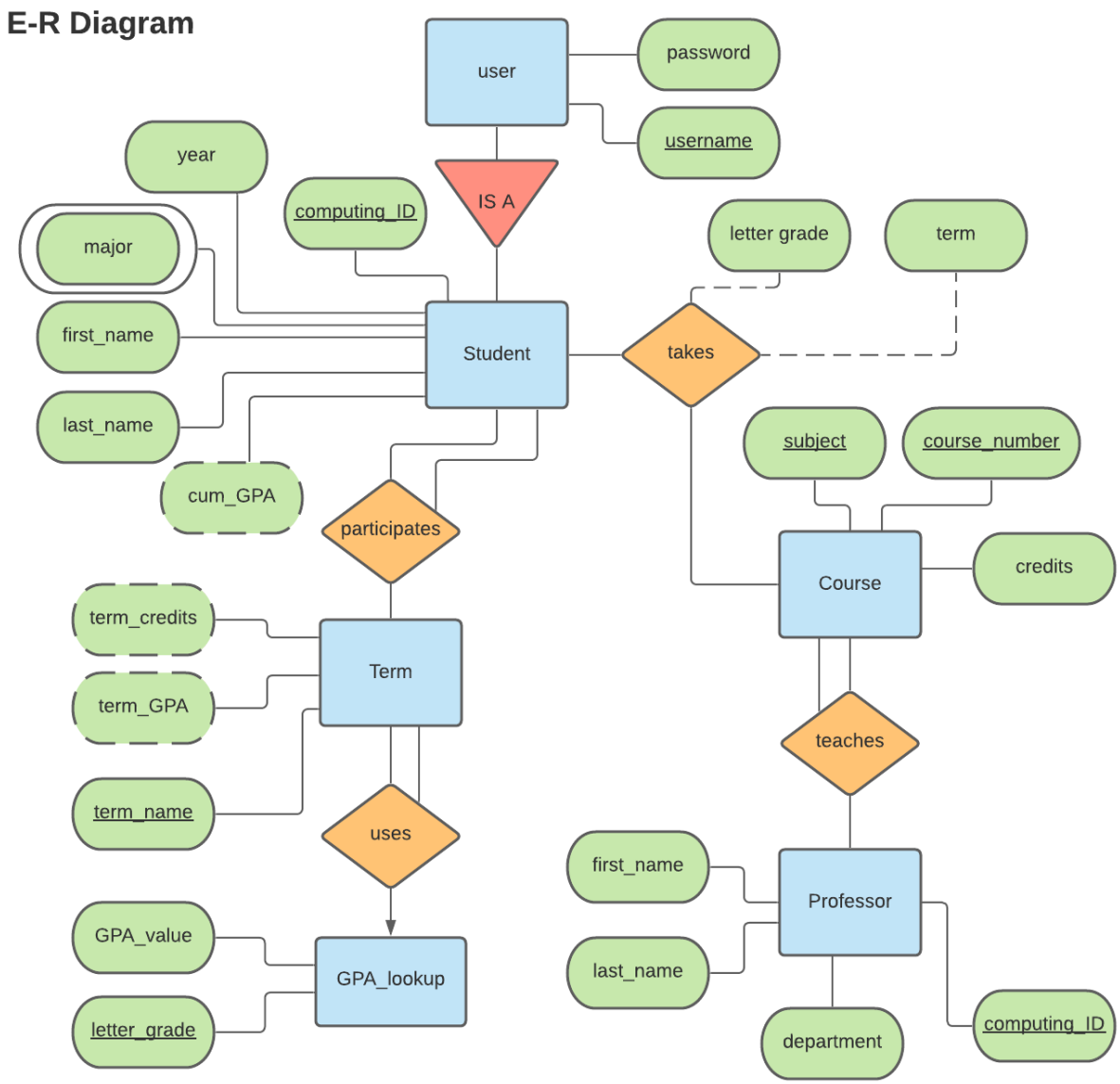
Group: Natural Join

- Syed Ahmad Hasan-Aamir (sah3tx)
- Anna Grace Calhoun (agc8a)
- Shoaib Rana (smr8gv)
- Carter Bristow (cb6xj)

Final Deliverable Report

1. Database Design

Include a final version of your E-R diagram



Include a final version of your tables written in schema statements

Schema Statements

1. course(subject, course number, credits)
2. student(computing ID, first_name, last_name, year, cumGPA)
3. student_major(computing_id, major)
4. participates(computing_ID, term_name)
5. term(term_name, credits, GPA)
6. GPA_lookup(letter grade, GPA_value)
7. professor(computing ID, first_name, last_name, department)
8. uses(term_name, letter_grade)
9. takes(computing ID, subject, course_number, letter_grade, term)
10. teaches(computing_ID, subject, course_number) user(username, password)

2. Database Programming

Specify where you host your database

We hosted our database on the CS server.

