL to create the tables for this assignment.