CS387 Final Project Report

Project name - ER2SQL

Group Members:

- Akshat Kumar (200050004)
- Jash Kabra (200050054)
- Shubham Santosh Bakare (200050133)
- Subarno Nath Roy (200050139)

Overview and Motivation:

Entity Relationship diagrams provide a clear and concise representation of the database's structure, including entities, attributes, and relationships. Thus for designing complex databases, it is often very convenient to model the database in the form of an ER diagram that captures all kinds of relationships between entities and then use this ER diagram to create a relational schema compatible with modern database management systems like Postgresql.
Very few tools are publicly available for such an important task which motivated us to create such a tool for ourselves through this project.
We have created a tool(application) that will provide database designers with a polished User Interface for creating ER diagrams and an option to export the ER diagrams to relational schema such that it is compatible with Postgres. It will support all features of ER diagrams that we have studied in our lectures such as Strong and Weak entity sets, relations of One-One, Many-One, One-Many and Many-Many types, cardinality constraints, primary keys, etc.

Plan of action:

We have implemented the project in the following **three phases** :
- Implement data structures for storing Entities and relationships along with constraints in a structured manner for easy representation
- Program an algorithm to convert the ER diagram stored as data structures from the previous phase into a relational schema for Postgres. The algorithm for this has been taught in the lecture, however, we couldn't find any implementation of a program(code) online that could convert ER diagrams to SQL schema. In our opinion, such a program will be very useful to database designers.
- Create a Graphical User Interface that will allow Database Designers to create ER diagrams. It will feature a canvas and Drag and Drop tools for creating entities and relationships between them. Our program should be able to then convert from the diagram thus created to the aforementioned data structures.

Data structures used:

We have stored and represented the entities and relationships using classes and used the appropriate class objects. JSON objects have been used to parse the ER diagram objects and then converted to our data structures. The classes can be enumerated as:

- **Entity Class** - name, attributes, weakness level
- **Attribute Class** - name, data_type, constraint type(if primary key), is null constraint
- **Relation Class** - name, entity1-2, cardinality 1-2, participation 1-2

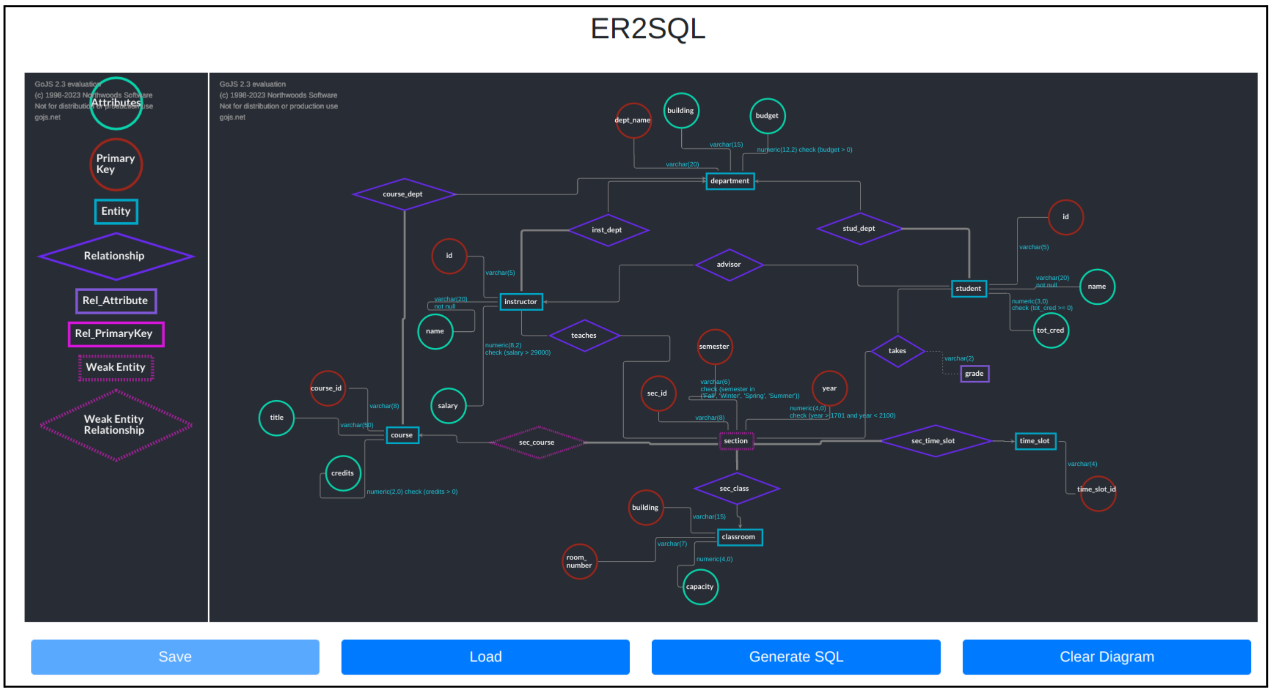
Algorithm implemented:

We will be implementing the following algorithm as the central driver of our application to convert the ER diagrams into relational database which will then later be converted into SQL queries using the standard procedure:

1. **Mapping of Regular Entity Types**
    a. For each entity type E in ER schema, create table T which includes all attributes of E.
    b. Set the primary key of T as the primary key of E.
2. **Mapping of Weak Entity Types**
    a. For each weak entity type W with owner entity type E, create table T which includes all attributes of W as attributes of T.
    b. Primary key of T is combination of primary key of owner E and primary key of weak entity type W.
    c. Include primary keys of owner E as foreign key in table T referencing table corresponding to owner entity E.
3. **Mapping of Many-to-Many Relation Type**
    a. For each many-to-many relationship type R in ER schema, identify the entities E and F participating in R.
    b. Create a new table T which includes all attributes of R as attributes of T
    c. The table T also includes the primary key of E as attributes. Primary key of E is also included as foreign key in table T and references the table corresponding to entity E.
    d. We do part c for entity F as well.
    e. Primary key of T is combination of primary key of E, F, R.
4. **Mapping of One-to-Many Relationship Types**
    a. For each binary one-to-many relationship type R, identify the entity M that represents the entity at the many sides of R and O representing the entity at the one side of R.
    b. Include primary key of O as attributes in table corresponding to M and as foreign key in table referencing table corresponding to entity E.
    c. If the relationship R has total participation on many entity side M and partial on one side O, the attribute added in part b is also made not null.
5. **Mapping of One-to-One Relationship Types**
    a. Treat any of the sides as many and follow the algorithm of one-to-many. If one of the entities has partial participation and the other has total participation, preferentially treat the entity having total participation as many.
6. **Printing SQL**
    a. We topologically sort the tables created according to foreign key dependencies and then print the appropriate SQL syntax.

Final implemented GUI:

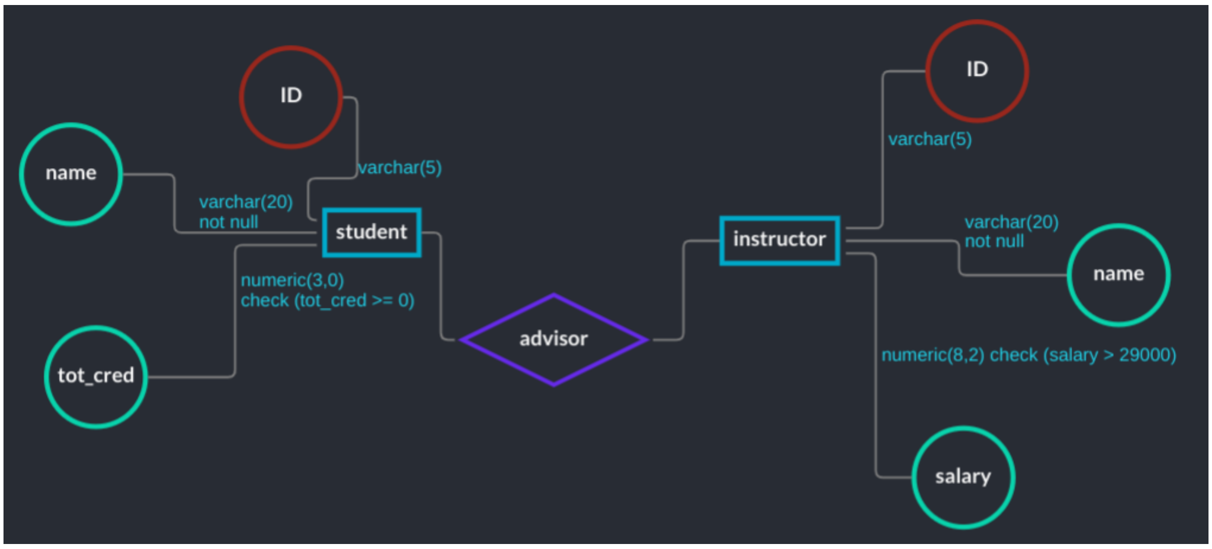
We have implemented the GUI tool for creating ER diagrams using the GOJS library in javascript. This is a sample (ER diagram for univ-db) drawn using our GUI tool:



