Week 10 Study Guide

RDBMS And Database Entity Objectives

Define what a relational database management system is

The RDBMS is a software application that you run that your programs can connect to so that they can store, modify, and retrieve data.

Describe what relational data is

Relational Data is data that is related in some way. For instance user data in a photo sharing application might have a relationship with the photo data and photo data and user data might both have a relationship with comment data.

There are three types of relationships you can have

1. One to One
2. One to Many
3. Many to Many

Define what a database is

A database is a collection of structured data stored in a Database "System" or "Server"

Define what a database table is

A database can have many tables, a table is made up of several `Columns`. Each `Column` is of a certain type. A table `Schema` describes the columns in a table. Tables also contain `Rows` which hold the actual data for the table.

Describe the purpose of a primary key

A primary key is a single unique column in a database table.

Describe the purpose of a foreign key

A *foreign key* is an integer column in a table which holds the value of a matching *primary key* from another table. A *foreign key constraint* insures that the id stored in the foreign key column is a valid primary key in the related table.

Connect to an instance of PostgreSQL with the command line tool psql

Usage of psql:

```
psql -U <database username> -h <hostname> <database name>
```

- `-U` defaults to the same as your unix username
- `-h` does not have a default value. If you leave it out, psql connects through a unix socket instead of connecting to a hostname through the network
- `<database name>` defaults to be the same as whatever `<database username>` is

You can set these two *environment variables* (`PGHOST` and `PGUSER`) in your shell to override the default behavior `-U` and `-h`

For example, to always connect to localhost with the database user `postgres` you would set them to this:

```
export PGHOST=localhost
export PGUSER=postgres
```

You can create a hidden file called `.pgpass` and put it into your unix home directory to set the password for psql, so you don't have to type it everytime.

Note: PostgreSQL comes with a default database called 'postgres' and a default superuser names 'postgres'

Identify whether a user is a normal user or a superuser by the prompt in the psql shell

You use the `\du` command. If a user is a superuser it will have the `Superuser` attribute.

```
postgres-# \du
                                   List of roles
       Role name        |                         Attributes                         | Member of
------------------------+------------------------------------------------------------+-----------
 postgres               | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 my_user                |                                                            | {}
```

Create a user for the relational database management system

```
CREATE USER <database username> WITH PASSWORD <password> <attributes>;
```

Attributes can be things like `SUPERUSER` or permissions like `CREATEDB` , or any of the other attributes listed in the documentation

Create a database in the database management system

Configure a database so that only the owner (and superusers) can connect to it

```
CREATE DATABASE <database name> WITH OWNER <database username>
```

Create Database Documentation

View a list of databases in an installation of PostgreSQL

You can use the `\l` command in psql to do this.

```
postgres-# \l
                                      List of databases
      Name       |         Owner          | Encoding |  Collate   |   Ctype    |   Access privileges
-----------------+------------------------+----------+------------+------------+-----------------------
 aa_times        | aa_times               | UTF8     | en_US.utf8 | en_US.utf8 |
 postgres        | postgres               | UTF8     | en_US.utf8 | en_US.utf8 |
 project_manager | project_management_app | UTF8     | en_US.utf8 | en_US.utf8 |
 recipe_box      | recipe_box_app         | UTF8     | en_US.utf8 | en_US.utf8 |
 template0       | postgres               | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
                 |                        |          |            |            | postgres=CTc/postgres
 template1       | postgres               | UTF8     | en_US.utf8 | en_US.utf8 | =c/postgres          +
                 |                        |          |            |            | postgres=CTc/postgres
(6 rows)
```

Create tables in a database

The basic format of `CREATE TABLE` is this:

```
CREATE TABLE <table name> (
  <column name> <data type>,
  <column name> <data type>,
  ...
  <column name> <data type>
);
```

You can find a list of possible data types in the PostgreSQL documentation

View a list of tables in a database

You can use the `\dt` command in psql to do this

```
aa_times-# \dt
 public | people   | table | aa_times
 public | sections | table | aa_times
 public | stories  | table | aa_times
```