Specify where you host your app — the deployment environment needed to deploy and run your project

We hosted the app on netlify. The link can be found on our Github repository's ReadMe page.

Describe and include all instructions / steps needed to deploy and run your project

Extract all files from the zip folder. Access the project though the netlify link. Because the app is hosted on Netlify, the user need not deploy it. We, the developers, took the following steps to set everything up:

Created and hosted the database on the CS server.
Connected the CS server database to a node.js server, which is hosted on Heroku.
Hosted the app on Netlify.
Connected our React app to the node.js server.

The app is hosted locally. In order to use it, visit the link on the Github readme file.
Source Code for the Project is available at: https://github.com/annagracecalhoun/gpa-calc

Discuss how the advanced SQL commands are incorporated in your app, what features of your app use them, and how they reflect the database

First, we added a trigger (see code below) so credits are added to term total when new courses are added to a student's course history or course grades are updated. This trigger is activated when people use the app to add a new course or update one of their

course grades. By automatically changing the total credits, it means the modal showing total credits can update simply by simply pulling this value from the database, rather than looping through and summing all of the course credits.

```
DELIMITER //
CREATE TRIGGER addCredits
BEFORE INSERT on takes
FOR EACH ROW
UPDATE term tc
       SET tc.term_credits = (
              SELECT SUM(credits) student
              NATURAL JOIN takes
              NATURAL JOIN course
              WHERE computing_ID = "student computing ID" AND term = "desired term")
       WHERE term_name = "desired term";
COMMIT;
END //
DELIMITER;
```

Secondly, we added a check to make sure that no course's credits exceeds the maximum (assumed to be 5 credits). This takes place when someone tries to add a new course. Their addition will be rejected if the credits are greater than 5, prompting the user to try again. By having this check embedded in the database, it meant we didn't have to check it in the application code.

```
CREATE ASSERTION valid_grade
CHECK (5 > (SELECT credits FROM course));
```

Lastly, we checked that students don't have over 3 majors. Again, doing this on the database side made our application development simpler. This will prevent the user from adding a 3rd major to their account.

```
CREATE ASSERTION maj_count
CHECK (4 > (SELECT COUNT(major) FROM student_major GROUP BY computing_ID));
```

3. Database Security at the Database Level

Specify whether the security is set for developers or end users

End users

Discuss how you set up security at the database level (access control)

Certain log-in credentials only grant user-level access. This is intended to bolster security at the database level by preventing malicious modifications. Users' access is limited--they cannot create or drop tables. Further, they cannot modify gpa_lookup, professors, or other tables that don't contain their personal information. We kept these permissions in mind as we did security

at the database level, creating a role of “studentRole,” which has privileges to do certain insert, select, update, and execute queries. The end user only has object privileges, not system privileges. Taken together, these restrictions ensure the integrity of the database and prevent inadvertent modifications that could sink the app.

Submit the SQL commands you use to limited / set privileges

```
CREATE USER 'studentRole'@'%' IDENTIFIED VIA
mysql_native_password USING '***';
// create studentRole with a password

GRANT SELECT ON takes to studentRole;
GRANT SELECT ON term to studentRole;
GRANT SELECT ON course to studentRole;
GRANT SELECT ON student to studentRole;
GRANT SELECT ON student_major to studentRole;
GRANT SELECT ON user to studentRole;

GRANT UPDATE ON user to studentRole;
GRANT UPDATE ON takes to studentRole;
GRANT UPDATE ON course to studentRole;
GRANT UPDATE ON student to studentRole;
GRANT UPDATE ON student_major to studentRole;

GRANT EXECUTE ON user to studentRole;
GRANT EXECUTE ON takes to studentRole;
GRANT EXECUTE ON student_major to studentRole;
GRANT EXECUTE ON student to studentRole;

GRANT DELETE ON user to studentRole;
GRANT DELETE ON takes to studentRole;
GRANT DELETE ON student_major to studentRole;
GRANT DELETE ON student to studentRole;
```