Here we have the shape pane on the left hand side from where we can select and add new elements of desired types. Attributes are circles, primary key attributes have a different colour and other elements are as shown in the diagram. We also have Save, Load and Clear Diagram options as shown.

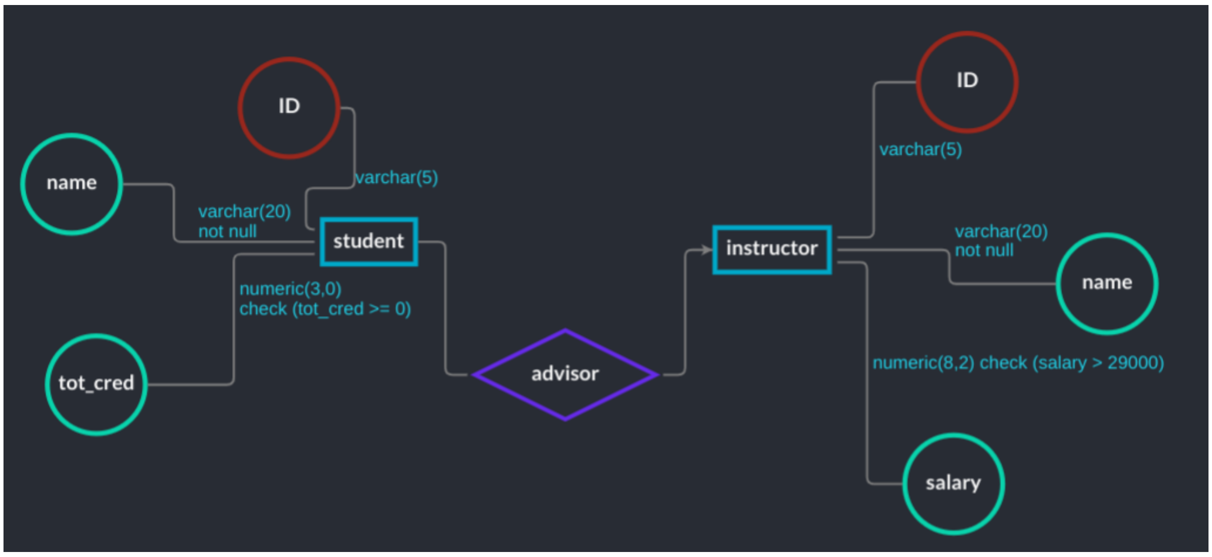
Sample Outputs generated by our application:

1. Many to Many relation



```sql
CREATE TABLE instructor (
   instructor_ID varchar(5),
   name varchar(20) not null,
   salary numeric(8, 2) check (salary > 29000),
   PRIMARY KEY (ID)
);

CREATE TABLE student (
   ID varchar(5),
   name varchar(20) not null,
   tot_cred numeric(3, 0) check (tot_cred >= 0),
   PRIMARY KEY (ID)
);

CREATE TABLE advisor (
   ID varchar(5),
   instructor_ID varchar(5),
   PRIMARY KEY (ID, instructor_ID),
   FOREIGN KEY (ID) REFERENCES student,
   FOREIGN KEY (instructor_ID) REFERENCES instructor
);
```

