TEAM 23

COSC3380: Database ER design, Normalization and Web App An Enterprise Airline Database System

HOMEWORK 3 REPORT

Team 23, and our assignment is target 3: airport flight management

1. Our web page contain 7 parts:

Each table should be able to be reached through the web application. This can be done by clicking each individual tab on the web application. The tabs will change to a crimson color when hovering over them, and when clicked the information section will display the designated table to the user. Here is what one can do on each of the individual tabs. VIDEO LINK: https://www.youtube.com/channel/UCfkSQt0u2HwOPYCjLj6Zkig/videos

- Search flights tab
  - The user can insert a flight id and all the information regarding that flight id will be displayed on the screen upon clicking search, if the flight id does not exist nothing will display and an alert will tell the user of the case.
- All flights tab
  - Here the user can add a new flight by inserting new values to the flight table. A user must input flight id, date1, time1, date2, time2, airport code, airport code, and plane id. Upon insertion the information will be inserted into the table and user will be able to see immediately. By creating a new flight, the web app also inserts a new row for flight extras, and plane support, which the user should update the information later. The user is also able to update and delete existing flights, this is the only place on the web app where the user can delete something. All values must be acceptable in order for a transaction to go through, for example if an airport code

is not an actual airport code defined through the airport table, a flight cannot be added or updated.

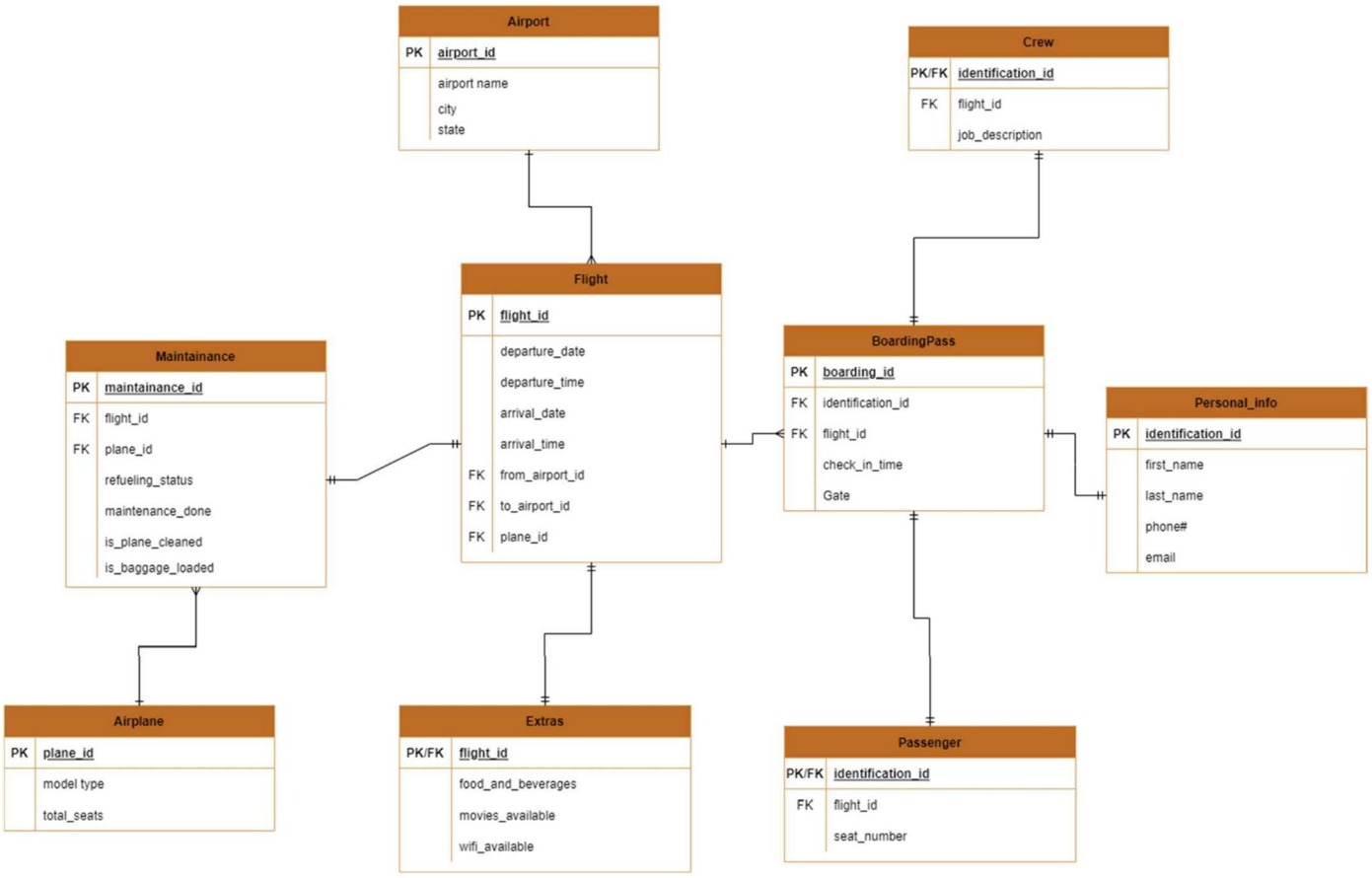
- Flights extras
  - Here the user can see what will be available during a flight such as food andbeverages, movies and wifi. If something turns out to be available that was currently not, it can be updated, same as if something turns out to no be available. On the edit modal, there is options for each value, several options for food and beverage, yes and no only for movies and Wi-Fi. This prevents the user from having to type this information manually and make a mistake.
- Plane support
  - Here the user can see whether the plane is ready for take off or not. It displays fuel status, maintenance needed, if the plane is cleaned, and if the baggage has beenloaded or not. The user is also able to update this to keep up with work done for a flight. On the edit modal, there is options for each value, options for fuel status, maintenance, cleaning, and baggage. This prevents the user from having to type this information manually and make a mistake. The only manual typing value is a plane id, and if the plane id does not refer to a proper plane model through the planes table, the transaction should not go through.
- Boarding information
  - Here the user can see all the boarding information for each crew member and passenger. It is just a display, and no information can be edited here as crew and passenger information is to be handled by another database.
- Search passenger/ search crew
  - Here the user can use the identification id of a person to look them up and get their personal information, check in time, and gate. If the identification id does not exist or is inputted wrong, an alert will tell the user of the case
- Airports

