Mini Project 1 by Derek Shultz and Mingjun Zhou

**(a)** Our system uses a command-line interface that the user interacts with to search for and book flights, as well as see their booked flights and cancel bookings.

The system will first prompt the user to login to the database. If their login username is the same as the one in brackets next to the word “Username”, they can press enter. If not, they can enter their username. Then, they enter their password in next to “Password:” and press enter. They will then be presented with a menu with the options “Login”, “Register”, and “Exit”. They simply enter the number next to the menu option they want to access that option (ex. Enter 1 and press enter to login). Login allows them to log in with an existing username and password, which they enter when prompted. Register allows them to enter a new username and password to access the system with. When done entering the username and password, they are logged in as that user. Exit will quit the program. When logged in, the user will have another menu of options, consisting of “Search For and Book Flights”, “List or Cancel Existing Flights”, “Logout”, and “Search for Round Trip Flights”. If they have logged in as an airline agent, they will also see “Record a Flight Departure” and “Record a Flight Arrival”. They then enter the number next to the option they would like to use. “Search for and Book Flights” will prompt them for a source and destination, for which they can enter airport codes or airport city names. Then it will ask for departure date, which they enter in the form of DD/MM/YYYY. Then, all flights with matching specifications are listed, along with the options “sort by number of connections”, “make a booking”, and “go back to menu”. “Sort by number of connections” sorts the flights by the number of connections. “make a booking” will put the system in booking mode, where they enter the number next to one of the flights to book it. “Go back to menu” goes back to menu. “List or Cancel Existing Flights” will list all the flights that the user has bookings for. If they enter the number next to one of the flights, they will be presented with all information about that flight and put the user back at the menu. They can also hit the number next to “cancel one of your flights” to go into cancel mode. If they enter the number next to one of the flights listed before, they will cancel their booking and ticket for that flight. Or they can go back to the menu. “Logout” will bring them back to the first login menu. “Search for Round Trip Flights” works the same way as “Search for and Book Flights”, but also include flightno3 and flightno4 and fare3 and fare4, which represent the flightno(s) and fare(s) for the return flight(s). “Record a flight departure” will prompt the user for the flight number and departure date, followed by the departure date and time (they must provide the date because when late-night flights are delayed, they could be pushed into the next day). “Record a flight arrival” works the same way, but it asks for the date and time of a flight’s arrival.

**(b)** Our software is divided among 5 files. Following, I will briefly describe the function of each file and how the functions work, focusing on those that deliver the major functions

1. **main.py** – handles all of the functions that are not directly related to the major functionalities of the code:

*conToDB*: Gets the user’s username and password in order to connect to the database. It keeps the password from being typed in plaintext by using a library called getpass.

*sqlWithReturnDesc*: Connects to the oracle database, runs an input sql command, closes the oracle connection, and returns the output of the query, as well as the description of the output of the query

*sqlWithReturn*: Connects to the oracle database, runs an input sql command, closes the oracle connection and returns the output of the query

*sqlWithNoReturn*: Connects to the database, runs an input sql command, closes the connection and doesn’t return anything.

*init*: Displays the login menu to the user and gives them the options to login, register, or exit. Tells them if they enter an incorrect input.

*logIn*: Takes user input for email and password (using getpass again) and checks if the entered username and password are present in the users table. If so, move onto the menu. If not, tell the user that the login failed.

*register*: Prompt the user to input a new email and password and insert the given values into the database. Doesn’t work if the email already exists or if the email or password is null.

*menu*: Tries to create the views available\_flights and good\_connections if they do not already exist. Gives the user a menu of options to select from, plus the options to record a flight’s departure or arrival if the user is in the airline\_agents table. Then checks the user’s input for validity and, if valid, sends it to the correct function.

If main.py is called by python3, the program automatically runs conToDB to start the program.

1. **search.py**- all the functions that have anything to do with searching for a one-way flight and booking one of those flights:

*printInfo*: takes user inputs to find suitable flights. It does this using the union of slightly modified versions of the available\_flights and good\_connections views specified in assignment 2. It has separate queries to check for if the entered src and dst are airport codes or parts of city/airport names. There is also another query for sorting by stops and price. Displays the flights information and provides the user with the options of sorting the results by the number of connections, go into booking mode, or go to the menu. Different queries will be called depending on the input. If the user chooses to make a booking, it prompts them to put in the number next to the flight they’d like to book. But, since the flight they want could have filled up before booking, it compares the current rows to the selected row from the original rows presented. It tells the user if their selected flight has filled up.

*booking*: When the user chooses a flight to book from the search menu, this function is called. If they are not yet in the passengers table, they are prompted to enter their name and country, which are entered into the passenger table with their email. It generates a new ticket number by finding the max ticket number and adding one. Then it inserts the booking into the tickets table once and bookings table possibly twice if there are 2 flights. Tells the user if booking successful or not.

*search*: The first function accessed in search mode that gets input from the user.

1. **existing.py**- all the functions that have to do with listing the existing bookings and cancelling them:

*existing*: Uses an sql command to find all the bookings in the booking table that belong to the current user of the system by checking if the email in the passengers table is equal to the current users table and finding tickets that have that passenger name, and bookings with those ticket’s ticket numbers and flight\_fares fares with those bookings’ fares and the flightnos of those. If that sql command works and returns a non-zero number of bookings, it will print them. It then prints the users options and prompts them for input. If the user selects the number of one of the rows, it will move to the more\_info function, otherwise it will move to cancel mode or the menu, depending on the user’s input.

