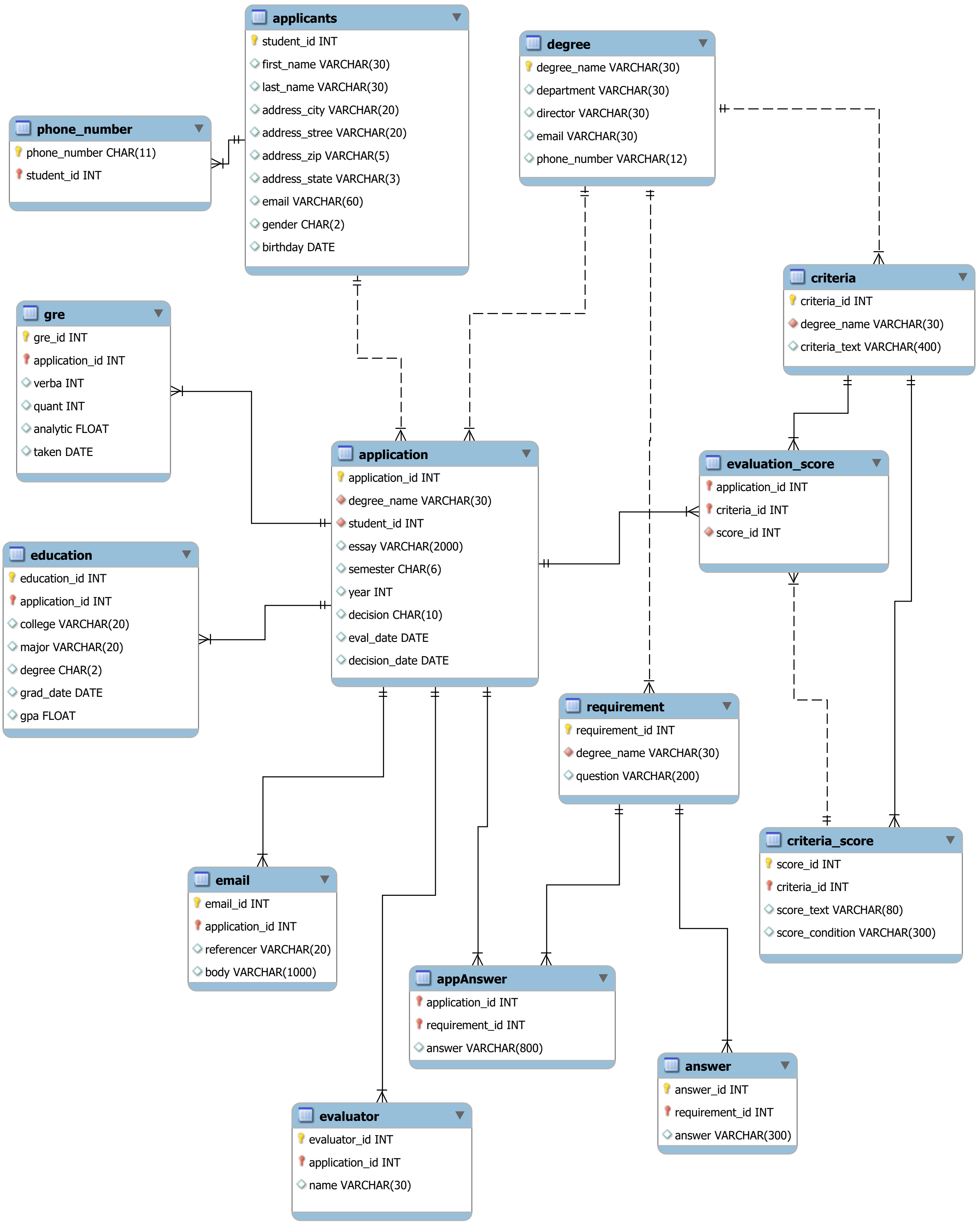
CMSC 461 Databases Final Project Phases C, D, E, F

Khang Ngo
khang4@umbc.edu

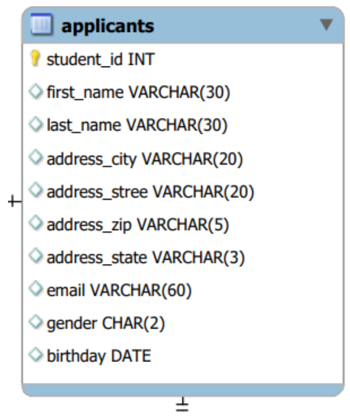
Phase C – Logical Design

In this phase, the tables that will be made were designed. Transformations and modifications were made to Phase B's ER diagram to make it into a more realistic model usable to create the database system. The final model is shown on the following page.


applicants
student_id INT
first_name VARCHAR(30)
last_name VARCHAR(30)
address_city VARCHAR(20)
address_stree VARCHAR(20)
address_zip VARCHAR(5)
address_state VARCHAR(3)
email VARCHAR(60)
gender CHAR(2)
birthday DATE
degree
degree_name VARCHAR(30)
department VARCHAR(30)
director VARCHAR(30)
email VARCHAR(30)
phone_number VARCHAR(12)
phone_number
phone_number CHAR(11)
student_id INT
criteria
criteria_id INT
degree_name VARCHAR(30)
criteria_text VARCHAR(400)
gre
gre_id INT
application_id INT
verba INT
quant INT
analytic FLOAT
taken DATE
application
application_id INT
degree_name VARCHAR(30)
student_id INT
essay VARCHAR(2000)
semester CHAR(6)
year INT
decision CHAR(10)
eval_date DATE
decision_date DATE
evaluation_score
application_id INT
criteria_id INT
score_id INT
education
education_id INT
application_id INT
college VARCHAR(20)
major VARCHAR(20)
degree CHAR(2)
grad_date DATE
gpa FLOAT
requirement
requirement_id INT
degree_name VARCHAR(30)
question VARCHAR(200)
criteria_score
score_id INT
criteria_id INT
score_text VARCHAR(80)
score_condition VARCHAR(300)
email
email_id INT
application_id INT
referencer VARCHAR(20)
body VARCHAR(1000)
appAnswer
application_id INT
requirement_id INT
answer VARCHAR(800)
answer
answer_id INT
requirement_id INT
answer VARCHAR(300)
evaluator
evaluator_id INT
application_id INT
name VARCHAR(30)

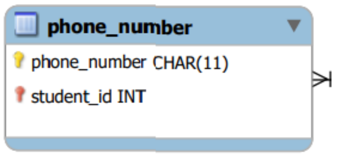
Description of Tables

Applicants



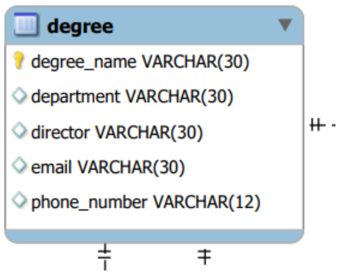
The applicant object has remained largely the same as from the initial design, as it is a rather stand-alone entity with unique attributes that simply must be present. The Address attribute, which was intended to be multivariable, has been split into multiple attributes to make sense for the relational data model. Something to note here is that I decided to keep the zip code as a varchar instead of an integer – this makes it more convenient for a case where someone might need to put something that is not a number into the zip code slot for some reason. I decided to make state and gender chars as they are relatively small, and in pretty much all cases entered values should not exceed those sizes. The primary key is a unique student id.