Root user pw: G96u9B7hjmbI7BH9
Deployed service [default] to [https://cs4750-332714.uk.r.appspot.com]

4. Database Security at the Application Level

Discuss how database security at the application level is incorporated in your project

1) Guarded against SQL injection attacks on the login screen

2) Only allowed for insertions related to the account user (for example, couldn't insert a class taken by another student)
3) Have type checks for inputs (can't input a non-numeric value for course credits)

We used methods discussed in lecture to prevent SQL injection attacks on the login screen. Specifically, we only allowed insertions related to the user account and implemented input type checking. The former prevents insertions that could threaten the security of the application. The latter prevents injection attacks that exploit naive code that does not perform rigorous type checks.

Submit code snippet(s) to illustrate how security aspect is implemented and to support your discussion

1) In places where injection attacks could occur, we used `mysql`'s (the package our node server uses to connect to the database) method for protecting against injection attacks - their built in `escape` function. An example of how we used this in the server endpoints is shown below. You can read more about this function in the `mysql` documentation.

```
app.get('/api/create', (req, res) => {
   const useName = req.body.useName;
   const pw = req.body.pw;

   const userInsert = 'SELECT * FROM user where username = ' + connection.escape(useName) + 'AND password = ' + connection.escape(pw) + ';';
   //const userInsert = 'CONNECT user (username, password) VALUES (?, ?);'
    db.query(userInsert, (err, result) => {
        console.log(err);
    })
});
```

2) While we also limited access through the database credentials, we also paid attention to this on the application side. In our node.js server (the layer connecting our app and the sql database), rather than having endpoints which accepted general parameters and constructed a wide variety of queries and commands, we created specific endpoints only for the abilities we wanted end users to be able to have. These specific endpoints are detailed below.
    a) The app can only "get" or read from the user table (to either validate or reject their login), and the term, takes, gpaVal, student_major, and course table to get the academic data relevant to the logged in user. An example endpoint, which retrieves the different gpa values of different grades is shown below.

```
app.get('/api/gpaVal', (req, res) => {
    const getGpa = 'SELECT * FROM gpa_lookup;'
    db.query(getGpa, (err, result) => {
        res.send(result);
    })
})
```

    b) Users can only post to (modify or add to) the user table (when creating a new account), takes, student_major, and student courses. All of the update endpoints take parameters from the site, specifically the computing id of the logged in

student, so that the user can only modify data relating to their account. Example post endpoints are shown below.

```
app.post('/api/updateCourse', (req, res) => {
   const compID = req.body.compID;
   const subject = req.body.subject;
   const courseNum = req.body.courseNum;
   const letGrade = req.body.grade;

   const courseUp = 'UPDATE takes set letter_grade = ? WHERE computing_ID = ? AND subject = ? AND course_number = ?;'
    db.query(courseUp, [letGrade, compID, subject, courseNum], (err, result) => {
        console.log(err);
    })
});
```

```
app.post('/api/delCourse', (req, res) => {
   const compID = req.body.compID;
   const subject = req.body.subject;
   const courseNum = req.body.courseNum;
   const letGrade = req.body.grade;

   const courseDel = 'DELETE FROM takes WHERE computing_ID = ? AND subject = ? AND course_number = ? AND letter_grade= ?;'
    db.query(courseDel, [compID, subject, courseNum, letGrade], (err, result) => {
        console.log(err);
    })
});
```

3) For inputs which should be numeric, we used `!inNan(input)` to check whether the inputs were a valid number before proceeding with further action. An example is shown below.

```
const changeGrade = (e) => {
    if (!isNaN(e.target.value)) {
    setcGrade(e.target.value)
};
};
```

We also checked to make sure accounts being created had usernames which were correctly formatted uva emails.

```
const createAccount = () => {
  const valUsername = '/@virginia.edu';
  if (userName.includes('@virginia.edu')) {
  Axios.post('http://localhost:3001/api/create', {useName: userName, pw: passWord}).then(() => {
    setuserName('');
    setpassWord('');
  }); }
}
```