o Here the user can see all the airports by code and see their name and what city and state they belong to.

- Planes

o Here the user can see all the planes by code and see what model and how many total seats it has. The user is also able to insert new plane models but only if the plane model is not already there.

ER diagram:



2. Database design

Our database as seen has a total of 9 tables, flights being the main table used through the web app. For each flight there is one support ticket and one extras ticket to show more information about the flight. This gives those tables a one-to-one relationship. Multiple maintenance ticket can refer to different airplane model more than once, not individual single planes but the plane type, giving this relationship one to many. Two other one to many is from

flights to airports and flights to boarding passes. There are many flights that go to from one single airport to another, and one flight contains many passengers and crew members (aka people boarding).

We choose to do extras (food and beverages, Wi-Fi, and movies) per flight instead of per passenger. This was because passengers and crew information were to be prefilled in the data base. If we went and did it by passenger, it would mean more information to be prefilled and less information to be altered with in the web app. Other information that we decide to be prefilled was the airports information. Airports are persistent and don't disappear or appear out of nowhere, compared to flights. Where new flights appear each day and current flights can be cancelled or moved to a different date/time. Same goes to current airplane models for use, new plane models are not produced like they are iPhone models, and once manufactured they are here to stay for the long run. Therefore we allowed new plane models to be added if needed, but other than some preexisting planes were prefilled.

As for our normalization we strived to reach the highest normalization form. None of the tables are in 1nf or 2nf. To avoid being in 3nf and not BCNF, we separated passenger individuals and crew members from being in one single table and connected them through their boarding pass. A crew member should still need a boarding pass to get on the plane they are going to work on, as a passenger needs one to get on the plane they are going to ride on. To allow 3nf, we broke of some of that information off and made the personal table. This table holds all the personal information of an individual which all depends on the primary key (identification #) making it bcnf. By doing this the board pass table is BCNF and not 3nf now.

