Data Admin Concepts & Database Management

Table of Contents

Lab 03 – Logical Modeling

Overview

This lab is the third of ten labs in which we will build a database using the systematic approach covered in the asynchronous material. Each successive lab will build upon the one before and can be a useful guide for building your own database projects in a systematic way.

In this lab, we will convert conceptual entity relationship diagrams (ERDs) into more fully dressed logical model diagrams, also known as enhanced ERDs. Part 1 of this lab explores three independent case studies and asks you to model the logical design with an increasing level of independent work required to identify the logical model components in the conceptual designs. Part 2 returns to our VidCast database and asks you to convert a version of the conceptual model to a logical model diagram.

For this lab, we will continue our work in Draw.io. There are other software tools that can accomplish this task (Visio, Vertabelo, and Lucidchart to name a few). If you would prefer to use these other tools, consult with your course section lead.

Read this lab document once through before beginning.

Learning Objectives

In this lab you will

- Demonstrate understanding of the logical model
- Demonstrate ability to convert conceptual ERDs to logical model diagrams (E-ERDs)
- Identify the appropriate cardinality and degree of relationships
- Identify data types using narrative elements and prior domain knowledge

Lab Goals

This lab consists of two sections. The first section presents three case studies for building logical model diagrams. Each consecutive case study will present less and less technical information. By the end of this section, you should be able to convert an ERD into a logical model fit for coding a database. In the second section, you will apply your ability to build a logical model for our ongoing database project, VidCast.

What You Will Need to Begin

- This document
- An active Internet connection
- Visit draw.io (https://www.draw.io/) and ensure you can create a blank diagram. Ideally, connect draw.io to your Google Account, OneDrive, or local disk. If you are new to using online software tools, the recommendation is to connect draw.io to your Google Drive. If you do not have a drive, visit drive.google.com (https://drive.google.com) to create one for free before beginning the second part of this lab.
- A blank Word (or similar) document into which you can place your answers. Please include your name, the current date, and the lab number on this document. Please also number your responses, indicating which part and question of the lab to which the answer pertains. Word docx format is preferred. If using another word processing application, please convert the document to pdf before submitting your work to ensure your instructor can open the file.
- To have completed Lab 02 – Conceptual Modeling
- To be proficient with ERD drawing tools available in Draw.io.
- Understanding of primary keys, surrogate keys, and foreign keys
- A basic understanding of data types and when to select which data type

Part 1 – Logical Modeling Case Studies

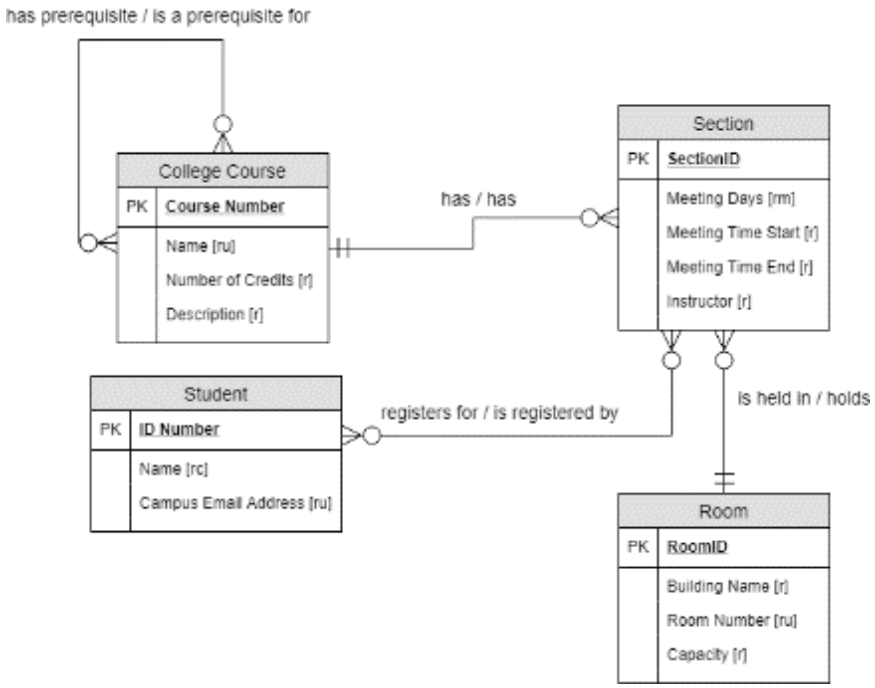
Setup

The following case studies have been adapted to allow you to build and demonstrate aptitude in translating ERDs to logical model enhanced ERDs (E-ERDs). The first case study walks you through step-by-step instructions for identifying facts, or business rules, in the narrative to be modeled. By the third

case study, you should be able to apply these principles on your own to design the solution. Each time you complete a case study, export the diagram as an image file (PNG or similar) and insert it into your answer doc.

Case Study 1 – Obligatory College Classes Modeling

We've elaborated on our college courses example from last time. From the diagram below, we will create a logical model diagram, or E-ERD that could potentially be used to code a database. This first example will provide a step-by-step guide for how to do this.



Mapping from a conceptual ERD to a logical model E-ERD can be broadly broken down into three main steps:

1. Map any existing entities into tables
2. Map any attributes into columns
3. Map any relationships between entities as relationships between tables

Step 1 – Map any entities into existing tables

Starting with a blank Draw.io diagram (or a new page in the existing diagram, if you'd like to keep everything together), create an "ER Table 1" shape and change each shape's title to correspond to the entity it represents.

A note on naming:

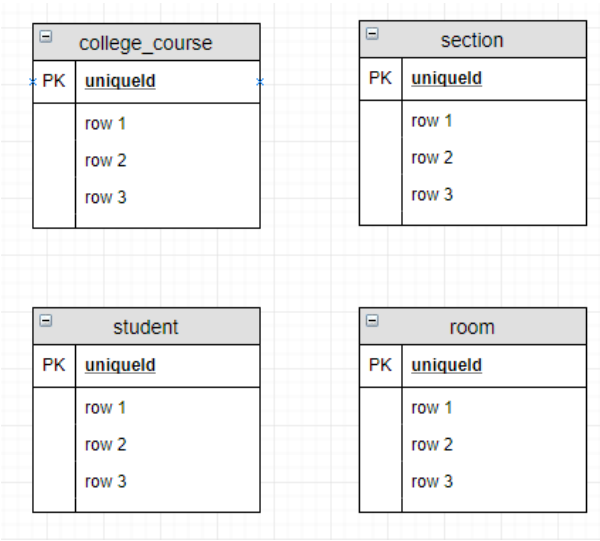
It's at this stage that we need a little forethought of how these tables and columns will look in our database. Since databases and the applications that use them are not particularly fond of spaces, from here on out, we'll remove spaces and rely on a convention that helps us visually separate words in our table and column names. This lab, and the rest of the course, uses an underscore, _ , to separate words instead of a space, making "College Course" into "college_course". Another convention you may see is

to use “camel Case” in which the first letter of each word is capitalized to distinguish itself from the other words in the name, making “College Course” into “CollegeCourse”. Still other, arguably more anarchist, conventions eschew such helpful tips and just smash the words together, making “College Course” into “collegecourse”. Whichever method you choose is up to the database programmer but note that if you deviate from the conventions used in the lab you’ll also have to make the change for all instances of that object in this and subsequent labs. The most important aspect of this decision is to **be consistent**. Try to not mix naming styles.

Additionally, most RDBMSs are case insensitive when it comes to object names. This means that the software will see College_Course and college_course as indistinguishable from one another. This lab uses all lower-case letters, but feel free to use whichever style looks best to you.

Lastly, we’re also going to try to avoid using table and column names that correspond to reserved words in our RDBMS. Usually, this can be done by putting the table name in front of the column name when defining a column. For instance, “Name” is a SQL Server reserved word: it has a specific meaning and use in SQL Server, so we’re going to avoid using it as a column name as is. As such, instead of the Course table having a Name column, we’re going to call that column course_name to avoid conflict with that reserved word. At this point, you may not know what all those words are, but where applicable within this lab, we’ll point them out.

After creating and renaming your shapes, your diagram should look like this (note that your arrangement may differ. It’s the presence of these shapes that matters, not their precise location on the page):



That’s it for step 1!

