

COMS W4111: Introduction to Databases

Spring 2024, Sections 002/V02

Homework 3

Introduction

- This notebook contains HW3. **Both Programming and Nonprogramming tracks should complete this homework.**
- You will submit **PDF and ZIP files** for this assignment. Gradescope will have two separate assignments for these.
- For the PDF:
 - The most reliable way to save as PDF is to go to your browser's menu bar and click `File -> Print`. Switch the orientation to landscape mode, and hit save.
 - **MAKE SURE ALL YOUR WORK (CODE AND SCREENSHOTS) IS VISIBLE ON THE PDF. YOU WILL NOT GET CREDIT IF ANYTHING IS CUT OFF.** Reach out for troubleshooting.
 - **MAKE SURE YOU DON'T SUBMIT A SINGLE PAGE PDF.** Your PDF should have multiple pages.
- For the ZIP:
 - Zip a folder containing this notebook and any screenshots.
 - You may delete any unnecessary files, such as caches.

Setup

```
In [ ]: %load_ext sql
        %sql mysql+pymysql://root:dbuserdbuser@localhost
        %sql SELECT 1
```

The sql extension is already loaded. To reload it, use:

```
%reload_ext sql
mysql+pymysql://general_user:***@localhost/s24_hw3
* mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
```

Out[]: 1

1

In []: %%sql

```
drop schema if exists s24_hw3;
create schema s24_hw3;
use s24_hw3;
```

```
mysql+pymysql://general_user:***@localhost/s24_hw3
* mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
4 rows affected.
1 rows affected.
0 rows affected.
```

Out[]: []

In []: `import copy`
`import math`

`import pandas`
`import pymysql`
`from sqlalchemy import create_engine`

```
sql_conn = pymysql.connect(
    user="root",
    password="dbuserdbuser",
    host="localhost",
    port=3306,
    cursorclass=pymysql.cursors.DictCursor,
    autocommit=True,
)
engine = create_engine("mysql+pymysql://root:dbuserdbuser@localhost")
```

```
cur = sql_conn.cursor()
res = cur.execute("SELECT 1")
res = cur.fetchall()
res
```

Out[1]: [{'1': 1}]

Written

- As usual, try to keep things short. Do not bloviate.
- You may use external resources, but you should cite your sources.

W1

Explain and list some differences between

- RAM
- Solid state drives
- Hard drives

RAM is used by programs that are running on the computer. It is a volatile kind of storage but is the fastest of the three.

SSDs and HDDs are both types of non-volatile storage.

HDDs store data on a physical disk that spins. There is a reading/writing head that is over the part of memory that is being accessed. Because of the physical design of HDDs (disk), interacting with the data stored there has three phases: 1. Seek: move the head to cylinder or track we want to access 2. Rotation: wait for sector to get under head (disk spins) 3. Transfer: Move data. To optimize data transfer, we try to position blocks spatially so that seeking and rotation time is minimal.

SSDs use NAND flash memory to store data persistently. Because they don't have moving parts they have significantly faster access times than HDDs. When writing to an SSD, you have to write and then erase the original block.

W2

With regards to disk drives, define

- Seek time
- Rotational latency time
- Transfer time/data transfer rate

answer

- seek time is the time it takes to reposition the arm over the correct track.
- rotational latency time is the time it takes for the sector to be accessed to appear under the head.
- transfer time is the rate at which data can be retrieved from or stored to the disk.

W3

Explain the concepts of

- Logical block addressing
- Cylinder-head-sector addressing

answer

- Logical block addressing: LBA is a common scheme for specifying the location of blocks of data (1). It replaces the CHS scheme, and uses a disk controller to translate the block's logical address to a physical address. Blocks are located by index, not their physical location.
- Cylinder-head-sector addressing CHS addressing is a 3D coordinate system for physical addressing that uses vertical, horizontal, and angular coordinates (head, cylinder, and sector, respectively). This system doesn't use a disk controller.

Sources: https://datacadamia.com/io/drive/lba?redirectId=data_storage%3Alba&redirectOrigin=bestEndPageName Wikipedia for CHS

W4

Define and list some benefits of

- Fixed-length records
- Variable-length records
- Row-oriented storage
- Column-oriented storage

Fixed-length records: when records are all the same size.