If we would have left that information on the board pass table, we would have a transitive dependency as the information would depend on the identification # and not the pk (board_pass_id). This goes along with the flights table and the airports table. If we included the airports table information in the flights table as one whole, the airport name, city, and state

would depend on the airport code and not the pk (flight_id). We separated information off into their own individual tables as necessary to try and make them all BCNF.

3. Transactions and SQL

To show and display tables on the web app we used simple select * from queries such as this one:

//This is a very basic query to display the airports

```
SELECT * FROM airports ORDER BY state, airport_id;
```

The only somewhat complicated select statement to display information was this one right here :

//This is used to display the information in the boarding information tab

```
SELECT * FROM board_pass
JOIN personal ON board_pass.identification_id = personal.identification_id
JOIN passenger ON personal.identification_id = passenger.identification_id
UNION
SELECT * FROM board_pass JOIN personal ON board_pass.identification_id =
personal.identification_id
JOIN crew ON personal.identification_id = crew.identification_id
ORDER BY board_id;
```

There are more basic select * from statements and functions to display the other tables, and those functions and queries are ran on the startup of the web app and when a table is updated by the user. Such as right here, the function selectFlight(); is ran and calls index.js through fetch and gets results through response

```
selectFlight();
async function selectFlight() {
    try {
        // GET all records from "http://localhost:5000/flights_table"
        const response = await fetch("http://localhost:5000/flights_table")
        const jsonData = await response.json();
        setFlight(jsonData);
        displayFlight();
    } catch (err) {
        console.log(err.message);
    }
}
```

index.js then calls the query to the sql server through pool.query() and sends back the results

```
app.get('/flights_table', async (req, res) => {
    try {
        let sql = 'SELECT * FROM flights ORDER BY flight_id;\n';
        const allRows = await pool.query(sql);
        writeSQL(sql);
        res.json(allRows.rows);
    } catch (err) {
        console.log(err.message);
    }
});
```

This is allows us to select the information from the sql table and send it to an html table through a function like this to allow the user to see it in the web app (multiple functions for each table were built like this):

```
const displayFlight = () => {
    flights_table.sort((a, b) => {
        return a.key - b.key;
    });
    const dispTable = document.querySelector('#flights-table');
    let tableHTML = "";
    flights_table.map(flights => {
        tableHTML +=
            `<tr key=${flights.flight_id}>
    <th>${flights.flight_id}</th>
    <th>${flights.departured.split('T')[0]}</th>
    <th>${flights.departure}</th>
    <th>${flights.arrivald.split('T')[0]}</th>
    <th>${flights.arrival}</th>
    <th>${flights.from_airport}</th>
    <th>${flights.to_airport}</th>
    <th>${flights.plane_id}</th>
    <th><button class="btn btn-warning" type="button" data-toggle="modal" data-target="#edit-modal-flights" onclick="editFlight(${flights.flight_id})">Edit</button></th>
    <th><button class="btn btn-danger" type="button" onclick="deleteFlight(${flights.flight_id})">Delete</button></th>
    </tr>`;
    })
    dispTable.innerHTML = tableHTML;
}
```

Now for a transactions to occur, all information that is being processed must be validated before the transaction can take place. We validate the information piece by piece by piece so we could identify where the user was going wrong specifically. We did this through one main function, which is this one:

```
async function checkid (id, tname, idname) {
    bod = { identification_id: id, table_name: tname, id_type: idname };
    const checkID = await fetch("http://localhost:5000/IDcheck", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(bod)
    });
    return await checkID.json();
}
```

This function will take a column value, the table name, and the name of the column and check if that value is valid by running the sql through the index.js file as follows:

```
app.post('/IDcheck', async (req, res) => {
    try {
        const { identification_id, table_name, id_type } = req.body;
        let sql = 'SELECT * FROM ' + table_name + ' WHERE ' + id_type + " = " + identification_id + ';';
        if (table_name == 'airports' || table_name == 'planes') {
            sql = 'SELECT * FROM ' + table_name + ' WHERE ' + id_type + " = \'" + identification_id + '\';';
        }
        writeSQL(sql);
        const Row = await pool.query(sql);
        let x = true;
        if (Row.rows[0] == undefined) {
            x = false;
        }
    res.json(x);
    } catch (err) {
        console.log(err.message);
    }
});
```