phone_number



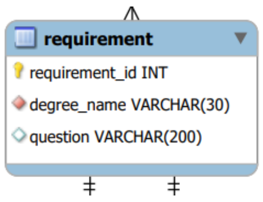
This table is necessary for an applicant to have more than one phone number. I decided to make the phone number attribute a char instead of an int as this allows for more leeway with data entry. Since there are no queries which really need to work with phone numbers as a number, it is easier to keep it as a char. This also allows users to include dashes if they want to, or spaces or anything else not-normal. Since I do expect most phone numbers to be a fixed format, in this case no spaces, I set the data type to a char, as a full phone number should be close to that set size in correct cases. This entity is a weak entity, and has the student_id of which it belongs to in addition to the phone number itself as a primary key. This allows many students to share numbers, but no single student can have the same number twice.

degree



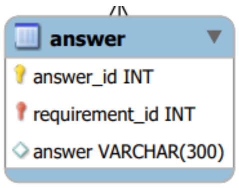
As the second "main" entity, this object has not changed much from its initial conception. This entity uses the degree_name as the primary key, as it is heavily implied that we should do this from the project requirements. Although I did see quite a few downsides from making it the primary key, I decided to go with it anyway as it does make sense, and also prevents the user from making two degree programs that are the same. I decided to put all of degree program's various attributes straight in the entity – this does mean the degree program can only have one director, one email, one phone number, ect. This is justified by the fact that the requirements do not say there should be many of these things, when for something like Applicant it specified that there could be multiple phone numbers, for example.

requirement



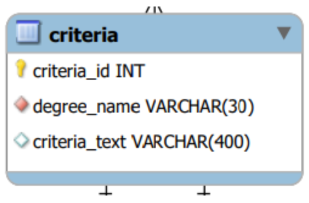
A required entity, the requirement table holds a question that an application must provide an answer to. I decided give this entity a unique identifier as its primary key, requirement_id, not only because it is a main entity, but because it makes a lot of other tables simpler and easier to handle.

answer



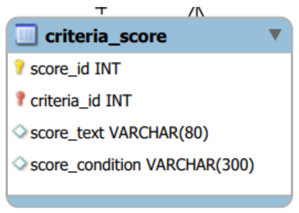
As each requirement must be able to have multiple answers, an answer table is required. It is a weak entity and is related to the requirement, the combination foreign key and discriminator is what makes its primary key.

criteria



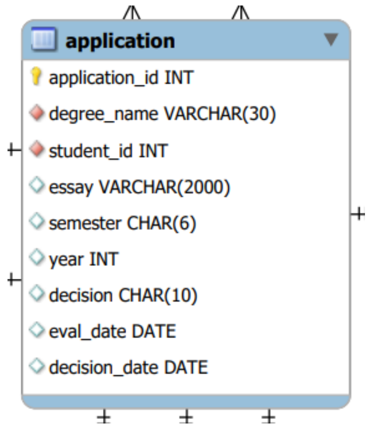
This table represents criteria that could be present in a rubric for a degree program. Essentially, this table, along with the criteria_score table, will replace the functionality of the rubric table present in the initial design phase. After careful review of the requirements, it was concluded that each degree program can only have ONE rubric, the rubric of which wholly consists simply of criteria entities. Therefore, it did not make sense to have a rubric entity when the criteria entity could be directly related to the degree entity the rubric entity would in turn have been related to. As such, we have this criteria entity, which consists of a unique criteria ID which is its primary key. Although I could have made the criteria ID a discriminator instead, making criteria a weak entity reliant on the degree object, I decided against it, as keeping criteria as a strong entity makes other entities later discussed much smaller and easier to deal with. Also, since it is replacing rubric, the criteria object can be considered one the main required entities, and can be justified that it deserves its unique identifier.

criteria_score



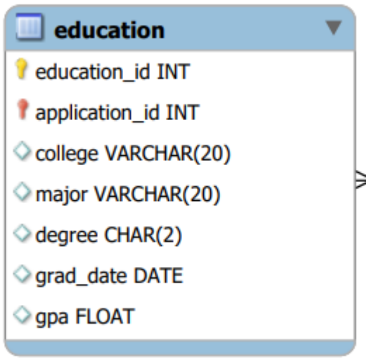
This table contains the possible scorings for each criterion. It represents a weak entity connected to the criteria object and uses a score_id as a discriminator. It is very loosely described what a score is in the requirements, and so in the end I chose to go with a string based score system which allows the user to pretty much input any kind of score they want. The score goes along with a score condition, which describes what an applicant must do or fulfil to get the aforementioned score. For example, for a criteria "models all entities", a score could be "proficient: modelled all entities", or "poor: "modelled only 20% of entities", where the text after the colon is the score condition a user would input when creating an entry in criteria_score. Each one of these score-condition pairs would be an entity with a score_id relative to the criteria it belongs to.

application



The application object is core to the database and has changed greatly from the initial design. For one, many attributes have been moved out of the application entity as the database must be able to support multiple instances of them, for example, education and GRE scores. Something to note is the fact that the evaluation entity has been incorporated and merged into the application object. After additional reading of the requirements, it would seem that although many professors can contribute to evaluating an application, those professors do not submit multiple evaluations – there is only ever one evaluation for every application. As such, there was no need to put the evaluation on its own, and so I made the more static fields of evaluation, such as decision and the dates, a part of the application object. The application object has a unique id as its primary key – it is a main entity. Applicants are able to create multiple applications – the requirements require more applications than applicants, so it must be that an applicant can make multiple applications.

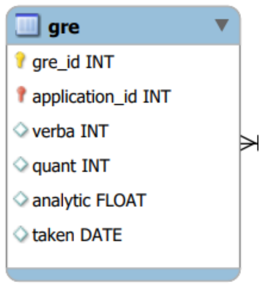
education



The education entity satisfies the requirement that applications have education of the student. While the reading the requirements, I was quite confused as to if a student should be able to have multiple educations, and if the education information should really be present on the application object. I had the thought that it was rather odd for education to be attached to the application, when it is the applicant that the education applies to, and so I considered for a moment, of allowing applicants to create educations objects that would attach to them; the applicant could then pick which education object they wanted to be present on an application they created thereafter. However, could also be seen as a wandering-off from the requirements, and so I refrained, keeping the education object related to the application entities. Users are able to add multiple educations to an application, but cannot attach the same education to another application.

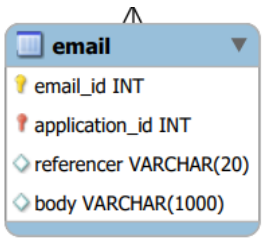
This is an unfortunate potential redundancy that is recognised, but as it is not a priority and time is short it must be overlooked.

gre



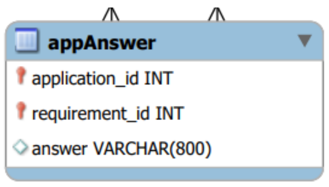
The gre object's conversion to a table follows similar logic to the education entity's normalisation. It was concluded that a user should be able to enter multiple GRE scores should they take multiple tests, but to fit the requirements I kept it attached to the application object instead of putting it onto the applicant. Similar to the education object it is a weak entity; its discriminant and the application id unique identifies it from other gre entities.

email



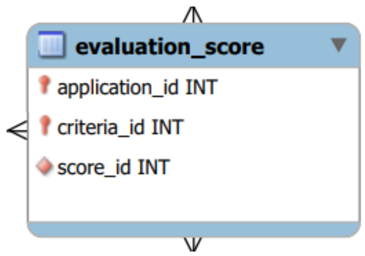
The email object is not the email of the application, but rather the reference emails that an application object must have as specified by the requirements. As there must be many email references, it must be present in its own table. The email object is a weak entity with a discriminator; this combined with the key of the application it is paired with forms its primary key. The email entity simply contains the sender and the body of the email.