- Accessing records is simple (we know where they should be)
- handling fixed-size records tends to be more computationally efficient
- If the deletes/inserts balance out over time, we can make this more efficient by storing free records in a linked list without having to move them, which gives us the benefit of quick access *and* quick updates/deletion.

Variable-length records: when the records have differing sizes

- Useful when the records' attributes themselves have differing sizes, like if they can be `NULL` or `varchar`

Row-oriented storage: storing rows in the same block.

- helpful when you need to insert a bunch

Column-oriented storage: storing blocks column-wise

- Good for when only some attributes are accessed (aggregation & analysis)

W5

Explain and list some differences between

- RAID 0
- RAID 1

- RAID 5

RAID 0: two+ physical disks make one single, logical fast disk. The data is split evenly between the two physical disks.

- Has no parity information or redundancy: failure of one physical disk means that there will be total data loss.
- striping with no mirroring or parity

RAID 1: two+ physical disks make one single, logical reliable disk. The data on one disk is copied over to the other one (redundant).

- Has redundancy, so there will be no data loss unless all physical disks fail.
- The array can only be as big as the smallest disk.
- mirroring, with no striping or parity

RAID 5: one big logical disk that is actually made up of 5 smaller "logical" disks.

- The blocks are stored across all 5 disks along with a parity component as well. Parity & redundancy allows us to rebuild data in the event of a failure.
- block-level striping with distributed parity

SQL

Overview

- The `data` directory contains a file `People.csv`. The columns are
 - `nameFirst`
 - `nameLast`
 - `birthYear`
 - `birthCountry`
 - `deathYear`
 - `deathCountry`

- For Nonprogramming students, note that this `People.csv` differs from the one you loaded in HW2. Do not mix the two files.
- **There is no one right answer for this section.** You can come up with and document your own design (as long as they satisfy the requirements).

Create Table

- Create a table based on the structure of `People.csv`
 - **You must add an additional attribute, `personID`, which has type `char(9)`**
 - `personID` should be the primary key of your table
 - `nameFirst` and `nameLast` cannot be null. The other (non-PK) columns can be null.
 - You should choose reasonable data types for the attributes
 - Do not use the `year` data type for `birthYear` or `deathYear`. The [range for year](#) is too small.
 - Your table will be empty for the next few sections. We will insert data later.

In []: `%%sql`

```
CREATE TABLE People (  
    personID CHAR(9) PRIMARY KEY,  
    nameFirst VARCHAR(50) NOT NULL,  
    nameLast VARCHAR(50) NOT NULL,  
    birthYear INT UNSIGNED,  
    birthCountry VARCHAR(50),  
    deathYear INT UNSIGNED,  
    deathCountry VARCHAR(50)  
);
```

```
mysql+pymysql://general_user:***@localhost/s24_hw3  
* mysql+pymysql://root:***@localhost  
mysql+pymysql://root:***@localhost/s24_hw3  
0 rows affected.
```

Out []: []

Person ID Function

- `personID` is formed using the following rules:
 1. The ID consists of three sections: `[lastSubstr][firstSubstr][number]`
 2. `lastSubstr` is formed by lowercasing `nameLast`, then taking the first 5 letters. If `nameLast` is less than 5 letters, use the entire `nameLast`.
 3. `firstSubstr` is formed by lowercasing `nameFirst`, then taking the first 2 letters. If `nameFirst` is less than 2 letters, use the entire `nameFirst`.
 4. For a specific combination of `[lastSubstr][firstSubstr]`, `number` starts from 1 and increments. `number` should be padded to have length 2.
 5. `nameFirst` and `nameLast` may contain periods ".", hyphens "-", and spaces " ". You should remove these characters from `nameFirst` and `nameLast` **before** doing the above substring processing.
- As an example, starting from an empty table, below is what `personID` would be assigned to the following names (assuming they were inserted in the order that they are shown)

<code>nameFirst</code>	<code>nameLast</code>	<code>personID</code>
Donald	Ferguson	fergudo01
David	Aardsma	aardsda01
Doe	Fergue	fergudo02
J. J.	Park	parkjj01

- Write a SQL function that generates a person ID using the above rules
 - You should determine what parameters and return type are needed
 - This function will be called by triggers in the next section. **It is up to you which logic you put in the function and which logic you put in the triggers.**
 - That is, if you plan to place the bulk of your logic in your triggers, then your function could be a few lines.
 - You may define helper functions
 - You may add additional attributes to your table if it helps