*more\_info*: Find all the info about the booking the user selected from all of the tables that contain info that the user would be interested in. Returns to the main menu.

*cancel*: Prompts the user to enter the number of the row they’d like to cancel. If valid input, removes rows from bookings and tickets that contain the ticket number of the booking that’s being cancelled.

1. **roundtrip.py**- all functions that have to do with listing and booking roundtrip flights:

*roundTrip*: Gets input data from user and moves onto printRoundInfo.

*printRoundInfo*: Works almost the same as printInfo in search.py, but the queries in this one are unioned with similar queries that are different mainly because the src and destination are reversed, finding the flights that are going in the opposite direction. It calls the return flight(s) flightno3 and flightno4.

*roundBooking*: Works pretty much the same as booking from search.py. But in this one, it generates a new ticket number and inserts it into the tickets table and into the bookings table possibly 4 times, if there are 4 flights, but at least 2 because a round trip must have at least 2 flights.

1. **agents.py**- all functions that have to do with recording actual arrival and departure times:

*recordDepart*: Gets flightno and dep\_date from user, making sure neither is null, then checks if a flight of that number exists on that departure date in table sch\_flights. Then, the user must input the actual date and time of its departure (a delayed flight could be delayed until the next day, so we wanted to make sure the date is right. That value is then inserted into act\_dep\_time in the sch\_flights table and the user is told if it was successfully updated.

*recordArr*: Does the exact same thing as recordDepart in this file, but updates act\_arr\_time instead of act\_dep\_time.

**(c)** Our testing strategy largely involved creating data to work with and then seeing if our program interacted with the data in an expected way. Most of the errors and testing were caused by sql query errors. We had some strange unexpected errors due to our occasional misuse of python syntax. For example, we were initially importing our other files with “from <filename> import \*”, which caused our program to tell us that certain functions did not exist in files that they did, in fact exist in. So, we changed to “import <filename>” statements and called functions from that file with “<filename>.<function>”, which solved that error. Another strange behavior was the program jumping back to previously accessed functions when we tried to exit the program. This was caused by our forgetting to return functions when we call the next function to use, causing the rest of that function beyond that function call to happen when the system tries to exit. That was easily remedied.

To test sql queries, we modified some of the student-provided test data from assignment 2 and added a few extra lines into some tables. Once we had set up the tables and populated them with data, we went into sql and tested queries until we found one that provided consistent, desirable results. We’d then copy that from splplus into our program’s code.

Once our program was functioning reasonably well, we tested it with unusual inputs (null inputs, letters where numbers should be, etc.) and fixed it any time it broke. One case that took a bit of testing before we got it right was checking to see if a flight had filled up before a booking had been made. For that, we had to have more than one user log in, put one in booking mode and have the other book all the seats in the desired flight. Then, we had to try to book the flight. The bug experienced here was that when a flight has all of its seats booked, it disappears from the available\_flights view, shifting all the rows below up to take it’s place, and so the user selected index could inadvertently book a seat on the incorrect flight. This was solved by recording the rows when they are first listed and comparing these to the present rows right as its being booked to see if the flight is still in the table and if it is, to book it and not another.

Once we were satisfied that all of our functions could take most typical user inputs, we looked at our test data and predicted what doing each of the tasks in marking rubric would do to it. We then executed each of the functionalities listed in the provided marking rubric and checked to see if the displayed values were as expected. Finally, once all the tasks in the marking rubric were successfully completed, we exited the program and went back into sqlplus to check if the data was affected as expected. After all of our previous testing of our sql and python, this part didn’t have a huge number of errors.

**(d)** Mingjun started the project before we coordinated when we were going to meet. He spent about 2 to 3 hours implementing the connection to the database, completing the login screen and the login menu.

We met 3 times when we were both available to work on the project. In the first meeting, Mingjun spent 3 hours working on the search and booking functions, implementing a lot of their basic functionality, while Derek spent 3 hours working on the listing of the existing flights, completing the functionality without testing. The next meeting, Mingjun spent 4 and a half hours finishing the implementation and starting to debug the search and booking functions while Derek spent 4 and a half hours completing the testing the listing of the existing flights function and starting to implement the basic functionality of the cancel function. Outside of these first 2 meetings, Mingjun spent about 2 hours working on and finishing his search and booking functions. Derek spent about an hour and a half finishing and testing his canceling function.

Derek got sick over the weekend before the project was due, so they could not meet one of the times they had meant to. During this period, Mingjun spent between 7 and 8 hours implementing the functionalities of airline agents and round trip functions. Derek spent 3 and a half hours completing the testing of airline agents and various functions in search.py, existing.py, and main.py, as well as making some of the output and code more consistent.

Our final meeting consisted of 5 hours of testing all of our program, with Mingjun heavily editing the search and booking functions for both round trips and regular flights so that they were totally complete. Derek, once done his portion of the testing and improving the comments in airline\_agents.py, existing.py, and half of the ones in main.py, started the documentation. After the meeting, Derek spent 5 hours finishing and revising the documentation and Mingjun spent an hour finishing the commenting on the remaining files.

To keep both of us up-to-date with what the other was doing and, therefore, what still needed to be done, we used a git repository on github.com. At the start, we decided on a general division of labor that constantly adapted, based on the progress each made. We communicated before we started working on any part of the project to make sure the other wasn’t already working on that part and so that they knew what was being taken care of.