

## College Preparation Program Database



By: Alex Hamilton, Adrian Ruelas, Andres Zamora

Presented to:  
Prof Nick Toothman  
CMPS 3420  
Phase 5

# Table of Contents

<b>Phase 1: Conceptual Database Design</b>	<b>2</b>
1.1 Fact-Finding/Data Collection Techniques and Information Gathering	2
1.1.1 Introduction to Enterprise/Organization	2
1.1.2 Description of Fact-Finding Techniques	2
1.1.3 Scope of Conceptual Design	3
1.1.4 Description of Entity Sets and Relationship Sets	3
1.1.5 User Groups, Data Views and Operations	5
1.2 Conceptual Database Design	5
1.2.1 Entity Type Description	5
1.2.2 Relationship Type Description	11
1.2.3 Related Entity Type	13
1.2.4 E-R Diagram (Rough Draft)	13
<b>Phase 2: From Conceptual Model to Logical Model</b>	<b>14</b>
2. Conceptual Database and Logical Database	14
2.1 E-R model and Relational model:	15
2.1.1 Descriptions of E-R model and Relational model	15
2.1.2 Comparison of the two models	16
2.2 From Conceptual Database to Logical Database	17
2.2.1 Converting Entity Types to Relations	17
2.2.2 Converting Relationship Types to Relations	18
2.2.3 Database Constraints	21
3. Convert Your E-R/Conceptual Database into a Relational/Logical Database	22
3.1 Relation Schema for Your Local Database	22
3.2 Sample Data of Relation	29
4. Sample Queries to Your Database	38
4.1 Design Of Queries	38
4.2 Relational Algebra Expressions for Queries	38
4.3 Tuple Relational Calculus Expressions for Queries	42
<b>Phase 3: Create Logical and Physical Database</b>	<b>44</b>
5. Normalization of Relations and PSQL	44
5.1 What is normalization	44
5.1.1 Definition of the Normal Forms	44
5.1.2 Anomalies	45
5.2 Check our relations	46
5.2.1 Normal forms of relations	46
5.3 Purpose and functionality of PSQL	50
5.4 PostgreSQL Schema Objects	52
5.5 Relational Schema	54
5.6 SQL Queries	61
5.7 Data Loader	62
5.7.1 Description of Data loading methods	62
5.7.2 Description of Java DataLoader Program	62
5.7.3 Implementing formulas/features to generate and import data	63
<b>Phase 4: Stored Procedures and Triggers</b>	<b>63</b>

6.1 Programming logic for SQL	63
6.1.1 Introduction to your DBMS SQL	63
6.1.2 Definitions of and usages for views, functions, procedures, and triggers	63
6.1.3 Syntax for creating views, functions, procedures, and triggers	64
6.1.4 Commands to view views, functions, procedures, and triggers	65
6.2 Implementations	65
6.2.1 Views	65
6.2.2 Stored Procedures and/or Functions	66
6.2.3 Triggers	68
6.2.4 Testing Results of Views, Procedures, and/or Functions	69
6.3 Syntax of Stored Functions and Procedures and Triggers of Two DBMS	72
6.3.1 Syntax similarities and differences between the two	72
6.3.2 Any discernible advantages/disadvantages between the two	73
<b>Phase 5: Graphical User Interface Design and Implementation</b>	<b>74</b>
7.1 GUI Application Description	74
7.1.2 Walkthrough with Pictures	74
7.2 Programming Sections	78

## Phase 1: Conceptual Database Design

### 1.1 Fact-Finding/Data Collection Techniques and Information Gathering

#### 1.1.1 Introduction to Enterprise/Organization

Our database concept will be applied to an after school college preparation program for high school students. As former high school students, between signing up for the SAT's, applying to colleges, and keeping good grades, we felt overwhelmed. We believe generally that there is a lack of guidance and resources available to students and that schools should prioritize the funding of programs that help students prepare for the future. The motivation behind this idea was to provide students an organized program to assist with taking that next step to further their education. This database will be used so the program advisors can keep track of student performance and provide them with the tools they'll need to improve their grades and academic skills. Students will be able to see their grades as well as set up meetings with their advisor, teacher, or tutor.

#### 1.1.2 Description of Fact-Finding Techniques

To learn more about what the database needs, we questioned ourselves since we're students. We asked ourselves: what kind of data do we need to see as students, what do we need to keep track of, and what do we need to do? We also questioned Andres's

sister since she is an advisor for a college readiness afterschool program. Along with questioning people to find the needs of the database we also researched other school database systems to see what's included in them. We also looked into school programs that are currently in use such as Blackboard and Canvas. Most of the data we're going to be using is mock data due to filling in names, addresses, and numbers. If this database would be implemented the actual data would be found from school records and the data would be inputted in by either an advisor or someone responsible for school records. A teacher would input grade data, students would be able to look at all their grades, courses, and GPA. Advisors would be able to see the grades of the students and be able to produce report cards. Advisors would have a primary role in viewing and manipulating data by viewing grades of the students, registering them for classes, and registering them for tutoring among other things.

### 1.1.3 Scope of Conceptual Design

Many school databases are massive due to tracking hundreds of students, their grades, courses, faculty and more. Our database will focus on a small section of this. Our database will concentrate on an afterschool program where some students have decided to sign up and a small number of faculty are helping. In the after school program we are going to be dealing with advisors, tutors, teachers, and students. Advisors, Teachers, and Tutors will be subentities of a Faculty entity. Students will have an entity as well.

### 1.1.4 Description of Entity Sets and Relationship Sets

---

#### Entity Types:

**Students** - StudentID, firstName, lastName, address, email, grade, phone, userName, password

Description: A **student** is enrolled in high school and has the essential contact needs of a high school student.

**Faculty** - FacultyID, Name, Email, Phone, userName, password

Description: The **faculty** is a superclass in which it contains the contact information of each staff member.

**Advisors** -

Description: An **advisor** is a specialization of a faculty member in which an advisor can schedule an event for the students.

**Tutors** - CourseType

Description: A **tutor** is a specialization of a faculty member in which these tutors would tutor students.

### **Teachers** - CourseType

Description: A **teacher** is a specialization of a faculty member in which they get to teach students.

### **Courses** - CourseID, TeacherName, CourseName, CourseType, CourseInfo, ClassroomNo

Description: The **course** contains the information of the name of the teacher and the information of the courses description.

### **Events** - EventID, Date, Time, ClassroomNo

Description: An **event** is superclass that an advisor schedules an event for the students. It contains the information and place and time of the events.

### **Workshops** - WorkshopName, WorkshopDescription

Description: The **workshops** is a specialization of the events. These workshops contains the name and the description.

### **Tests** - TestType

Description: The **tests** is a specialization of the events in which it contains the type of test that a student would take.

---

### Relationship Types:

#### **Relationship1 - Teaches**

Description: The teacher **teaches** the students.

Entity types: Teacher and Student

Constraint: Total for Teacher Total for Student

#### **Relationship2 - Assigns**

Description: Tutors and teachers are **assigned** to courses.

Entity Types: Tutor, Teacher, and Course

Constraint: Total for Tutor, Total for Teacher, Total for Course

#### **Relationship3 - Tutoring**

Description: Tutors **tutor** students.

Entity Types: Tutors and Student

Constraint: Total for Tutor, Partial for Student

#### **Relationship4 - Enrolled \_nto**

Description: Students are **enrolled into** a course.

Entity Types: Student and Course

Constraint: Total for Student, Total for Course

### Relationship5 - Signs\_Up

Description: Students **signs up** for tests.

Entity Types: Student and Tests

Constraint: Partial for Tests, Partial for Student

### Relationship6 - Schedules

Description: An advisor **schedules** events for students.

Entity Types: Advisor and Events

Constraint: Partial for Advisor, Total for Events

## 1.1.5 User Groups, Data Views and Operations

The groups of people who will need to access this database would-be students and advisors. Advisors will need their own view and client for important operations such as enrolling students in courses, generating reports, scheduling events, and assigning teachers or tutors to students. Advisors will be able to see all of the students, teachers, and grades for each course as well as see important dates such as SAT testing and college signup dates. When students sign in they would be able to see their grades, GPA, courses, teachers, test scores, and their assigned advisor. In addition, the student client will have different functions such as setting up appointments with advisors, teachers, and tutors.

## 1.2 Conceptual Database Design

### 1.2.1 Entity Type Description

---

#### **Students - Strong Entity**

Description: Represents the students taking part in the program and taking courses

Primary Key: StudentID

Candidate Keys: StudentID, email, username

Fields to be indexed: --

Attributes:

Name	StudentID	firstname	lastname	Address
Description	Unique identification value for students. Mostly used by faculty	Separate field for student firstname	Separate field for student lastname	Student's Street, City, Zip, State

Domain/Type	integer	varchar	varchar	varchar
Value-range	0000-9999	(50)	(50)	none
Default Value	None	None	None	None
Null Value Allowed	No	No	No	None
Unique	Yes	No	No	No
Single/Multivalued	Single	Single	Single	Single
Simple/Composite	Simple	Simple	Simple	Composite

Name	email	grade	phone	username	password
Description	Student's school email	9th, 10th, 11th, or 12th grade	Home phone or guardian phone for student	Student's login name for online program client	Student's login password for online program client
Domain/type	varchar	integer	varchar	varchar	varchar
Value-range	50, Any valid email	9-12	(10)	Any available username	Between 5-16 characters
Default value	None	None	None	None	None
Null Value Allowed?	No	No	No	No	No
Unique?	Yes	No	No	Yes	None
Single/ Multivalued	Single	Single	Single	Single	Single
Simple/ Composite	Simple	Simple	Simple	Simple	Simple

#### **Faculty-Strong Entity (Super Class)**

Description: Made up of only Teachers, Tutors, and Advisors. Each of the subclasses are disjoint from one another.

Primary Key: FacultyID

Candidate Keys: FacultyID, email, username

Fields to be indexed: --

Attributes:

Name	FacultyID	name	email	phone
Description	Unique identification value for faculty	First and last name	Faculty contact info	Office phone for faculty
Domain/Type	integer	varchar	varchar	varchar
Value-range	0000-9999	(50)	Any Valid email	(10)
Default Value	None	None	None	None
Null Value Allowed	No	No	No	No
Unique	Yes	No	Yes	No
Single/Multivalued	Single	single	Single	Single
Simple/Composite	Simple	composite	Simple	Simple

Name	username	password
Description	Faculty login name for online program client	Faculty login password for online program client
Domain/Type	varchar	varchar
Value-range	Any available username	Between 5-16 characters
Default Value	None	None
Null Value Allowed	None	No
Unique	Yes	None



Single/Multivalued	Single	single
Simple/ Composite	Simple	Simple

#### **Advisor - Weak Entity (Subclass of Faculty)**

No attributes

#### **Tutors - Weak Entity (Subclass of Faculty)**

---

Name	CourseType
Description	Subject Specialization of the Tutor
Domain/Type	varchar
Value-range	Math, english, etc
Default Value	None
Null Value Allowed	No
Unique	No
Single/Multivalued	Single
Simple/Composite	Simple

---

#### **Teachers - Weak Entity (Subclass of Faculty)**

---

Name	CourseType
Description	Subject Specialization of the Teacher
Domain/Type	varchar
Value-range	Math, english, etc
Default Value	None
Null Value Allowed	No

Unique	No
Single/Multivalued	Single
Simple/Composite	Simple

### **Courses - Strong Entity**

Description: Represents the students taking part in the program and taking courses

Primary Key: CourseID

Candidate Keys: courseID, courseName

Fields to be indexed: --

Attributes:

Name	CourseID	courseName	courseType
Description	Unique identification value for course	Name of the course	Specific Subject Type
Domain/Type	integer	varchar	varchar
Value-range	0000-9999	(50)	Math, english, etc
Default Value	None	None	None
Null Value Allowed	No	No	No
Unique	Yes	Yes	No
Single/Multivalued	Single	Single	Single
Simple/Composite	Simple	Simple	Simple

Name	courseInfo	classroomNo
Description	Description of the course	ClassroomNo
Domain/Type	varchar	integer
Value-range	(255)	000-999
Default Value	None	None

Null Value Allowed	No	No
Unique	No	No
Single/ Multivalued	Single	Single
Simple/ Composite	Simple	Simple

### **Events-Strong Entity (Super Class)**

Description: Represents the students taking part in the program and taking courses

Primary Key: EventID

Candidate Keys: EventID

Fields to be indexed: --

Attributes:

Name	EventID	date	Time	classroomNo
Description	Unique identification value for an event	Date of event	Time of event	ClassroomNo
Domain/Type	integer	date	time	integer
Value-range	0000-9999	any	any	000-999
Default Value	None	None	None	None
Null Value Allowed	No	No	No	No
Unique	Yes	No	No	No
Single/Multivalued	Single	Single	Single	Single
Simple/ Composite	Simple	Simple	Composite	Simple

### **Workshops - Weak Entity (Subclass of Events)**

Name	workshopName	workshopInfo
Description	Name of the workshop	Description of workshop

Domain/Type	varchar	varchar
Value-range	(50)	(255)
Default Value	None	None
Null Value Allowed	No	No
Unique	No	No
Single/Multivalued	Single	single
Simple/Composite	Simple	single

---

### **Tests - Weak Entity (Subclass of Events)**

Name	testName
Description	Name of Test event
Domain/Type	varchar
Value-range	SAT, ACT, AP Tests (Calc, Bio, etc)
Default Value	None
Null Value Allowed	No
Unique	No
Single/Multivalued	Single
Simple/Composite	Simple

### **1.2.2 Relationship Type Description**

---

#### **Teachers**

Description: Teachers are assigned to students based on the courses the student's taking

Entity set involved: Teacher and Student

Mapping cardinality: M:N (Many teachers teach many students)

Descriptive fields/attributes:

Participation constraint: Total for Teacher, Total for Student

### **Assigns**

Description: Tutors and teachers are assigned to courses that they are knowledgeable in by an advisor.

Entity set involved: Tutor, Teacher, Course

Mapping cardinality: 1:1:N

Descriptive fields/attributes:

Participation constraint: Total for Tutor, Total for Teacher, Total for Course

### **Tutors**

Description: Tutors help students with the course that the student is in and that the tutor is responsible for.

Entity set involved: Tutor and Student

Mapping cardinality: M:N (Students have multiple tutors that represent multiple course types)

Descriptive fields/attributes:

Participation constraint: Total for Tutor, Partial for Student

### **Enrolled\_Into**

Description: Students are enrolled into courses by and advisor based on what subjects they need to work on or what subjects they want to specialize in.

Entity set involved: Course and Student

Mapping cardinality: M:N (Students can be enrolled in many courses and courses can have many students)

Descriptive fields/attributes:

Participation constraint: Total for Course, Total for Student

### **Signs\_Up**

Description: Students see test dates and sign up for important exams such as SAT's and ACT's.

Entity set involved: Tests and Student

Mapping cardinality: (Many students can be assigned to many test) I say M:N

Descriptive fields/attributes:

Participation constraint: Partial for Tests, Partial for Student

### **Schedules**

Description: Advisors input event times, dates, and locations into the database so students can see what important events they may want to attend.