In []: `%%sql`

```
CREATE FUNCTION GeneratePersonID(
```



```

    firstName VARCHAR(50),
    lastName VARCHAR(50)
) RETURNS CHAR(9) deterministic
BEGIN
    DECLARE lastSubstr VARCHAR(5);
    DECLARE firstSubstr VARCHAR(2);
    DECLARE paddedNumber VARCHAR(2);
    DECLARE newPersonID CHAR(9);
    DECLARE newIdNumber INT;
    DECLARE maxIdNumber INT;

    SET firstName = REPLACE(REPLACE(REPLACE(REPLACE(firstName, "'", ''), '. ', ' '), '- ', ' '), ' ', '');
    SET lastName = REPLACE(REPLACE(REPLACE(REPLACE(lastName, "'", ''), '. ', ' '), '- ', ' '), ' ', '');

    SET lastSubstr = LEFT(LOWER(lastName), 5);
    SET firstSubstr = LEFT(LOWER(firstName), 2);

    # Current max id for this username
    SELECT MAX(CAST(SUBSTRING(personID, -2) AS UNSIGNED)) INTO maxIdNumber
    FROM People
    WHERE LEFT(personID, LENGTH(personID) - 2) = CONCAT(lastSubstr, firstSubstr);

    IF maxIdNumber IS NULL THEN
        SET newIdNumber = 1;
    ELSE
        SET newIdNumber = maxIdNumber + 1;
    END IF;

    SET paddedNumber = LPAD(newIdNumber, 2, '0');
    SET newPersonID = CONCAT(lastSubstr, firstSubstr, paddedNumber);
    RETURN newPersonID;
END;

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
* mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
0 rows affected.

```

Out[1]: []

Insert and Update Triggers

- We want to automatically generate `personID` using the function above whenever a row is inserted. The user should not need to manually specify it.
- Write a SQL trigger that runs every time a row is inserted
 - The trigger should generate a person ID for the row based on its `nameFirst` and `nameLast` ; it should then set the `personID` for that row.
 - This should occur even if the user attempts to manually set `personID` . The user's value for `personID` is ignored.
 - You should call the function you wrote above
- Write another SQL trigger that runs every time a row is updated
 - There is no `immutable` keyword in MySQL; however, we can simulate immutability using a trigger. If the user attempts to modify `personID` directly, throw an exception.
 - If the user modifies `nameFirst` or `nameLast` such that the `personID` is no longer valid based on the rules in the previous section (specifically, if `[lastSubstr][firstSubstr]` is no longer the same as before), you should re-generate `personID` and re-set it.
 - You should call the function you wrote above
- **You are writing two SQL triggers for this section**

In []: `%%sql`

```
create trigger genID before insert
on People for each row
begin
    set new.personID = GeneratePersonID(NEW.nameFirst, NEW.nameLast);
    set @generated_person_ID = new.personID;
end;

create trigger updatePersonId
before update
on People
for each row
begin
    if old.personID <> new.personID then
        SIGNAL sqlstate '45000'
```

```

        set message_text = 'you cannot modify personID directly.';
    elseif left(old.nameFirst, 2) <> left(new.nameFirst, 2) or left(old.nameLast, 5) <> left(new.nameLast,
        set new.personID = GeneratePersonID(new.nameFirst, new.nameLast);
    end if;
end;

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
* mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
0 rows affected.
0 rows affected.

```

Out[]: []

Create and Update Procedures

- You must implement two stored procedures
 1. createPerson(nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry, personID)
 - A. personID is an out parameter. It should be set to the ID generated for the person.
 - B. All the other parameters are in parameters
 2. updatePerson(personID, nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry, newPersonID)
 - A. newPersonID is an out parameter. It should be set to the ID of the person after the update (even if it didn't change).
 - B. All the other parameters are in parameters.
 - a. personID is used to identify the row that the user wants to update. The other in parameters are the values that the user wants to set.
 - b. **Ignore null in parameters.** Only update an attribute if the in parameter is non-null.
- Depending on how you implemented your triggers, these procedures could be as simple as calling insert / update and setting the out parameters