Step 2 – Map any attributes into columns

The mapping of attributes to columns can be broken down into four (4) distinct steps:

1. Add primary keys to tables
2. Create new tables for multivalue attributes (we’ll link them back up later)
3. Decompose any composite attributes into their constituent parts

4. Map the rest of the attributes to columns

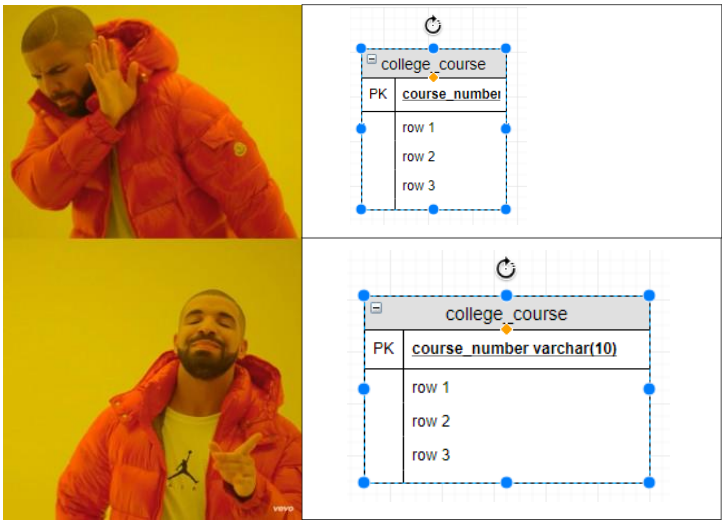
Step 2.1 – Add primary keys to tables

It is possible that we have already identified primary keys for each table. If this is the case, then we just map that attribute into the uniqueid entry in the new diagram's table. If we haven't identified a primary key yet, and if there is no sufficient attribute or set of attributes that would qualify as a primary key (see rules about primary keys elsewhere), you can use a surrogate key. Our ERD already has surrogate keys on the Section and Room entities. If we didn't specify these before in our business rules, we would add them here.

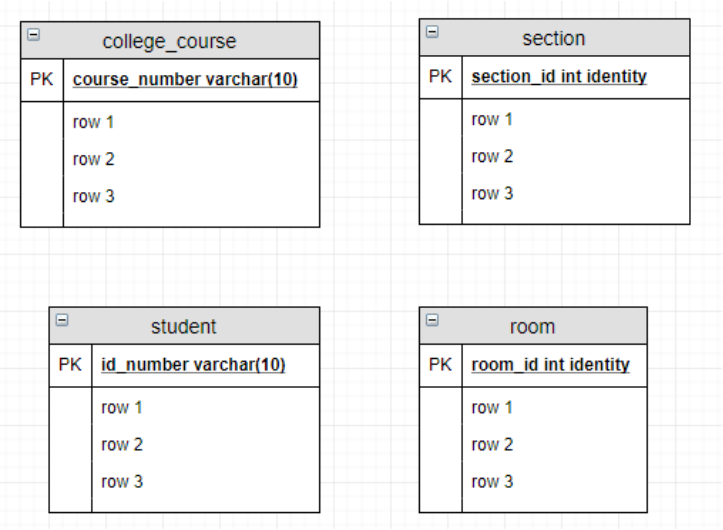
For each primary key (PK) in the ERD, change the uniqueidentifier entry in the new diagram to match using the naming conventions described above. We're also going to select data types at this point. It's best-practice to use "int identity" as the data type for any surrogate keys (for SQL Server). In a more perfect world, all primary keys would be surrogate keys, but we'll work with our business rules to make a case either way.

For our diagram, the PKs for the College Course and Student tables will have a datatype of `varchar(10).` The PKs for the Room and Section tables are surrogate keys and will get `int identity` as data types.

Some modeling tools will automatically resize each shape to fit all the text within. Draw.io doesn't do this, so when you have text that overflows and can't be read, simply resize the shape to accommodate the text. To do so, click the title of the shape once and click-and-drag the blue handles to size as needed.

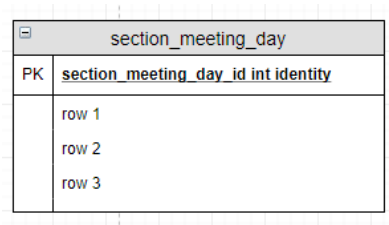


After changing your uniqueidentifier entries and resizing your shapes, your diagram should look like this:

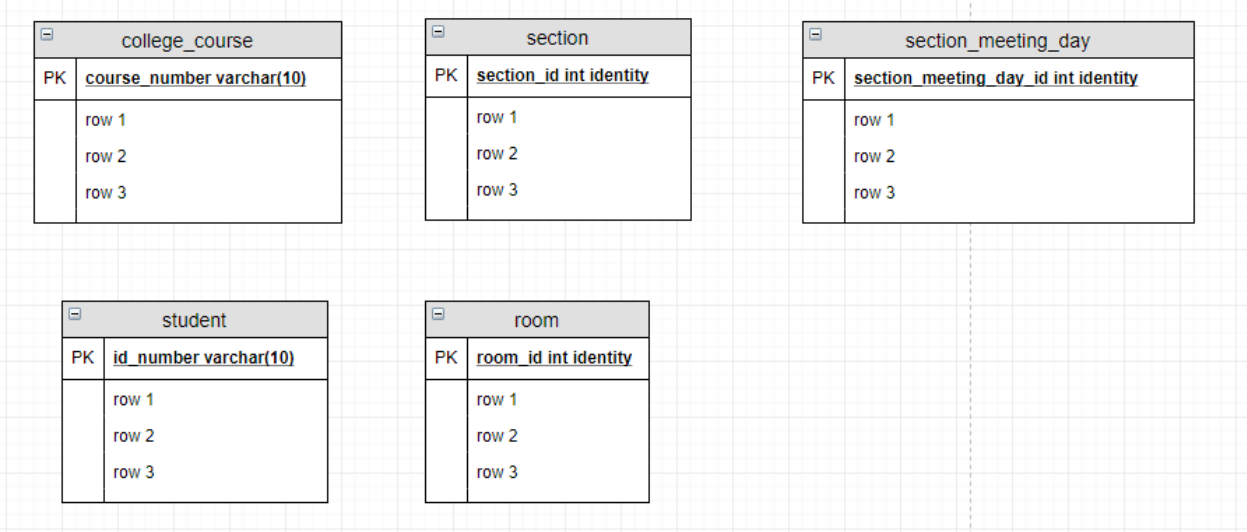


Step 2.2 - Create new tables for multivalue attributes

Because we don't have a way to store multiple values for one attribute in a table, we need to create a new table for those. In most cases, we can combine the names of the original entity and the attribute to build a good name for this new table. We will connect them later using relationships and foreign keys, but for now we'll create a new table for each. In our ERD, we have only one multivalue attribute, the meeting days attribute of the section entity. Add a new ER Table 1 to the diagram (near the section shape) and rename it to "section_meeting_day". While we're at it, let's give it a surrogate key of "section_meeting_day_id int identity". Resize the shape to accommodate the new text.



If we had more multivalue attributes, we would repeat this step for each of those. Since we do not, we can move along. Your diagram should now look like this:



Step 3 - Decompose any composite attributes into their constituent parts

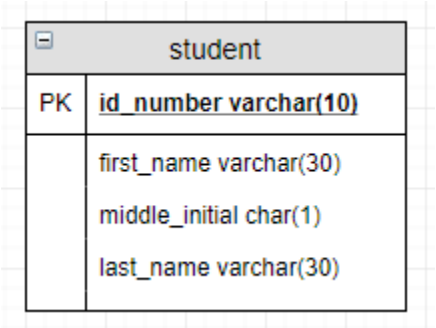
In the course of conceptual modeling, we may have identified attributes which, while in the ERD are only a single attribute, are actually composed of multiple components. For instance, in good database design, we don't want to store just "name" for a person. Instead, we want their "first_name", "last_name", and potentially other parts. For an ERD it is sufficient to say "Name [c]" to indicate it is a composite, but it is now time to break that apart into its constituent columns.