Entity set involved: Advisor and Events

Mapping cardinality: 1::N (Advisor can schedule many events, but events are scheduled by one advisor)

Descriptive fields/attributes:

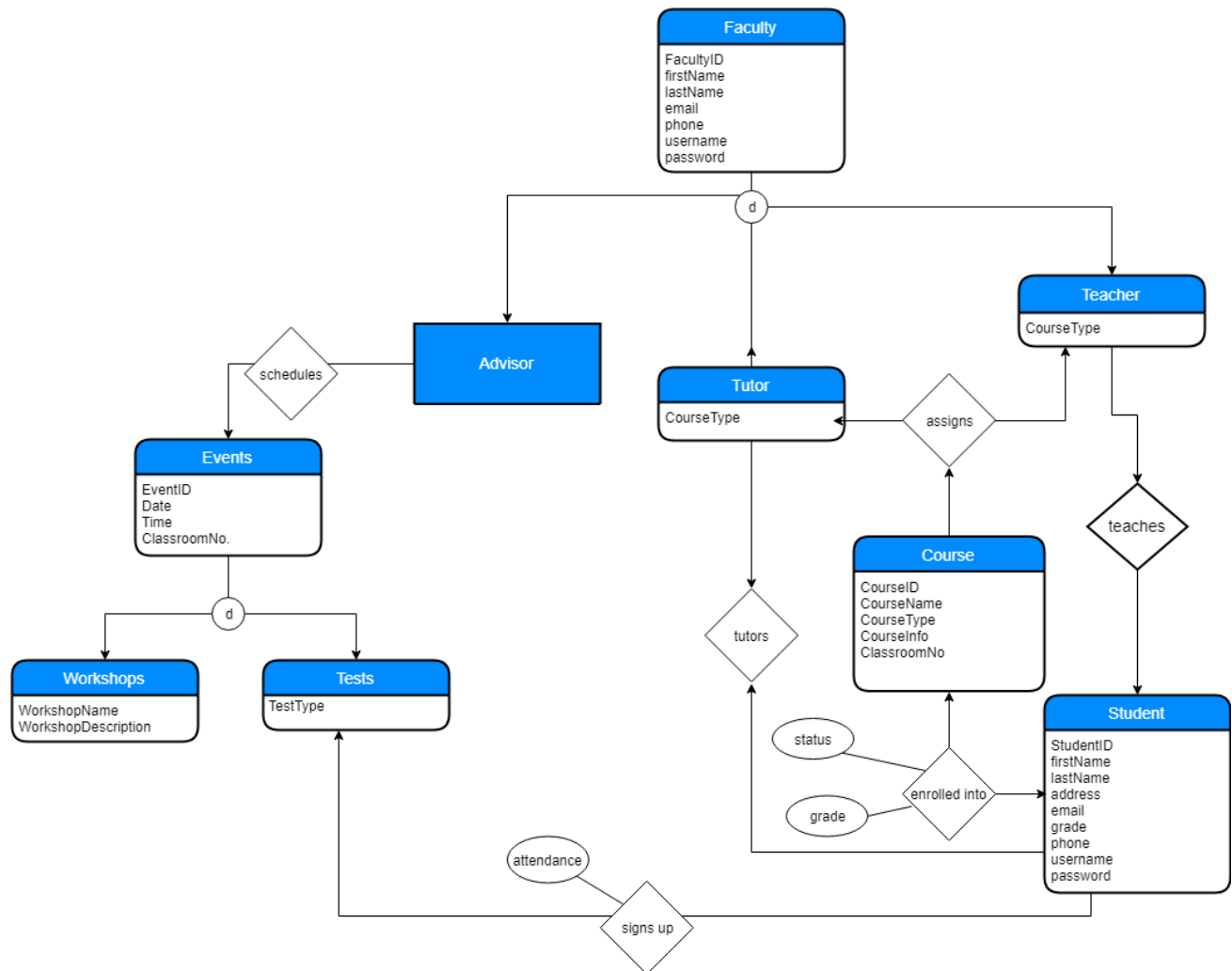
Participation constraint: Partial for Advisor, Total for Events

### 1.2.3 Related Entity Type

Specialization is when you have an entity such as Faculty, a superclass, and split it into subclasses or subtypes in which each member of the subclass not only has common traits stated in the superclass, but also more specific traits that each of the subclass members don't share with each other. Our database is the perfect example of this because Faculty acts as the superclass while teachers, tutors, and advisors are subclasses.

Generalization is essentially the opposite process of specialization where entities that share sufficient common traits can be grouped into a superclass. This leaves only the traits which differentiate them remaining in those entities. In our database the specialization of Faculty being broken up into teachers, tutors, and advisors are disjoint. This means that school faculty members, for example, cannot be both a teacher and a tutor. The participation constraint in this case would be total because we're assuming that the faculty for this after school program can only be teachers, tutors, or advisors. Remember that the scope of this database is centered only around this after school program.

### 1.2.4 E-R Diagram (Rough Draft)



## Phase 2: From Conceptual Model to Logical Model

### 2. Conceptual Database and Logical Database

In Phase I, we established our Conceptual Database (ER model). In Phase II, we aim to convert our ER model to a relational model (Logical Database). This will be done in section 2 by converting our entity types and relationship types into relations and establishing our database constraints. After establishing our Relational Model, we move on to section 3 where we need to specify the relation schema and create sample data (tuples, table body) for each relation. Finally, in section 4 we want to design 10 challenging queries for our database that can each be expressed in Relational Algebra, Tuple Relational Calculus, and Domain Relational

Calculus. These queries will be used to test our knowledge of these concepts and be useful for testing our database for the next phases.

## 2.1 E-R model and Relational model:

### 2.1.1 Descriptions of E-R model and Relational model

The first E-R diagram developed for database purposes was made by Peter Chen in the 1970s, a faculty member at Carnegie-Mellon University in Pittsburgh. Part of Peter Chen's inspiration for his ER model came from Charles Bachman who is credited with developing a type of Data Structure Diagram which is also known as the Bachman Diagram today. Chen revealed this model to the public in his well-known paper *The Entity-Relationship Model – Toward a Unified View of Data*. Because of his work, the ER model is now named an ANSI (American National Standards Institute) standard. The UML (Unified Modeling Language) language which is widely used in a lot of software design today has stemmed from the ER model and Bachman's work. Chen's ER model is also now prominent in many information modeling applications and has many uses besides database applications. ER diagrams can also be used in business information systems, education, and for research purposes. They can also be used in software engineering and serve as tools for establishing requirements for a project.

The E-R (Entity-Relationship) in database design model acts as a conceptual database. An ER model is used to map entities and relationships that are to be later converted to relations in a logical database. An entity type defines a group of entities while entities are groups that attributes can be derived from. For example, Employee is an entity type because it can specify managers who are also employees or other specific employees. The relationships in an ER model are typically labeled as verbs that describe the relationship between the entity types. ER models can also show cardinalities to specify if a relationship is one-to-one, many-to-one, or many-to-many. Strong entities in ER models are entities that have sufficient number of attributes to derive a primary key. Weak entities can't stand on their own because they lack key attributes and a unique identifier. Attributes can be simple or composite and single or multivalued. Simple attributes can't be broken up while composite attributes can be broken up into several parts. For example, address can be used as a composite attribute because you can break it up into Street address, ZIP, State. Multivalued attributes can have multiple options such as color. Simple and single attributes cannot be broken down or expanded on.

The first relational model was developed by computer scientist Edgar F. Codd at IBM. He proposed this model to be used in a database management



system in his paper *A relational model of data for large shared data banks*. Codd based his model off mathematical notation and thought that data should be grouped in relations and represented as a set of tuples. He defined a relation as a table with rows and columns while attributes would be a named column of a specific relation. We now see the relational model used in many database management systems such as SQL, MySQL, and Oracle.

In the relational model, relations or tables are entity sets where the rows (tuples) are individual entities. Each tuple is unique, and the cardinality of the relation corresponds to the number of tuples. The columns in the relational model translate to attributes which each have a unique name. Some columns have unique domains such as length limits or being restricted to characters or integers. These domain constraints specify what sort of data the attribute is allowed to contain. A relation schema contains the relation name, attributes names and domains, integrity constraints, and default values.

### 2.1.2 Comparison of the two models

#### **Advantages and Disadvantages:**

One advantage of the ER model is that it's an easy baseline to convert to other data models such as the relational model. It's also a straightforward, easy to understand representation of data. The ER model has many advantages however it does lack some necessary details such as domains, constraints, and any kind of mathematical notation, like the Relation Model, to fully represent every element in a database.

A fundamental advantage of the Relational Model is that it's backed by mathematical notation and can be explained with relational algebra and relational calculus. Another benefit to the relational model is that it is also easy to understand with the table format while also being very detailed. The relational model is also easy to maintain because of this. The downside of a relational database is that it isn't as easy to represent on paper than the ER model. It requires a higher amount of effort to initially develop and it might be harder to read than the ER model for those not familiar with the different entities and attributes in the database. It also doesn't handle composite or multivalued attributes as well as the ER model.

#### **Differences and Similarities:**

If one were to convert from the ER model to the Relational model they should notice some key differences. First the ER model represents a collection of entities and the relationships between those entities while the relational model

represents the collection of tables and the relation between those tables. The ER model describes and handles data as entity sets, relationship sets, and attributes while the relational model describes its data in tables which contain domain, constraints, attributes, and tuples. The Relational model, however, doesn't list mapping cardinalities like the ER model. Simply put, the ER model is more abstract with entities and attributes while the relational model specifically defines relations with domains, constraints, etc. which makes it considerably easier when implementing a database. The ER model, however, does serve its purpose as being a great starting tool for database design.

While both models have their differences, they also have some similarities. Relational Schema includes attributes and tuples which are the entities from the ER model. Both models have structured information. They include types and instances which describe what kind of data is stored and the purpose of every attribute (tuple).

## 2.2 From Conceptual Database to Logical Database

In this section we will focus on converting our ER model to the Relational model (logical database). This will require us to convert our entity types and relationship types into relations. To convert our entity types we need to know if they are weak/strong entities. We also need to know which attributes of each of our entities are simple/composite and single/multivalued in order to convert. For example, a multivalued attribute in an ER model is equivalent to a relation and foreign key in the Relational model. The value range of our attributes will now be known as the domain.

In order to convert our relationship types into relations we need to determine the cardinalities (one-to-many, many-to-many, etc). For example a 1:N relationship type will act as a foreign key (relationship relation) in the relational model. We next also have to potentially map "IsA" or "hasA" relationships which act as super classes and subclasses for our relationships that have attributes.

### 2.2.1 Converting Entity Types to Relations

#### **Mapping of Regular Entity Types**

First, we want to convert our strong entities into relations. Here we want to create a relation for each entity set. It can still have the same name and same set of attributes. The key of the entity set will be used as the primary key of the relation. Any other keys become candidate keys or secondary keys. Each entity's attributes should become fields of our tables with their respective domains and data types. We also only want to include the simple attribute components of composite attributes if we have any. After the relations are created from mapping

the entity types, each tuple will represent an entity instance. These are known as entity relations.

### **Mapping of Weak Entities**

The next step is to convert any weak entities into relations. We want to again create a relation for every entity type and include all simple attributes or simple components of composite attributes. Since weak entities can't stand alone, we want our relation (R) to include the primary key of the relation for the weak entity's owner as a foreign key attribute. The primary key of our relation (R) is the combination of the primary key(s) of the owner and the partial key(s) of the weak entity type.

### **Mapping of Multivalued Attributes**

To convert our multivalued attributes, we need to create a separate relation schema since there's no such thing as a multivalued attribute in the relational model. For a multivalued attribute (M) in entity set (E) create a relation R to store M, with attribute A corresponding to M where A is a single attribute version of our multivalued attribute M. The relation R will include A plus the primary key attribute of entity E. The primary key of the Relation R will include all attributes of R (Each value in M for entity E must be unique). There will be a foreign key constraint from R to E on primary key E attributes.

### **Mapping of Simple and Composite Attribute**

The relational model also doesn't handle composite attributes. The simple attribute components of the entity's composite attributes, if any, become separate attributes in the relation schema.

## **2.2.2 Converting Relationship Types to Relations**

### **Mapping of Binary 1:1 Relationship Types**

For each 1:1 relationship type R from the ER schema, identify the relations A and B that equate to the entity types participating in the relationship R. There are three different approaches for converting this relationship.

#### **Foreign key approach:**

In this approach we choose one of the relations S and include a foreign key in that Relation with a primary key of K. It's more appropriate to choose a relationship that has total participation and include all simple attributes with the 1:1 cardinality R as attributes of relation S. This is probably the most appropriate approach.

#### **Merged relation approach:**

An alternative approach is to merge the two entity types involved in the relationship into a single relation R. We can do this when we have total participation, so the two tables will have the same number of tuples.

Cross-reference approach:

The final approach involves setting up a third relation in order to cross reference the primary keys of two entity types. This is also called the relationship relation approach. It's useful if neither of the entity types have total participation.

**Mapping of Binary 1:N Relationship Types**

Foreign Key Approach:

For each 1:N relationship type identify the relation R that represents the participating entity type that is on N-side of the relationship. Include as foreign key in R the primary key of the relation S that represents the entity on the other side of the relationship type. Basically, each entity instance on the N side is related to at most one entity instance on the 1 side of the relationship type. Simple attributes of this relationship type should be included as attributes of the relation.

Cross-Reference Approach:

We can use the cross reference or relationship relation approach here again as an option. We create a third relation R again whose attributes are primary keys of both relations involved in the relationship. These will also be foreign keys to both relations. The primary key of this third relation R is the same as the primary key of one of the relations S. This approach is useful if few tuples in S participate to avoid having NULL values in the foreign key.

**Mapping of Binary M:N Relationship Types**

Foreign Key Approach:

For every M:N relationship in our ER model we want to create a new relation and include primary keys of all relations as primary keys of the new relation. (Include as foreign key attributes in the new relation the primary keys of the relations that represent entity types participating in the relationship. The combination of this will form the primary key of the new relation). Basically from the M:N relationship we end up with a new Relationship Relation with two foreign keys. Also include any simple attributes of the M:N relationship as attributes in the new relation.

Cross-Reference Approach:

We can use the cross-reference approach again when few relationship instances exist so we can avoid NULL values in foreign keys. The primary key of the

relationship relation will be the only one of the foreign keys that reference the participating entity relations.

### **Mapping of N-ary Relationship Types**

To convert N-ary relationships we need to create a relation representing the relationship. Include the primary key of each participating entity set. As a result, we should end up with a new relationship relation and n foreign keys.

### **Mapping of Specialization or Generalization**

There are multiple methods of converting or mapping a number of subclasses that together form a specialization.

#### **Multiple relations – superclass and subclasses ('IsA'):**

This method creates a relation for both the super and subclass. The superclass relation contains the attributes of the superclass entity. The subclass relation holds the attributes of the subclass entity and the primary key of the superclass as a foreign key.

#### **Multiple relations – subclass relations only:**

This method creates a relation for only subclass entities and appends (union) the superclass to each subclass. The primary key will be the super classes. This option only works when every entity in the superclass belongs to at least one of the subclasses (total). It is also only recommended if the specialization has disjointness constraint, however, if it is overlapping, the same entity can be copied in multiple relations.

#### **Single relation with one type attribute:**

In this approach one relation is created which holds the combination of attributes from superclasses and subclasses. The relation also contains a type attribute whose value indicates the subclass to which each tuple belongs. This method can be used only with disjoint specialization and if there's the possibility of generating many NULL values from specific attributes.