In []: %%sql

```

CREATE PROCEDURE createPerson(

```

```

    IN nameFirst VARCHAR(255),
    IN nameLast VARCHAR(255),
    IN birthYear INT,
    IN birthCountry VARCHAR(255),
    IN deathYear INT,
    IN deathCountry VARCHAR(255),
    OUT out_personID VARCHAR(255)
)
BEGIN
    INSERT INTO People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
    VALUES (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry);

    SET out_personID = @generated_person_ID;
END;

CREATE PROCEDURE updatePerson(
    IN in_personID VARCHAR(255),
    IN in_nameFirst VARCHAR(255),
    IN in_nameLast VARCHAR(255),
    IN in_birthYear INT,
    IN in_birthCountry VARCHAR(255),
    IN in_deathYear INT,
    IN in_deathCountry VARCHAR(255),
    OUT out_personID VARCHAR(255)
)
BEGIN
    # -- get a new id if the first characters of in_nameLast are different from the first
    # -- 5 of in_personID, OR if the first two characters of in_nameFirst are different from
    # -- the last two alphabetic letters of the name.

    DECLARE lastNameSubstring varchar(5);
    DECLARE firstNameSubstring varchar(2);

    DECLARE idLastName varchar(5);
    DECLARE idFirstName varchar(2);

    SET lastNameSubstring = LEFT(lower(in_nameLast), 5);
    SET firstNameSubstring = LEFT(lower(in_nameFirst), 2);

    SET idLastName = left(in_personID, length(lastNameSubstring));
    SET idFirstName = right(left(in_personID, length(firstNameSubstring) + length(lastNameSubstring)), leng

```

```

IF lastNameSubstring <> idLastName or firstNameSubstring <> idFirstName
  THEN
    SET out_personID = GeneratePersonID(in_nameFirst, in_nameLast);
  ELSE
    # -- Otherwise, retain the existing personID
    SET out_personID = in_personID;
  END IF;

# -- Update the person's information
UPDATE People
SET
  nameFirst = COALESCE(in_nameFirst, nameFirst),
  nameLast = COALESCE(in_nameLast, nameLast),
  birthYear = COALESCE(in_birthYear, birthYear),
  birthCountry = COALESCE(in_birthCountry, birthCountry),
  deathYear = COALESCE(in_deathYear, deathYear),
  deathCountry = COALESCE(in_deathCountry, deathCountry)
WHERE personID = in_personID;

END;

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
* mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
0 rows affected.
0 rows affected.

```

Out []: []

Security

- You must create a new user `general_user` and use security to allow it to perform only `select` and `execute` operations (i.e., no `insert`, `delete`, and `update` operations)

In []: %%sql

```

CREATE USER 'general_user'@'localhost' IDENTIFIED BY 'dbuserdbuser';

# Grant SELECT privileges on your_table_name to general_user
GRANT SELECT ON s24_hw3.People TO 'general_user'@'localhost';

```

```
# Grant EXECUTE privileges on the stored procedures to general_user
GRANT EXECUTE ON PROCEDURE s24_hw3.createPerson TO 'general_user'@'localhost';
GRANT EXECUTE ON PROCEDURE s24_hw3.updatePerson TO 'general_user'@'localhost';

# Remove other privileges from general_user
REVOKE INSERT, DELETE, UPDATE ON s24_hw3.* FROM 'general_user'@'localhost';
```

```
mysql+pymysql://general_user:***@localhost/s24_hw3
* mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1396, "Operation CREATE USER failed for 'general_user'@'localhost'")
[SQL: CREATE USER 'general_user'@'localhost' IDENTIFIED BY 'dbuserdbuser'];]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

Inheritance Using Views

- A person can be a player or manager
 - That is, a player is-a person, and a manager is-a person
- Describe how you could implement this inheritance relationship given that you already have your `people` table
 - No code is necessary

The `people` table could have an additional field called `isPlayer` which is a boolean and true if the person is a player and false if the person is a manager.

`player` and `manager` would be views on this modified `people` table, where we would `SELECT` out rows based on if `isPlayer` is true or not: if `isPlayer` the row should be in the `player` view, and so on.

Data Insertion Testing

- The cells below load data from `People.csv` to your database
 - No code is required on your part. Make sure everything runs without error.