This allows us to see if a current value is or is not in the table specified, which allows us to decide whether we should let the user insert or make a change with this value. If all specification are met, than a transaction can take place. This can be seen here when inserting a new flight:

```
async function insertFlight() {
    // read data from input
    const inputID = document.querySelector('#k1').value;
    const inputDEPARTURED = document.querySelector('#a1').value;
    const inputDEPARTURE = document.querySelector('#b1').value;
    const inputARRIVALD = document.querySelector('#c1').value;
    const inputARRIVAL = document.querySelector('#d1').value;
    const inputFROM = document.querySelector('#e1').value;
    const inputTO = document.querySelector('#f1').value;
    const inputPLANE_ID = document.querySelector('#g1').value;

    // want x to be false to allow this new flight
    let x = await checkid(inputID, 'flights', 'flight_id');
    // want y, z, and k to be true
    let y = await checkid(inputFROM, 'airports', 'airport_id');
    let z = await checkid(inputTO, 'airports', 'airport_id');
    let k = await checkid(inputPLANE_ID, 'planes', 'plane_id');
    let j = true;
    if (inputDEPARTURED > inputARRIVALD) {
        j = false;
    }
    if (x == false && 99999 < inputID && inputID < 1000000) {
        if (y && z && k && j) {
            try {
                const body = { flight_id: inputID, departured: inputDEPARTURED, departure: inputDEPARTURE, arrivald: inputARRIVALD, arrival: inputARRIVAL, from_airport: inputFROM, to_airport: inputTO, plane_id: inputPLANE_ID };

                const response = await fetch("http://localhost:5000/flights_table", {
                    method: "POST",
                    headers: { "Content-Type": "application/json" },
                    body: JSON.stringify(body)
                });
                const newRow = await response.json();
                flights_table.push(newRow);
                displayFlight();
                inputID.value = '';
                inputDEPARTURED.value = '';
                inputDEPARTURE.value = '';
                inputARRIVALD.value = '';
                inputARRIVAL.value = '';
                inputFROM.value = '';
                inputTO.value = '';
                inputPLANE_ID.value = '';
            } catch (err) {
                console.log(err.message);
            }
        } else if (j == false) {
            alert('departured date cannot be ahead of arrival date');
        } else if (y == false) {
            alert(inputFROM + ' is not a proper airport code,\n refer to the airports table for the codes');
        } else if (z == false) {
            alert(inputTO + ' is not a proper airport code,\n refer to the airports table for the codes');
        } else {
            alert(inputPLANE_ID + ' is not an available model plane,\n refer to the planes table for available aircraft');
        }
    } else if (x) {
        alert(inputID + ' is already flight\n try another 6 digit #');
    } else {
        alert(inputID + ' is not a 6 digit #, \ntry again');
    }
}
```

Here we can see that we check for statements x, y, z, k and j. For example x tell us if a flight id is already in use in the flights table. To insert a new flight we want this to be false because we do not want duplicate flight ids. So this goes through an if statement if x = false and x is a 6-digit number than we proceed to insert the flight assuming all the other statements pass as well. The function takes in the user inputs and builds the transaction through index.js file here:

```
app.post('/flights_table', async (req, res) => {
    try {
        const { flight_id, departured, departure, arrivald, arrival, from_airport, to_airport, plane_id } = req.body;
        let tran = 'INSERT INTO flights\n' +
            'VALUES(' + flight_id + ',\'' + departured + '\',\'' + departure + '\',\'' + arrivald + '\',\n\'' + arrival + '\',\'' + from_airport + '\',\'' + to_airport + '\',\'' + plane_id + '\');\n';
        let newST = 'INSERT INTO support VALUES (' + (flight_id - 77777) + ',\'' + plane_id + '\',' + flight_id + ',\n\'UNKNOWN\', \'NEEDS TO BE CHECKED\', \'UNKOWN\', \'UNKOWN\');\n';
        let flightExtr = 'INSERT INTO extras VALUES ('+flight_id+', \'UNKNOWN\',\'UNKNOWN\',\'UNKOWN\');\n'
        let select = 'SELECT * FROM flights WHERE flight_id = ' + flight_id + ';\n';
        await pool.query(Stran);
        await pool.query(tran);
        await pool.query(newST);
        await pool.query(flightExtr);
        const newROW = await pool.query(select);
        await pool.query(Ctran);
        // if transaction successful type to transaction file
        let x = '' + Stran + tran + newST + flightExtr + select + Ctran;
        writeTran(x); lastSQL(x);
        res.json(newROW.rows[0]);
    } catch (err) {
        console.log(err.message);
    }
});
```

