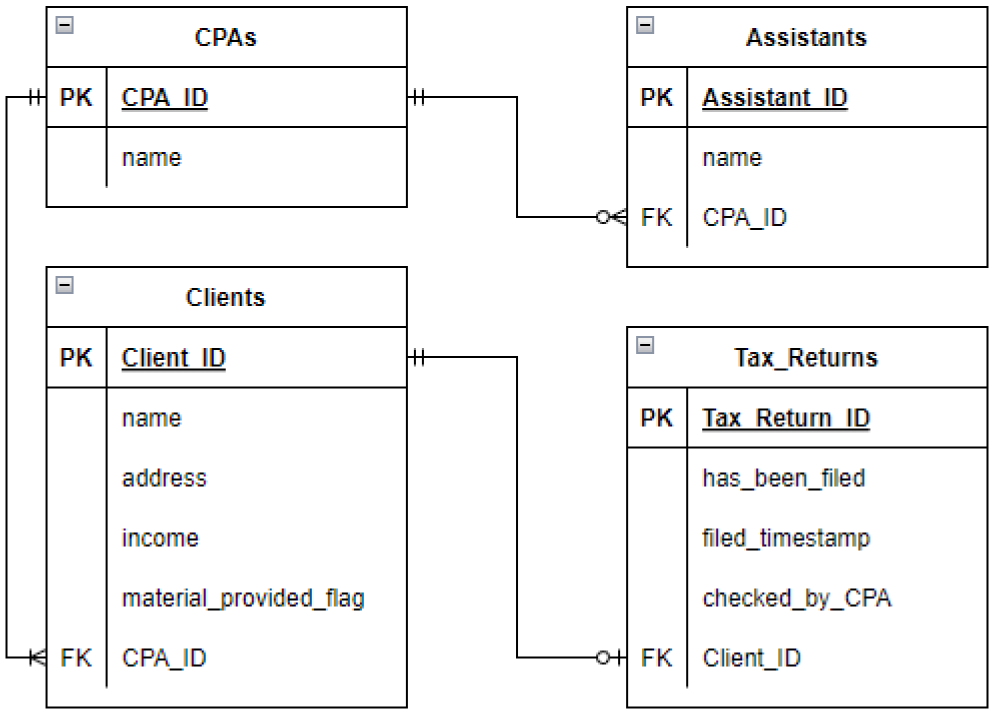
James McKoin

INSC 484: Database Applications

Homework 3

Database Design:



I chose this design for my client/tax return system because it successfully accomplishes the goals of separating concerns and being fully normalized (well 3NF). Additionally, I wanted the system to be CPA centric. This design demands that a CPA be entered into the database first before clients, assistants, or tax return data can be stored. This design also allows for the potential of determining how many clients each CPA has, allows client information to exist in the database without a tax return on file, and allows CPAs to exist without having an assistant. Conversely, clients cannot exist in the database without a CPA data, assistants are also dependent on the existence CPAs in the database, and tax returns can not be filed without any client information.

Software Design:

prompt_create_cpa():

This function is intended to prompt the user to add a CPA into the database and is the only menu option that will successfully run with an empty database. It asks the user for the CPA name and passes that value into the CPA object call (where it is stored by a cpa variable in the main). Once the CPA name is passed to the CPA object is initialized with the name provided and an id of None. Back in the man the CPA object method 'save' is called. This is where a connection to the database is created the, the user provided name data is passed to the database.add_cpa function. Within the database module the add_cpa function then uses the cursor.execute method to pass in the string data insert into returning id SQL statement as well as the name data. Once the name data is inserted into the database an incrementing id value is created and returned back to the add_cpa function. This value is caught by the cpa_id variable and returned back to the CPA object. The cpa_id value that was returned back to the CPA object is caught by the new_cpa_id variable and assigned as the self.id, after the connection is closed.