```
In [ ]: # Load People.csv into a dataframe.
        # You may see NaNs in the non-null columns. This is fine.

people_df = pandas.read_csv("data/People.csv")
```

```
people_df.head(10)
```

```
Out[ ]:
```

	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
0	Ed	White	1926.0	USA	1982.0	USA
1	Sparky	Adams	1894.0	USA	1989.0	USA
2	Bob	Johnson	1959.0	USA	NaN	NaN
3	Johnny	Ryan	1853.0	USA	1902.0	USA
4	Jose	Alvarez	1956.0	USA	NaN	NaN
5	Andrew	Brown	1981.0	USA	NaN	NaN
6	Chris	Johnson	1984.0	USA	NaN	NaN
7	Johnny	Johnson	1914.0	USA	1991.0	USA
8	Albert	Williams	1954.0	Nicaragua	NaN	NaN
9	Ed	Brown	NaN	USA	NaN	NaN

```
In [ ]: def add_person(p):
        """
        p is a dictionary containing the column values for either a student or an employee.
        """
        cur = sql_conn.cursor()

        # This function changes the data, converting nan to None.
        # So, we make a copy and change the copy.
        p_dict = copy.copy(p)
        for k, v in p_dict.items():
            if isinstance(v, float) and math.isnan(v):
                p_dict[k] = None

        # print(f"{p_dict['nameFirst']} {p_dict['nameLast']} ->", end=" ")

        # This provides a hint for what your stored procedure will look like.
        res = cur.callproc(
            "s24_hw3.createPerson",
            # The following are in parameters
            (
```

```

        p_dict["nameFirst"],
        p_dict["nameLast"],
        p_dict["birthYear"],
        p_dict["birthCountry"],
        p_dict["deathYear"],
        p_dict["deathCountry"],
        # The following are out parameters for personID.
        None,
    ),
)

# After the procedure executes, the following query will select the out values.
res = cur.execute("""SELECT @_s24_hw3.createPerson_6""")
result = cur.fetchall()

sql_conn.commit()
cur.close()
# print(result[0]['@_s24_hw3.createPerson_6'])
return result[0]["@_s24_hw3.createPerson_6"] # Return personID

```

- Below is the main data insertion logic
 - add_person calls your createPerson procedure
 - The data directory also contains a file People_Ids.csv, which is the expected personID for each row after it is inserted. We'll use this to check your createPerson implementation.

```

In [ ]: %sql truncate table s24_hw3.people

expected_ids_df = pandas.read_csv("data/People-Ids.csv", header=None)
expected_ids = [e[0] for e in expected_ids_df.values.tolist()]

for i, (p, e_id) in enumerate(zip(people_df.to_dict(orient="records"), expected_ids)):
    p_id = add_person(p)
    assert p_id == e_id, \
        f"Row {i}: Expected {e_id}, but got {p_id} for {p['nameFirst']} {p['nameLast']}"

print("Successfully inserted all data")

```



```
mysql+pymysql://general_user:***@localhost/s24_hw3
* mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
0 rows affected.
Successfully inserted all data
```

Data Updating Testing

- The following cells test your update trigger and `updatePerson` implementation
 - No code is required on your part. Make sure everything runs as expected.
 - The tests assume you just finished the Data Insertion Testing section. You may run into issues if you run the Data Updating Testing section multiple times without resetting your data.

```
In [ ]: # Switch back to root
%sql mysql+pymysql://root:dbuserdbuser@localhost/s24_hw3

def transform(d):
    # %sql returns dict of attributes to one-tuples.
    # This function extracts the values from the one-tuples.
    return {k: v[0] for k, v in d.items()}

def is_subset(d1, d2):
    # Checks if d1 is a subset of a d2
    for k, v in d1.items():
        if k not in d2 or str(d2[k]) != str(v):
            return False
    return True
```

```
In [ ]: # Create new person to test on

%sql call createPerson("Babe", "Ruth", null, null, null, null, @ruthID)
res1 = %sql select * from people p where p.personID = @ruthID
res1_d = transform(res1.dict())
expected_d = dict(
    personID="ruthba01",
    nameFirst="Babe",
    nameLast="Ruth",
    birthYear=None,
```

```

        birthCountry=None,
        deathYear=None,
        deathCountry=None
    )

    print(res1)

    assert is_subset(expected_d, res1_d), \
        f"Row has unexpected value. Expected {expected_d}, but got {res1_d}"

    print("Success")

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.

```

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
ruthba01	Babe	Ruth	None	None	None	None

Success