In our ERD, we have only one (1) composite attribute, the name of the student. Let's break that apart into three columns in our logical diagram: first_name, middle_initial, and last_name. We also need to pick a data type for each of these at this point. These types need not be the same for all the components, so we handle them individually. For first_name and last_name, let's use varchar(30) to allow for up to 30 alphanumeric characters. Middle initial is only one character long, so let's use char(1).
** We don't need varchar here because it's only 1-character long.*

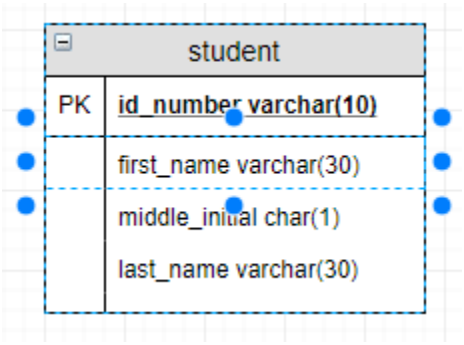
On the student shape, change the row 1, row 2, and row 3 text to

- `first_name varchar(30)`
- `middle_initial char(1)`
- `last_name varchar(30)`

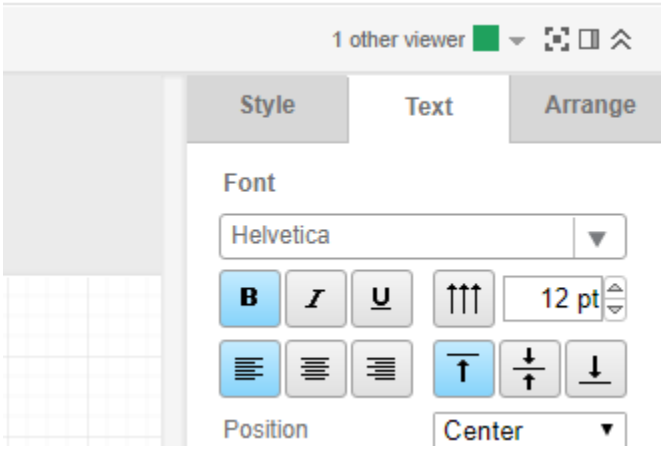
respectively.



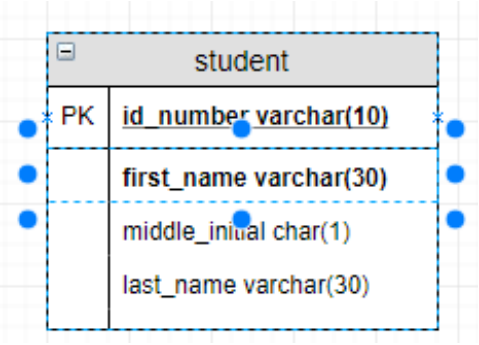
In our ERD, the Student's Name also has the "required" property, meaning all or part of this attribute must be provided when adding data (that is, each student must have a name!) we indicate this in our logical diagram by making the column name and data type **bold faced**. To do this, click on the column that should be bold to select just that part of the shape. Try it with first_name. You should click once on first_name and it should be surrounded by blue handles, indicating first_name is selected.



Next, in the properties on the right hand side of the screen, click the Text tab, and click the **B** icon below the font name.

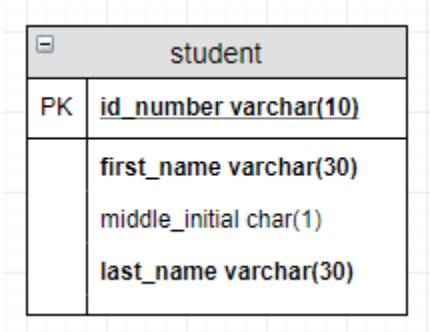


Now the text of the column is bold faced, indicating it is required.



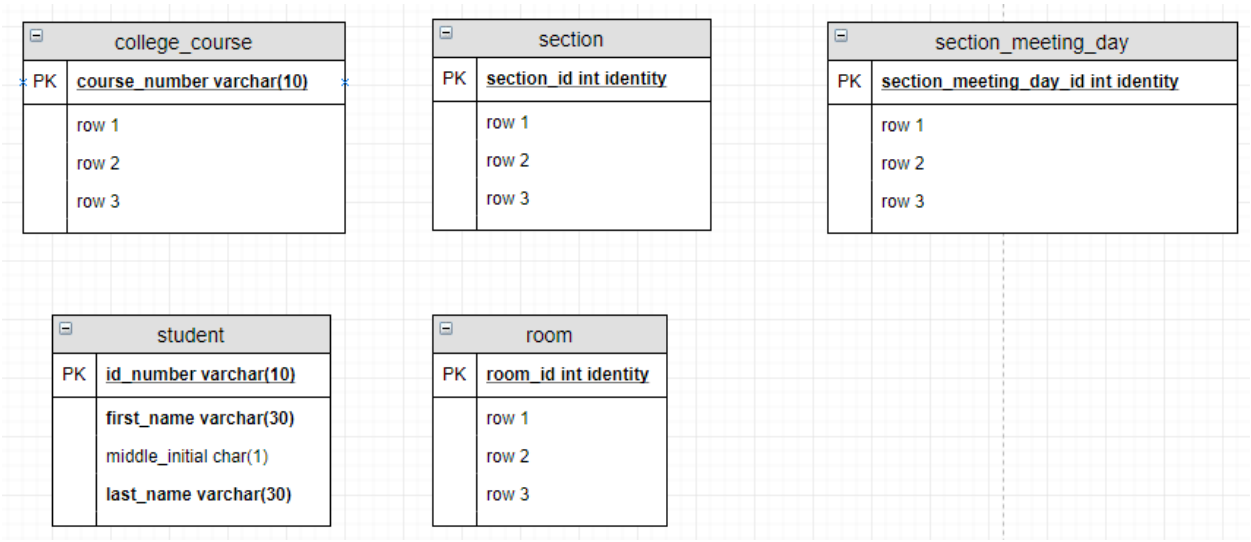
Do this to the last_name column as well. We will leave the middle_initial as is. While the narrative doesn't tell us which of these are a required component of the Name attribute, or even what the components of the Name attribute are for that matter, we can use our prior domain knowledge of

names to assume that not everyone has a middle initial, but all our students should have a first and last name. After doing this, your student table should look like this



If we had more composite attributes in our diagram, we would repeat this for each of those. Sonce we do not, we are ready to move onto step 4!

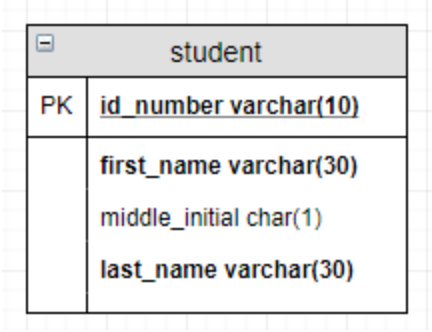
Our diagram so far:



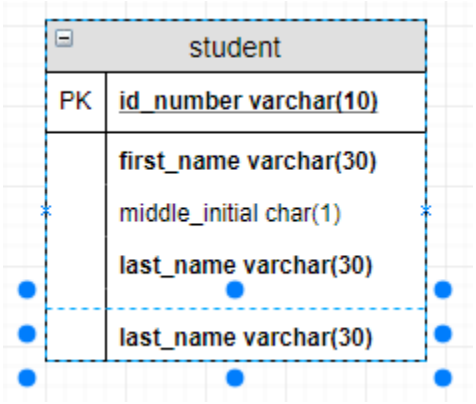
Step 2.4 - Map the rest of the attributes to columns

Having handled most of our special case attributes, (composite, multivalue, primary key), we can now move onto all the other attributes. We still have a couple special cases to handle, so let's work on them in turn.

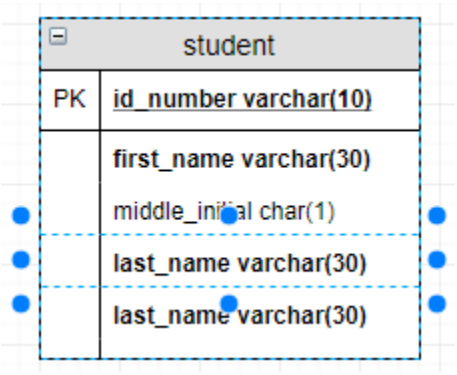
Campus Email Address is a required and unique attribute of the Student entity, so let's start there. Unfortunately, we're our of spots on our student shape:



The easiest method to add another spot is to duplicate one that's already there. Right click on the last_name column name and click Duplicate. This will add a copy of last_name to the bottom of the shape.

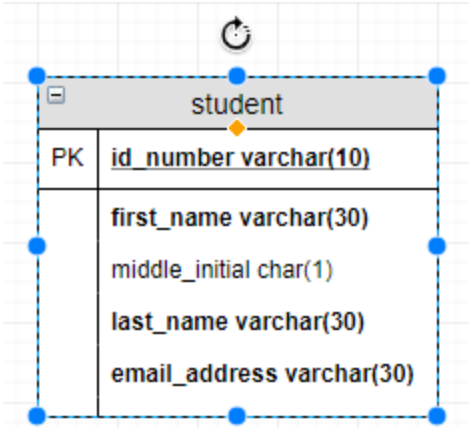


This new entry appears below the helpful spacer at the bottom of our shape, so let's move it up. With the new copy of last_name selected (click once on it if it's not. It will look like the above image when it is selected), click and drag last name so that it is above the spacer.