#### **Single relation with multiple type attributes ('HasA'):**

This option involves one relation being created that holds the combination of attributes from the superclasses and subclasses like with one type attribute. However, it also contains a Boolean attribute that indicates whether or not a tuple belongs to a subclass (based on true or false values). This method only works with overlapping subclasses specialization and disjoint specialization.

### **Mapping of Union Types (Categories)**

In order to map a category whose defining superclasses have different keys, we need to create a new key attribute called a surrogate key to correspond to the

different superclass entities. We can't use any keys exclusively to identify all entities in the category because they're unique to the defining classes. The relation tuples that coincide with the superclass entity will share the same value of the surrogate key depending on if many entities are superclasses of the same entity.

### 2.2.3 Database Constraints

Constraints are generally known as restrictions. In database context they are restrictions we put on our actual values in our relational model and database. The constraints we put on our domains and default values ensure that our input/output is correct according to what our data type is. Data types/fields also need constraints like strings and integers to ensure it only accepts the correct data type.

#### Entity constraint:

The entity integrity constraint tells us that no primary key can have a NULL value. We can't have a NULL value for our primary key because it's the unique identifier to identify individual tuples in a relation. This would mean that we couldn't identify some tuples if we had a null value. This constraint is automatically enforced by the DBMS by giving an error when you try to INSERT a NULL value into the field.

#### Primary key and unique key constraints:

A primary key constraint will ensure a unique non-NULL value and that each tuple has unique values. It also ensures that there exists only one primary key attribute per table. This is enforced by the DBMS by checking for NULL values as the primary key and checking for duplicates in the tuples.

#### Referential constraints:

These types of constraints are specified between two relations and is used to maintain consistency between tuples in the two relations. A tuple in one relation must reference an existing tuple in another relation. This means we need a foreign key to reference the primary key of the tuple in another relation. A DBMS can enforce referential either by deleting the foreign key rows to keep integrity or providing an error.

#### Check Constraints and Business Rules:

A check constraint is another integrity constraint that defines valid data when adding or updating an entry in a table. This constraint is applied to every tuple. If true the value may be inserted, otherwise it is rejected. Business rules are constraints that aren't expressed directly in the model, but rather semantic based

constraints made by the business or application-based constraints. These are explained by the business or organization that uses the database as well as enforced by the application programs. An example of one of these constraints is the salary of an employee not exceeding a certain amount because of business guidelines. These constraints are both enforced through mechanisms called triggers and assertions in the DBMS.

Domain constraints:

This specifies a certain atomic value that is allowed in each attribute. For example, if our domain is constrained to only integer values, we couldn't insert any characters, floating point numbers, Boolean values, etc. Domain constraints are important because they limit the allowed data types to be input for a certain field which ensures the correct data is being stored. A DBMS will enforce this by checking if the given input matches with the domain or value range created for the attribute.

### 3. Convert Your E-R/Conceptual Database into a Relational/Logical Database

Now, with all the knowledge we learned from the previous section, it's time to start building our relation schema. In 3.1 we'll list out each entity and relation relationship table. These tables will contain their primary keys, foreign keys, attributes, and those attribute's domain. Listed will also be the constraints on the attributes along with the constraints on the relation tables themselves.

In section 3.2 we give examples of how the tables we create from the relation schema would look like along with sample data in those tables.

#### 3.1 Relation Schema for Your Local Database

Legend
<u>PrimaryKey</u>
<u>Primary&amp;ForeignKey</u>
ForeignKey
Attribute

**Students (strong)**

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
<u>StudentID</u>	integer, 0000-9999, primary key	Unique, Not Null	Primary
firstName	varchar(50)	Not Null	No
lastName	varchar(50)	Not Null	No
streetName	varchar(50)	Not Null	No
city	varchar(50)	Not Null	No
zip	Integer(5), 00000-99999	Not Null	No
state	varchar(2), any abbreviation of USA states	Not Null	No
email	varchar(50), any valid email	Unique, Not Null	Yes
grade	Integer, 9-12	Not Null	No
phone	varchar(10)	Not Null	No
username	varchar(18), any available username	Unique, Not Null	Yes
password	varchar(16), 5 to 16 characters	Not Null	No

- **Business Constraint:** Password must be at least 5 characters long
- **Reference Constraints:** None

#### Faculty (strong)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
<u>FacultyID</u>	Integer, 0000-9999	Unique, Not Null	Primary
firstName	varchar(50)	Not Null	No
lastName	varchar(50)	Not Null	No
email	varchar(50), any valid email	Unique, Not Null	Yes



phone	varchar(10)	Not Null	No
username	varchar(25), any available username	Unique, Not Null	Yes
password	varchar(16), 5 to 16 characters	Not Null	No

- **Business Constraints:** Password must be at least 5 characters long
- **Reference Constraints:** None

### Courses (strong)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
<u>CourseID</u>	Integer, 0000-9999	Unique, Not Null	Primary
courseName	varchar(50)	Not Null	No
courseType	varchar(8), Academic_Department_Names	Not Null	No
courseInfo	varchar(255)	Not Null	No
classroomNo	Integer(3), 000-999	Not Null	No

- **Business Constraint:** CourseType must be an abbreviation of one of our Academic Departments. CourseInfo must be a detailed description of the course.
- **Reference Constraints:**

### Events (Strong)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
<u>EventID</u>	Integer, 0000-9999	Unique, Not Null	Primary
date	Date	Not Null	Composite with: ClassroomNo
sTime	Time	Not Null	No
eTime	Time	Not Null	No

classroomNo	Integer(3), 000-999	Not Null	Composite with: Date/Time
FacultyID	integer, 0000-9999	Foreign Key	No

- **Business Constraint:** ClassroomNo must be a number of a classroom that exists on our campus. sTime must be less than eTime.
- **Reference Constraints:** FacultyID must reference FacultyID from Advisor

#### Advisors (weak)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
AdvisorID	integer, 0000-9999	Unique, Not Null	Primary

- **Business Constraint:**
- **Reference Constraints:** AdvisorID must reference FacultyID from Faculty

#### Tutors (weak)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
TutorID	integer, 0000-9999	Unique, Not Null	Primary
courseType	varchar(8), Academic_Department_Names	Not Null	No

- **Business Constraint:** CourseType must be an abbreviation of one of our Academic Departments.
- **Reference Constraints:** TutorID must reference FacultyID from Faculty

#### Teachers (weak)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
TeacherID	integer, 0000-9999	Unique, Not Null	Primary
courseType	varchar(8), Academic_Department_Names	Not Null	No

- **Business Constraint:** CourseType must be an abbreviation of one of our Academic Departments.
- **Reference Constraints:** TeacherID must reference FacultyID from Faculty

#### Workshops (weak)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
EventID	Integer, 0000-9999	Unique, Not Null	Primary
workshopName	varchar(50)	Not Null	No
workshopInfo	varchar(255)	Not Null	No
FacultyID	integer, 0000-9999	Foreign Key	No

- **Business Constraint:**
- **Reference Constraints:** EventID must reference EventID from Events, FacultyID must reference AdvisorID from Advisor

#### Tests (weak)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
EventID	Integer, 0000-9999	Unique, Not Null	Primary
testName	varchar(7), Test_Type	Not Null	No
FacultyID	integer, 0000-9999	ForeignKey	No

- **Business Constraint:** TestName must be the name of one of our many Test Types, i.e. SAT, AP Bio, CST
- **Reference Constraints:** EventID must reference EventID from Events, FacultyID must reference AdvisorID from Advisor

#### Teaches (relation: M:N)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
TeacherID	integer, 0000-9999	Unique pair with: StudentID; Not Null	Composite primary

<u>StudentID</u>	integer, 0000-9999	Unique pair with: FacultyID; Not Null	Composite primary
------------------	--------------------	---------------------------------------	-------------------

- **Business Constraint:**
- **Reference Constraints:** TeacherID must reference TeacherID from Teacher, StudentID must reference StudentID from Student

#### Assigns (N:1:1)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
<u>CourseID</u>	Integer, 0000-9999	Unique, Not Null	Primary
TeacherID	integer, 0000-9999	Foreign Key	No
TutorID	integer, 0000-9999	Foreign Key	No

- **Business Constraint:**
- **Reference Constraints:** CourseID must reference CourseID from Course, TeacherID must reference FacultyID from Teacher, TutorID must reference FacultyID from Tutor

#### Tutoring(relation: M:N)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
<u>TutorID</u>	integer, 0000-9999	Unique pair with: StudentID; Not Null	Composite primary
<u>StudentID</u>	integer, 0000-9999	Unique pair with: FacultyID; Not Null	Composite primary

- **Business Constraint:**
- **Reference Constraints:** TutorID must reference TutorID from Tutor, StudentID must reference StudentID from Student

#### Enrolled\_Into (relation M:N)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
-------------	--------------	--------------	-----------------

<u>CourseID</u>	integer, 0000-9999	Unique pair with: StudentID; Not Null	Composite primary
<u>StudentID</u>	integer, 0000-9999	Unique pair with: CourseID; Not Null	Composite primary
status	varchar(11), "FAILED", "IN_PROGRESS", or "COMPLETED"	Not Null	No
grade	char, "A", "B", "C", "D", or "F"	Not Null	No

- **Business Constraint:** Status must be one of the 3 words specified. Grade must be one of the 5 grade letters specified.
- **Reference Constraints:** CourseID must reference CourseID from Course, StudentID must reference StudentID from Student

#### Signs\_Up(relation M:N)

Attributes:	Type/Domain:	Constraints:	Candidate Keys:
<u>EventID</u>	integer, 0000-9999	Unique pair with: StudentID; Not Null	Composite primary
<u>StudentID</u>	integer, 0000-9999	Unique pair with: EventID; Not Null	Composite primary
attendance	Boolean, 1 for attended, 0 for absent, null for event has not yet happened	None	No

- **Business Constraint:** Attendance must be one of the 3 values: 1 for attended, 0 for absent, or null if the event has not happened yet.
- **Reference Constraints:** EventID must reference EventID from Events, StudentID must reference StudentID from Student

## 3.2 Sample Data of Relation

Student											
studentID	firstName	lastName	streetName	city	zip	state	email	grade	phone	username	password
1	Winni	Drohan	Morningstar	Columbia	65211	MO	wdrohan0@merriam-webster.com	12	5735627163	wdrohan0	Fd5vuuq9u9q
2	Zea	Levi	Rigney	Baltimore	21229	MD	zlevi1@who.int	9	4432901908	zlevi1	EnsMqCdNWIFJ
3	Issie	Stockley	Caliangt	Silver Spring	20910	MD	istockley2@instagram.com	9	2403031020	istockley2	x0MxZVtQS
4	La verne	Kinglesyd	Haas	Washington	20397	DC	lkinglesyd3@ebay.co.uk	12	2027052024	lkinglesyd3	41H3mntbD5pD
5	Steve	Kitchenside	Magdeline	San Jose	95160	CA	skitchenside4@unc.edu	9	4082135381	skitchenside4	55l4Mv
6	Adela	Sporner	Forest	Dallas	75216	TX	asporner5@examiner.com	11	2141701176	asporner5	rvQhxZCn
7	Bayard	Forsdyke	Green Ridge	Sarasota	34276	FL	bforsdyke6@msn.com	11	9416560631	bforsdyke6	jFz0E3x
8	Bonni	Wensley	Mccormick	Cincinnati	45264	OH	bwensley7@fda.gov	10	5133932620	bwensley7	9qhfhRn
9	Chico	Biagini	Mitchell	Saginaw	48609	MI	cbiagini8@goo.gl	12	9895429996	cbiagini8	s6OC1c
10	Karine	Maciak	Knutson	Boise	83711	ID	kmaciak9@weebly.com	9	2087289481	kmaciak9	vEbp2k3vS7

Faculty						
FacultyID	firstName	lastName	email	phone	username	password
1	Rici	Walasik	rwalasik0@cisco.com	5708750994	rwalasik0	asG6BxvNs
2	Aubrette	O'Hanley	aohanley1@storify.com	9352127092	aohanley1	j2Siot
3	Happy	Sturley	hsturley2@nhs.uk	3277437058	hsturley2	ErhjeXzX4U
4	Saidee	McGinley	smcginley3@rakuten.co.jp	3059289976	smcginley3	otGI58tp
5	Lyell	Primrose	lprimrose4@theglobeandmail.com	6504498825	lprimrose4	BxISry
6	Nowell	O'Farris	nofarris5@clickbank.net	4052535915	nofarris5	w0uDZAEr
7	Dionisio	Grigoriev	dgrigoriev6@unc.edu	8124404436	dgrigoriev6	HBNq4xIPZahH
8	Den	Simoneton	dsimoneton7@sitemeter.com	8648258028	dsimoneton7	0GYKKEpmJywj
9	Bobbe	Kennifeck	bkennifeck8@squarespace.com	4113363414	bkennifeck8	R6EjvN
10	Booth	Slavin	bslavin9@virginia.edu	2399841719	bslavin9	lnENVH

Course				
CourseID	courseName	courseType	courseInfo	classroomNo
1	Math Readiness	MATH	Preparing student's for high school level math	342
2	English Radiness	ENGL	Preparing student's for high school level english	636
3	Calculus 1	MATH	The beginnings of calculus, students will learn about graphing	634
4	Calculus 2	MATH	Continuing calculus, students will learn about limits and bounds	559
5	Persuasive Writing	ENGL	Students will learn how to write compelling, persuasive papers	798
6	Intro to Programming	CMPS	Students will learn about the world of programming using C++	44
7	Geometry	MATH	Students will learn about finding the perimeter and area of different shapes	352
8	PE 1-2	PE	Student will stay in shape playing dodgeball	167
9	Typing	GENR	Students will learn how to type quickly	517
10	Creative Writing	ENGL	Students will learn how to write the stories in their heads onto paper	392

Events					
EventID	Date	sTime	eTime	classroomNo	FacultyID
1	2/10/2019	2:11	5:10	818	2504
2	5/31/2019	14:33	20:51	856	5392
3	6/19/2019	1:20	7:33	455	5355
4	5/19/2019	5:43	21:29	35	6793
5	10/18/2018	22:08	23:00	786	385
6	12/17/2018	1:54	14:02	140	764
7	2/5/2019	6:15	15:40	88	7076
8	10/30/2018	11:32	19:31	628	2870
9	5/11/2019	8:56	14:14	708	2025
10	10/18/2018	13:49	15:16	840	8642

Advisor
AdvisorID
837
1211
2648
3321
3476
4299
4576
4776
7853
8673

Tutor	
TutorID	courseType
1603	MATH
1604	MATH
1422	ENGL
4036	CMPS
6452	PE
6497	ART
7000	BUISN
7033	PHIL
8074	CHEM
9092	BIO

Teacher	
TeacherID	courseType
286	MATH
620	ENGL
1154	CMPS
2737	PE
3333	COOK
3433	MUSC
8719	HSTRY
8777	SPAN
9643	FRNCH
9792	PE



Workshops			
EventID	workshopName	workshopInfo	FacultyID
1	CSUB Fast Register	Students will meet with CSUB advisors to get registered for classes	2344
2	Healthy Eating	Learn what cheap healthy foods you can eat at home	5585
3	Modeling	Learn the secrets models use at runways	8863
4	Taking Good Photos	Meet with Photographer Smith to learn about taking better photos	504
5	Dog Training	Learn how to start training your dog at home	3908
6	Building Good Study Habits	Learn what effective study habits are right for you	1271
7	Learning About Stock	Learn how the stock market works	7449
8	Making A Website	Learn how to make a website	589
9	Dealing With Bullies	Learn about the ways you can deal with bullies	9437
10	Building Relationships	Learn how to build life long relationships with people	7812

