CIT 483/583 Lab – Sqlite and Active Record

Fall 2018

**Instructions**

Open and save this file in any MS word-compatible format as Lab09\_*Firstname*\_*Lastname*.<ext> and place your answers in that document. Do a **Save-As** and retain all of my content. Keep the document safe in case your submission fails, or you discover an error prior to the due date and wish to re-submit. Submit your document to the Lab09 dropbox in Canvas. The due date and any other pertinent information are noted in the Canvas item.

This lab should be completed on students.cs.nku.edu as a reference implementation and as a means of testing your answers.

**Place the answers in or immediately following each question and make sure your answers stand out from the questions by using a different font color.**

You may find it helpful to have two terminals open, both connecting to students.cs.nku.edu—one for running queries against your sqlite3 DB and one for running Ruby commands in IRB that use Active Record)

1. Copy the file authors.sql from my home directory to a location under your home directory. Create the DB authors.db by running the following command:

$ sqlite3 authors.db < authors.sql # should show an error at the end

Despite the error, the DB will have been created. First, check to see that the file exists in the directory from which you invoked the command.

$ ls –l authors.db

NOTE: It's worth understanding the source of the error. View the file authors.sql using vi, less, more, or any other program you like for reading files. Notice that AUTHORS\_BOOKS has a foreign key relationship to both AUTHORS and BOOKS:

CREATE TABLE authors\_books(

id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

author\_id INTEGER,

book\_id INTEGER,

**FOREIGN KEY(author\_id) REFERENCES authors(id),**

**FOREIGN KEY(book\_id) REFERENCES books(id)**

) ;

To enable enforcement of those relationships, we have to tell Sqlite to do so:

-- enable foreign key constraint

PRAGMA foreign\_keys = ON;

The insert statement that references a non-existent ID 99 causes the error, and it fails to create the row in AUTHORS\_BOOKS.

INSERT INTO authors\_books (author\_id, book\_id)

VALUES (99,1);

If we had not enabled that functionality, e.g., if we commented out the PRAGMA line with --, and then removed authors.db and recreated it, you would see that the row would exist, even though it is logically incorrect. You do not have to prove it in this lab, but you do have to notice that Sqlite doesn't automatically enforce foreign key relationships.

1. Now we can open the DB using sqlite3 and do some basic commands to see what it contains. The sqlite> prompt indicates that you are connected to the DB. Any of three commands will disconnect and exit the program: .exit .or .quit or Ctrl-D

$ sqlite3 authors.db

sqlite> .help # capture the last 4-5 lines only

sqlite> .schema

sqlite> .tables

sqlite> .exit

1. As well as showing metadata about the DB, sqlite3 allows standard SQL commands to be run.

$ sqlite3 authors.db

sqlite> .schema AUTHORS

sqlite> select \* from AUTHORS;

sqlite> select \* from AUTHORS where last\_name = 'Blake';

sqlite> select \* from AUTHORS where last\_name like 'D%';

sqlite> select \* from AUTHORS where last\_name like 'B%';

sqlite> select \* from AUTHORS where id = 1;

1. Let's focus on the mapping table Authors\_Books. First study the data a bit so we can recreate the records later. In fact, we'll drop the table so we can recreate it using Active Record later, so be sure to capture the table contents.

sqlite> select \* from AUTHORS\_BOOKS;

sqlite> drop table authors\_books;

sqlite> .schema

1. The first step in creating or modifying a database with Active Record is connecting. Open a second puTTY or Mac terminal session on students.cs.nku.edu so you can interact with the database using SQLite in the first terminal session while you are altering it in the second session using IRB.

>> denotes the IRB prompt as usual.

>> require 'active\_record'

require 'sqlite3'

>> ActiveRecord::Base.establish\_connection(adapter: 'sqlite3', database: 'authors.db')

1. We know we can create tables with Active Record, but the normal rules still apply. If a table already exists, you can't create it. You may truncate the error output to the first few lines.

>> ActiveRecord::Schema.define do

create\_table :authors do |t|

t.string :first\_name, :null => false

t.string :last\_name, :null => false

end

end

1. We can recreate the table we dropped earlier, though.

>> ActiveRecord::Schema.define do

create\_table :authors\_books do |t|

t.integer :author\_id, :null => false

t.integer :book\_id, :null => false

end

end

We can also try to add the foreign key relationships that existed in our original definition.

ActiveRecord::Schema.define do

add\_foreign\_key :authors\_books, :authors

end

ActiveRecord::Schema.define do

add\_foreign\_key :authors\_books, :books

end

1. Go back to the first terminal using sqlite3 and check that the table now exists. Notice, however, that the foreign key relationship is not reflected in the table description.

sqlite> .schema Authors\_Books

1. Even with the foreign keys defined, and the PRAGMA to enable the functionality, we find that Active Record and SQlite don't seem to enforce the schema constraints. First, we need an object to be the "object" to represent a single row. In the IRB session:

>> class Author\_Book < ActiveRecord::Base

self.table\_name = 'Authors\_Books' # overrides default

end

Next, we create an object of that class and commit to the DB using the .save method inherited from ActiveRecord::Base

>> r = Author\_Book.new( :author\_id => 1, :book\_id => 1)

>> r.save

Notice that we never had to define the fields author\_id and book\_id, or even the primary key id. It was all derived from the table definition. We can verify the record was created using SQL in the SQLite terminal, or in IRB using ActiveRecord. There are a few methods to extract those fields, including auto-generated attr\_reader methods for each field.

>> Author\_Book.find(1).attributes

>> Author\_Book.find(1).id

>> Author\_Book.find(1).author\_id

>> Author\_Book.find(1).book\_id

1. Test the foreign key handling. There is no book with id = 999, so it should fail. But it doesn't.

>> r = Author\_Book.new( :author\_id => 1, :book\_id => 999)

>> r.save

>> Author\_Book.all.each { |a| puts a.attributes }

1. As it turns out, we can enforce the foreign key relationship in the definition of the Author\_Book class.

First, though we need classes to represent Author and Book entities:

>> class Author < ActiveRecord::Base

has\_and\_belongs\_to\_many :books

end

>> class Book < ActiveRecord::Base

has\_and\_belongs\_to\_many :authors

end

>> class Author\_Book < ActiveRecord::Base

self.table\_name = 'Authors\_Books'

belongs\_to :author

validates :author, presence: true

belongs\_to :book

validates :book, presence: true

end

>> r = Author\_Book.new( :author\_id => 1, :book\_id => 998)

>> r.save

1. Go back to the SQLite session and query AUTHORS\_BOOKS to see that the first "bad" record was created, but the second was not. Let's get rid of the bad record while we are at it.

sqlite> select \* from authors\_books;

sqlite> delete from authors\_books where id = 2 ;

sqlite> select \* from authors\_books;

1. For the last step, see if you can use ActiveRecord to add AUTHORS\_BOOKS records so that each AUTHOR is paired with at least one BOOK. [Use the output of the select \* from AUTHORS\_BOOKS statement in the SQLite section above.] Capture the IRB commands and the output.