Here is a sample transaction when inserting a flight:

// all values were checked to ensure they were valid before initiating the transaction

```
BEGIN TRANSACTION;
INSERT INTO flights
VALUES(123456,'2021-12-01','21:05','2021-12-01',
'23:05','ABQ','IAH','B2354');
INSERT INTO support VALUES (45679,'B2354',123456,
'UNKNOWN', 'NEEDS TO BE CHECKED', 'UNKOWN', 'UNKOWN');
INSERT INTO extras VALUES (123456, 'UNKNOWN','UNKNOWN','UNKOWN');
SELECT * FROM flights WHERE flight_id = 123456;
COMMIT;
```

To update information, the process is similar to that of inserting. The updated values are checked to make sure they are valid through the checkID() function, and once checked and valid, the transaction is built and sent through the sql server.

```
//Here is a transaction to update some information

BEGIN TRANSACTION;
UPDATE extras SET food_and_bev = "Snacks and Water",
movies = "YES",
wifi = "NO"
WHERE flight_id = 100001;
COMMIT;
```

One problem we were facing was when searching for a specific flight or person. The reason for our problems was probably because we were trying to reach information form more than one table. To solve our problem we created temporary tables to hold that information that we were searching for and updated it as we searched. Another problem was when trying to update the html table through JS. We soon figured out that when searching for a new object, we had to clear the table and insert the new information to only output one object and not more than one. We did this by popping the old object out first, before pushing the new one. Here was our code for doing just that:

```
async function getFlight() {

    // read data from input
    const inputID = document.querySelector('#f_id').value;
    // makes sure key is unique
    const x = await checkid(inputID, 'flights', 'flight_id');
    if (x) {
        try {
            const body = { flight_id: inputID };

            const response = await fetch("http://localhost:5000/searchF_table", {
                method: "POST",
                headers: { "Content-Type": "application/json" },
                body: JSON.stringify(body)
            });
            const newRow = await response.json();
            searchF_table.pop();
            searchF_table.push(newRow);
            displaySflight();
            searchF_table.pop();
            inputID.value = '';
        } catch (err) {
            console.log(err.message);
        }
    } else {
        alert('Flight ' + inputID + ' does not exist\nplease make sure you are inputting the correct 6 digit #' +
            '\nto create a new flight head over to the flights tab');
    }
}
```

Here is a transaction for updating the temporary table for searching for a flight:

```
//To search for a flight, a temporary table is used to hold the information //and a transaction is
used to update and retrieve that information
BEGIN TRANSACTION;
DELETE FROM hold3;
INSERT INTO hold3
SELECT flights.flight_id, departured, departure, arrivald, arrival, from_airport,
to_airport,
flights.plane_id, food_and_bev, movies, wifi, support_no, refueling, maintenance,
cleaning, baggage
FROM flights
JOIN extras ON flights.flight_id = extras.flight_id
JOIN support ON flights.flight_id = support.flight_id
WHERE flights.flight_id = 100001;
SELECT * FROM hold3;
DELETE FROM hold3;
COMMIT;
```

4. Summary

Throughout the web app there are several tables from the data base that can be reached. Each table is referred through a different tab and reached through different JS functions. In each tab, the user is able to either search for information, see information, or update/add/delete information. At the bottom of the screen there is a show me button, this button allows the user to see the last ran sql query ran by the web app. We created our database to have tables at the

highest of normalization as possible, allowing us to have table that are many to one or one to many. We learned how to put to use what we learned in class from sql to ER diagrams to normalization. We also got the chance to work with html and java script to a whole new extent, or at least our group did, as we have never worked with these coding languages like we did for this web app.