Tests		
EventID	testName	FacultyID
4368	SAT	3564
9819	CST	5540
1513	AP Bio2	4845
9696	AP Calc1	5819
225	SAT	680
2330	SAT	2013
5721	CST	5329
1763	GDAL	678
2047	ACT	1170
8378	ACT	7347

Teaches	
TeacherID	StudentID
5166	3454
7736	7612
5231	1441
6597	3394
8467	2171
9616	9517
2501	3991
796	8827
5160	6204
3903	1860
8785	8519
2734	3784
1977	8680
9984	5893
8180	8093
8949	7881
3353	8157
6980	4555
728	9865
4273	9046
2094	8633
7092	2054
4897	7186
2832	1041
3484	8876
4301	169
6888	169
6685	3407
8681	7993
8785	3309
462	4220
2417	644
2549	3691
3935	9941
7734	2193
7985	393
7434	3061
2752	6618
1884	8275
6890	6247

Assigns		
CourseID	TeacherID	TutorID
1	7134	4666
2	5046	6144
3	4545	2030
4	6865	4387
5	5337	8389
6	6162	8694
7	6098	8848
8	4088	680
9	4217	7946
10	5323	9417
11	5286	9223
12	6634	5226
13	3113	8155
14	4453	2107
15	9948	7920
16	8715	2039
17	6650	1880
18	6405	966
19	7178	7394
20	9998	9966
21	9182	399
22	8249	3459
23	7692	794
24	3654	6124
25	4168	2036
26	8366	7682
27	551	7893
28	2925	1729
29	6480	1038
30	8042	7080
31	1466	8904
32	8342	6880
33	2199	3723
34	2289	7816
35	4794	166
36	8020	1512
37	2467	6636
38	3898	4926
39	4096	8910
40	920	3921

Tutors	
TutorID	StudentID
8698	2683
7819	2005
1969	2040
9809	1152
701	4892
8737	3085
8221	4653
3043	814
4396	1999
5080	3722
8696	3004
8246	7372
2523	1593
1311	8261
106	7799
9738	6357
4358	7276
8501	5060
6972	3746
9970	7954
4153	8381
5678	5450
1042	4928
6211	7096
9753	2028
9779	9767
2934	340
4771	4331
8993	9451
6263	9874
7750	9265
9346	5610
9008	7477
6513	1534
7315	6379
1161	8583
7923	3850
1204	8602
9221	9429
4841	386

Enrolled Into			
CourselD	StudentID	status	grade
2839	257	IN PROGRES	B
146	8534	FAILED	D
856	9279	FAILED	D
1907	7768	COMPLETED	C
7027	9564	COMPLETED	B
9453	450	IN PROGRES	F
5331	6210	IN PROGRES	D
564	9872	FAILED	F
8651	4326	COMPLETED	F
527	4009	IN PROGRES	B
3069	7593	COMPLETED	C
7845	1487	COMPLETED	A
2571	9268	COMPLETED	B
1719	5803	COMPLETED	A
3007	6472	IN PROGRES	D
7207	541	COMPLETED	B
3391	2998	IN PROGRES	C
3772	758	COMPLETED	F
5279	1747	COMPLETED	F
1685	1981	COMPLETED	A
7667	2648	COMPLETED	C
3146	1079	IN PROGRES	A
1606	940	COMPLETED	B
7912	4049	COMPLETED	C
5266	9313	IN PROGRES	A
8573	3960	FAILED	F
7482	6647	IN PROGRES	B
2185	4190	COMPLETED	B
4157	5170	IN PROGRES	D
8704	3272	COMPLETED	A
2607	2740	IN PROGRES	F
8512	3264	COMPLETED	D
3204	4054	FAILED	F
3584	3653	COMPLETED	C
9800	8813	COMPLETED	C
5104	8756	FAILED	F
4996	3421	COMPLETED	B
5572	5545	IN PROGRES	D
6028	6416	COMPLETED	A
7263	3750	IN PROGRES	A

Signs Up		
EventID	StudentID	attendance
9019	1632	1
9188	9120	1
3073	3845	NULL
5913	5810	1
1727	6694	1
7327	9001	0
7467	6762	NULL
7023	2732	0
8522	4179	NULL
707	6415	0
845	8518	1
5122	2180	1
3961	2147	1
2625	4737	NULL
1954	1955	NULL
8379	3931	1
5619	6021	1
8034	1833	NULL
4855	4150	0
7799	2683	NULL
6357	9650	0
5565	4787	NULL
5842	6787	NULL
9250	4793	1
6666	7586	0
307	4845	0
7527	9146	1
5141	6542	NULL
9582	4210	NULL
555	3502	NULL
1821	2716	0
4552	1875	0
8547	3424	1
9945	5646	NULL
277	8865	1
7532	414	0
4662	2938	NULL
6450	187	1
9355	124	0
5217	2798	0



## 4. Sample Queries to Your Database

In this section, we want to test our relational database by developing 10 challenging queries that require the use of different operations from Relational Algebra and Tuple Relational Calculus. These queries will be designed to test the bounds of our database and ensure the conversion process from ER to relational was successful.

### 4.1 Design Of Queries

Relational algebra is a formal language used to describe the basic operations used in relational databases. These operations are used to manipulate the database and produce tables of data. These tables of data are relevant to whatever data a user could be looking for.

- 1. List all students who are enrolled in AP Calculus who currently signed up for the AP Calculus Exam.**
- 2. List all Workshops Names that were scheduled during 9/1/2019-12/17/2019.**
- 3. List all events that John Doe has signed up for during 9/1/2019-12/17/2019.**
- 4. List juniors who are taking the tutor named Amber Carroll.**
- 5. Find all events that start at 2pm and the names of the advisors who created them, except those created by advisor Joe Mama.**
- 6. List all tutor names that student John Doe has been assigned.**
- 7. List all junior students with a grade “C” in any English course.**
- 8. Retrieve the names of students who are enrolled into all the courses of student Joe Smith.**
- 9. List the names of students who have a “completed” course status and received an “A” grade in math courses.**
- 10. Find the event with the highest student attendance**

### 4.2 Relational Algebra Expressions for Queries

Relational algebra is the basic set of operations for the relational model. Relational algebra expressions form a sequence of relational algebra operations that represents

the results of a database query. Relational algebra is procedural so the order of operators is important.

Operators:  $\pi$ ,  $\sigma$ ,  $\rho$ ,  $\times$ ,  $\bowtie$ ,  $*$ ,  $\div$ ,  $\wedge$ ,  $\vee$ ,  $\neg$

1. List all students who are enrolled in Calculus 1 who currently signed up for the AP Calculus 1 Exam.

```
T1 ← Enrolled Into  $\bowtie$  Course  
           CourseID = CourseID  
T2 ←  $\sigma$  (T1.Status = "In Progress"  $\wedge$  T1.CourseName = "Ap Calculus" ) (T1)  
T3 ←  $\pi$  (CourseID, CourseName, Status, StudentID) (T2)  
S1 ← Signs up  $\bowtie$  Tests  
           EventID = EventID  
S2 ← T3  $\bowtie$  S1  
           StudentID = StudentID  
R ←  $\sigma$  (testName = "AP Calculus") (S2)
```

---

2. List all Workshops Names that were scheduled during 9/1/2019-12/17/2019.

```
T1 ← Workshops * Events  
           EventID = EventID  
T2 ←  $\sigma$  (Date > 9/1/2019  $\wedge$  Date < 12/17/2019) (T1)  
R ←  $\pi$  (WorkshopNames, Date) (T2)
```

---

3. List all events that John Doe has signed up for during 9/1/2019-12/17/2019.

```
T1 ← Student * Signs Up  
           StudentID = StudentID  
T2 ← Events * T1  
           EventID = EventID  
T3 ←  $\sigma$  (Date > 9/1/2019  $\wedge$  Date < 12/17/2019  $\wedge$  firstName = "John"  $\wedge$  lastName = Doe) (T2)  
R ←  $\pi$  (EventID, firstName, lastName, Date) (T3)
```

---



4. List juniors who are taking the tutor named Amber Carroll.

**T1 ← Student \* Tutors**

**StudentID = StudentID**

**T2 ←  $\rho$ (sFirstName, sLastName, FacultyID)  $\pi$  (firstName, lastName, TutorID) (T1)**

**T3 ← Faculty \* T2**

**FacultyID = FacultyID**

**T4 ←  $\rho$ (tFirstName, tLastName)  $\pi$  (firstName, lastName) (T3)**

**T5 ←  $\sigma$  (tFirstName= "Amber"  $\wedge$  tLastName = "Carrol"  $\wedge$  grade= "11") (T4)**

**R ←  $\pi$  (tFirstName, TLastName, grade, sFirstName, sLastname) (T5)**

---

5. Find all events that start at 2pm and the names of the advisors who created them, except those created by advisor Joe Mama.

**T1 ← Events \* Faculty**

**FacultyID = FacultyID**

**T2 ←  $\sigma$  (firstName= "Joe"  $\wedge$  lastName = "Mama") (T1)**

**T3 ← T1 - T2**

**T4 ←  $\sigma$  (sTime= "14:00") (T3)**

**R ←  $\pi$  (EventID, sTime, FacultyID, firstName, lastName) (T4)**

---

6. List all tutor names that student John Doe has been assigned.

**T1 ←  $\sigma$  (firstName= "John"  $\wedge$  lastName = "Doe") (Student)**

**T2 ← Tutors \* T1**

**FacultyID = FacultyID**

**T3 ←  $\rho$ (sFirstName, sLastName, FacultyID)  $\pi$  (firstName, lastName, TutorID) (T2)**

**T4 ← Tutor \* T3**

**FacultyID = TutorID**

**R ←  $\pi$  (FacultyID, firstName, lastName) (T4)**

---

7. List all junior students with a grade “C” in any English course.

**T1**  $\leftarrow$  **Student**  $\bowtie$  **Enrolled Into**  
                    **StudentID**= **StudentID**  
**T2**  $\leftarrow$   $\rho$ (sGrade, cGrade)  $\pi$  (student.grade, Enrolled Into.grade) (**T1**)  
**T3**  $\leftarrow$  **Course**  $\bowtie$  **T2**  
                    **CourseID**= **CourseID**  
**T4**  $\leftarrow$   $\sigma$  (sGrade = “11”  $\wedge$  cGrade = “C”  $\wedge$  CourseType = “ENGL”) (**T3**)  
**R**  $\leftarrow$   $\pi$  (StudentID, sGrade, cGrade) (**T4**)

---

8. Retrieve the names of students whos enrolled into all the courses of student Joe Smith.

**J1**  $\leftarrow$   $\sigma$  (firstName = “Joe”  $\wedge$  lastName = “Smith” ) (**Student**)  
**J2**  $\leftarrow$  **Enrolled Into** \* **J1**  
                    **CourseID** = **CourseID**  
**J3**  $\leftarrow$   $\pi$  (CourseID) (**J3**)  
**T1**  $\leftarrow$  **Enrolled Into** \* **Student**  
                    **CourseID** = **CourseID**  
**T2**  $\leftarrow$  **T1** - **J2**  
**T3**  $\leftarrow$   $\pi$  (CourseID, firstName, lastName) (**T2**)  
**R**  $\leftarrow$  **T3**  $\div$  **J3**

---

9. List the names of students who have a “completed” course status and received an “A” grade in math courses.

**T1**  $\leftarrow$  **Enrolled Into** \* **Course**  
                    **CourseID** = **CourseID**  
**T2**  $\leftarrow$   $\sigma$  (CourseType = “Math”  $\wedge$  status= “COMPLETED” ) (**T1**)  
**T3**  $\leftarrow$   $\pi$  (CourseType, status, grade) (**T2**)  
**S1**  $\leftarrow$  **Student** \* **T3**  
                    **StudentID** = **StudentID**

$$R \leftarrow \pi (\text{StudentID, firstName, lastName}) (S2)$$
$$R \leftarrow \pi (\text{EventID}) (S1)$$

3. List all events that John Doe has signed up for during 9/1/2019-12/17/2019.

**{e\* | Event(e) ^ (∃)(Student(s) ^ signsUp(si)) ^ s.studentID = si.studentID ^ (∃)(Events(e)) and e.eventID = si.eventID ^ e.date > 9/1/2018 ^ e.date < 12/17/2018 ^ s.firstname = "John" ^ s.lastname = "Doe"}**

4. List junior students who are taking the tutor named Amber Carroll.

**{s | Student(s) ^ s.grade = 11 ^ ((∃ t)(Tutors(t) ^ t.StudentID = s.StudentID ^ ((∃ f)(Faculty(f) ^ f.FacultyID = t.TutorID ^ f.firstName = 'Amber' ^ f.lastName = 'Carrol'))))}**

5. Find all events that start at 2pm and the names of the advisors who created them, except those created by advisor Joe Mama.

**{e | Events(e) AND ((∃ f)(Faculty(f) AND e.FacultyID = f.FacultyID AND e.sTime = 1400 AND ((∀ f) NOT f.firstName = 'Joe' AND f.lastName = 'Mama'))}**

6. List all tutor names that student John Doe has been assigned.

**{f.firstName, f.lastName | Faculty(f) AND ((∃ t)Tutors(t) AND f.FacultyID = t.TutorID AND (∃ s)Student(s) AND s.StudentID = t.StudentID AND s.firstName = 'John' AND s.lastName = 'Doe')}**

7. List all junior students with a grade "C" in any English course.

**{s | Student(s) AND s.grade = 11 AND ((∃ e)EnrolledInto(e) AND s.StudentID = e.StudentID AND e.grade = 'C' AND (∃ c)Course(c) AND e.CourseID = c.CourseID AND c.CourseType = 'ENGL')}**

8. Retrieve the names of students who are enrolled into all the courses of student Joe Smith.

**{s.firstName, s.lastName | Student(s) ^ NOT (s.firstName = "Joe" ^ s.lastName = "Smith") ^ ((∀ e)EnrolledInto(e) ^ e.StudentID = s.StudentID ^ s.firstName = "Joe" ^ s.lastName = "Smith") }**

9. List the names of students who have a "completed" course status and received an "A" grade in math courses.

**{s.firstName, s.lastName | Student(s) AND (( $\exists$  e)EnrolledInto(e) AND s.StudentID = e.StudentID AND e.status = 'COMPLETED' AND e.grade = 'A' AND ( $\exists$  c)Course(c) AND c.CourseID = e.CourseID AND c.CourseType = 'MATH')}}}**

10. Find the event with the highest student attendance

**{e | Events(e) AND ( $\exists$  s)SignsUp(s) AND e.EventID = s.EventID AND...**

## Phase 3: Create Logical and Physical Database

### 5. Normalization of Relations and PSQL

In this phase we will work on normalizing our relations by following the three normal forms and checking for any anomalies. We will also be using PostgreSQL as our relational database management software to inject our newly made relational database from phase 2. Finally we will show the schema for each table and convert our queries into SQL. We will show and provide the results for each query to demonstrate the effectiveness of our database.