```

In [ ]: # Update birth country and year
%sql call updatePerson(@ruthID, null, null, 1895, "USA", 1948, "USA", @ruthID)

test_res = %sql select * from people p where p.nameLast = "Ruth"
print(transform(test_res.dict()))
res2 = %sql select * from people p where p.personID = @ruthID
res2_d = transform(res2.dict())
expected_d = dict(
    personID="ruthba01",
    nameFirst="Babe",
    nameLast="Ruth",
    birthYear=1895,
    birthCountry="USA",
    deathYear=1948,
    deathCountry="USA"
)

```

```

)

print(res2)

assert is_subset(expected_d, res2_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res2_d}"

print("Success")

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
{'personID': 'ruthba01', 'nameFirst': 'Babe', 'nameLast': 'Ruth', 'birthYear': 1895, 'birthCountry': 'USA',
'deathYear': 1948, 'deathCountry': 'USA'}
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+-----+-----+-----+-----+-----+-----+-----+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+-----+-----+-----+-----+-----+-----+-----+
| ruthba01 | Babe     | Ruth    | 1895     | USA         | 1948     | USA          |
+-----+-----+-----+-----+-----+-----+-----+
Success

```

```

In [ ]: # Checking that null is a noop
%sql call updatePerson(@ruthID, null, null, null, null, null, null, @ruthID)
res3 = %sql select * from people p where p.personID = @ruthID
res3_d = transform(res3.dict())

print(res3)

assert is_subset(expected_d, res3_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res3_d}"

print("Success")

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
+-----+-----+-----+-----+-----+-----+-----+
| personID | nameFirst | nameLast | birthYear | birthCountry | deathYear | deathCountry |
+-----+-----+-----+-----+-----+-----+-----+
| ruthba01 | Babe     | Ruth    | 1895     | USA         | 1948     | USA         |
+-----+-----+-----+-----+-----+-----+-----+
Success

```

```

In [ ]: # Try to manually set personID
        # Note: You should get an OperationalError. If you get an AssertionError, then
        # your trigger is not doing its job.

```

```

res4 = %sql update people set personID = "dff9" where personID = "ruthba01"

assert res4 is None, "Your trigger should throw an exception"

print("Success")

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1644, 'you cannot modify personID directly.')
[SQL: update people set personID = "dff9" where personID = "ruthba01"]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
Success

```

```

In [ ]: # Check that update trigger updates personID if name changes

```

```

%sql call updatePerson(@ruthID, "George", "Herman", 1920, "USA", 2005, "USA", @ruthID)
res5 = %sql select * from people p where p.personID = @ruthID
res5_d = transform(res5.dict())
expected_d = dict(
    personID="hermage01",
    nameFirst="George",
    nameLast="Herman",

```

```

        birthYear=1920,
        birthCountry="USA",
        deathYear=2005,
        deathCountry="USA"
    )

print(res5)

assert is_subset(expected_d, res5_d), \
f"Row has unexpected value. Expected {expected_d}, but got {res5_d}"

print("Success")

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.

```

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
hermage01	George	Herman	1920	USA	2005	USA

Success

```

In [ ]: %sql call updatePerson(@ruthID, "George", "Herman", 1920, "USA", 2005, "USA", @ruthID2)
%sql select @ruthID2, @ruthID;

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.

```

```

Out [ ]: @ruthID2    @ruthID
hermage01 hermage01

```

Security Testing

- Write and execute statements below to show that you set up the permissions for `general_user` correctly
 - You should show that `select` and `execute` work, but `insert`, `update`, and `delete` don't

```
In [ ]: # Connect to database as general_user
%%sql mysql+pymysql://general_user:dbuserdbuser@localhost/s24_hw3
```

```
In [ ]: %%sql
SELECT * FROM s24_hw3.People WHERE People.nameFirst = "Don";
```

```
* mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
11 rows affected.
```

```
Out[ ]:  personID  nameFirst  nameLast  birthYear  birthCountry  deathYear  deathCountry
         -----
johnsdo02      Don      Johnson      1911           USA           2000           USA
johnsdo03      Don      Johnson      1926           USA           2015           USA
leppedo01      Don      Leppert      1931           USA           None           None
leppedo02      Don      Leppert      1930           USA           2021           USA
mcmahdo02      Don      McMahon      1930           USA           1987           USA
nichodo02      Don      Nicholas      1930           USA           2007           USA
raderdo01      Don      Rader        1893           USA           1983           USA
wallado01      Don      Wallace      1940           USA           None           None
whitedo01      Don      White        1919           USA           1987           USA
willido01      Don      Williams     1931           USA           2011           USA
willido02      Don      Williams     1935           USA           1991           USA
```