appAnswer



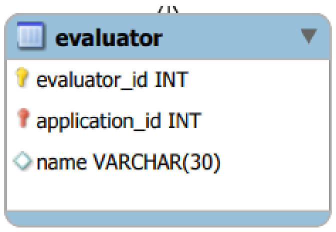
This object represents an answer made by an applicant in response to one of the questions that is a part of the degree program the applicant is applying for. As such it must have relationships with an application and such requirement. As stated earlier, requirements have their own unique identifier; this allows the appAnswer object to relate with the requirement object without having to store the degree name, which is already present in the application object that the appAnswer entity is related with. This helps to reduce a redundancy. The primary key of the entity is formed by the combination of the application id and the requirement id. This was decided as each application can only have one answer to each requirement question of the degree to which the application is applying to.

evaluation_score



As discussed in the application section, the evaluation entity has been removed an incorporated into the main application object, as there can only be one evaluation per application. However, as required by the requirements, an evaluation must have scores for the application that match those specified by the criteria of the degree the application is applying for. As such, it must be possible for there to be multiple evaluation scores attached to an application. That forms this object, which represents a score for an application, linked to the criteria it is a score for, and linked to a score id which tells us what the score actually is. The primary key of this entity consists of the application id and the criteria id – each application can only have one score for each criteria id that is part of the degree program of the application. As you may notice, since we made criteria a uniquely identified object, we do not have to store degree name anywhere in this object. This is useful as the accesses done for this object, as discussed in later sections, are rather annoying, and having criteria have a unique id really helped.

evaluator



The evaluator stores the names of those people who are performing the evaluation of an application. As it uses a discriminator+application ID for its primary key, evaluators with the same name are permitted. I was initially thinking of creating a faculty system, where users could add faculty, but since we do not have any requirements on saving information of evaluators aside from their name, I refrained from doing so.

Other decisions

ISA classes

I decided not to use any class structures, as I was constantly evolving the database as I worked on it, and never really settled on a clear vision of what the database should be. This prevented me from having any foresight of what could be useful as an ISA class. In hindsight, looking at the database from the end, it doesn't really seem like any classes would really be necessary, as it would seem many of our data entities are rather unique.

Multivalued Attribute Mapping

As described in detail in the Table Description section above, many of the attributes that were multivalued, or could be arrays, were moved into their own table.

Weak Entity Mapping

As seen in the Table Description section, there are many weak entities that are planned to be present in the database. These entities are discriminated with a discriminator, which when combined with the primary key of the parent table they are related to forms the primary key.

Normalisations

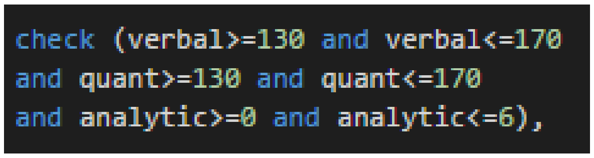
Just implementing the required entities alone already gives a rather normalised table, so I would say that most of the tables described above are in a normalised form. Tables are related to one another by primary keys, so data is not duplicated.

Phase D – Physical Design

Table Creation Scripts

The create table scripts are modelled directly after the ER diagram present in the previous section (or to be more precise, the model was made after the tables in this section). Therefore we will not go over the code for every single table, as it is simply the sql code for implementing the ER diagram shown before. There are some special mentions, however:

gre



```
check (verbal>=130 and verbal<=170
and quant>=130 and quant<=170
and analytic>=0 and analytic<=6),
```

As the scores of gre must be set in a specific range, I made sure to use some sql check statements to prevent the user from inputting incorrect scores. However, as I learned much too late, the MySQL we are using does not support check statements, so it doesn't actually do anything. But it's the thought that counts, I suppose?

education/application

I did consider making education's degree attribute and application's semester an enumeration, between (bs,ba,ms,ma) and (spring,fall), respectively. However, as the user interface will offer a multiple choice anyway, I decided not to double the limitation in SQL. This also helped during implementation, if I decided to change anything; I wouldn't have had to do it in multiple places.

education.gpa

The GPA value is limited to the normal range of 0-4. I considered not having this at first, as there might be a time where a different GPA scale is being used. However, since we do compare the GPAs in a later query, decided that GPAs must be forced to be on the same scale. But again, apparently, MySQL does not support check statements, so it doesn't actually do anything.

degree_name primary keys

As recommended by the requirements, the degree name was made the primary key. This might have been a bad idea, as the degree name is not only used as a foreign key in many other tables, but as a usable data field in degree object. As such, any changes to this degree name must be cascade through the database. Looking back, I probably would have changed degree_name to not be a primary key, and instead used a unique constraint on it. I realised this a little late, and don't really want to refactor the entire database.

Table Deletion

Not much special going on here, except that I tried to make sure tables are dropped in reverse order from when they are created, so that foreign keys don't fail.

Data loading Scripts

The data loading scripts were created by using the program to enter in data, and then exporting the data with MySQL Workbench's data export feature, which could export the data as a file of sql statements. I decided to do this instead of making up insert statements myself as the database program handles the incrementation of IDs for me, as it was designed to.

User Interface Design

As we are allowed to make a pretty bad user interface, which will be thoroughly described later, I decided to go with a text based interface. By navigating through menus and submenus the user should be able to access and edit information for all the required tables without having to memorise too many things. I did not want the user to have every choice up front, that would make it too confusing, and I didn't want to make a command line interface, that would be too much memorising.

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 the list of questions and answers alongside 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 (editing, 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

I decided not to create any additional indices, as the database at its current state is not expected to be very complex, and as it is rather normalised in my opinion. Just by having primary keys the database should operate at an acceptable speed.

User Requirements Satisfaction

As these tables were designed to fit the ER diagram shown in the previous phase, which was created a transformation of the ER diagram in the previous-previous phase, I would think that the data requirements are still being satisfied by this design. I even kept education and GRE scores attached to application, although it might not be a good idea, to ensure that that particular requirement is not being skirted.

Phase E - Development

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

For application evaluation information, I wanted to keep the dates for the evaluation null or empty until they were filled. Because of the way my general add function was implemented, and the fact that I got to implementing evaluations a little late, I didn't really have time to figure out how to get those date values to be some kind of null value when creating the application object. As such, they are set to a time far in the past, and the program checks if the date is before a certain time, it counts the field as empty. This makes logical sense, as never will a user enter for an evaluation date created a time before the current time. This also acts as an impromptu error check – should the user attempt to enter an illogical date, it will show up as empty. But I wouldn't advertise this feature, as error checking is virtually absent everywhere else.

Code Structure

I can't say I'm too proud about the way I implemented the program, it is rather arbitrary if a submenu is going to be in its own function or not, and there are relatively few comments. If there any clarification is necessary please do not hesitate to contact me.

User Interface Discussion

In this section I write about some of my thoughts and processes while implementing the UI. How to use the UI is explained further in the Usage Guide phase.

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 separator up front.

Student Mode

A theme you will see throughout the program is the ability to "select" items. Further 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. Luckily, 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. Unfortunately, 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. Noticeably, 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 multiple 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.

Query Mode

The Query mode is rather straightforward, the user can pick a query and the program gathers additional information from the user if necessary and outputs the result. I decided to make the output seem as English-like as possible, but I was probably pretty inconsistent with that.

Queries Discussion

The queries required of us were often confusing in some aspects; in this section I explain my assumptions and approach to the queries.

Query 1: Names of applicants for degree program and period

For this query, it seems like the requirements want the query to take in a degree program and period, and then return the names of applicants to that period. This is efficiently accomplished as the application object of the program contains all the relevant information needed by the query. However, since applicants can submit multiple applications, a "select distinct" SQL statement had to be used to make sure an applicant's name would not show up multiple times. In the Python program when this query is used it is possible to exclude any one of the search options (degree, semester, year), and it will only search for the given constraints.

Query 2: total applicants by program and period

This query shows the number of applicants that applied in each degree for each individual semester of each individual year. It does not require any user input, as the requirements seem to suggest that all degree, semester, and year combinations should be listed, as unlike the previous query, the words "given degree" is not present. As a side note, many of the queries we have to implement spread our database kind of thin, this is highly noticeable as we do not have a lot of sample data put into the database.