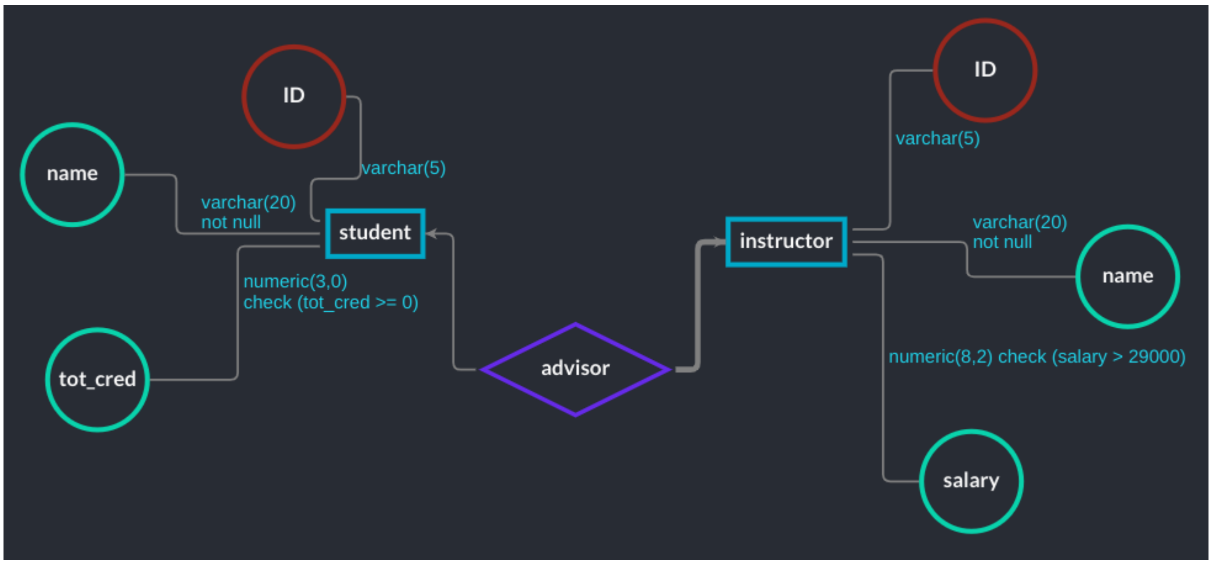
2. **One to Many relation**



```sql
CREATE TABLE instructor (
   ID varchar(5),
   name varchar(20) not null,
   salary numeric(8, 2) check (salary > 29000),
   PRIMARY KEY (ID)
);
```

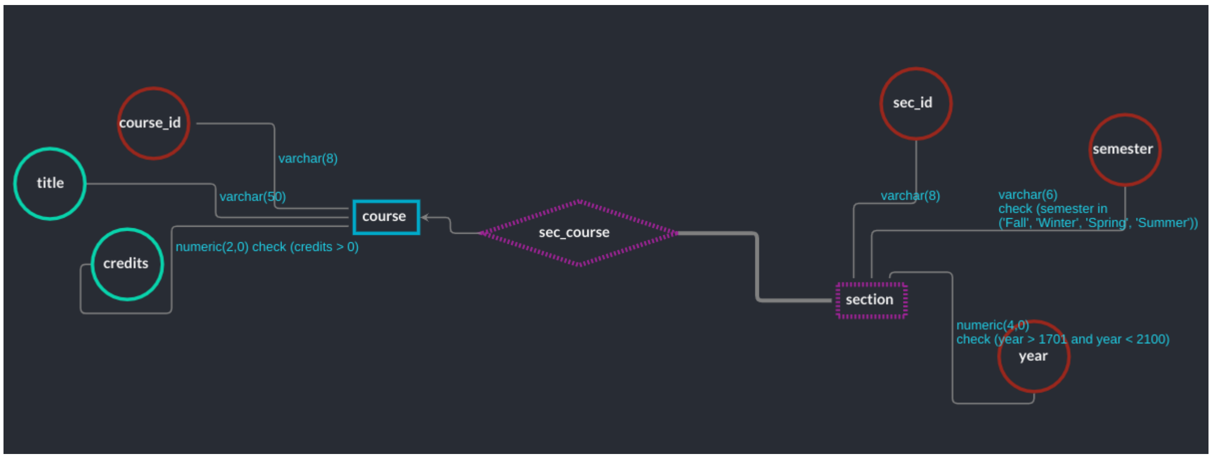
```sql
CREATE TABLE student (
   ID varchar(5),
   name varchar(20) not null,
   tot_cred numeric(3, 0) check (tot_cred >= 0),
   instructor_ID varchar(5),
   PRIMARY KEY (ID),
   FOREIGN KEY (instructor_ID) REFERENCES instructor
);
```