This process of creating an object that is initialized by user input and None as the id, until the save method is called to allow the database to autoincrement and return the assigned value, where it is then stored as the self.id, occurs with all the objects created in my python code and inserted into the elephant SQL database. Within every object I also overwrite the __repr__ method to provide descriptive feedback to the user. The general approach was to detail all attribute information of the object when called in the main (then printed to the user).

prompt_create_assistant:

This function behaves vary similarly to the create CPA function, but I implemented some try/except blocks to catch the valueerror that occurs when a user fails to input an integer for the CPA id. I also implement a try/except block in the save method for the client object, where the try block tries to pass the user information to the database and returns a 1 if successful and catches the foreign key violation that occurs if the CPA id does not exist in the database and returns None. I have some if/else logic in the main dealing with the return values, by informing the user that the CPA doesn't exist or printing the client information if it is successful inserted into the database.

I have a nested if/elif/else statement dealing with whether the user wants to see a list of available CPAs. If the user enters 'y' the list_avail_cpas function is called. This a short function that triggers a for loop to increment through the CPA object's all method. The all method calls the database.get_cpas function that performs a select all SQL statement and returns a fetchall back to the main. In the main the CPA id and CPA names are listed out.

prompt_create_client:

The client creation is performed almost identically to the creating the assistant object. Where foreign key error is caught, and the user is prompted with the option to view all the CPAs in the database. I did create a set_material_flag function to deal with the materials provided flag, just to keep the create clint function looking cleaner. The set_material_flag function just ask for y/n input and returns T/F.

prompt_mark_client_materials:

In this function I implement try/except blocks to ask the user for client id. The except block deals with if the client id doesn't exist in the database or if the user fails to input an integer. The user input id is passed to the Client.get().mark() method. The first method call get returns the client information form the database that corresponds to the id provided, then the second method call mark, in short, calls a database function that updates the material_provided_flag attribute to True. If successful, the client information is printed out and a notice to the user that the client information was updated.

prompt_check_client_materials:

This function behave similarly to mark_client_materials, but once the object information is pulled from the database, a if/else statement is used to determine whether or not the tax material has been provided. The Boolean data type of this attribute makes the comparison relatively simple.

prompt_file_tax_return:

This function is where the try/except marathon continues. I implement a nested try/except block within my original try block. The outermost try/except is just to catch the valueerror that prompts if a user fails to input an integer for the client id. The nested try/except block used to determine if a tax return belonging to that client id already exists in the database. Only one tax return per client. The interior try block calls the get_by_client TaxReturn method and informs the user that a tax return for this client already exist if successful. The interior except block catches the typeerror that prompts when the user provide client id does not correspond a tax return in the database and creates a new tax return instance with the client id provided and a timestamp of the time that it was filed. There is a nested function call determine_cpa_or_assistant. This asked the user if they are a CPA and returns a true or false value to the check_by_cpa variable that is used when initializing the TaxReturn object.

prompt_check_client_return:

I implement try/except blocks for the first time to check if a client has a tax return based on the client id provided by the user. The try block runs the get_by_client method and informs the user that a tax return has been filed for the client if successful. There is a except block to catch the valueerror if an integer isn't input, and to catch the typeerror if there isn't a tax return belonging to a client with the id provide. Within the valueerror except block I implement another try/except block to inform the user if the client id they provided exist in the client table of the database. I did this to contextualize whether a tax return doesn't exist because there hasn't been one filed for an existing client or that there is no client with that id in the database.

prompt_mark_cpa_checked:

Here I implement a try block to ask the user for the tax return id. The TaxReturn get method is called to return the tax return information with the corresponding id, then the mark method is called to update the checked_by_cpa value to True. If invalid input is provided or if the tax return id doesn't exist, the error is caught and printed out to the user.

prompt_check_cpa_checked:

The function performs similarly to the mark_cpa_checked function. The try block asks for tax return id, then the get method is called where tax return information from the database is used to construct the TaxReturn object. Once created, the attribute information is used to determine if the tax return has been check by a CPA. Valueerrors are caught and printed to the user. Typeerrors inform the user that no tax return with the id provided exist in the database.