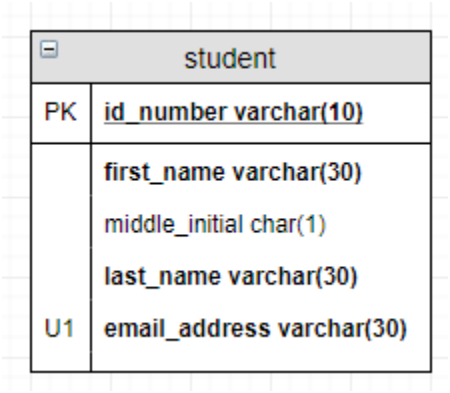
If you're new to the software, this can take some trial and error, but it will get easier with practice.

Rename the column email_address and leave the data type the same.



The email address should also be unique. To ensure this, we will need to add a Unique constraint to the column. We will cover more about the technical aspects of unique constraints in coming weeks, but for now, we can indicate that using a U1 in the space to the left of the vertical bar (under the letters PK but left of the email_address line).

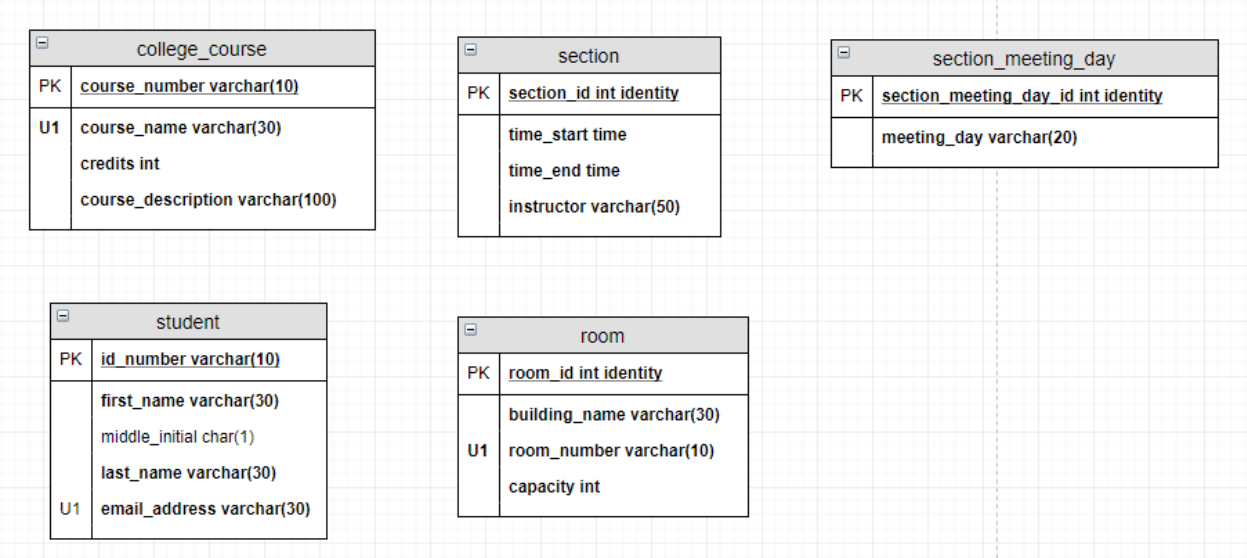
Double click in the space to the left of the email_address, but under the "PK" and you will be able to type in that box. The shape should look like this when done:



Why "U1"?

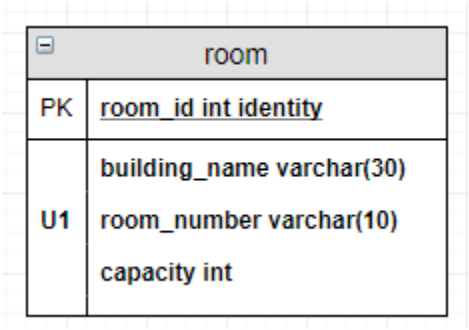
The U in U1 indicates that this is a unique constraint. Because we may have more than one such constraint in the table, we use an ordinal after the U to differentiate them. We will see the impact of this momentarily.

Continue mapping the attributes of the other entities. Use the following diagram for advice on which data type to choose. When you're finished, your diagram should look like this:



Note that when we mapped the multivalue attribute, Meeting Days, we mapped it into the new table we created in previous steps.

Let's look at the room table:



There are a couple of observations and one edit we need to make. The first is the choice of varchar as the data type for the room number. Even though it has "number" in its name, that isn't necessarily a clue to use a numeric data type. Very often, buildings will use letters in their room numbers (ie "B100" to denote room 100 in the basement). Also, many buildings will use a zero (0) in the beginning of the room number (i.e. 013 is a computer lab in Hinds Hall at Syracuse University).

In the case of the former, we will not be able to store the alphabetic portion of the number, compromising our data integrity. In the case of the latter, when storing a number with a zero (0) at the front, the software will zero-suppress the value, showing 013 as 13. Because the 0 is important to the number, we would like to ensure that it is always shown and stored with the data.

The same is true for postal codes. US postal codes, called zip codes, are numeric (if the hyphen isn't considered). Zip codes in the northeast begin with zero (0). Also, postal codes in Canada and the UK, for example, include letters.

Phone numbers, social security numbers (SSNs), and other numbers used for identification of real world things should also allow alphanumeric input as the position of each digit or groups of digits is meaningful.

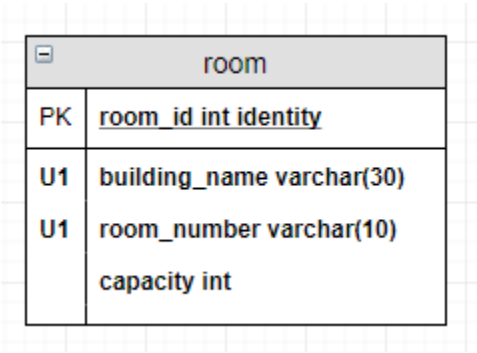
As a rule, if a column is to store numbers, but arithmetic operations on those numbers (addition, subtraction, multiplication, division, etc.) do not yield a meaningful result, err on the side of caution and use char or varchar for the data type.

The last problem with our room table is that we have made the room number unique. A closer look reveals that this can be a problem if more than one building has a room of the same number. If Building A has room 100, that will work. If we then add Building B and it has a room 100, we have a problem. We have made the room number unique, so only one building can ever have room number 100.

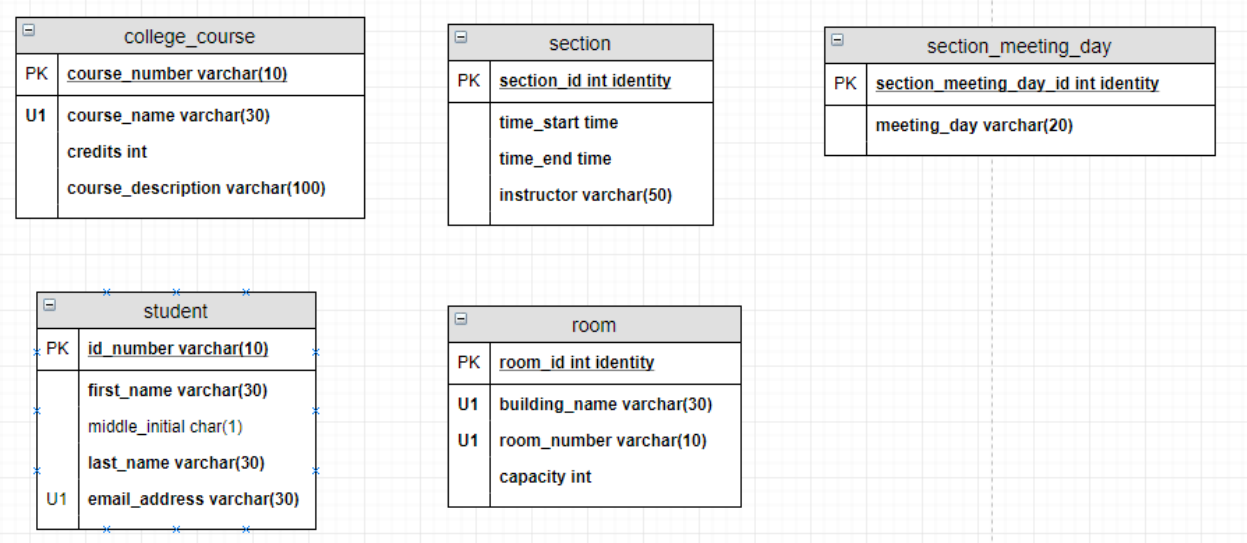
To solve this, let's look at it from a different perspective and change how we read the business rule. A room number must be unique for any given building. Building A can only ever have one room numbered 100. Building B may also have a room 100, but only once.