#### 5.1 What is normalization

Around 1970 Edgar F. Codd introduced the normalization process for relational schema in databases. Normalization is the process of analyzing relational schema and seeing if it satisfies a specific normal form. This is done by analyzing each relation and testing it against the criteria for normal forms. If the relation schemas don't meet the criteria and normal form tests, they are broken down into smaller relation schemas that do meet criteria and possess the properties needed. Codd proposed three normal forms (first, second, and third). Codd later developed, along with Raymond Boyce, the Boyce-Codd normal form, an extension of the third normal form. The goal with normalization is to eliminate redundant data to minimize storage space and update anomalies in order to obtain unambiguous and intended results.

##### 5.1.1 Definition of the Normal Forms

###### **First Normal Form**

The first normal form holds the criteria for a basic relational model. The first normal form states that the domain of an attribute must include only simple or atomic values (non-composite). It also states that the domain of an attribute cannot be multivalued, but only single valued. The relation

should also not have nested relations. In order to remedy this, one must form new relations for each multivalued attribute or nested relation.

### **Second Normal Form**

The Second Normal Form is based on full functional dependency which states with a functional dependency of  $X \rightarrow Y$ , if removal of any attribute  $A$  from  $X$  means that the dependency does not hold any more; that is, for any attribute  $A \in X$ ,  $(X - \{A\})$  does not functionally determine  $Y$ . This means for relations where the primary key contains multiple attributes, no nonkey attribute should be functionally dependent on a part of the primary key. To normalize these relations, one should decompose and set up a new relation for each partial key with its dependent attribute(s).

### **Third Normal Form**

The Third normal form is based on transitive dependency which states if there exists a set of attributes  $Z$  in  $R$  that is neither a candidate key nor a subset of any key of  $R$ , and both  $X \rightarrow Z$  and  $Z \rightarrow Y$  hold. This means relations should not have a nonkey attribute functionally determined by another nonkey attribute. There should be no transitive dependency of a nonkey attribute on the primary key. In order to remedy this, one must decompose and set up a relation that includes nonkey attributes that functionally determines other nonkey attributes.

### **Boyce Codd Normal Form**

Boyce Codd Normal Form is an extension of the third normal form but more strict. A relational schema is BCNF if whenever a nontrivial functional dependency  $X \rightarrow A$  holds in  $R$ , then  $X$  is a superkey of  $R$ . All BCNF relations are also 3NF, but not all 3NF relations are BCNF.

## **5.1.2 Anomalies**

Anomalies are issues in relations that have yet to be normalized. An update anomaly is a data inconsistency that results from data redundancy and a partial update. Update anomalies can be broken down into different categories: insertion anomalies, deletion anomalies, and modification anomalies.

### **INSERT Anomalies**

Many times you may need to use Insertion anomalies for a badly designed relation. To insert a new tuple for a relation it must include attribute values in which the said attribute value has to have the presence of other attribute values. However, it can also contain NULL when the presence is either not applicable or is unknown. A second insertion anomaly situation is to have a tuple with all NULL values. This occurs when the given situation is not applicable or unknown.

### **DELETE Anomalies**

If a deletion anomaly occurs for a tuple that is related to another relation then the information for the said tuple would be lost. For example if we dropped a tutorid from tutors that facultyid would be dropped in the faculty table.

### **Modification Anomalies**

In a relation where a tuple gets updated its other relations that are related to it must also be changed to emphasize the updated tuple. For example if we changed the last name for a teacher we would have to update it in faculty beforehand. This is done to ensure consistency between tables.

## **5.2 Check our relations**

We will check our relations to see if it is in the third-normal-form or Boyce-Codd Normal Form.

### **5.2.1 Normal forms of relations**

#### **Students-**

##### **Functional Dependencies:**

FD1- Trivial; {**StudentID**} → {firstName, lastName, streetName, city, zip, state, email, grade, phone, username, password}

FD2- {email} → {**StudentID**, firstName, lastName, streetName, city, zip, state, grade, phone, username, password}

FD3- {username} → {**StudentID**, firstName, lastName, streetName, city, zip, state, email, grade, phone, password}

##### **Candidate Keys:**

**StudentID**, email, username

Normal Forms:

1NF is satisfied because all of the attributes have atomic domains.

**Faculty-**

Functional Dependencies:

FD1- Trivial; {**FacultyID**} → {firstName, lastName, email, phone, username, password}

FD2- {email} → {**FacultyID**, firstName, lastName, phone, username, password}

FD3- {username} → {**FacultyID**, firstName, lastName, email, phone, password}

Candidate Keys:

**FacultyID**, email, username

Normal Forms:

**Courses-**

Functional Dependencies:

FD1- Trivial; {**CourseID**} → {courseName, courseType, courseInfo, classroomNo}

Candidate Keys:

**CourseID**

Normal Forms:

**Events-**

Functional Dependencies:

FD1- Trivial; {**EventID**} → {date, sTime, eTime, classroom, facultyID}

Candidate Keys:

**EventID**, **FacultyID**, date, classroomNo

Normal Forms:

**Advisors-**

Functional Dependencies:



FD1- Trivial; {**AdvisorsID**} → {firstName, lastName, email, phone, username, password}

FD2- {email} → {**AdvisorsID**, firstName, lastName, phone, username, password}

FD3- {username} → {**AdvisorsID**, firstName, lastName, email, phone, password}

Candidate Keys:

**AdvisorsID**, email, username

Normal Forms:

### **Tutors-**

Functional Dependencies:

FD1- Trivial; {**TutorID**} → {couretype}

Candidate Keys:

**TutorsID**

Normal Forms:

### **Teachers-**

Functional Dependencies:

FD1- Trivial; {**TeacherID**} → {courseType}

Candidate Keys:

**TeacherID**

Normal Forms:

### **Workshops-**

Functional Dependencies:

FD1- Trivial; {**EventID**} → {workshopName, workshopInfo, FacultyID}

Candidate Keys:

**EventID**

Normal Forms:

### **Tests-**

#### Functional Dependencies:

FD1- Trivial; {**EventID**}  $\rightarrow$  {testName, FacultyID}

#### Candidate Keys:

**EventID**

#### Normal Forms:

### **Teaches-**

#### Functional Dependencies:

FD1- Trivial; {**TeacherID**}  $\rightarrow$  {**StudentID**}

FD1- {**StudentID**}  $\rightarrow$  {**StudentID**}

#### Candidate Keys:

**TeacherID, StudentID**

#### Normal Forms:

### **Assigns-**

#### Functional Dependencies:

FD1- Trivial; {**CourseID**}  $\rightarrow$  {TeacherID, TutorID}

#### Candidate Keys:

**CourseID**

#### Normal Forms:

### **Tutoring-**

#### Functional Dependencies:

FD1- Trivial; {**TutorID**}  $\rightarrow$  {**StudentID**}

FD1- {**StudentID**}  $\rightarrow$  {**TutorID**}

#### Candidate Keys:

**TutorsID, StudentID**

Normal Forms:

### **Enrolled\_Into-**

Functional Dependencies:

FD1- Trivial; {**CourseID, StudentID**} → {status, grade}

Candidate Keys:

**CourseID, StudentID**

Normal Forms:

### **Signs\_Up-**

Functional Dependencies:

FD1- Trivial; {**EventID, StudentID**} → {attendance}

Candidate Keys:

**EventsID, StudentID,**

Normal Forms:

## 5.3 Purpose and functionality of PSQL

### **What is PSQL?**

PostgreSQL or psql is an open source database management system that uses the SQL language to hold and organize a lot of data for a database project. PostgreSQL has been known for its reliability, data integrity and expansive features. Since it's open source the community continues to build upon postgres in order to expand its capabilities and to make it an overall exceptional platform for database management. PostgreSQL runs on all the major operating systems, so it will be convenient tool to use for our database projects.

### **History of PostgreSQL**

Postgres was originally created by Michael Stonebraker, a computer science professor at UC Berkeley. Project Postgres was however inspired heavily first by a project named INGRES. INGRES turned out to be a proof-of-concept for relational databases which would later develop into a company called Ingres

in the 1980s. Later however Ingres was bought by a company called Computer Associates Inc. in 1994 or CA Technologies which is what it is known as today.

The original Postgres development lasted from 1986 to 1994 and used INGRES to develop concepts, object orientation and the query language rather than using the original code as the basis. By the end of this development cycle Postgres was commercialized and later bought by IBM. In 1994 SQL support was added to Postgres and Postgres95 was released with this the following year. In 1996 Postgres was re-released as PostgreSQL 6.0 to reflect its new innovations with the support of SQL. Since then the community and volunteers known as The PostgreSQL Global Development Group have maintained the software and have made continuous innovations over time.

### **Features/Functionality**

PostgreSQL includes many features that will help us create tables for our database including useful data types, data integrity, concurrency or performance, reliability or disaster recovery, security, and extensibility.

### **Data Types**

PostgreSQL has extensive library of datatypes that we can use to describe each one our attributes. There are primitives which includes integers, numeric, string, Boolean types. Structured data types which can be either date/time, array, range, UUID. Document data types such as JSON or XML. It also has customizations of attributes such as composite or custom types.

### **Data Integrity**

PostgreSQL also has measures to keep data integrity by specifying UNIQUE or NOT NULL, establishing primary keys/foreign keys, listing constraints for each table, and having explicit locks.

### **Concurrency, Performance**

PsqI supports B-trees, expression, and multicolumn indexing. It also has a useful query planner, and table partitioning. PsqI also supports Just-in time compilation of expressions which is a method of compilation during run time rather than before execution of the program.

### **Reliability**

PsqI supports Write-ahead Logging (WAL) which is a family of techniques for providing atomicity and durability in database systems. It also includes point-in-time-recovery which is where an administrator can restore or recover a set of data or a particular setting from a time in the past.

### **Security**

PsqI uses a robust access-control system, column and row-level security, and different methods of Authentication such as GSSAPI, SSPI, LDAP.

### **Extensibility**

PostgreSQL supports stored functions and procedures which will play a heavy part of our back-end work. The languages that are supported to write and execute these stored procedures include Java, Python, Perl etc. Another extensibility feature to note are foreign data wrappers which give one the ability to connect to other databases or streams with a SQL UI

## 5.4 PostgreSQL Schema Objects

### **Table**

Tables in PSQL are fundamental database objects that let us store our data in a row/column format. Each column represents a single attribute while each row can contain data types to describe the domain of the attribute and whether or not the attribute is null/not null.

Syntax:

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    column3 datatype,
    .....
    columnN datatype,
    PRIMARY KEY( one or more columns )
);
```

### **View**

Views are virtual tables or pseudo-tables that are based on a stored query that gets data from already existing base tables. A view can even represent joined tables and contain all rows of a table or selected rows from one or more tables.

Syntax:

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
```

```
WHERE [condition];
```

## **Indexes**

Indexes are special lookup tables that the database can use to quickly access rows and speed up data retrieval. An index can help queries, that are selected, perform more efficiently. However, it does slow down input, such as UPDATE and INSERT commands. Indexes also have multiple types such as single-column, multicolumn, unique, partial, and implicit indexes.

Syntax:

```
CREATE INDEX index_name ON table_name;
```

## **Sequence**

The sequence schema object creates a new sequence number generator that creates and initializes a single row table. Each time the sequence is requested it responds with the next number in the sequence. This can help generate unique primary keys automatically and move these keys across different rows and tables. The sequence object can have many parameters such as increment, minvalue, maxvalue, start, cache, cycle and owned by.

Syntax:

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ]  
name [ INCREMENT [ BY ] increment ]  
      [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE  
maxvalue | NO MAXVALUE ]  
      [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ]  
CYCLE ]  
      [ OWNED BY { table_name.column_name | NONE } ]
```

## **Triggers**

Triggers are callback functions in the database that are invoked or “triggered” when a specific event occurs based on the logic given. A trigger can be invoked before the operation is attempted on a row, after the operation has completed or instead of the operation.

Syntax:

```
CREATE TRIGGER trigger_name [BEFORE|AFTER|INSTEAD OF]
event_name
ON table_name
[
  -- Trigger logic goes here....
];
```

## 5.5 Relational Schema

The schema and contents of each relation will be given

Advisors:

```
ahamilto=> \d advisors
      Table "public.advisors"
      Column | Type | Modifiers
      -----+-----+-----
      advisorid | integer | not null
Indexes:
    "advisors_pkey" PRIMARY KEY, btree (advisorid)
Foreign-key constraints:
    "fk_advisors_advisorsid" FOREIGN KEY (advisorid) REFERENCES faculty(facultyid) ON DELETE CASCADE
Referenced by:
    TABLE "events" CONSTRAINT "fk_events_advisorid" FOREIGN KEY (advisorid) REFERENCES advisors(advisorid) ON DELETE SET NULL
    TABLE "tests" CONSTRAINT "fk_tests_advisorid" FOREIGN KEY (advisorid) REFERENCES advisors(advisorid) ON DELETE SET NULL
    TABLE "workshops" CONSTRAINT "fk_workshops_advisorid" FOREIGN KEY (advisorid) REFERENCES advisors(advisorid) ON DELETE SET NULL
```

```
LINE 1: SELECT *FROM advisors
          ^
ahamilto=> SELECT *FROM advisors;
 advisorid
-----
         1
         3
         7
(3 rows)
```

Assigns:

```
ahamilto=> \d assigns
      Table "public.assigns"
      Column | Type | Modifiers
      -----+-----+-----
      courseid | integer | not null
      teacherid | integer |
      tutorid | integer |
Indexes:
    "assigns_pkey" PRIMARY KEY, btree (courseid)
Foreign-key constraints:
    "fk_assigns_courseid" FOREIGN KEY (courseid) REFERENCES courses(courseid) ON DELETE CASCADE
    "fk_assigns_teacherid" FOREIGN KEY (teacherid) REFERENCES teachers(teacherid) ON DELETE SET NULL
    "fk_assigns_tutorid" FOREIGN KEY (tutorid) REFERENCES tutors(tutorid) ON DELETE SET NULL
```

```
ahamilto=> SELECT *FROM assigns;
 courseid | teacherid | tutorid
-----+-----+-----
         1 |          4 |        2
         2 |          8 |        5
         3 |          4 |        2
         4 |          4 |        2
         5 |          8 |        5
         6 |         10 |        6
         7 |          4 |        2
         8 |          9 |
         9 |         10 |        6
        10 |          8 |        5
(10 rows)
```

## Courses:

```
ahamilton=> \d courses

Table "public.courses"
  Column      |      Type      |      Modifiers
-----+-----+-----
 courseid    | integer        | not null default nextval('courses_courseid_seq'::regclass)
 coursename  | character varying(50) | not null
 coursetype  | academic_department_names | not null
 courseinfo  | character varying(255) | not null
 classroomno | integer        | not null

Indexes:
  "courses_pkey" PRIMARY KEY, btree (courseid)
Check constraints:
  "chk_courses_classroomno" CHECK (classroomno >= 0 AND classroomno <= 999)
  "chk_courses_courseid" CHECK (courseid >= 0 AND courseid <= 9999)
Referenced by:
  TABLE "assigns" CONSTRAINT "fk_assigns_courseid" FOREIGN KEY (courseid) REFERENCES courses(courseid) ON DELETE CASCADE
  TABLE "enrolled_into" CONSTRAINT "fk_enrolled_into_courseid" FOREIGN KEY (courseid) REFERENCES courses(courseid)
```