Query 3: Most popular major in a given year

This query's requirement text was completely unspecific about what "year" actually is. Since we had no data requirement that the application object need a date on it, I had to assume that year referred to the graduation date of the applicant, and so this query must be a report on the major that the highest amount of applicants graduated from in a given year. Since the requirement does not seem to specify that the query must return all the top majors, I limited the result to 1. Many of the other requirements explicitly specify that multiple results are to be returned, and so I figured that if it did not say so, I did not have to. I also considered that "year" might pertain to the application's applying year, but I thought it would be more interesting to see how many people graduated with a certain major in given year.

Query 4: Lowest GPA applicants accepted in the current period

Since the requirement seemed to want to return multiple applicants, I assumed that this must mean the query must return all applicants who have the same as the lowest GPA. I assumed “current period” to refer to the current year and current semester, as defined in the first query’s requirement text. The decision date does not have a field for semester, so I decided that a fall semester would consist of the months 7-12, while spring is 1-6.

For this query, denormalisation might have been useful, as the education object must trace through the application object, and then to the applicant table before a name can be acquired. Luckily, most of the processing is done before this traversal of tables, the traversal is only necessary for the conversion from application ID to applicant name.

Query 5: number of students applying to a degree program, grouped by major

This requirement was quite confusing and I still do not really think I did it right. The requirement states, "for each degree program by prior degree", but does not really explain what "prior degree" is. We had no data requirements of storing any kind of "prior degree", and I would think that most applicants would have no prior degree, as they are entering a degree program right now, to get a degree. The only other instance of a separate degree being identified is in the application's education requirements, where it mentions a "degree (BS, BA, MS, MA)". I consider the degree field in education to actually be degree *type*, and the name of the "degree" to be the field major. Therefore, I concluded that the requirement must actually mean, prior major, which by extension, could represent a prior degree. And so this query returns a report of the number of applicants that have applied to each degree with a certain major. Since the requirement does state "for each", I made sure the query returns information on all degree programs.

Query 6: non evaluated applications [till a certain date]

This requirement was also not very elaborated upon; I was completely unable to figure out what "certain date" means. As applications do not have dates (besides the year they are applying for), the only other dates pertaining to evaluation are evaluation creation date and decision date. However, the evaluation creation date can only be set once an evaluation is made, and the decision date can only be set after the evaluation is made. Therefore, if an application has not been evaluated, it will have no dates. So there is no date to compare that I could think of in this case. I was also thinking that it could mean, applications that have been made before a certain date that have not been evaluated, if this were the case, the query could be used to find old applications that have been waiting for a long time to be evaluated. However, we had no requirements to save a date for application creation, and so this cannot the intended usage of the query. In the end, I just made the query return all the application ids for applications that have not been evaluated. As a side note, the requirement states to find "applications", but does not specify what information to return in the application. Although I return an application ID, as that represents an application, it is kind of difficult to imagine how this would be useful to a user who knew nothing of the inner workings of the database, as we do.

Query 7: Evaluation decisions by program and year

This query returns the number of decisions made for each degree for each year. As this information is present within the application object, this query does not have to reach far to create its result. I would also like to note that this query stretches our test database quite sparsely, as we did not have to put much sample data put into it.

Query 8: Referencers that appeared most frequently in accepted applications

The assumption I made for this query was that by "find the emails" it actually meant "find the email addresses". Although it would be useful to get the entire email (body and address), somehow, I feel like this is too much information, especially as the query does not seem to offer any filters. As such, this query finds the number of accepted applications each referencer that appears in the email table has made, and returns the emails of those who were a part of the most accepted applications. Due to the way the email objects are handled, I chose not to fit in a multiple choice system for choosing email referencers, and so for an email address to be counted multiple times the user must correctly spell the email address when adding a new reference email. You may notice that I often use the term “referencer” instead of email address or email. This is as the data requirements did not actually specify that we needed to have specifically an email address, so instead the email object has a general field simply called “referencer”. In this field the user can put either just a name, or, an email address, which is what seems to be the correct thing to put in. Since the program doesn’t require that this field be specifically an email address, I decided to be non-misleading and state that the return value is “referencer”.

Query 9: GRE statistics

For this query I assumed "application year" means the year the application is applying for. Since this year is rather arbitrary and depends on the choice of the applicant who made the application, I can't really see how it has anything to do with GRE scores. But the query was still do-able using the various aggregate functions already included in SQL. Since there are three different GRE scores, it returns three different values for each unique degree/year pair.

Query 10: Most attended colleges in past 5 years

I was once again unsure of what date to compare in this query. After some thought I settled on all applicants that had a graduation date anytime after the current year minus five, as if they graduated before and are not in another college than they obviously do not fit the "attended college" bill. As such, this query counts the number of applicants per college given that date restriction, and returns all the colleges that are equal to the max.

Additional Limitations/Issues

Besides the many limitations already discussed thus far, these are some of the others I could think of that couldn't quite fit smoothly into any of the other sections:
-only one rubric per degree, but this is stated in the project description
-not really consistent when giving user choices: sometimes it is the id of the item, sometimes it is presented in numeric order
-a lot of sub menus to get to information
-when editing a field, have to reenter information, cant just edit it in place
-it might not be immediately clear to a new user where they should go to be able to create or edit something, given that this new user knows nothing of the structure of the system
-the essay can't be editted in a traditional sense, when you are able to navigate the original text and make modifications, you can only reinput a whole new essay which updates the field. But it wasn't really in the requirements that we must be able to directly edit the essay
- Inconsistent UI choices. I didn't really pay attention to if the choices started at 0 or 1, and it kind of randomly switches around.
-The character limits for the varchar attributes in the database might be considered rather small.
-no "best to worst" ordering for criteria. This isn't a requirement, but it would make sense for the possible scores to be ordered in some way. However, since it was not required, the database has no support for reordering of answers, and since answer scores are text/description based (e.g., "very good" or "poor" instead of a number), there can be no automatic sorting

Possible Improvements

Aside from improvements stated in any of the other sections, these are some additional overall improvements that could be made:
-Input validation at all levels
-ability to cancel more operations
-more things could be converted to a create-and-choose system. For example, when creating a email referencer, that name could appear as a choice to the user
-More general functions in the code. I think this program might be the ugliest Python code I've ever written

Phase F: Usage Manual

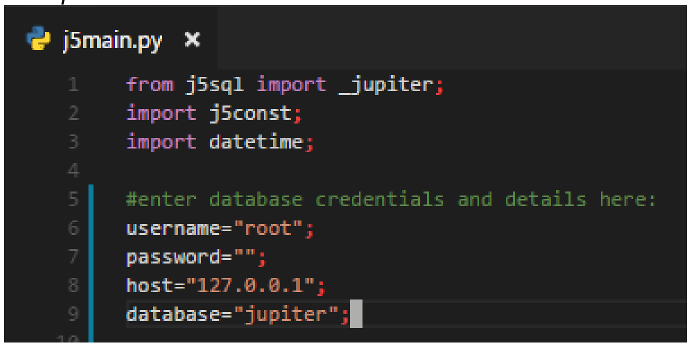
Setting up the database

A MySQL database must be running and a database must be created prior to usage of the program. Please consult the nearest database professor for how to set up a database server and create a database on the server. The program does not create a database or database server. When creating a database on the database server, **the name of the database must be "jupiter".**

Create the Tables and load the Data

The tables must be created before the application can be run. Using MySQL workbench or something similar, run the createAll.sql on your running database to create tables in the database called jupiter. Again, **a database called jupiter must already be created.** Use your database program and call loadAll.sql to load sample data into the database.