Identify and describe the common data types used in PostgreSQL

Here are some of most common datatypes

- SERIAL
- VARCHAR
- TEXT
- NUMERIC
- INTEGER
- BOOLEAN
- TIMESTAMP

Describe the purpose of the UNIQUE and NOT NULL constraints, and create columns in database tables that have them

UNIQUE Constraint

Unique constraints ensure that the data contained in a column, is unique among all the rows in the table.

```
CREATE TABLE products (
    id integer UNIQUE,
    name text,
    price numeric
);
```

UNIQUE Documentation

NOT NULL Constraint

A not-null constraint simply specifies that a column must not assume the null value.

```
CREATE TABLE products (
    id integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

NOT NULL Documentation

Create a primary key for a table

A primary key constraint indicates that a column can be used as a unique identifier for rows in the table. This requires that the values be both unique and not null.

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name text,
    price numeric
);
```

PRIMARY KEY Documentation

Create foreign key constraints to relate tables

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table.

You can use the `REFERENCES` keyword:

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    product_id integer REFERENCES products(id),
    quantity integer
);
```

Or you can use the `FOREIGN KEY` syntax on a separate line

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    id SERIAL PRIMARY KEY,
    product_id integer,
    quantity integer,
    FOREIGN KEY (product_id) REFERENCES products(id)
);
```

FOREIGN KEY Documentation

Explain that SQL is not case sensitive for its keywords but is for its entity names

Both of these are valid SQL, although it is a convention to uppercase the SQL keywords.

```
SELECT name, quantity FROM orders;
```

```
select name, quantity from orders;
```

This is NOT the same:

```
SELECT "Name", "Quantity" FROM "Orders";
```

(Note, in PostgreSQL, if we do not put double quotes around the column and table names, postgres will lowercase them first before running the query)

So a query like this:

```
SELECT Name, Quantity FROM Orders;
```

Will be turned into this before postgres runs it.

```
SELECT name, quantity FROM orders;
```

So if you actually have capital letters in your column and table names, make sure to always double-quote them.

SQL Objectives

1. How to use the SELECT … FROM … statement to select data from a single table.

```
SELECT population
FROM countries;
```

2. How to use the WHERE clause on SELECT, UPDATE, and DELETE statements to narrow the scope of the command.

```
SELECT population
FROM countries
WHERE name = 'France';
```

```
UPDATE countries
SET population = 1000
WHERE name = "Vatican City";
```

```
DELETE FROM planets
WHERE name = 'Pluto';
```

3. How to use the JOIN keyword to join two (or more) tables together into a single virtual table.

```
SELECT player_name
FROM players
JOIN teams ON teams.id = players.team_id
WHERE team_name = 'Lakers';
```

4. How to use the INSERT statement to insert data into a table.

```
INSERT INTO nfl_players
VALUES (DEFAULT, 'Joe Burrow', 'Bengals');
```

5. How to use a seed file to populate data in a database.

```
-- seed_file.sql
CREATE TABLE IF NOT EXISTS bands (
    id SERIAL,
    name VARCHAR(50) NOT NULL,
    vocalist VARCHAR(50),
    PRIMARY KEY (id)
);

INSERT INTO bands
VALUES (DEFAULT, 'The Beatles', 'John Lennon');

INSERT INTO bands
VALUES (DEFAULT, 'Queen', 'Freddie Mercury');

INSERT INTO bands
VALUES (DEFAULT, 'U2', 'Bono');
```

SQL Learning Objectives Pt 2

1. How to perform relational database design

Database design is a difficult subject with few absolutes. Experience building applications and resolving design issues will help you to make judgement calls.