3. One to One with Total participation



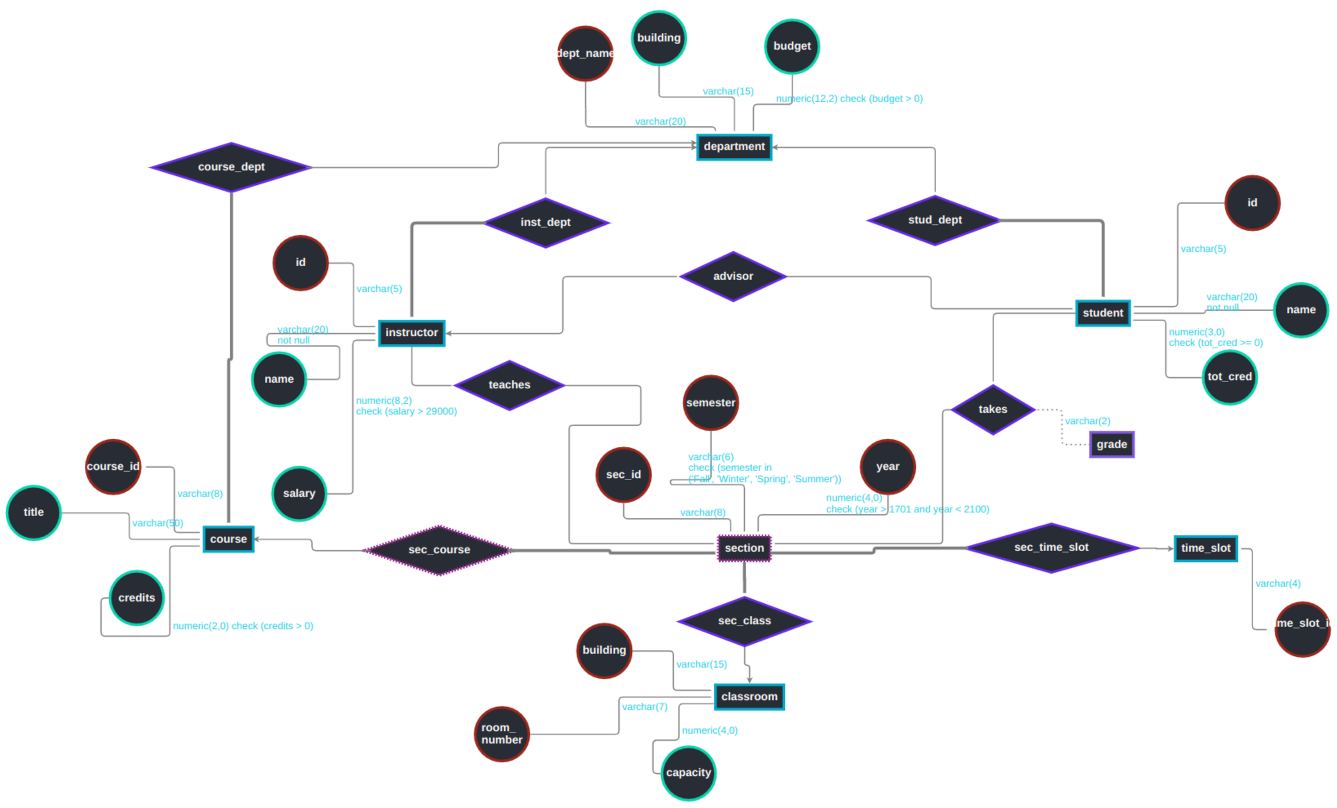
```sql
CREATE TABLE student (
   ID varchar(5),
   name varchar(20) not null,
   tot_cred numeric(3, 0) check (tot_cred >= 0),
   PRIMARY KEY (ID)
);

CREATE TABLE instructor (
   ID varchar(5),
   name varchar(20) not null,
   salary numeric(8, 2) check (salary > 29000),
   student_ID varchar(5) NOT NULL,
   PRIMARY KEY (ID),
   FOREIGN KEY (student_ID) REFERENCES student
);
```