We can assert that the combination of building_name and room_number must be unique for all rows in room. This is called a composite unique constraint. To indicate that on the diagram, add another U1 to the left of business name.

This indicates that U1 as a constraint is defined as the combination of any column with a U1 next to it.



That wraps up our mapping of attributes. After all of that, your diagram should look like this:



Now we're ready for…

Step 3 - Map any relationships between entities as relationships between tables

Rules for mapping relationships are as follows:

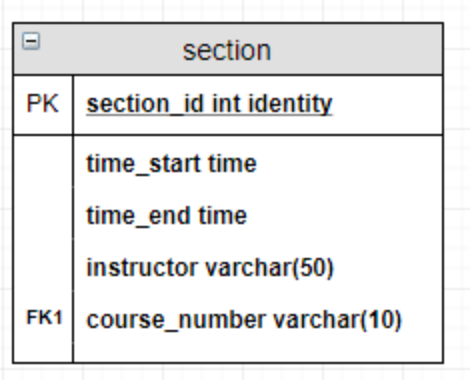
- Add a foreign key to the "many" side of each one-to-many relationship
- The cardinality of one-to-many relationships remains the same
- Relationships between tables created when mapping multivalue attributes are one of the new table to many of the original table
- Many to many relationships need to be broken up using associative tables.
- One to one relationships exist, but are rare

Start by mapping the one-to-many relationship between college_course and section. Begin by adding a new column to section named course_number with a data type of `varchar(10)`. It is probably easiest to duplicate an existing column in section (instructor is a good choice) and moving it up above the bottome spacer.

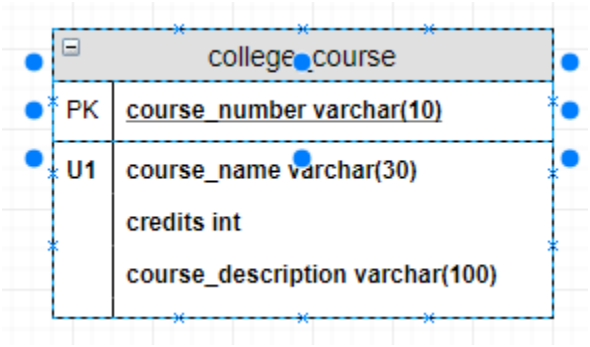
Then, in the space to the left of the column name, put FK1 to indicate that this is a Foreign Key. You may have to use the Font properties to the right side of the screen to decrease the font size to get the full FK1 to show in the box.

Like the unique constraint, we use FK to denote Foreign Key and we use an ordinal to differentiate this Foreign Key from any others on the table. While composite foreign keys are rare, they do exist in the wild.

Your section table should look like this:

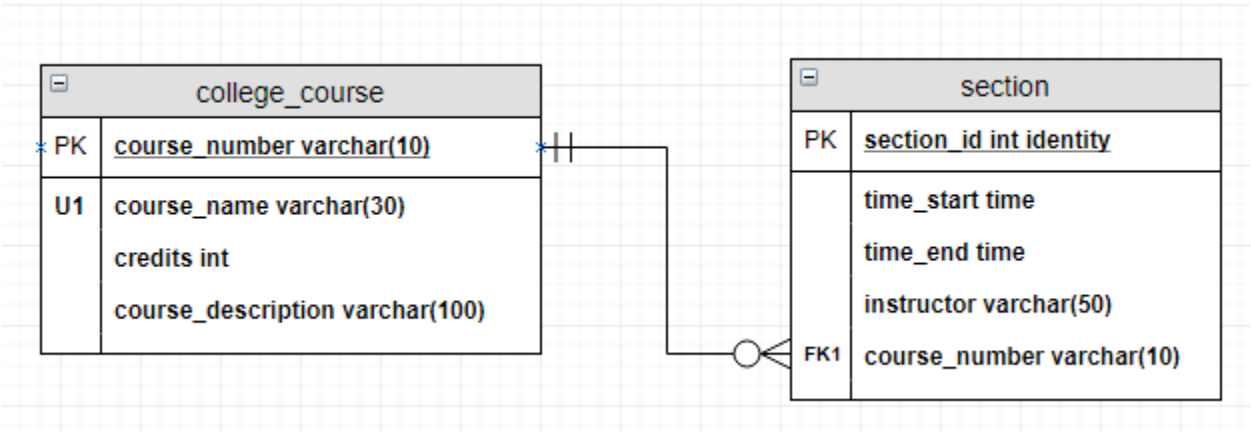


Next, select the course_number in college_course



If you then hover over the college_course column in the shape, the arrow indicators appear, much like the arrows when mapping during conceptual modeling. Drag a line between this column and the new column in section. The new column will show with a blue outline when you are in the correct spot with your cursor.

Change the line start and line ends as necessary to match the cardinality of the ERD.

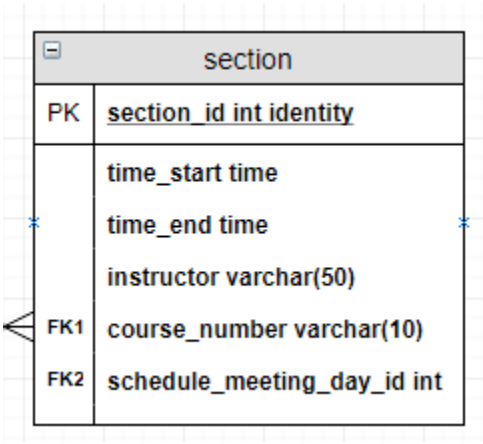


Having this level of specificity will aid in coding these tables later.

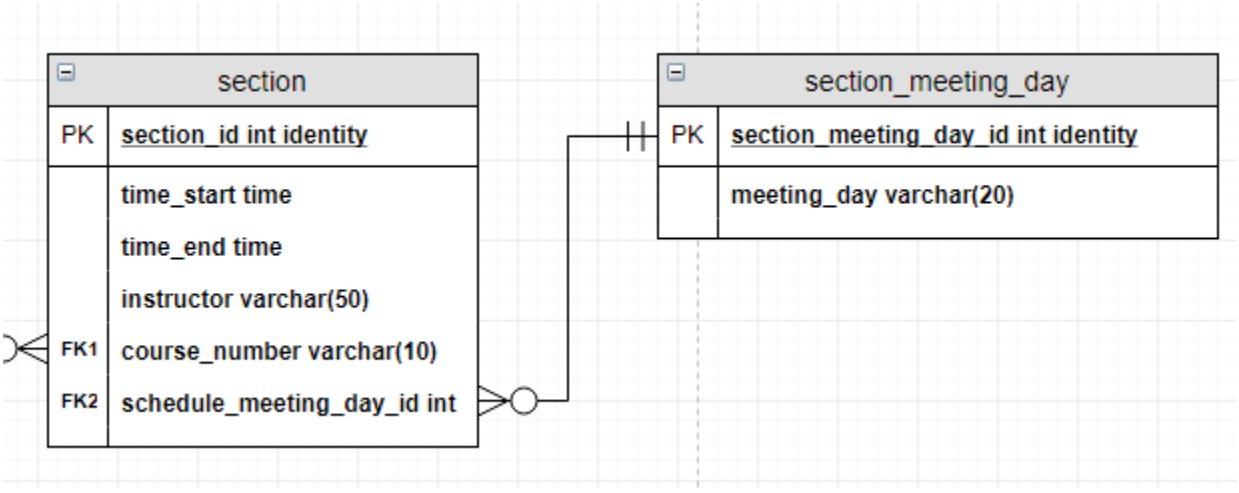
Add another column to section called schedule_meeting_day_id and make its data type `int`. Make it FK2 instead of FK1.

Wait, why not "`int identity`"? The data type of foreign keys must match their referenced primary keys, but "identity" is a property of the column in schedule_meeting_day. It is not part of the data type.