But as a starting point, start with these four steps:

```
1. What are the main entities in my application (nouns)?
2. How are they related to one another?
3. Can I normalize any information?
```

2. How to use transactions to group multiple SQL commands into one succeed or fail operation

```
```js
async function transferFunds(pool, account1, account2, amount) {
    const balanceQ = 'select balance from "AccountBalances" where account_id = $1';
    const updateBalanceQ = 'update "AccountBalances" set balance=$1 where account_id = $2';

    await pool.query("BEGIN;");
    try {
        const balance1 = await pool.query(balanceQ, [account1]);
        if (balance1 < amount) {
            throw ("Not enough funds");
        }
        const balance2 = await pool.query(balanceQ, [account2]);

        await pool.query(updateBalanceQ, [balance2 + amount, account2]);
        await pool.query(updateBalanceQ, [balance1 - amount, account1]);
        await pool.query("COMMIT;");
    } catch (e) {
        await pool.query("ROLLBACK;");
    }

}
```
```

3. How to apply indexes to tables to improve performance

Indexes are used to optimize queries. We add indexes to columns, in order to allow the Query Planner to more efficiently filter matching rows.

By evaluating the WHERE clause of a poorly performing query, we can determine which columns are involved in the query, and which indexes the Query Planner might be able to take advantage of. Knowing which indexes (there are lots of types) to add and when is an advanced topic in Database Management.

4. Explain what and why someone would use EXPLAIN

`EXPLAIN` and `EXPLAIN ANALYZE` are the tools that we have to improve poorly performing queries. By applying these keywords to the front of a query we are able to learn about which indexes Postgres is able to utilize, in comparison to which tables must be sequentially scanned.

Using `EXPLAIN` is also an advanced topic in Database Management. We should know that these tools exist, that they can give you insight into the performance of our queries, but we should not be expected to use them.

5. Demonstrate how to install and use the node-postgres library and its Pool object to query a PostgreSQL-managed database

To install node-postgres:

```
```bash
$ npm install node-postgres
```
```

Somewhere in your JS:

```
```js
const { Pool } = require('pg');

const pool = new Pool({
    database: <mydbname>,
    hostname: <mydbhostname>,
    username: <username>,
```
```

````
    password: <password>
});

const result = await pool.query("SELECT 1;");
```
````

6. Explain how to write prepared statements with placeholders for parameters of the form "$1", "$2", and so on

Prepared Statements allow us to create queries that don't need to know the constant values needed for the where clause in advance. Consider:

````
```js
const loginQuery = 'SELECT * FROM users WHERE username = $1 and password = $2;';

async function loginUser(username, password) {
    const results = await pool.query(loginQuery, [username, password]);
    ...
}
```
````

In addition the benefit of not having to use template strings, the `node-postgres` library is providing us a *huge* hidden security benefit! Prepared statements exist primarily to protect our applications from [https://developer.mozilla.org/en-US/docs/Glossary/SQL_Injection](SQL-injection attacks) - one of the most common type of *hacking* attacks that we see on the web.

ORM Objectives

How to install, configure, and use Sequelize, an ORM for JavaScript

We use Sequelize 5 in this course.

Installing Sequelize 5

Use npm to install `sequelize`, `sequelize-cli`, and `pg` because we are using PostgreSQL.

```
npm install sequelize@^5 sequelize-cli@^5 pg
```

Configuring sequelize

First you must initialize your project with sequelize

```
npx sequelize-cli init
```

Then edit the `config/config.json` file to add the appropriate database name, username, password and dialect.

The dialect should be `postgres` because that's the RDBMS we are using.

Using Sequelize

You generate Models, edit the models and migration files to your liking, and then use the model classes in your code to query, insert, update and delete data.

```
npx sequelize model:generate --name Cat --attributes "firstName:string,specialSkill:string"
```

This will generate two files, a migration file and a model file.

Now we can edit both to add any custom columns, constraints or associations (relationships).

After running our migrations, we can do queries.

We import our model like so:

```
const { Cat } = require('./models');
```

And now we can do queries against the model.