```
ahamilton=> SELECT * FROM courses;
 courseid | coursename | coursetype | courseinfo | classroomno
-----+-----+-----+-----+-----
 1 | Math Readiness | MATH | Preparing student's for high school level math | 342
 2 | English Readiness | ENGL | Preparing students for high school level english | 636
 3 | Calculous 1 | MATH | The beginnings of calculous, students will learn about graphing | 634
 4 | Calculous 2 | MATH | Continuing calculous, students will learn about limits and bounds | 559
 5 | Persuasive Writing | ENGL | Students will learn how to write compelling, persuasive papers | 798
 6 | Intro to Programming | CMPS | Students will learn about the world of programming using C++ | 44
 7 | Geometry | MATH | Students will learn about finding the perimeter and area of different shapes | 352
 8 | PE 1-2 | PE | Student will stay in shape playing dodgeball | 167
 9 | Typing | CMPS | Students will learn how to type quickly | 517
10 | Creative Writing | ENGL | Students will learn how to write the stories in their heads onto paper | 392
(10 rows)
```

## Enrolled Into:

```
ahamilton=> \d enrolled_into

Table "public.enrolled_into"
  Column      |      Type      |      Modifiers
-----+-----+-----
 courseid    | integer        | not null
 studentid   | integer        | not null
 status      | class_status   | not null
 grade       | class_grade    | not null

Indexes:
  "enrolled_into_pkey" PRIMARY KEY, btree (courseid, studentid)
Foreign-key constraints:
  "fk_enrolled_into_courseid" FOREIGN KEY (courseid) REFERENCES courses(courseid)
  "fk_enrolled_into_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE
```

```
ahamilton=> SELECT *FROM enrolled_into;
 courseid | studentid | status | grade
-----+-----+-----+-----
 1 | 2 | COMPLETED | A
 1 | 5 | FAILED | F
 1 | 7 | IN_PROGRESS | C
 2 | 1 | COMPLETED | C
 2 | 2 | COMPLETED | B
 2 | 5 | FAILED | F
 3 | 8 | FAILED | F
 3 | 9 | IN_PROGRESS | F
 3 | 10 | IN_PROGRESS |
 4 | 2 | IN_PROGRESS | B
 4 | 3 | IN_PROGRESS | A
 5 | 4 | COMPLETED | A
 5 | 5 | COMPLETED | A
 5 | 6 | COMPLETED | D
 6 | 7 | FAILED | F
 6 | 8 | IN_PROGRESS | A
 7 | 9 | IN_PROGRESS |
 7 | 2 | FAILED | F
 8 | 1 | FAILED | F
 8 | 10 | IN_PROGRESS | A
 9 | 10 | COMPLETED | A
10 | 1 | COMPLETED | A
10 | 3 | FAILED | F
 9 | 3 | IN_PROGRESS | C
 4 | 4 | IN_PROGRESS | D
(25 rows)
```

## Events:



```

ahamilton=> \d events

```

Column	Type	Table "public.events"	Modifiers
eventid	integer	not null	default nextval('events_eventid_seq'::regclass)
date	date	not null	
stime	time without time zone	not null	
etime	time without time zone	not null	
classroomno	integer	not null	
advisorid	integer	not null	

```

Indexes:
    "events_pkey" PRIMARY KEY, btree (eventid)
Check constraints:
    "chk_events_etime" CHECK (etime > stime)
Foreign-key constraints:
    "fk_events_advisorid" FOREIGN KEY (advisorid) REFERENCES advisors(advisorid) ON DELETE SET NULL
Referenced by:
    TABLE "tests" CONSTRAINT "fk_tests_eventid" FOREIGN KEY (eventid) REFERENCES events(eventid) ON DELETE CASCADE
    TABLE "workshops" CONSTRAINT "fk_workshops_eventid" FOREIGN KEY (eventid) REFERENCES events(eventid) ON DELETE CASCADE

```

```

ahamilton=> SELECT * FROM events;

```

eventid	date	stime	etime	classroomno	advisorid
1	2019-02-19	02:11:00	05:10:00	818	1
2	2019-05-23	14:20:00	15:30:00	856	1
3	2019-06-19	13:20:00	13:50:00	455	7
4	2019-05-19	15:30:00	16:00:00	35	
5	2019-10-22	17:00:00	18:30:00	786	1
6	2019-12-17	06:15:00	08:30:00	140	7
7	2019-02-05	11:20:00	12:30:00	88	7
8	2019-05-11	08:56:00	14:14:00	628	3
9	2019-10-31	20:00:00	22:00:00	333	3
10	2019-12-11	08:00:00	12:00:00	430	1

(10 rows)

## Faculty:

```

ahamilton=> \d faculty

```

Column	Type	Table "public.faculty"	Modifiers
facultyid	integer	not null	default nextval('faculty_facultyid_seq'::regclass)
firstname	character varying(50)	not null	
lastname	character varying(50)	not null	
email	character varying(50)	not null	
phone	character varying(10)	not null	
username	character varying(18)	not null	
password	character varying(16)	not null	

```

Indexes:
    "faculty_pkey" PRIMARY KEY, btree (facultyid)
    "faculty_email_key" UNIQUE CONSTRAINT, btree (email)
    "faculty_username_key" UNIQUE CONSTRAINT, btree (username)
Check constraints:
    "chk_faculty_facultyid" CHECK (facultyid >= 0 AND facultyid <= 9999)
    "chk_faculty_password" CHECK (length(password::text) > 5)
    "ckh_faculty_phone" CHECK (phone::text !~ '%[0-9]%'::text)
Referenced by:
    TABLE "advisors" CONSTRAINT "fk_advisors_advisorsid" FOREIGN KEY (advisorid) REFERENCES faculty(facultyid) ON DELETE CASCADE
    TABLE "teachers" CONSTRAINT "fk_teachers_teacherid" FOREIGN KEY (teacherid) REFERENCES faculty(facultyid) ON DELETE CASCADE
    TABLE "tutors" CONSTRAINT "fk_tutors_tutorid" FOREIGN KEY (tutorid) REFERENCES faculty(facultyid) ON DELETE CASCADE

```

```

ahamilton=> SELECT * FROM faculty;

```

facultyid	firstname	lastname	email	phone	username	password
1	Rici	Walasik	rwalasik0@cisco.com	5708750994	rwalasik0	asG6BxvNs
2	Aubrette	O'Hanley	aohanley1@storify.com	9352127092	aohanley1	j2S1ot
3	Happy	Sturley	hsturley2@nhs.uk	3277437058	hsturley2	ErhjeXzX4U
4	Saidee	McGinley	smcginley3@rakuten.co.jp	3059289976	smcginley3	otGI58tp
5	Lyell	Primrose	lprimrose4@theglobeandmail.com	6504498825	lprimrose4	Bx1Sry
6	Nowell	O'Farriis	nofarriis5@clickbank.net	4052535915	nofarriis5	w0u0ZAEr
7	Dionisio	Grigoriev	dgrigoriev6@unc.edu	8124404436	dgrigoriev6	HBnQ4x1PZahH
8	Den	Simoneton	dsimoneton7@sitemeter.com	8648258028	dsimoneton7	0GYKKEpmJywJ
9	Bobbe	Kennifeck	bkennifeck8@spacespace.com	4113363414	bkennifeck8	R6EJvN
10	Booth	Slavin	bslavin9@virginia.edu	2399841719	bslavin9	1nENVH

(10 rows)

## Signs\_up:

```

ahamilton=> \d signs_up
Table "public.signs_up"
Column | Type | Modifiers
-----+-----+-----
eventid | integer | not null
studentid | integer | not null
attendance | boolean |
Indexes:
    "signs_up_pkey" PRIMARY KEY, btree (eventid, studentid)
Foreign-key constraints:
    "fk_signs_up_eventid" FOREIGN KEY (eventid) REFERENCES tests(eventid) ON DELETE CASCADE
    "fk_signs_up_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE

ahamilton=>

```

```

ahamilton=> SELECT * FROM signs_up;
eventid | studentid | attendance
-----+-----+-----
2 | 2 | t
2 | 3 | t
2 | 6 | t
2 | 10 | t
3 | 1 | t
3 | 2 | f
3 | 3 | t
3 | 3 | t
3 | 4 | f
3 | 5 | f
4 | 6 | t
4 | 7 | t
4 | 8 | t
4 | 10 | f
4 | 9 | f
6 | 2 |
6 | 3 |
6 | 5 |
6 | 6 |
10 | 1 | t
10 | 4 | t
10 | 5 | f
10 | 7 | f
10 | 8 | t
10 | 9 | t
(24 rows)

```

## Students:

```

ahamilton=> \d students
Table "public.students"
Column | Type | Modifiers
-----+-----+-----
studentid | integer | not null default nextval('students_studentid_seq'::regclass)
firstname | character varying(50) | not null
lastname | character varying(50) | not null
streetname | character varying(50) | not null
city | character varying(50) | not null
zip | integer | not null
state | character varying(2) | not null
email | character varying(50) | not null
grade | integer | not null
phone | character varying(10) | not null
username | character varying(18) | not null
password | character varying(16) | not null
Indexes:
    "students_pkey" PRIMARY KEY, btree (studentid)
    "students_email_key" UNIQUE CONSTRAINT, btree (email)
    "students_username_key" UNIQUE CONSTRAINT, btree (username)
Check constraints:
    "chk_students_grade" CHECK (grade >= 9 AND grade <= 12)
    "chk_students_password" CHECK (length(password::text) > 5)
    "chk_students_state" CHECK (state::text = ANY (ARRAY['AL'::character varying::text, 'AK'::character varying::text, 'AZ'::character varying::text, 'AR'::character varying::text, 'CA'::character varying::text, 'CO'::character varying::text, 'CT'::character varying::text, 'DE'::character varying::text, 'FL'::character varying::text, 'GA'::character varying::text, 'HI'::character varying::text, 'ID'::character varying::text, 'IL'::character varying::text, 'IN'::character varying::text, 'IA'::character varying::text, 'KS'::character varying::text, 'KY'::character varying::text, 'LA'::character varying::text, 'ME'::character varying::text, 'MD'::character varying::text, 'MA'::character varying::text, 'MI'::character varying::text, 'MN'::character varying::text, 'MS'::character varying::text, 'MO'::character varying::text, 'MT'::character varying::text, 'NE'::character varying::text, 'NV'::character varying::text, 'NH'::character varying::text, 'ND'::character varying::text, 'NM'::character varying::text, 'NY'::character varying::text, 'NC'::character varying::text, 'ND'::character varying::text, 'OH'::character varying::text, 'OK'::character varying::text, 'OR'::character varying::text, 'PA'::character varying::text, 'RI'::character varying::text, 'SC'::character varying::text, 'SD'::character varying::text, 'TN'::character varying::text, 'TX'::character varying::text, 'UT'::character varying::text, 'VT'::character varying::text, 'VA'::character varying::text, 'WA'::character varying::text, 'WV'::character varying::text, 'WI'::character varying::text, 'WY'::character varying::text]))
    "chk_students_studentid" CHECK (studentid >= 0 AND studentid <= 99999)
    "chk_students_zip" CHECK (zip >= 0 AND zip <= 99999)
    "chk_students_phone" CHECK (phone::text !~ '%[^0-9]%'::text)
Referenced by:
    TABLE "enrolled_into" CONSTRAINT "fk_enrolled_into_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE
    TABLE "signs_up" CONSTRAINT "fk_signs_up_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE
    TABLE "teaches" CONSTRAINT "fk_teaches_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE
    TABLE "tutoring" CONSTRAINT "fk_tutoring_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE

```

```

ahamilton=> SELECT * FROM students;
studentid | firstname | lastname | streetname | city | zip | state | email | grade | phone | username | password
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
1 | Winni | Drohan | Morningstar | Columbia | 65211 | MO | wdrohan0@merriam-webster.com | 12 | 5735627163 | wdrohan0 | Fd5vuug9u9q
2 | Zea | Levi | Rigney | Baltimore | 21229 | MD | zlevil@who.int | 9 | 4432901908 | zlevil | EnsMqCdNMWIFJ
3 | Issie | Stockley | Caliangt | Silver Spring | 20910 | MD | istockley2@instagram.com | 9 | 2403031020 | istockley2 | x0MxZVtQS
4 | La verne | Kinglesyd | Haas | Washington | 20397 | DE | lkinglesyd3@ebay.co.uk | 12 | 2027052024 | lkinglesyd3 | 41H3mntbD5pD
5 | Steve | Kitchenside | Magdelaine | San Jose | 95160 | CA | skitchenside4@unc.edu | 9 | 4082135381 | skitchenside4 | 5514Wv
6 | Adela | Spooner | Forest | Dallas | 75216 | TX | aspooner5@examiner.com | 11 | 2141701176 | aspooner5 | ruQnvZCn
7 | Bayard | Forsdyke | Green Ridge | Sarasota | 34276 | FL | bforsdyke6@msn.com | 11 | 9416560631 | bforsdyke6 | jFz0E3x
8 | Bonni | Wensley | McCormick | Cincinnati | 45264 | OH | bwensley7@fda.gov | 10 | 5133932620 | bwensley7 | 9qhFhRn
9 | Chico | Biagini | Mitchell | Saginaw | 48609 | MI | cbiagini8@goo.gl | 12 | 9895429996 | cbiagini8 | s6OC1c
10 | Karine | Maciak | Knutson | Boise | 83711 | ID | kmaciak9@weebly.com | 9 | 2087289481 | kmaciak9 | vEbp2k3vS7
(10 rows)

```

## Teachers:

```
ahamilton=> \d teachers
Table "public.teachers"
  Column |          Type          | Modifiers
-----+-----+-----
 teacherid | integer                | not null
 coursetype | academic_department_names | not null
Indexes:
    "teachers_pkey" PRIMARY KEY, btree (teacherid)
Foreign-key constraints:
    "fk_teachers_teacherid" FOREIGN KEY (teacherid) REFERENCES faculty(facultyid) ON DELETE CASCADE
Referenced by:
    TABLE "assigns" CONSTRAINT "fk_assigns_teacherid" FOREIGN KEY (teacherid) REFERENCES teachers(teacherid) ON DELETE SET NULL
    TABLE "teaches" CONSTRAINT "fk_teaches_teacherid" FOREIGN KEY (teacherid) REFERENCES teachers(teacherid) ON DELETE CASCADE
```

```
ahamilton=> SELECT * FROM teachers;
 teacherid | coursetype
-----+-----
          4 | MATH
          8 | ENGL
          9 | PE
         10 | CMPS
(4 rows)
```

## Teaches:

```
ahamilton=> \d teaches
Table "public.teaches"
  Column | Type | Modifiers
-----+-----+-----
 teacherid | integer | not null
 studentid | integer | not null
Indexes:
    "teaches_pkey" PRIMARY KEY, btree (teacherid, studentid)
Foreign-key constraints:
    "fk_teaches_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE
    "fk_teaches_teacherid" FOREIGN KEY (teacherid) REFERENCES teachers(teacherid) ON DELETE CASCADE
```

```

ahamilto=> SELECT * FROM teaches;
teacherid | studentid
-----+-----
         4 |          1
         8 |          2
         9 |          3
        10 |          4
         4 |          5
         4 |          6
         8 |          7
         9 |          8
        10 |          9
        10 |         10
         4 |          7
        10 |          2
        10 |          3
         9 |          9
         9 |          1
         8 |          1
         4 |         10
         9 |          2
        10 |          5
         4 |          2
(20 rows)

```

Tests:

```

ahamilto=> \d tests
          Table "public.tests"
  Column   | Type   | Modifiers
-----+-----+-----
 eventid   | integer | not null
 testname  | testname | not null
 advisorid | integer |
Indexes:
    "tests_pkey" PRIMARY KEY, btree (eventid)
Foreign-key constraints:
    "fk_tests_advisorid" FOREIGN KEY (advisorid) REFERENCES advisors(advisorid) ON DELETE SET NULL
    "fk_tests_eventid" FOREIGN KEY (eventid) REFERENCES events(eventid) ON DELETE CASCADE
Referenced by:
    TABLE "signs_up" CONSTRAINT "fk_signs_up_eventid" FOREIGN KEY (eventid) REFERENCES tests(eventid) ON DELETE CASCADE

```

```

ahamilto=> SELECT * FROM tests;
eventid | testname | advisorid
-----+-----+-----
        2 | SAT      |          1
        3 | CST      |          7
        4 | AP_BIO   |
        6 | AP       |          7
       10 | SAT      |          1
(5 rows)

```

Tutoring:

```

ahamilto=> \d Tutoring
          Table "public.tutoring"
  Column   | Type   | Modifiers
-----+-----+-----
 tutorid   | integer | not null
 studentid | integer | not null
Indexes:
    "tutoring_pkey" PRIMARY KEY, btree (tutorid, studentid)
Foreign-key constraints:
    "fk_tutoring_studentid" FOREIGN KEY (studentid) REFERENCES students(studentid) ON DELETE CASCADE
    "fk_tutoring_tutorid" FOREIGN KEY (tutorid) REFERENCES tutors(tutorid) ON DELETE CASCADE

```

```

ahamilto=> SELECT * FROM tutoring;
tutorid | studentid
-----+-----
      2 |         2
      2 |         5
      2 |         7
      2 |         3
      5 |         1
      5 |         2
      5 |         5
      5 |         6
      5 |         7
      2 |         1
      6 |         7
      6 |         8
      6 |        10
      6 |         3
      6 |         1
(15 rows)

```

Tutors:

```

ahamilto=> \d tutors
          Table "public.tutors"
  Column      |      Type      | Modifiers
-----+-----+-----
  tutorid     | integer         | not null
  coursetype   | academic_department_names | not null
Indexes:
    "tutors_pkey" PRIMARY KEY, btree (tutorid)
Foreign-key constraints:
    "fk_tutors_tutorid" FOREIGN KEY (tutorid) REFERENCES faculty(facultyid) ON DELETE CASCADE
Referenced by:
    TABLE "assigns" CONSTRAINT "fk_assigns_tutorid" FOREIGN KEY (tutorid) REFERENCES tutors(tutorid) ON DELETE SET NULL
    TABLE "tutoring" CONSTRAINT "fk_tutoring_tutorid" FOREIGN KEY (tutorid) REFERENCES tutors(tutorid) ON DELETE CASCADE

```

```

ahamilto=> SELECT * FROM tutors;
tutorid | coursetype
-----+-----
      2 | MATH
      5 | ENGL
      6 | CMPS
(3 rows)

```

Workshops:

```

ahamilto=> \d workshops
          Table "public.workshops"
  Column      |      Type      | Modifiers
-----+-----+-----
  eventid     | integer         | not null
  workshopname | character varying(50) | not null
  workshopinfo | character varying(255) | not null
  advisorid   | integer         |
Indexes:
    "workshops_pkey" PRIMARY KEY, btree (eventid)
Foreign-key constraints:
    "fk_workshops_advisorid" FOREIGN KEY (advisorid) REFERENCES advisors(advisorid) ON DELETE SET NULL
    "fk_workshops_eventid" FOREIGN KEY (eventid) REFERENCES events(eventid) ON DELETE CASCADE
ahamilto=>

```

```

ahamilton> SELECT * FROM Workshops;
eventid | workshopname | workshopinfo | advisorid
-----|-----|-----|-----
1 | CSUB Fast Register | Students will meet with CSUB advisors to get registered for classes | 1
5 | Healthy Eating | Learn what cheap healthy foods you can eat at home | 1
7 | Modeling | Learn the secrets models use at runways | 7
8 | Taking Good Photos | Meet with Photographer Smith to learn about taking better photos | 3
9 | Dog Training | Learn how to start training your dog at home | 3
(5 rows)

```

## 5.6 SQL Queries

This is where we'll convert our queries into sql and printscreen paste the result

**1. List all students who are enrolled in AP Calculus who currently signed up for the AP Calculus Exam.**

**2. List all Workshops Names that were scheduled during 9/1/2019-12/17/2019.**

```

SELECT B.workshopName,B.date
FROM (Workshops INNER JOIN Events ON Workshops.EventID =
Events.EventID) AS B
WHERE B.date >= 9/01/2019 AND B.date <= 12/17/2019;

```

**3. List all events that John Doe has signed up for during 9/1/2019-12/17/2019.**

```

SELECT B.EventID
FROM (Students INNER JOIN Signs_UP ON Students.StudentID =
Signs_Up.StudentID) AS A
INNER JOIN Events ON A.EventID = Events.EventID AS B
WHERE B.date >= 9/01/2019 AND B.date <= 12/17/2019 AND B.firstName =
"John" AND B.lastname = "Doe";

```

**4. List juniors who are taking the tutor named Amber Carroll.**

```

SELECT studentid, firstname, lastname
FROM Student INNER JOIN Tutoring ON
WHERE grade = "11" AND firstname = Amber AND lastname = "Carroll"

```

**5. Find all events that start at 2pm and the names of the advisors who created them, except those created by advisor Joe Mama.**

```

SELECT A.EventID, A.sTime, A.firstName, A.lastName
FROM Events INNER JOIN Faculty ON Events.AdvisorID = Faculty.FacultyID AS
A
WHERE NOT (A.firstName = "Joe" AND A.lastName = "Mama") AND A.sTime =
"14:00"

```

**6. List all tutor names that student John Doe has been assigned.**

**7. List all junior students with a grade “C” in any English course.**

```
SELECT
FROM (Students INNER JOIN Enrolled_Into ON Students.StudentID =
Enrolled_Into.StudentID AS A) INNER JOIN Courses ON A.CourseID =
Courses.CourseID) AS B
WHERE B.grade = “C” AND A.grade = 11 AND B.courseType = “ENGL”;
```

**8. Retrieve the names of students who are enrolled into all the courses of student Joe Smith.**

**9. List the names of students who have a “completed” course status and received an “A” grade in math courses.**

**10. Find the event with the highest student attendance**

## 5.7 Data Loader

### 5.7.1 Description of Data loading methods

Insert Into allows us to insert new rows or tuples into a selected table in psql. We also have a value list using the VALUES command. The values must be in the same order specified by the column list with commas separating every value.

```
INSERT INTO TABLE_NAME (column1, column2,
column3,...columnN)
VALUES (value1, value2, value3,...valueN);
```

In order to insert data that exists in another table we need to also select and specify FROM and WHERE. FROM specifies the other table we’re drawing from while WHERE specifies a certain condition of the attribute we’re trying to select. AND can also be used to add an extra condition to a WHERE statement.

```
INSERT INTO table(column1,column2,...)
SELECT column1,column2,...
FROM other_table
WHERE condition;
```

### 5.7.2 Description of Java DataLoader Program

The DataLoader.java program, developed and provided by Dr. Huaqing Wang, allows us to load some of our records or sample data into our

tables in the database from a text file. This allows us to import our data in either csv or xml format. The program allows the user to specify how to separate the sample data from the text file into columns, so we can make sure the sample data is grouped with the correct table to ensure the data transfer is accurate. This method is convenient because it allows us to customize how our data is imported and speeds up the overall process, so we don't have to manually insert test data for every table.

### 5.7.3 Implementing formulas/features to generate and import data

We, in the end, chose to use an online tool, mockaroo.com, to generate and export our data. Mockaroo is a free random data generator and API mocking tool that allows us to create custom datasets in many different formats such as CSV, JSON, SQL, and EXCEL. It also lets us easily cooperate by giving us the ability to enter data for the same schema and dataset without overwriting anything. We found that the features provided helped us format and customize our data to better suit our database.

## Phase 4: Stored Procedures and Triggers

### 6.1 Programming logic for SQL

#### 6.1.1 Introduction to your DBMS SQL

PL/pgSQL is an extension of the Postgres DMBS that supports its own procedural programming language which can be used to manipulate data through the use of control statements, cursors, stored procedures, loops, triggers, and construct complex functions. The benefit of using the procedural language along with a postgres database is that we can group our stored procedures/functions along with our queries inside the server. This is done in order to avoid manually inserting every individual query that has to be read, processed, and executed by the server before outputting a result. Using PL/pgSQL to write functions, triggers, and stored procedures makes this process more efficient by grouping blocks of query computations.

#### 6.1.2 Definitions of and usages for views, functions, procedures, and triggers

---

##### **Stored procedure**



Stored procedures in PL/pgSQL, allow us to compute and execute our queries multiple times without constantly having to contact the database server. These procedures will also fundamentally increase overall efficiency because they will be precompiled and stored in our PostgreSQL server. A stored procedure may or may not return a value, but it will return a result set. The body consists of declarations, execution statements and exception handlers if necessary.

### **Stored function**

Stored functions are similar to stored procedures except for the key difference being that stored functions will return a value where it is called. Functions, however, do not return values as OUT parameters unlike stored procedures. These more closely resemble functions we see in other programming languages.

### **Trigger**

Triggers are callback functions in the database that are invoked or “triggered” when a specific event or condition returns true based on the logic given. A trigger can be invoked before, after, or instead of when the operation is attempted on a row. This means that triggers can be executed before, after, or instead of an insert, delete, or update statement.

### **View**

Views are virtual tables or pseudo-tables that are based on a stored query that gets data from already existing base tables. A view can even represent joined tables and contain all rows of a table or selected rows from one or more tables.

## 6.1.3 Syntax for creating views, functions, procedures, and triggers

---

### **Stored procedure**

```
CREATE [OR REPLACE] FUNCTION <procedure name> (arguments)
    DECLARE
        declaration;
        [...]
    BEGIN
        < procedure_body >
        [...]
    END;
```

---

### **Stored function**

---

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS $variable_name$
    DECLARE
        declaration;
        [...]
    BEGIN
        < function_body >
        [...]
        RETURN { variable_name | value }
    END; LANGUAGE plpgsql;
```

### **Trigger**

```
CREATE TRIGGER trigger_name [BEFORE|AFTER|INSTEAD OF]
event_name
ON table_name
[
    -- Trigger logic goes here....
];
```

### **View**

---

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

#### 6.1.4 Commands to view views, functions, procedures, and triggers

**View:** \dv, Select \* FROM pg\_view

**Functions and Procedures:** \df, Select \* FROM pg\_function();

**Trigger:** \dy, Select \* From pg\_trigger;

## 6.2 Implementations

### 6.2.1 Views

#### **1. A view for a certain table with certain query conditions.**

//Shows all students from the senior class while hiding sensitive info like  
//their passwords.

```
CREATE VIEW senior_class
SELECT StudentID, firstName, lastName, grade, phone
```

```
FROM Students
WHERE grade = 12;
```

## 2. A view for a join between two tables

//Shows the names of the creators of events and their contact info

```
CREATE VIEW event_contact_info
SELECT EventID, firstName, lastName, email, phone
FROM Faculty
INNER JOIN Events
ON Faculty.FacultyID = Events.FacultyID;
```

## 3. A view for a join between three tables

//Shows the course name and its teacher along with their contact info

```
CREATE VIEW course_contact_info
SELECT
a.CourseID, c.courseName, f.FacultyID, f.firstName, f.lastName,
f.email, f.phone
FROM Faculty AS f
INNER JOIN assigns AS a
ON f.FacultyID = a.TeacherID
INNER JOIN Courses AS c
ON a.CourseID = c.CourseID;
```

## 6.2.2 Stored Procedures and/or Functions

### 1. A stored procedure for inserting a new record into one of your tables. The field values are passed to the procedure through the input parameters.

//This procedure allows advisors to create events

```
CREATE OR REPLACE FUNCTION create_event
(_date DATE, _sTime TIME, _eTime TIME, _room INT, _ID INT)
RETURNS void AS $$
BEGIN
    INSERT INTO
Events(date, sTime, eTime, classroomNo, FacultyID)
VALUES(_date, _sTime, _eTime, _room, _ID);
    COMMIT;
END;
$$ LANGUAGE plpgsql;
```

### 2. A stored procedure for deleting an existing record based on the primary key of your selected table.

//Will delete a tutor from the faculty table

```
CREATE OR REPLACE FUNCTION delete_tutor_faculty (tID INT)
RETURNS void AS $$
BEGIN
    DELETE FROM Faculty
    USING Tutors
    WHERE tID = FacultyID AND tID = TutorID;
END;
$$ LANGUAGE plpgsql;
```

### 3. A stored function which returns the average of a numerical fields of highest or lowest N records where N is the parameter for the function.

//Thinking of getting the average grade of a course,  
//get all the grades of students in a course, set A = 4, B = 3, C = 2, etc.  
// add them all up then divide by students for the avg.

```
CREATE FUNCTION top_of_class_avg(N INTEGER, cID INTEGER)
RETURNS DECIMAL AS $$
DECLARE
    average DECIMAL;
BEGIN
    SELECT AVG (Grade_Point)
    INTO average
    FROM (
        SELECT s.StudentID,
            CASE WHEN e.grade = 'A' THEN 4
                 WHEN e.grade = 'B' THEN 3
                 WHEN e.grade = 'C' THEN 2
                 WHEN e.grade = 'D' THEN 1
                 WHEN e.grade = 'F' then 0
                 ELSE NULL END
            AS Grade_Point
        FROM Students AS s
        INNER JOIN enrolled_into AS e
        ON s.StudentID = e.StudentID AND cID = e.CourseID
        ORDERED BY e.Grade_Point ASC LIMIT N
    )MyTable;
    RETURN average;
END;
$$ LANGUAGE plpgsql;
```

### 6.2.3 Triggers

#### 1. Deletion trigger

Whenever we delete a tutor, we will put the id they had and their name along with the timestamp they left into a table. When a tutor is deleted, the tables were set to cascade the change to the "tutors"(tutored\_by) table and to set itself to null in the "assigns" table.

```
--TRIGGER--
CREATE TRIGGER T_tutor_delete
BEFORE DELETE
ON Tutor FOR EACH ROW
EXECUTE PROCEDURE tutor_leave_date_insert();
-----

--FUNCTION--
CREATE OR REPLACE FUNCTION tutor_leave_date_insert()
RETURNS trigger AS $$
BEGIN
    INSERT INTO
former_tutors(TutorID,firstName,lastName,leftOn)
VALUES(OLD.TutorID,OLD.firstName,OLD.lastName,now());
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
-----
```

#### 2. Update trigger

When a teacher changes their last name, we update it and have a trigger so we know what it was before and when it was changed

```
--TRIGGER--
CREATE TRIGGER T_teacher_last_name
BEFORE UPDATE
OF lastName ON FACULTY FOR EACH ROW
EXECUTE PROCEDURE teacher_lastName_update();
-----

--FUNCTION--
CREATE OR REPLACE FUNCTION teacher_lastName_update()
RETURNS trigger AS $$
BEGIN
    IF OLD.lastName <> NEW.lastName THEN
        INSERT INTO
teacher_name_changes(FacultyID,oldLastName,newLastName,change
d_on)
```

```
VALUES (OLD.FacultyID,OLD.lastName,NEW.lastName,now());
        END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
-----
```