Incidentally, unlike the data type, the name of a foreign key column can be anything you like. It is a good practice to use a column name that is close to the referenced primary key to help keep things straight. Some software tools handle this automatically.

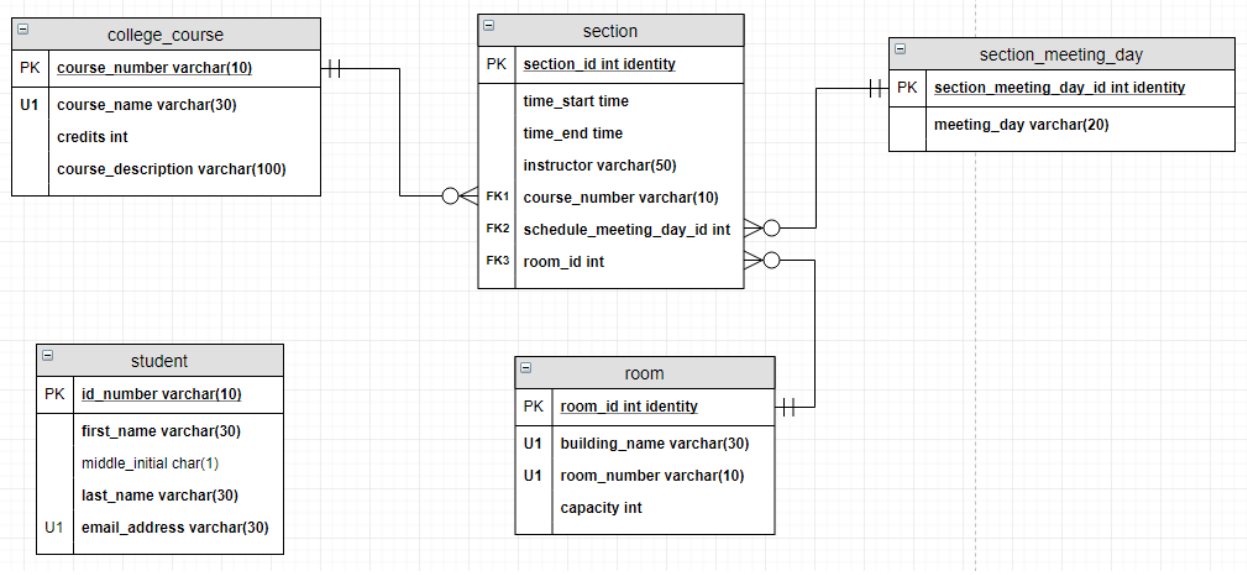


Drag a line between schedule_meeting_day_id from the schedule_meeting_day table to the new column in section. Change its cardinality to one and only one schedule_meeting_day to zero or more section.



Add a new foreign key column to section called room_id and make its data type int. Because this is a third foreign key for this table, indicate the FK as FK3. Make a one-to-many relationship between room_id of the room table and room_id of the section table.

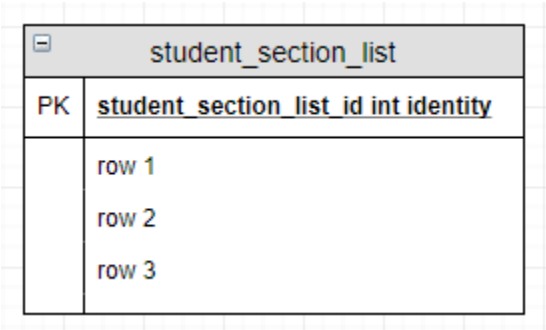
Your diagram should now look like this:



That is the process for mapping any one-to-many relationship in a logical model or E-ERD.

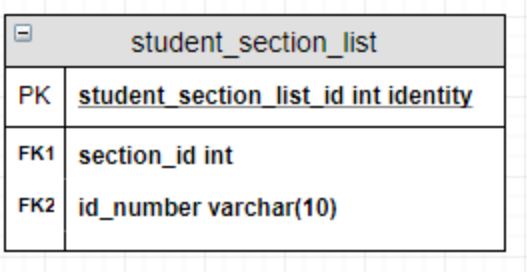
Because there is no functional way to implement a many-to-many in an RDBMS, we must use a bridge table, or technically speaking, a weak-associative table, between the participating tables in the many to many.

Begin by adding an ER Table 1 shape to the page between the student and section tables. Name this table student_section_list and give it a primary key of student_section_list_id with the int int identity data type / property. You may have to shuffle some shapes around to make space. Simply click and drag on the grey header to move a shape.

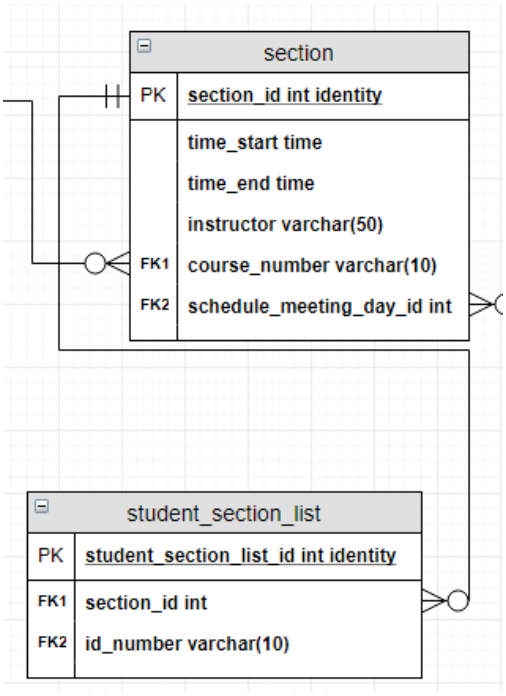


Next, add the following column names/datatypes/constraint indicators and bold-face them to make the required:

- section_id int; FK1
- id_number varchar(10); FK2

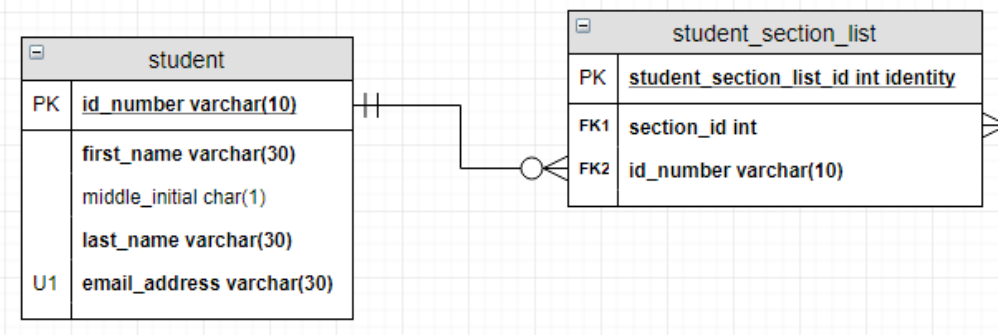


Next, draw a one to many relationship between section_id of the section table and section_id of the student_section_list table.

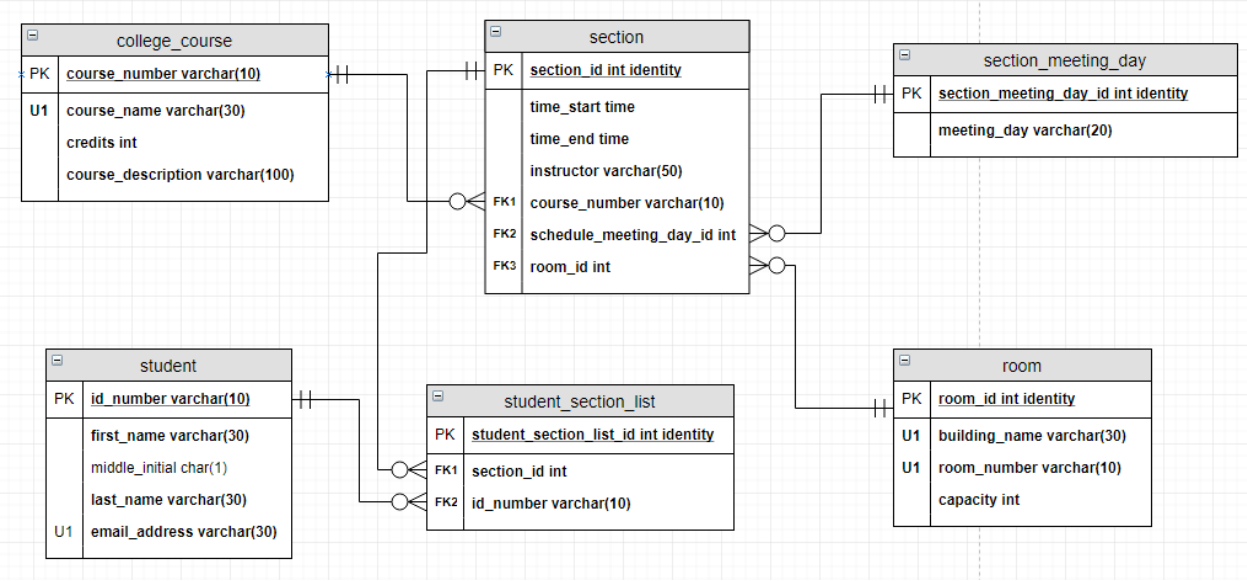


It's okay if lines cross one another at this point. We will adjust the spacing and orientation of tables later. If you do move tables and the relationship lines don't behave like you want them to, simply click on the line and adjust the path using the blue circles (handles) along the length of the line. Avoid moving the ends as those are the points that glue the shapes to the lines.