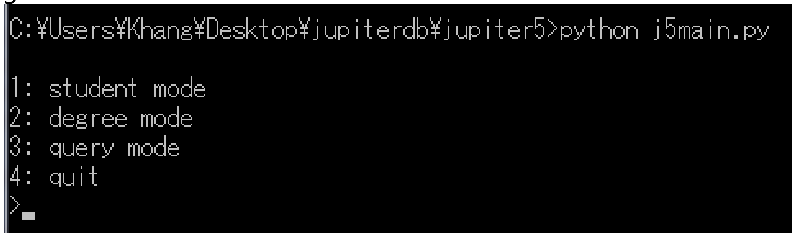
Getting the Program setup with the Database



```
j5main.py
from j5sql import _jupiter;
import j5const;
import datetime;

#enter database credentials and details here:
username="root";
password="";
host="127.0.0.1";
database="jupiter";
```

The program uses the python mysql connector module to connect to a mysql database. In order to do this, a **username, password, host address** must be provided by the user. These details must be entered near the top of the j5main.py file. Some defaults will already be there, it is safe to overwrite them with your own details.

Running the Program



```
C:¥Users¥Khang¥Desktop¥jupiterdb¥jupiter5>python j5main.py

1: student mode
2: degree mode
3: query mode
4: quit
>
```

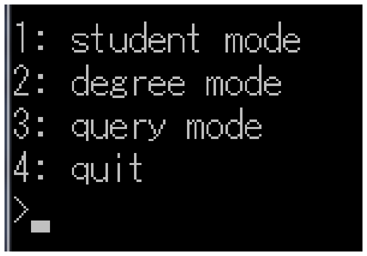
The program can be run by calling Python on the main python file, j5main.py, while all other python files are in the same directory (j5const.py and j5sql.py). Note that if the program fails to connect to the database, an error will have occurred at this point. Please re-check your inputted credentials. Additionally, please make sure to have the most update version of Python, and to install any missing modules.

General Usage Warnings

The UI of the application was rather roughly (very roughly) made and as a result has a 'few' quirks. As we were not required to make a good UI, there are a few things to watch out for:

1. If an SQL error occurs, the program (should) not crash, and the transaction will not occur. Please make sure to carefully check for any error messages that appear, the program **does not** pause to show messages, the message will appear, and the program will continue.
2. Many prompts will request that something be "selected". In pretty much all cases this means to enter an **integer** that corresponds to one of the choices visible above, but not necessarily *directly* above. For many of these prompts, the program will **crash** if given an incorrect input (out of range, not an integer, not one of the choices), so please be careful.
3. Many operations are **not cancellable**. This means that if you accidentally select it, you will have to either complete the operation or close the program. Luckily, it is generally easy to recover from these choices.
4. Concerning delete menu choices: they will either delete the currently selected item, or prompt to delete an item from a list above. I tried to make sure this is specified, but might have missed some prompts. In either case, most delete prompts **do not ask for confirmation**. Since deletes cascade, quite a decent amount of data may be lost by an accidental delete. Please be careful around delete choices.
5. It is **highly** recommended for the sample data to be loaded before using the program. The menus of the program were not designed for empty menus, for example the program will have to be closed should you try to select something when there is nothing to select. It is of course possible to use the program starting with an empty database, however, be warned that **many choices will lead to dead ends**, requiring the UI to be closed.
6. The UI is not guaranteed to work correctly given incorrect user input.
7. The database and its data should not be damaged should the program crash, so feel free to close the program via keyboard interrupt at any time. Often times, this may feel like the fastest way to get to the main menu when deep in submenus.

The main screen



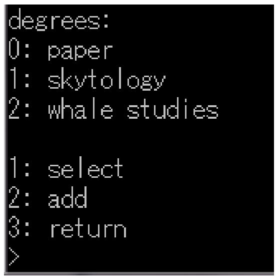
```
1: student mode
2: degree mode
3: query mode
4: quit
>
```

Once the program is running, you will be presented with the main screen. From this main screen you are able to navigate through menus in order to access various parts of the database.

Degree Mode

The degree mode has all the menus necessary to enter and modify details of a degree program in the database. It is recommended to start with this mode, as many operations in Student Mode require information created during Degree Mode.

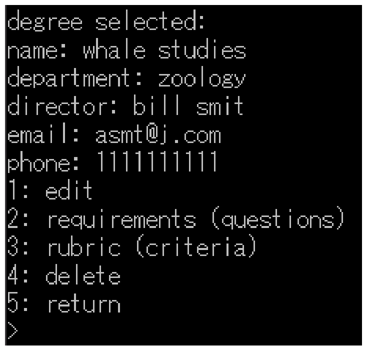
Degree Select



```
degrees:
0: paper
1: skytology
2: whale studies

1: select
2: add
3: return
>
```

Degrees can be created on this screen. Additional information of the degree can be entered after selecting the degree.

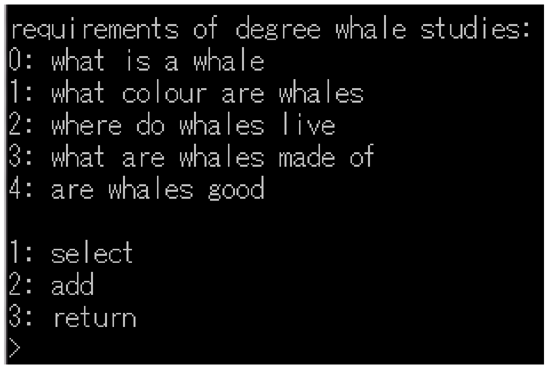
Selected Degree



```
degree selected:
name: whale studies
department: zoology
director: bill smit
email: asmt@j.com
phone: 1111111111
1: edit
2: requirements (questions)
3: rubric (criteria)
4: delete
5: return
>
```

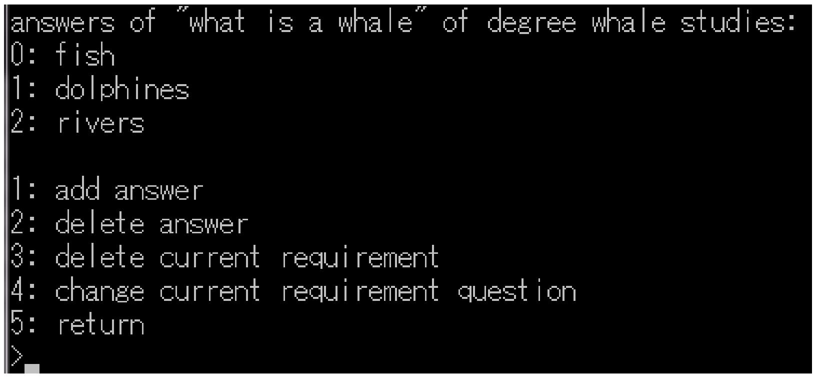
Here the user can edit the details of the degree, enter requirement questions, and set criteria. Selecting "delete" **will delete the currently selected degree.**

Selected Degree->Requirements Mode



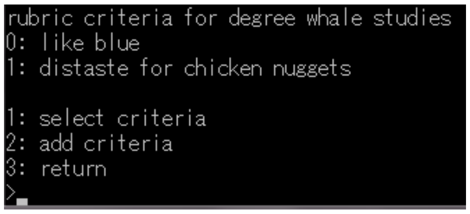
```
requirements of degree whale studies:
0: what is a whale
1: what colour are whales
2: where do whales live
3: what are whales made of
4: are whales good

1: select
2: add
3: return
>
```

Selecting the requirements option shows the requirement questions for the currently selected degree. To add a new requirement, select the add option and enter the question of the requirement. To edit the requirement, delete the requirement, and add answers, simply select the requirement using the select choice. There is no option to edit answers – as it does not matter what order answers come in, simply delete the offending answer and add a new one. This is is equivalent to editing an answer. Displayed below is the screen upon selecting a requirement:



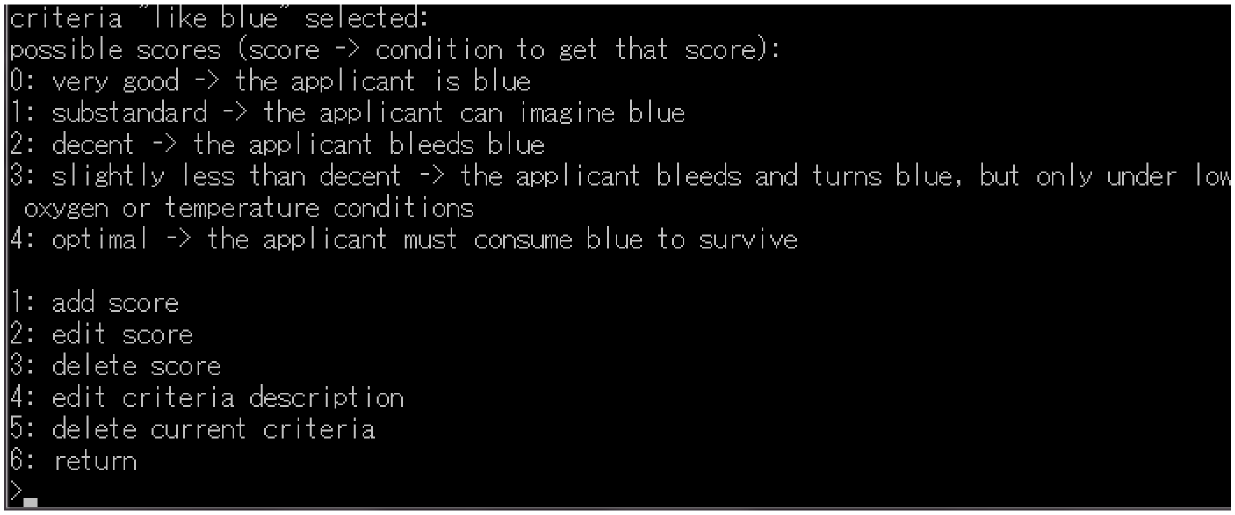
```
answers of "what is a whale" of degree whale studies:
0: fish
1: dolphines
2: rivers

1: add answer
2: delete answer
3: delete current requirement
4: change current requirement question
5: return
>
```

Selected Degree->Rubric Mode



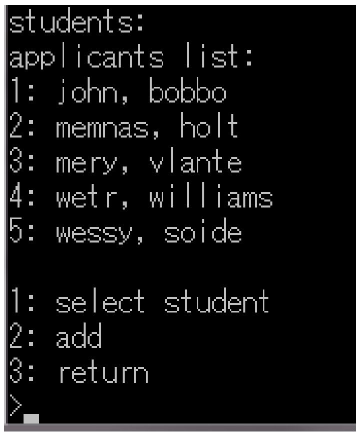
```
rubric criteria for degree whale studies
0: like blue
1: distaste for chicken nuggets

1: select criteria
2: add criteria
3: return
>
```

The rubric of a degree consists of its criteria. In this menu criteria can be added and selected in a similar manner to requirements. You will be prompted to enter the description of a criteria when creating one. Once a criteria is made and selected, scores can be assigned to the criteria in a similar manner to requirement's answers, the difference being a score has a description and a condition. The two fields of a score can be edited by selecting the relevant choice. When prompted to choose a number of a score, enter the number to the left of the score as it appears on the list above the menu prompts. Displayed below is the screen upon selecting a criteria:



```
criteria "like blue" selected:
possible scores (score -> condition to get that score):
0: very good -> the applicant is blue
1: substandard -> the applicant can imagine blue
2: decent -> the applicant bleeds blue
3: slightly less than decent -> the applicant bleeds and turns blue, but only under low
 oxygen or temperature conditions
4: optimal -> the applicant must consume blue to survive

1: add score
2: edit score
3: delete score
4: edit criteria description
5: delete current criteria
6: return
>
```

Student Mode

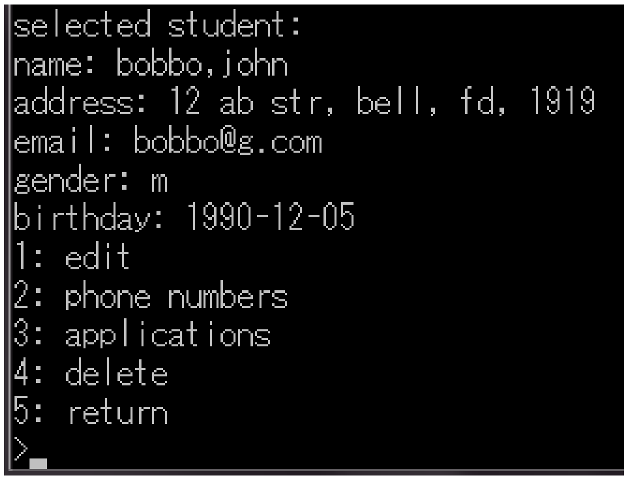


```
students:
applicants list:
1: john, bobbo
2: memnas, holt
3: mery, vlante
4: wetr, williams
5: wessy, soide

1: select student
2: add
3: return
>_
```

warning: once a create student operation has begun, it cannot be cancelled without ending the program. If you did not want to create a student, please follow through the process with arbitrary information to create a student, this student can be deleted later.

Students can be selected and created here. Phone numbers for a student can be added after the student is selected. When selecting a student, **type the number to the left** of the student desired.

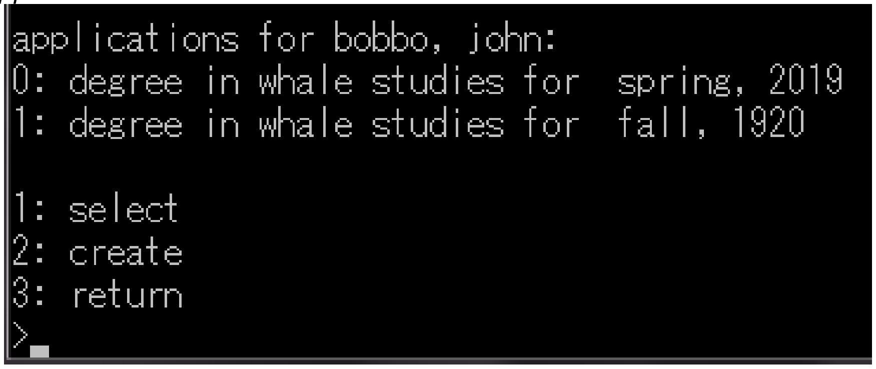
Selected Student Mode



```
selected student:
name: bobbo,john
address: 12 ab str, bell, fd, 1919
email: bobbo@g.com
gender: m
birthday: 1990-12-05
1: edit
2: phone numbers
3: applications
4: delete
5: return
>_
```

On this screen the phone numbers of a student can be added, and applications can be made.