4. Weak entity relationship



```
CREATE TABLE course (
   course_id varchar(8),
   title varchar(50),
   credits numeric(2, 0) check (credits > 0),
   PRIMARY KEY (course_id)
);

CREATE TABLE section (
   sec_id varchar(8),
   semester varchar(6) check (
       semester in ('Fall', 'Winter', 'Spring', 'Summer')
   ),
   year numeric(4, 0) check (
       year > 1701
       and year < 2100
   ),
   course_id varchar(8),
   PRIMARY KEY (sec_id, semester, year, course_id),
   FOREIGN KEY (course_id) REFERENCES course
);
```

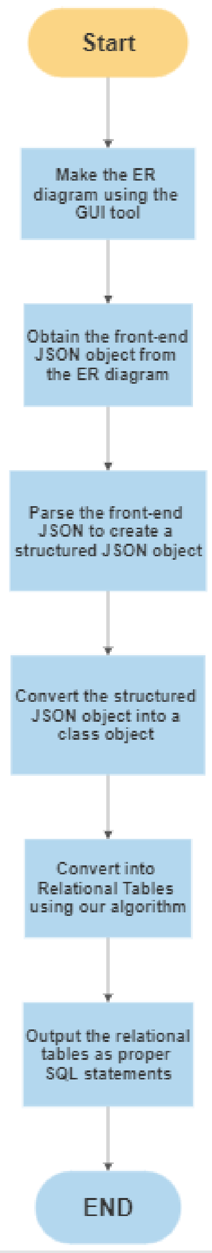
5. The entire Univ-DDL Relational Schema



```
CREATE TABLE time_slot (
    time_slot_id varchar(4),
    PRIMARY KEY (time_slot_id)
);

CREATE TABLE classroom (
    building varchar(15),
    capacity numeric(4, 0),
    room_number varchar(7),
    PRIMARY KEY (building, room_number)
);

CREATE TABLE section (
    sec_id varchar(8),
    semester varchar(6) check (
        semester in ('Fall', 'Winter', 'Spring', 'Summer')
    ),
    year numeric(4, 0) check (
        year > 1701
        and year < 2100
    ),
```

```sql
   building varchar(15) NOT NULL,
   room_number varchar(7) NOT NULL,
   time_slot_id varchar(4) NOT NULL,
   PRIMARY KEY (sec_id, semester, year),
   FOREIGN KEY (building, room_number) REFERENCES classroom,
   FOREIGN KEY (time_slot_id) REFERENCES time_slot
);

CREATE TABLE department (
   dept_name varchar(20),
   building varchar(15),
   budget numeric(12, 2) check (budget > 0),
   PRIMARY KEY (dept_name)
);

CREATE TABLE course (
   course_id varchar(8),
   title varchar(50),
   credits numeric(2, 0) check (credits > 0),
   sec_id varchar(8),
   semester varchar(6) check (
       semester in ('Fall', 'Winter', 'Spring', 'Summer')
   ),
   year numeric(4, 0) check (
       year > 1701
       and year < 2100
   ),
   dept_name varchar(20) NOT NULL,
   PRIMARY KEY (course_id, sec_id, semester, year),
   FOREIGN KEY (sec_id, semester, year) REFERENCES section,
   FOREIGN KEY (dept_name) REFERENCES department
);

CREATE TABLE instructor (
   id varchar(5),
   name varchar(20) not null,
   salary numeric(8, 2) check (salary > 29000),
   dept_name varchar(20) NOT NULL,
   PRIMARY KEY (id),
   FOREIGN KEY (dept_name) REFERENCES department
);

CREATE TABLE teaches (
   id varchar(5),
   sec_id varchar(8),
   semester varchar(6) check (
       semester in ('Fall', 'Winter', 'Spring', 'Summer')
   ),
   year numeric(4, 0) check (
       year > 1701
       and year < 2100
   ),
   PRIMARY KEY (id, sec_id, semester, year),
   FOREIGN KEY (id) REFERENCES instructor,
   FOREIGN KEY (sec_id, semester, year) REFERENCES section
);

CREATE TABLE student (
   id varchar(5),
   name varchar(20) not null,
   tot_cred numeric(3, 0) check (tot_cred >= 0),
   dept_name varchar(20) NOT NULL,
   instructor_id varchar(5),
   PRIMARY KEY (id),
   FOREIGN KEY (dept_name) REFERENCES department,
   FOREIGN KEY (instructor_id) REFERENCES instructor
);

CREATE TABLE takes (
   grade varchar(2),
   sec_id varchar(8),
   semester varchar(6) check (
       semester in ('Fall', 'Winter', 'Spring', 'Summer')
   ),
   year numeric(4, 0) check (
       year > 1701
       and year < 2100
   ),
   id varchar(5),
   PRIMARY KEY (sec_id, semester, year, id),
   FOREIGN KEY (sec_id, semester, year) REFERENCES section,
   FOREIGN KEY (id) REFERENCES student
);
```