```
const cat = await Cat.findByPK(1);
```

How to use database migrations to make your database grow with your application in a source-control enabled way

Migrations are a set of instructions written in JavaScript to create, update, and remove tables and columns from our database.

Migrations always run ONCE in the order of the timestamps that prefix the filenames.

You can generate a migration file either by generating a model, or by generating a stand-alone migration like so.

```
npx sequelize-cli migration:generate --name add_last_name_column_to_users
```
```

How to perform CRUD operations with Sequelize

CRUD = "Create, Read, Update and Delete"

Create

You can use the `<Model>.create()` method, or the `<Model>.build()` along with `<instance>.save()`

Read

You can use `<Model>.findOne()`, `<Model>.findAll()`, or `<Model>.findByPk()` to query
and read data

Update

You can just set properties on an instance and call `.save()` or you can use the `.update()` method.

Delete

You can use the `.destroy()` method on an instance to destroy a single row, or `<Model>.destroy()`
with a `where` clause to destroy multiple rows.

Check out the Sequelize Documentation or Sequelize Cheatsheet for more examples and details.

How to query using Sequelize

We call static methods on the Model to query the database.

These are some of the most common ones.

```
const <instance> = await <Model>.findOne(<query options>);

const <array> = await <Model>.findAll(<query options>);

const <instance> = await <Model>.findByPk(<query options>);
```

Check out the Sequelize Documentation or Sequelize Cheatsheet for more examples and details.

How to perform data validations with Sequelize

You define data validations on the Sequelize Model:

You add a validate property to a column definition, and use one of the many
built in validations to define what is allowed for a column.

This uses the common `notNull` and `notEmpty` validations:

```
const Cat = sequelize.define('Cat', {
   firstName: {
     type: DataTypes.STRING,
     allowNull: false,
     validate: {
       notNull: {
         msg: "firstName must not be null",
       },
       notEmpty: {
         msg: "firstName must not be empty",
       }
     },
   }
```

The documentation contains an exhaustive list.
html#validations) of all possbile validations:

Note: Try not to get confused by the documentation's use of ES6 `class` based Sequelize models, for validation, the property still applies the same way, just in the `init()` method instead of the `define()` method.

How to use transactions with Sequelize

You call the sequelize.transaction() function and pass is a callback.

The callback will receive a copy of the transaction id `tx`.

You pass this id to any sequelize methods (like save()) that you want to be
a part of the transation.

If this callback succeeds without errors, sequelize will commit the transaction.

In this example if either `save()` fails, the entire transaction will be rolled
back and the database will not be changed.

```
await sequelize.transaction(async (tx) => {
    // Fetch Markov and Curie's accounts.
    const markovAccount = await BankAccount.findByPk(
        1, { transaction: tx },
    );
    const curieAccount = await BankAccount.findByPk(
        2, { transaction: tx }
    );

    // Increment Curie's balance by $5,000.
```

```
    curieAccount.balance += 5000;
    await curieAccount.save({ transaction: tx });

    // Decrement Markov's balance by $5,000.
    markovAccount.balance -= 5000;
    await markovAccount.save({ transaction: tx });
});
```

It's a good idea to wrap the call to sequelize.transaction in a `try catch` block, so we can handle the transaction failing in a graceful way.

```
async function main() {
  try {
    // Do all database access within the transaction.
    await sequelize.transaction(async (tx) => {
      // Fetch Markov and Curie's accounts.
      const markovAccount = await BankAccount.findByPk(
        1, { transaction: tx },
      );
      const curieAccount = await BankAccount.findByPk(
        2, { transaction: tx }
      );

      // Increment Curie's balance by $5,000.
      curieAccount.balance += 5000;
      await curieAccount.save({ transaction: tx });

      // Decrement Markov's balance by $5,000.
      markovAccount.balance -= 5000;
      await markovAccount.save({ transaction: tx });
    });
  } catch (err) {
    // Do something useful here like log the error or send a message to the user
  }

  await sequelize.close();
}

main();
```