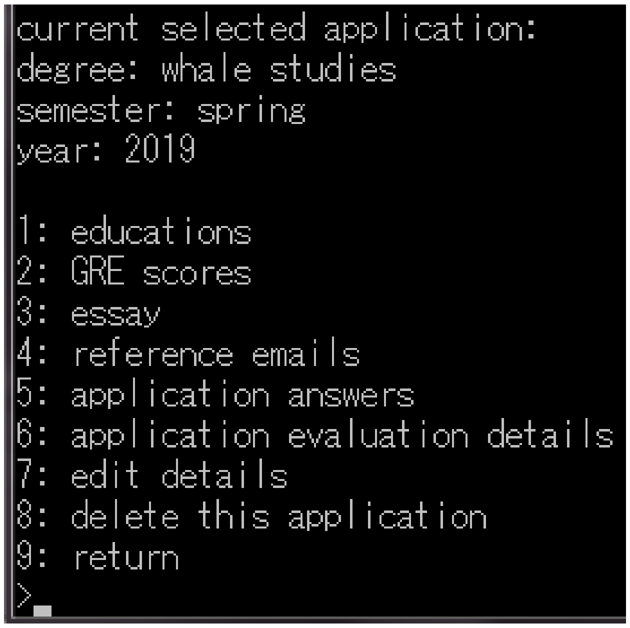
Adding an Application



```
applications for bobbo, john:
0: degree in whale studies for  spring, 2019
1: degree in whale studies for  fall, 1920

1: select
2: create
3: return
>
```

When adding an application, the user will be prompted to select a degree that had been created in the degree mode. If no degrees are present, refer to General Warning 5. Select the application after it has been created to begin adding more details.

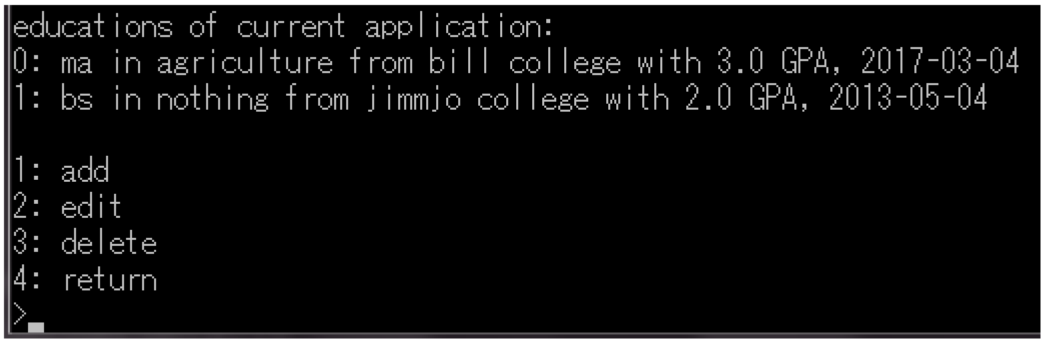
Application Mode



```
current selected application:
degree: whale studies
semester: spring
year: 2019

1: educations
2: GRE scores
3: essay
4: reference emails
5: application answers
6: application evaluation details
7: edit details
8: delete this application
9: return
>
```

From this screen, application information can be entered. Select an option to navigate menus which are able to access the relevant information. A note for the submenus: in this part of the program, many of the submenus, when asking for the number of some item to edit or delete, will **crash the program given incorrect input**. Although the user should **already be watching out** for this, incorrect input is slightly more important in this section of the program. A reminder that this is the worst UI ever.

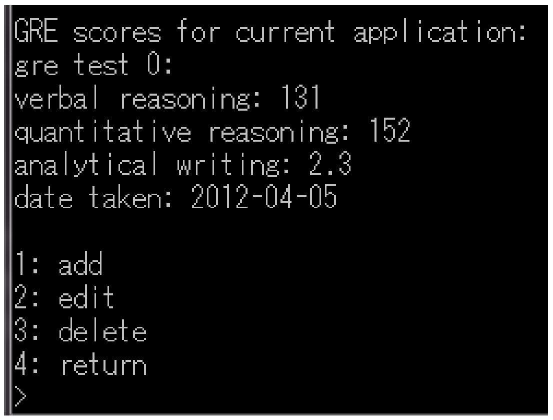
Application Mode->Education



```
educations of current application:
0: ma in agriculture from bill college with 3.0 GPA, 2017-03-04
1: bs in nothing from jimmjo college with 2.0 GPA, 2013-05-04

1: add
2: edit
3: delete
4: return
>
```

Applicant education can be entered here. Please note that although there are sql CHECKS in the table code, these checks do not actually work, so it may be possible to enter, if needed, GPA values above 4. However, this may lead to unexpected results when performing queries, so do so at your own risk. The delete and edit options on this menu will ask for a number, input the number immediately to the left of the desired education to perform the action upon it.

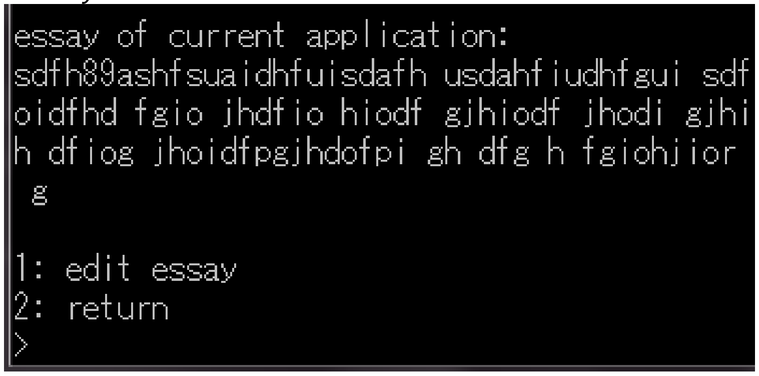
Application Mode->GRE scores



```
GRE scores for current application:
gre test 0:
verbal reasoning: 131
quantitative reasoning: 152
analytical writing: 2.3
date taken: 2012-04-05

1: add
2: edit
3: delete
4: return
>
```

The GRE mode is functionally identical to the education mode; it simply deals with different information. Although there are CHECK statements for the GRE score ranges, they do not work, and so the GRE scores entered are not checked for valid-ness. Enter illogical scores at your own risk.

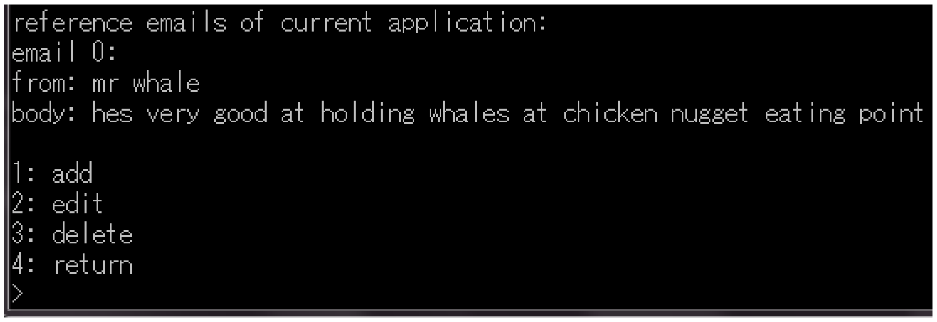
Application Mode->Essay



```
essay of current application:
sdfh89ashfsuaidhfuisdafh usdahfiudhfgui sdf
oidfhd fgio jhdfio hiodf gjhiodf jhodi gjhi
h dfiog jhoidfpgjhdofpi gh dfg h fgiohjior
 g

1: edit essay
2: return
>
```

The essay for the application can be viewed and edited here. It is separate as logically essays should be long, and would take up the screen if it weren't in its own submenu.

