**Physical Design Discussion**

*Denormalisations*

Denormalisations for this project seemed most relevant to the required queries we are supposed to do. However, for general usage - add, update, view, delete, and essentially everything do-able by navigating the menus of the program, the normalisations in place are effective. I tried to make sure that to get relevant data for most menu screens, the SQL connector does not have to select from multiple tables. Of course, for some of these transactions, notably the requirement-answer and criteria-evaluation transactions, in order to display both the list of questions and answers along side each other, the query had to access both tables. However, I decided to keep these tables normalised, as if you count the number of operations performable on each of the tables individually (editting, adding, ect.), it obviously outnumbers that of the operations performed when they are together (only displaying information alongside each other). For the queries, many of the queries will need access to multiple tables to accomplish their task. As we don’t really have that much data, so the queries will be fast, and I ultimately decided that the update and maintenance of data is more important than the queries, I decided not to denormalise anything even for the queries. That, and also the reason of I didn’t want to mess up my tables any more.

*Secondary Indexing*

**Implementation Discussion**

*Language*

As required, I used Python to implement the UI and database access features. According to my console, my version of Python is **3.6.3**. I know for sure the program won’t work on Python 2, but don’t know about anything between 3.0 and 3.6, or higher. If something seems to be going wrong, please try running the program using Python 3.6.3. According to the command “pip freeze”, the sql module I have installed is “mysql-connector==2.1.6”. I am not sure if this is the one we are supposed to be using, but to use it I followed the documentation given on the mysql website given by our teacher, and so it should be the same. If things seem to be going wrong (for example, python says the module is missing), please try “pip install mysql-connector==2.1.6” to install the module.

*Menuing System*

As we are allowed to make the worst UI ever, I decided to go with a command line UI with as much multiple choice as possible. Although I did try to make general menu functions at the beginning, there were too many times where the menu varied, so I couldn’t use it for all of my menus. Now, if I tried to make the program again, I probably would be able to make the code of the menuing more general, as I know many of the types of menus that are going to be used. As a result of this bad menuing, the main program seems quite long, lines of code wise.

*SQL Access*

I was rather confused at what SQL we are supposed to be using, and so in the end I got the mysql python module working and it seems to be able to communicate with a mysql server, which was downloaded from the same mysql website given to us by the teacher.

*SQL Database Access Code*

I decided to try to keep all usage of this mysql module (or to be precise, the “mysql.connector” module) in one file, so things don’t get too confusing. This file is the j5sql.py file, which creates an object I called “jupiter”, representing the jupiter database which it connects to. The program maintains a connection until the program is closed. In this file I created various functions that would perform specific SQL queries to the database, functions which would be used in the main program. I also created general functions for adding, updating, and deleting from the database, which helped me greatly through designing the program. If I did this project over, I would definitely think about trying to make more of these, but at the time I could not be certain of what I needed, and so it was hard to think up more general functions to make.

*SQL Error Handling*

I try to catch errors whenever a sql cursor call is made in the SQL python file. The error is printed, the transaction doesn’t occur and the program doesn’t crash.

*Input Validation*

I have probably written or will write a lot about this. Although I tried to put some validation in at the beginning, eventually it got pretty annoying and since we are allowed to make terrible Uis, I just stopped doing it. Although no data will be lost once an operation is complete (hopefully), please be careful as the program may completely crash given incorrect and unexpected input. You may be thinking of testing the program by giving it data that is correct but doesn’t make sense logically; you might wonder, will this crash the program? Well, as stated earlier, I tried to make the system as multiple choice as possible – this prevents the user from doing such a thing. For example, you may want to try to give an application a degree program that doesn’t exist. This is not possible, as upon creating an application you are given a multiple choice of degree program to add. The only way to break it is to enter a number that is not within range of the multiple choice – which I count as invalid input, and is thereby not supported.

*Constants*

Or, to be more precise, the lack of. At the beginning, I had thought of storing away important arrays of strings that might end up appearing multiple times or would have to be used multiple times. For example, my general SQL add, update and delete functions require that the primary keys of the tables being modified be given to them. I was planning on saving the primary keys away somewhere and by just giving some function the name of the table, would be able to get these keys. But as I didn’t really know where to start using that, I ended up not using it. So as you will see if you read my code, there are many arrays of strings that are initialised when they are used, and since menus loop they are initialised again and again. This may be seen as an inefficiency.

*Repeated Querying*

An inefficiency I ended up having is repeated querying while an item is selected. As will be discussed later, a user must select an item in the ui of the program by navigating menus before they can choose actions to act on that selected object. Information is displayed about the object everytime the user completes an action and returns to the menu with the selected object. Many times in my code, I will re-retrieve that object from the database to make sure that any updates made to it are correctly displayed. What I could have done was made the update functions update the item in memory, so I don’t have to re-query the database for the item again. Since we didn’t have any requirements or limits database queries, I didn’t spend much effort to correct this. But I believe it is worth mentioning this is an optimisation that could have been made.

*Null Dates*

**User Interface**

*Initial Screen*

After thinking about the project for a while, I decided that the actions the user should be able to perform falls into three categories, as seen above. A user will either be modifying details about students, or details about the degree programs which will be necessary for student information to be entered. Therefore it made sense to have this initial seperator up front.

*Student Mode*

A theme you will see through out the program is the ability to “select” items. Furthur action can be performed once an item has been selected, and this helps to keep [some] menus a little shorter and cleaner. On this screen the user will immediately see the list of students – in a real world scenario this would probably have to be changed, as if there are many students it will be quite annoying. Lucklily, we are allowed to make the UI as bad as we want. To select a student, the user must select the select option and then type the number next to the desired student as it appeared on the menu. This process, where there are numbers next to items that the user must select, is a recurring element in the program. Please note: when it prompts you to select a user, the program will crash if the input is invalid (for example, not an integer, not one of the choices). There were no requirements on input validation. Although there are some places with validation, there are many more where there is none.

*User Add and Add UI in general*

Adding a student is straightforward, it involves the filling in of some fields. Unfortuately, you cannot cancel an add operation once it has begun (short of closing the program), but if you see it through to the end, you can delete the student afterward. Noticably, for this add operation and many other add operations, there are some required fields that may not be prompted. This is because those fields are multi-valued (can have mulitple inputs) and so are addable by the user later, after they have selected the item they just created.

*Selected User Mode*

After selecting a student the general information of the user is displayed. From here the user can add, edit, and phone numbers of the student, edit the information of the student, and view the student’s applications.

*Field Edit Modes In General*

Editing a field prompts the user to pick a field, then enter a new value. This operation is reused for mostly all other edit operations for other tables, as well. Like the add operation, it unfortunately cannot be cancelled.

*Application Select*

The user can create or select applications. The application procress is quite simple as most fields are entered post-creation, when the user has selected the application.

*Selected Application Mode*

Upon selecting an application, the user is able to access the required data of the application object.