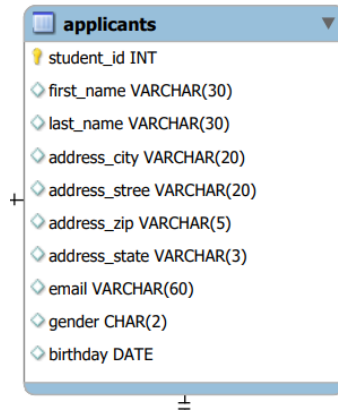


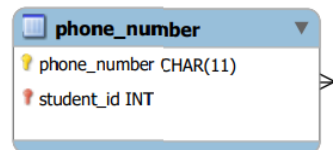
Description of Tables

Applicants



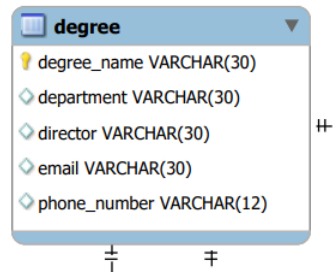
The applicant object has remained largely the same as from the initial design, as it is a rather stand-alone entity with unique attributes that simply must be present. The Address attribute, which was intended to be multivariable, has been split into multiple attributes to make sense for the relational 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



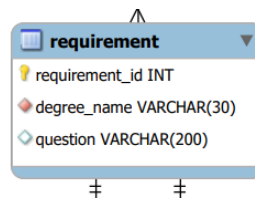
This table is necessary for an applicant to have more than one phone number. I decided to make the phone number attribute a char instead of an int as this allows for more leeway with data entry. Since there are no queries which really need to work with phone numbers as a number, it is easier to keep 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



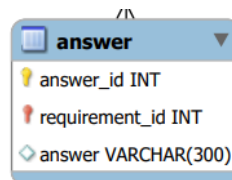
As the second "main" entity, this object has not changed much from its initial conception. This entity uses the degree_name as the primary key, as it is heavily implied that we should do this from the project requirements. Although I did see quite a few downsides from making it the primary key, I decided 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



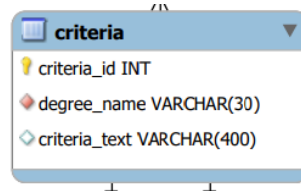
A required entity, the requirement table holds a question that an application must provide an answer to. I decided give this entity a unique identifier as its primary key, requirement_id, not only because it is a main entity, but because it makes a lot of other tables simpler and easier to handle.

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.

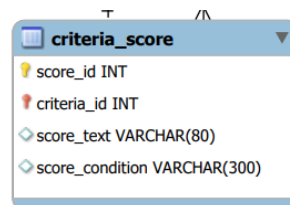
criteria



criteria	
criteria_id	INT
degree_name	VARCHAR(30)
criteria_text	VARCHAR(400)

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.

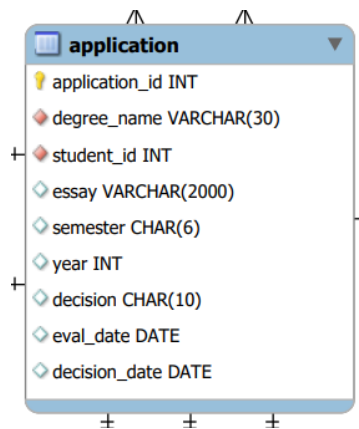
criteria_score



criteria_score	
score_id	INT
criteria_id	INT
score_text	VARCHAR(80)
score_condition	VARCHAR(300)

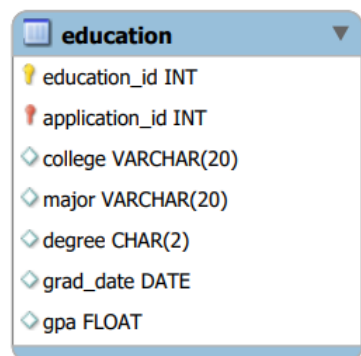
This table contains the possible scorings for each criterion. It represents a weak entity connected to the criteria object and uses a score_id as a discriminator. It is very loosely described what a score is in the requirements, and so in the end I chose to go with a string based score system which allows 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.

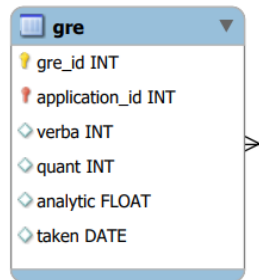
education



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.

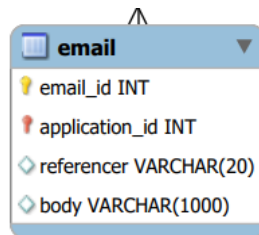
This is an unfortunate potential redundancy that is recognised, but as it is not a priority and time is short it must be overlooked.

gre



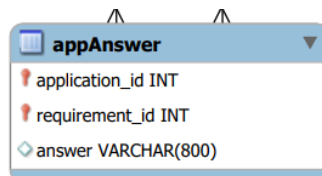
The gre object's conversion to a table follows similar logic to the education entity's normalisation. It was concluded that a user should be able to enter multiple GRE scores should they take multiple tests, but to fit the requirements I kept it attached to the application object instead of putting it 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.

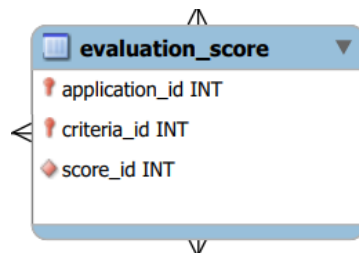
appAnswer



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

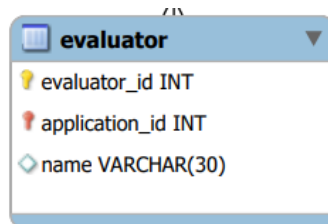
only have one answer to each requirement question of the degree to which the application is applying to.

evaluation_score



As discussed in the application section, the evaluation entity has been removed and 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.