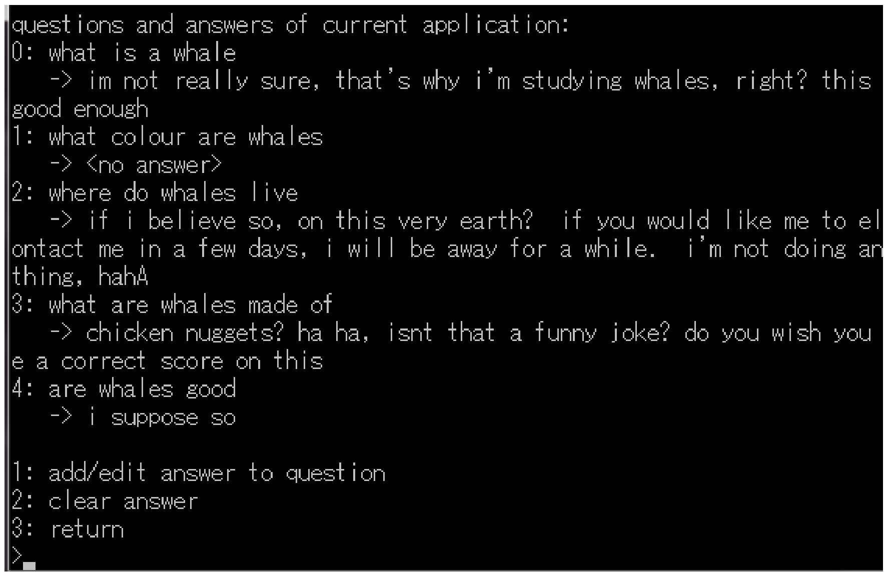
Application Mode->Reference Emails



```
reference emails of current application:
email 0:
from: mr whale
body: hes very good at holding whales at chicken nugget eating point

1: add
2: edit
3: delete
4: return
>
```

Reference Emails can be viewed, added, edited, and deleted here. Up to a maximum of three emails can be present at a time, please remove one if you have reached the limit.

Application Mode->Answers

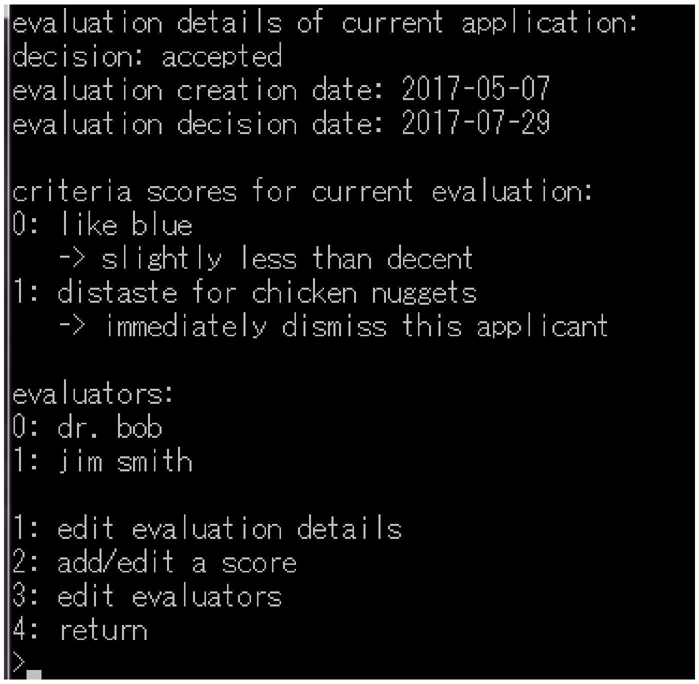


```
questions and answers of current application:
0: what is a whale
   -> im not really sure, that's why i'm studying whales, right? this
good enough
1: what colour are whales
   -> <no answer>
2: where do whales live
   -> if i believe so, on this very earth?  if you would like me to el
ontact me in a few days, i will be away for a while.  i'm not doing an
thing, hahA
3: what are whales made of
   -> chicken nuggets? ha ha, isnt that a funny joke? do you wish you
e a correct score on this
4: are whales good
   -> i suppose so

1: add/edit answer to question
2: clear answer
3: return
>
```

Note: this screenshot is cutoff to save space

Here, the user can supply answers to requirement questions of the degree the application is applying to. Every question is displayed immediately in this submenu; the user simply has to pick a question and type in their answer.

Application Mode->Evaluation



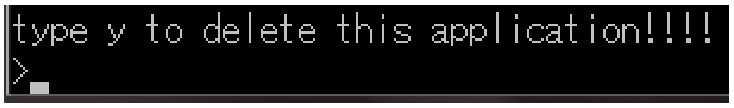
```
evaluation details of current application:
decision: accepted
evaluation creation date: 2017-05-07
evaluation decision date: 2017-07-29

criteria scores for current evaluation:
0: like blue
   -> slightly less than decent
1: distaste for chicken nuggets
   -> immediately dismiss this applicant

evaluators:
0: dr. bob
1: jim smith

1: edit evaluation details
2: add/edit a score
3: edit evaluators
4: return
>
```

Evaluation things such as final decision, evaluation creation date, and evaluation decision can be entered here. Select "edit evaluation details" to change these fields. The criteria of the degree of the currently selected application is also visible here. By selecting "add/edit score", scores can be assigned by evaluators. The scores available to assign are those created during Degree Mode->Rubric Mode. The score description will then appear below the criteria. Evaluator's names can also be added at this time using the "edit evaluators" option. When attempting to delete an evaluator, refer to the number next to the evaluator's name. Note for user: Dates are expected to be valid, and may result in strange queries if illogical dates are entered. For example, it is expected that a decision is to be accompanied with a decision date. It is not possible to enforce these with the menu UI, where a user can edit both fields one at a time.

Application Mode->Edit Details

The details entered at the start of the application creation process may be corrected here.

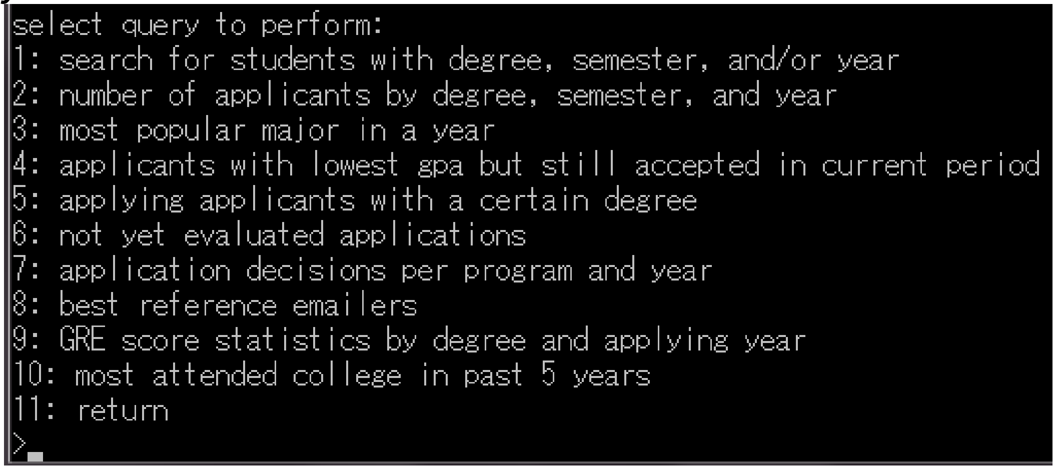
Application Mode->Delete Application



```
type y to delete this application!!!!
>
```

This will **delete the currently selected application and data attached to it**. Fortunately, we are proud to announce this particular delete option does in fact provide confirmation. Do not expect it on any other options.

Query Mode



```
select query to perform:
1: search for students with degree, semester, and/or year
2: number of applicants by degree, semester, and year
3: most popular major in a year
4: applicants with lowest gpa but still accepted in current period
5: applying applicants with a certain degree
6: not yet evaluated applications
7: application decisions per program and year
8: best reference emailers
9: GRE score statistics by degree and applying year
10: most attended college in past 5 years
11: return
>
```

Queries can be executed here. The queries are ordered in the same order as the requirements, just in case the descriptions are confusing. For this part of the program, the program pauses after each query result so the user can take note of it. At these times, follow the on-screen prompt and press the **enter** key to continue. It must be the **enter** key. Some queries will prompt for a piece of information before continuing, please continue to heed the warnings of invalid user input for these prompts as well.