To complete this relationship, add a one to many relationship line from id_number on the student table and id_number on the student_section_list table.



So far, so good:

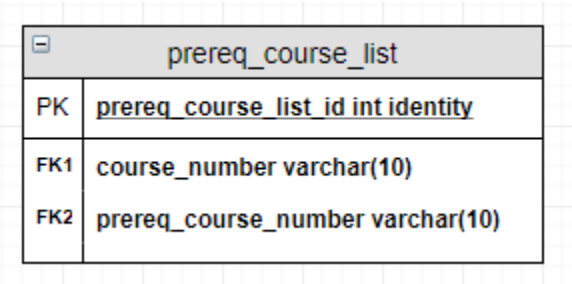


When mapping a many-to-many to a logical model diagram, add a bridge table, or weak associative table, between the participating tables and follow the rules for mapping a one to many from the original tables to this new table.

In the original ERD, College Course has a unary many-to-many relationship that represents a list of prerequisites. This is mapped in the same way as our binary many-to-many relationship, except both new relationships map to this new table.

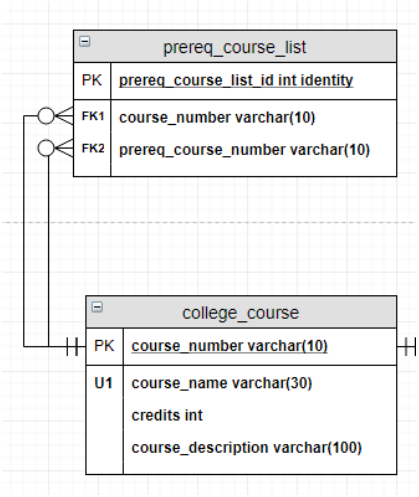
On your diagram, add a new ER Table 1 shape named prereq_course_list. Give it a surrogate key and these columns:

- course_number varchar(10) – make this FK1 and required
- prereq_course_number varchar(10) – make this FK2 and required

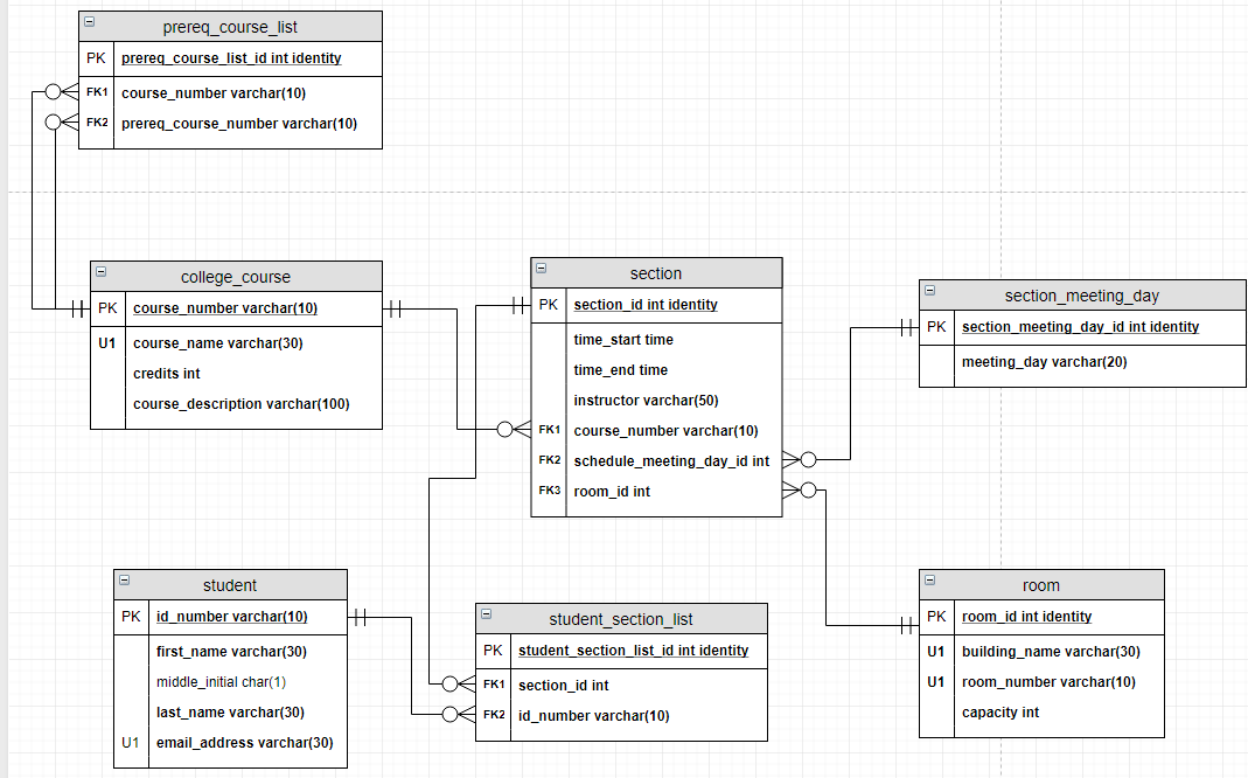


Create a one-to-many relationship between the course_number of the course table and course_number of the prereq_course_list. Create ANOTHER relationship between course_number of course and the prereq_course_number of the prereq_course_list.

What does this mean? Though the relationships both point to the same column name in the course table, it is assumed that the values in prereq_course_number and course_number of the prereq_course_list table will be different, as in different **instances** of the College Course entity from the ERD.



By now, your diagram should look like this



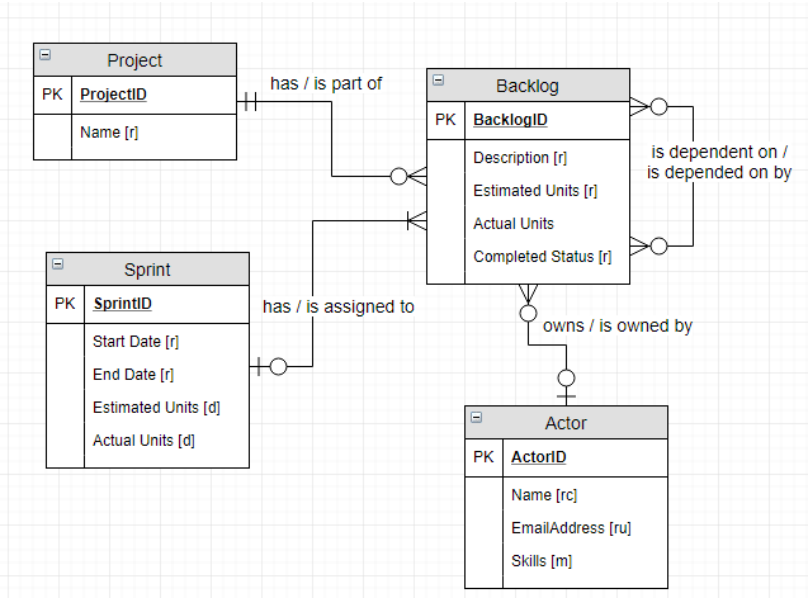
Feel free to practice shuffling things around to get a better fit, if you'd like. It is best to try one thing and then undo that action if it didn't behave as expected. Once you're satisfied with your diagram's layout, add a text box with your name in it to the diagram and export the diagram as a PNG and insert that PNG into your answer document.