Program Flow of our application:


Start
Make the ER diagram using the GUI tool
Obtain the front-end JSON object from the ER diagram
Parse the front-end JSON to create a structured JSON object
Convert the structured JSON object into a class object
Convert into Relational Tables using our algorithm
Output the relational tables as proper SQL statements
END

Limitations, Benefits of our algorithm and other learnings:

- Our algorithm can only implement/test specific cases of total participation of entity sets. We have included total participation on many-side of a many-to-one relationship or on either side of a one-to-one relationship. This is an improvement over the reference univ-ddl provided to us as it did not take into account any cases of total participation.

- If there is a one-to-many relation with partial participation on many-side, we do not create a new table unlike the reference univ-ddl. We thought it would be better to not add an extra table in our database, but this could result in null values in the table corresponding to many side entities of the foreign key referencing one side entity.
- If we were to add total participation in any other case, there would be circular functional dependencies and relational models in SQL cannot be implemented. For example, if there is total participation in a many-to-many relationship R, then we create a new table for R having primary keys of participating entities as foreign keys. If we want to take into account total participation, we would have to add the primary key of R as foreign key in the total entity and make it not null but it leads to a circular dependency.
- To avoid these circular dependencies, we thought of adding assertions to check our total participation constraints. If there is a one-to-one relation advisor having total participation on both sides- student and instructor, we can treat the student side as many and add primary key(inst_id) of instructor as attribute in student table: inst_id not null, and add inst_id as foreign key referencing department.
  To show total participation on instructor side, we can write the assertion:
  create assertion inst_total check (null not in (select instructor.id from instructor left outer join student)).
  However, Postgresql does not support assertions so we could not go ahead with this idea.
- We also took into account relationships, entities having the same name for their attributes by prepending the names of the entity or relationship to the attribute name wherever required.

Exploration and Extensions (Future Work):

After implementing the proposed ideas and designs for our application, the following ideas can be explored as extensions to our project to take our application to a better level suitable for a possible market launch:

- **Error Handling**: Our tool currently works on the assumption that the ER diagram made using the tool is accurate and the corresponding SQL can be generated. But as an extension we can also provide an error handling and rectification mechanism which will help us identify errors in the ER diagram.
- **Entity Sets Roles implementation**: Our tool currently doesn't support the addition of Roles to the Entity Sets in a relationship (where one entity has multiple arrows to a relation, a self loop). This feature can be integrated further as a future work.
- **Reverse Process**: We would also like to try to implement the reverse process, i.e given a relational schema, visualise it as an ER diagram. This will help visualise the relations between entities. However, this seems to be a highly complex goal as our program should differentiate between entities and relationships in a relational schema and understand the complex relationships between different entities.
- **Functional Dependencies**: We would like to implement an option for database designers to provide functional dependencies along with the ER diagram. We could develop a program to decompose the resulting Relational schema into BCNF or 3NF form. Although functional dependencies are not completely supported with Postgres, declaring them along with an ER diagram and creating a resulting normalised decomposition can help eliminate redundancy and improve data integrity.

- **Integrating Image Recognition**: Instead of using our custom GUI to input the ER diagram into our app, we will provide another method of input in the form of images of the ER diagrams which will then be converted into suitable forms first and then the SQL results would be given. This is a useful feature for organisations and developers with their ER diagrams in the form of diagrams available to them.

References:

- Slides Algo | Purdue
- inf1-da-1-2b.pdf (ed.ac.uk)
- Algorithmic Resolution of an ER Schema into Relational DDL Statements using Artificial Intelligence (ijser.org)
- Reference for GUI