### 3. "Instead of" trigger

Instead of changing the attributes in the view, we set the changes in the underlying tables that create the view as well.

```
CREATE TRIGGER instead_of_teacher_contact
INSTEAD OF UPDATE ON course_contact_info
FOR EACH ROW EXECUTE PROCEDURE update_view_teacher_contact();

--FUNCTION--
CREATE OR REPLACE FUNCTION
update_view_teacher_contact()
RETURNS trigger as $$
BEGIN
    IF OLD.phone <> NEW.phone THEN
        UPDATE Faculty
        SET phone = NEW.phone
        WHERE OLD.FacultyID = FacultyID;
    ENDIF;

    IF OLD.email <> NEW.email THEN
        UPDATE Faculty
        SET email = NEW.email
        WHERE OLD.FacultyID = FacultyID;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

## 6.2.4 Testing Results of Views, Procedures, and/or Functions

### Views:

course\_contact\_info:

```
ahamilton=> SELECT * FROM course_contact_info;
courseid | coursenam | facultyid | firstna | lastna | email | phone
-----+-----+-----+-----+-----+-----+-----
7 | Geometry | 4 | Saidee | McGinley | smcginley3@rakuten.co.jp | 3059289976
4 | Calculous 2 | 4 | Saidee | McGinley | smcginley3@rakuten.co.jp | 3059289976
3 | Calculous 1 | 4 | Saidee | McGinley | smcginley3@rakuten.co.jp | 3059289976
1 | Math Readiness | 4 | Saidee | McGinley | smcginley3@rakuten.co.jp | 3059289976
10 | Creative Writing | 8 | Den | Simoneton | dsimoneton7@sitometer.com | 8648258028
5 | Persuasive Writing | 8 | Den | Simoneton | dsimoneton7@sitometer.com | 8648258028
2 | English Readiness | 8 | Den | Simoneton | dsimoneton7@sitometer.com | 8648258028
8 | PE 1-2 | 9 | Bobbe | Kennifeck | bkennifeck8@squarespace.com | 4113363414
9 | Typing | 10 | Booth | Slavin | bslavin9@virginia.edu | 2399841719
```

event\_contact\_info:

```
ahamilto=> SELECT * FROM event_contact_info;
eventid | firstname | lastname | email | phone
-----+-----+-----+-----+-----
10 | Rici | Walasik | rwalasik0@cisco.com | 5708750994
5 | Rici | Walasik | rwalasik0@cisco.com | 5708750994
2 | Rici | Walasik | rwalasik0@cisco.com | 5708750994
1 | Rici | Walasik | rwalasik0@cisco.com | 5708750994
9 | Happy | Sturley | hsturley2@nhs.uk | 3277437058
8 | Happy | Sturley | hsturley2@nhs.uk | 3277437058
7 | Dionisio | Grigoriev | dgrigoriev6@unc.edu | 8124404436
6 | Dionisio | Grigoriev | dgrigoriev6@unc.edu | 8124404436
3 | Dionisio | Grigoriev | dgrigoriev6@unc.edu | 8124404436
(9 rows)
```

senior\_class:

```
ahamilto=> SELECT * FROM senior_class;
studentid | firstname | lastname | grade | phone
-----+-----+-----+-----+-----
1 | Winni | Drohan | 12 | 5735627163
4 | La verne | Kinglesyd | 12 | 2027052024
9 | Chico | Biagini | 12 | 9895429996
(3 rows)
```

## **Functions:**

create\_event:

```
ahamilto=> Select *FROM create_event ('11/28/2019', '04:15', '05:15', 818, 1);
create_event
-----
(1 row)

ahamilto=> Select * FROM events;
```

```

ahamilto=> Select * FROM events;
eventid | date       | stime    | etime    | classroomno | advisorid
-----+-----+-----+-----+-----+-----
1 | 2019-02-10 | 02:11:00 | 05:10:00 | 818 | 1
2 | 2019-05-31 | 14:30:00 | 15:30:00 | 856 | 1
3 | 2019-06-19 | 13:20:00 | 13:50:00 | 455 | 7
4 | 2019-05-19 | 15:30:00 | 16:00:00 | 35 | 
5 | 2019-10-22 | 17:00:00 | 18:30:00 | 786 | 1
6 | 2019-12-17 | 06:15:00 | 08:30:00 | 140 | 7
7 | 2019-02-05 | 11:20:00 | 12:30:00 | 88 | 7
8 | 2019-05-11 | 08:56:00 | 14:14:00 | 628 | 3
9 | 2019-10-31 | 20:00:00 | 22:00:00 | 333 | 3
10 | 2019-12-11 | 08:00:00 | 12:00:00 | 430 | 1
20 | 2015-11-28 | 04:59:02 | 12:56:32 | 818 | 1
23 | 2015-11-28 | 04:59:02 | 12:56:32 | 818 | 1
24 | 2015-11-28 | 04:59:00 | 06:59:00 | 818 | 1
25 | 2019-11-28 | 04:15:00 | 05:15:00 | 818 | 1
(14 rows)
--More--

```

delete\_tutor\_faculty

```

ahamilto=> Select * FROM tutors;
tutorid | coursetype
-----+-----
2 | MATH
5 | ENGL
6 | CMPS
(3 rows)

ahamilto=> Select * FROM delete_tutor_faculty ('2');
delete_tutor_faculty
-----
(1 row)

ahamilto=> Select * FROM tutors;
tutorid | coursetype
-----+-----
5 | ENGL
6 | CMPS
(2 rows)

```

top\_of\_class\_avg:

```

ahamilto=> SELECT * FROM top_of_class_avg(10, 2);
top_of_class_avg
-----
1.6666666666666667
(1 row)

ahamilto=> SELECT * FROM top_of_class_avg(10, 1);
top_of_class_avg
-----
2.0000000000000000
(1 row)

```



## 6.3 Syntax of Stored Functions and Procedures and Triggers of Two DBMS

For this project, the implementation, the procedures and the design of the database is all being worked on PL/pgSQL. There are many other DBMS such as MySQL.

### 6.3.1 Syntax similarities and differences between the two

#### Syntax to create a Function in MySql version 8.0

```
CREATE
[DEFINER = user]
FUNCTION sp_name ([func_parameter[,...]])
RETURNS type
[characteristic ...] routine_body
```

#### Syntax to create a Function in PostgreSQL version 9.2.24

```
CREATE [ OR REPLACE ] FUNCTION
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  [ RETURNS rettype
    | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
  [ WITH ( attribute [, ...] ) ]
```

#### Syntax to create a Trigger in MySql version 8.0

```
CREATE
[DEFINER = user]
TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW
[trigger_order]
trigger_body
```

### Syntax to create a Trigger in PostgreSQL version 9.2.24

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF }  
{ event [ OR ... ] }  
  ON table_name  
  [ FROM referenced_table_name ]  
  [ NOT DEFERRABLE | [ DEFERRABLE ]  
{ INITIALLY IMMEDIATE | INITIALLY DEFERRED } ]  
  [ FOR [ EACH ] { ROW | STATEMENT } ]  
  [ WHEN ( condition ) ]  
  EXECUTE PROCEDURE function_name ( arguments )
```

### Syntax to create a Procedure in MySql version 8.0

```
CREATE  
  [DEFINER = user]  
  PROCEDURE sp_name ([proc_parameter[,...]])  
  [characteristic ...] routine_body
```

### Syntax to create a Procedure in PostgreSQL version 11

```
CREATE [ OR REPLACE ] PROCEDURE  
  name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )  
{ LANGUAGE lang_name  
  | TRANSFORM { FOR TYPE type_name } [, ... ]  
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER  
  | SET configuration_parameter { TO value | = value | FROM CURRENT }  
  | AS 'definition'  
  | AS 'obj_file', 'link_symbol'  
} ...
```

## 6.3.2 Any discernible advantages/disadvantages between the two

The advantages that PostgreSQL offers is that it offers many data types than compared to MySQL for instance you can designed complex custom data types. The architecture of PostgreSQL is that it is an Object-Relational Database Management System. It's meant to be optimized for more complex queries such as databases that are big complex designed. A key feature for the server side of PostgreSQL is that it offers JSON data type support. PostgreSQL provides materialized view and temporary table. The disadvantages when compared to MySQL is that PostgreSQL consumes more power. Such that the performance can

take a haul. A key feature is that compared to MySQL PostgreSQL can be very slow.

The advantages that MySQL offers is that it is a relational database management system. MySQL is very fast when compared to PostgreSQL and it is very simple minded instead of being complex like PostgreSQL. Another key feature is that it supports many programming languages when compared to PostgreSQL. The disadvantages that MySQL has to offer is that it doesn't provide use for materialized views and temporary tables implemented. Another instance when comparing to PostgreSQL is that it doesn't provide the data domain object. It doesn't do well with large data volumes such that when having tons of users in the database all at once it can be a problem therefore it is best to use a complex database such as PostgreSQL.

## Phase 5: Graphical User Interface Design and Implementation

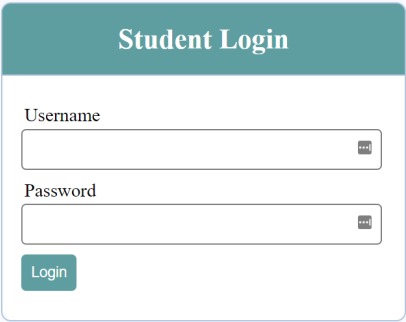
### 7.1 GUI Application Description

We have decided that we would make a website our GUI. This was chosen as it would fit best with what our database was set out to accomplish. A quick overview of the site is as follows:

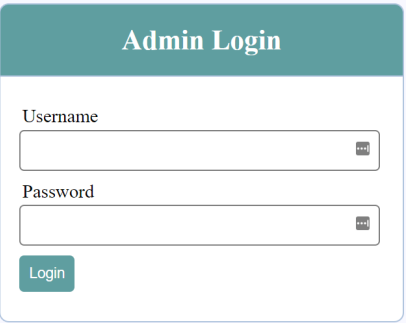
1. Two separate login pages where students and faculty can login
2. Separate views depending on your login
  1. Student
    - a. Will be able to see their basic general info
    - b. Under their general info they will be able to see their class grades
  2. Faculty
    - a. Will be able to see their basic general info
    - b. Will have access to certain pages to manipulate the database
    - c. In this case we are treating all faculty as Advisors so they can create events and manage Tutors.

#### 7.1.2 Walkthrough with Pictures

First, we come to the login page. We choose one of our logins based on what we are logging in as.



A login form titled "Student Login" with a teal header. It contains two input fields: "Username" and "Password", each with a small icon on the right. Below the fields is a teal "Login" button.



A login form titled "Admin Login" with a teal header. It contains two input fields: "Username" and "Password", each with a small icon on the right. Below the fields is a teal "Login" button.

When we login as a student, we can see a welcome text with the students name in the top right corner. In this quick main page we can see the student's basic info. We have chosen not to show their username or their password as those should be a bit more private. If we scroll down a bit more we can see the classes that the student has taken as well as their current ones along with their grades.

Welcome Winni

Toggle Menu

Home

Grades

## Home

ID: 1  
FirstName: Winni  
LastName: Drohan  
Address: Columbia, MO 65211  
Email: wdrohan0@merriam-webster.com  
Grade: 12  
Phone: 5735627163

## Grades

courseid	coursename	status	grade
2	English Readiness	COMPLETED	C
8	PE 1-2	FAILED	F
10	Creative Writing	COMPLETED	A

When login as an admin we would be brought to a different page. Currently we only have 3 working pages for admins, those pages being index.php, create\_event.php, and delete\_tutor.php. We will start out with index.php.

Welcome Frederic

Toggle Menu

Home

Create Events

Delete Tutors

## Basic Information

ID: 15  
FirstName: Frederic  
LastName: Musk  
Email: hahaha@yahoo.com  
Phone: 4205595555

Just like the Student page, we are brought into a page with our basic information. We gather this info from a query to the database using pg\_query. We decide to not show the faculty's username and password and here as well.

Welcome
Home
Create Events
Delete Tutors

Toggle Menu

## Create Events

Date
Start Time
End Time
Classroom #
AdvisorID
Submit

eventid	date	stime	etime	classroomno	advisorid
1	2019-02-10	02:11:00	05:10:00	818	1
2	2019-05-31	14:30:00	15:30:00	856	1
3	2019-06-19	13:20:00	13:50:00	455	7
4	2019-05-19	15:30:00	16:00:00	35	
5	2019-10-22	17:00:00	18:30:00	786	1
6	2019-12-17	06:15:00	08:30:00	140	7
7	2019-02-05	11:20:00	12:30:00	88	7
8	2019-05-11	08:56:00	14:14:00	628	3
9	2019-10-31	20:00:00	22:00:00	333	3
10	2019-12-11	08:00:00	12:00:00	430	1
20	2015-11-28	04:59:02	12:56:32	818	1
23	2015-11-28	04:59:02	12:56:32	818	1

Normally only an advisor would have access to this page but, we have little experience with web development so decided to mash all faculty into one page for now. At the bottom of the page we show a table with events that have been scheduled. At the top of the page we have boxes where we can fill in information and be able to create a new event after hitting submit. This page takes advantage of a function we created in the database called `create_event`. We take the info inputted into the boxes and send them as arguments to our function to create a new event.

Welcome
Home
Create Events
Delete Tutors

Toggle Menu

## Delete Tutors

TutorID
Fire

tutorid	coursetype
5	ENGL
6	CMPS
12	MATH

The next page is one for deleting tutors. Maybe a tutor has been slacking on the job or maybe they have just left for another job, we need a way to remove them from the system. Here we can simply type in a tutors ID number after finding it from the results at the bottom of the page. After

a simple click they should be gone from the database and a trigger goes off to remove them from other tables as well. This also takes advantage of a function we used called `delete_tutor_faculty`. The table being shown is from a query to Tutors and the function being used deletes the tutor from its parent tree Faculty. Since we can delete a tutor with the function and have them not appear on the page, it means that our cascade triggers are correctly being utilized.

## 7.2 Programming Sections

### 7.2.1 Server-side Programming

// The views and procedure stuff are pretty self explanatory, don't think we should go into much detail here and can just refer to our walkthrough or whatever.

### 7.2.2 Middle-tier Programming

- Code for database connection.

// Here we keep using `$conn = (insert parameters here)`, we probably could have just put it into a php file somewhere and just have kept referring to it when needed by including that file rather than constantly creating a new `$conn` variable and initializing it any time but it is what it is.

- Code that displays views.

// Views are just tables as well, for all our pages we show different tables with different parameters by pushing a `SELECT *` query into our database

- Method of calling stored procedure/function.

// Stored procedures are called with `delete tutor` and `create event` by running a query into the database

### 7.2.3 Client-side Programming

- Local searching, sorting/ordering.

- Saving data/reports to files.

- Loading or parsing data to insert into database if applicable.

- Any other code sections you would like to mention.

// a lot of the searching is done for you by clicking through the pages and such, you want to see the tutors to be able to delete one? The tables there. You want to put an event, you can see the other events too. You're a student and want to login, you probably just want to see your grades so they're already there for you. Basically our searching is done by clicking pages.