Case Study 2 – Project Management

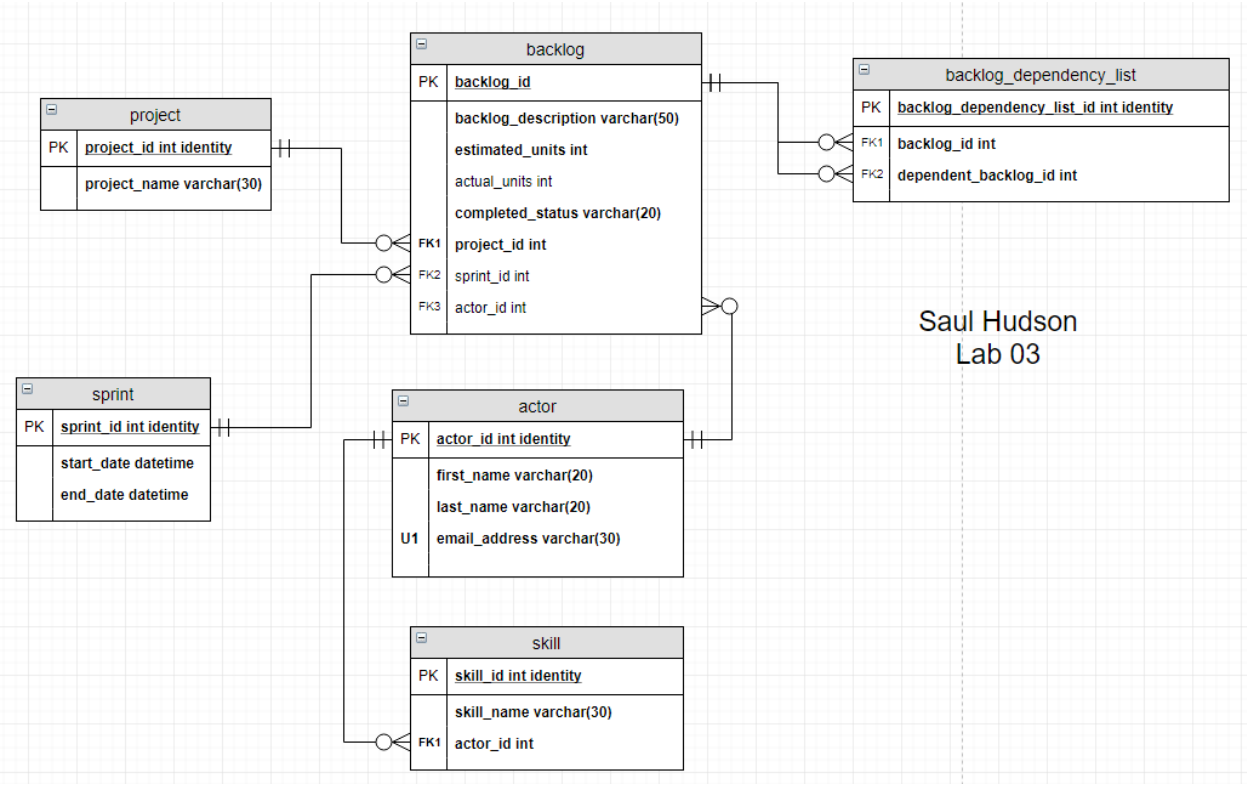
Use the same three-step process to map the project management ERD from Lab 02 into a logical model E-ERD. Add a text box with your name to the diagram.

This model includes some "optional one" to many relationships. In addition to selecting the current line-ends for those relationships, when you add the foreign keys for those relationships, do not make those required. There are also some "mandatory many" cardinalities. Simply make the line ends match.

Also, we will skip any derived attributes. We can calculate those at run time using queries (again, later). So we do not need to store them at all!



Your diagram should look like this:



Export the diagram to a PNG and add it to your answer document.

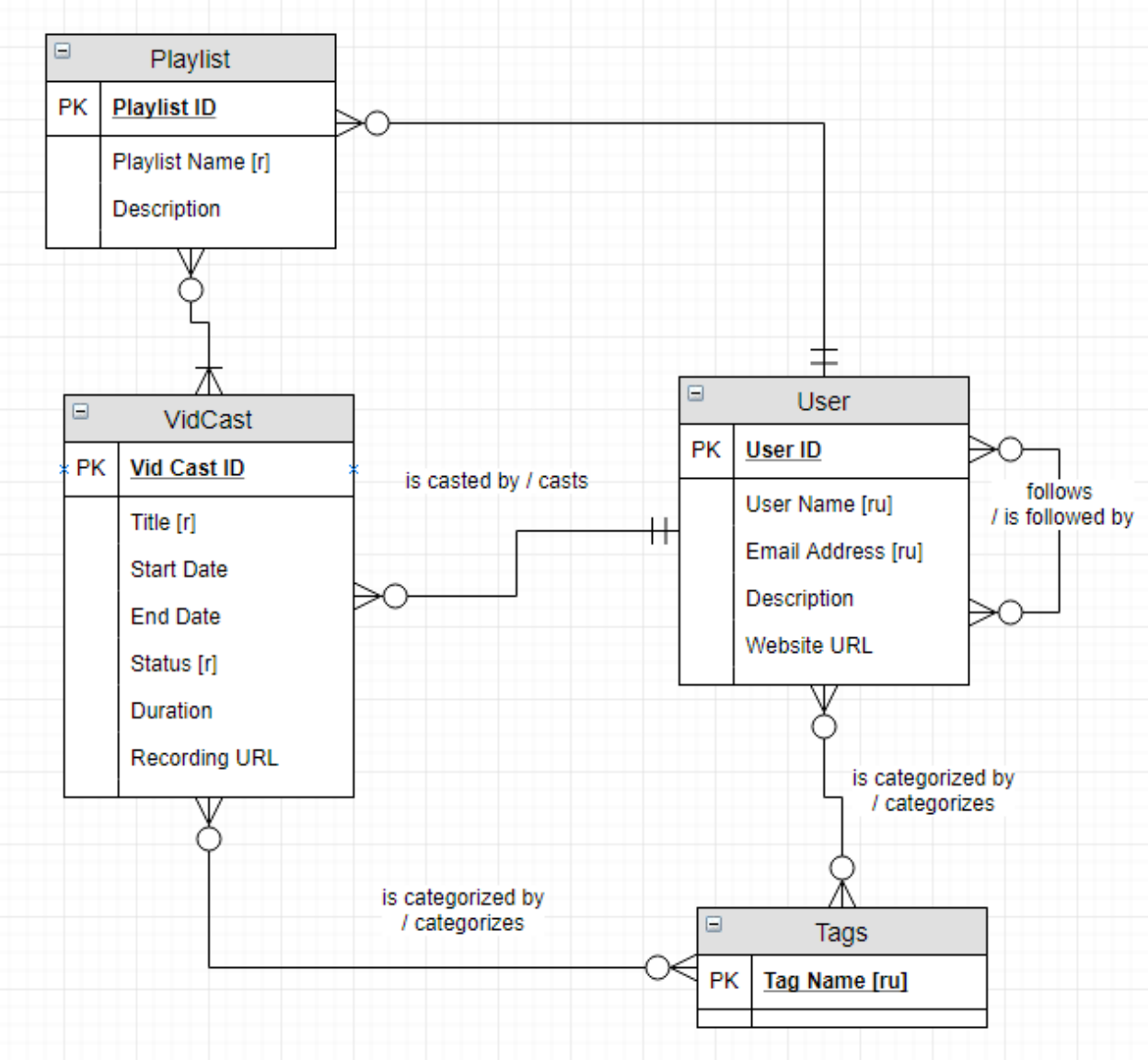
Case Study 3 – Book Publishing Database

Your turn: In Lab 02, you created an ERD for a book publishing database. Use your skills in mapping to logical model E-ERDs to create a logical model diagram for it. Use your best judgment for datatypes and such. Be sure to include a text box with your name on the diagram.

Part 2 – VidCast Logical Model

Setup

In the previous lab, we built a conceptual model ERD for our VidCast service. Since then, we have had some more discussions with the design team and have made a few modifications to the ERD. Each user can create a playlist containing one or more recorded VidCasts. A name is required for this playlist and each play list will be identified by a system generated playlist id. The new ERD looks like this:



To-do

For the above diagram, use draw.io to create a logical model E-ERD for the VidCast software. Once finished, export it as an image and place it at the end of your answer doc.

After completing Part 2, save and submit your answer doc.