```
In [ ]: %%sql
CALL s24_hw3.createPerson('John', 'Doe', 1990, 'USA', NULL, NULL, @newPersonID);

SELECT * FROM s24_hw3.People WHERE People.personID = @newPersonID;

* mysql+pymysql://general_user:***@localhost/s24_hw3
  mysql+pymysql://root:***@localhost
  mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
1 rows affected.
```

```
Out[ ]: personID  nameFirst  nameLast  birthYear  birthCountry  deathYear  deathCountry
-----
doejo01      John      Doe       1990       USA           None       None
```

```
In [ ]: %%sql
INSERT INTO s24_hw3.People (nameFirst, nameLast, birthYear, birthCountry) VALUES ('Jane', 'Smith', 1985, 'Canada');

* mysql+pymysql://general_user:***@localhost/s24_hw3
  mysql+pymysql://root:***@localhost
  mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1142, "INSERT command denied to user 'general_user'@'localhost' for table 'people'")
[SQL: INSERT INTO s24_hw3.People (nameFirst, nameLast, birthYear, birthCountry) VALUES ('Jane', 'Smith', 1985, 'Canada');]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

```
In [ ]: %%sql
UPDATE s24_hw3.People SET birthYear = 1980 WHERE nameLast = 'Doe';

* mysql+pymysql://general_user:***@localhost/s24_hw3
  mysql+pymysql://root:***@localhost
  mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1142, "UPDATE command denied to user 'general_user'@'localhost' for table 'people'")
[SQL: UPDATE s24_hw3.People SET birthYear = 1980 WHERE nameLast = 'Doe';]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

```
In [ ]: %%sql
DELETE FROM s24_hw3.People WHERE nameLast = 'Doe';
```

```
* mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1142, "DELETE command denied to user 'general_user'@'localhost' for table 'people'")
[SQL: DELETE FROM s24_hw3.People WHERE nameLast = 'Doe';]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

GoT Data Visualization

Data Loading

- Run the cell below to create and insert data into GoT-related tables

```
In [ ]: %sql mysql+pymysql://root:dbuserdbuser@localhost/s24_hw3

for filename in [
    "episodes_basics", "episodes_characters", "episodes_scenes"
]:
    df = pandas.read_json(f"data/{filename}.json")
    df.to_sql(name=filename, schema="s24_hw3", con=engine, index=False, if_exists="replace")

print("Success")
```

Success

Overview

- In this section, you'll be combining SQL and Dataframes to create data visualizations
 - You may find [this notebook](#) helpful
 - You may also find the [Pandas docs](#) helpful
- **For all questions, you need to show the SQL output and the visualization generated from it.** See DV0 for an example.

DV0

- This question is an example of what is required from you
- Create a bar graph showing the amount of time each season ran for (in seconds)
- You should use the `episodes_scenes` table
- Note: `season_running_time <<` in the following cell saves the output of the SQL query into a local Python variable `season_running_time`

```
In [ ]: %%sql

season_running_time <<

with one as (
    select seasonNum, episodeNum, sceneNum, sceneEnd, time_to_sec(sceneEnd) as sceneEndSeconds,
           sceneStart, time_to_sec(sceneStart) as sceneStartSeconds,
           time_to_sec(sceneEnd)-time_to_sec(sceneStart) as sceneLengthSeconds
    from episodes_scenes
),
two as (
    select seasonNum, episodeNum, max(sceneEnd) as episodeEnd, max(sceneEndSeconds) as episodeEndSeconds
    from one
    group by seasonNum, episodeNum
),
three as (
    select seasonNum, cast(sum(episodeEndSeconds) as unsigned) as totalSeasonSeconds,
           sec_to_time(sum(episodeEndSeconds)) as totalRunningTime
    from two
    group by seasonNum
)
select * from three;

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
8 rows affected.
Returning data to local variable season_running_time
```

```
In [ ]: # You must show the SQL output
```

```
season_running_time = season_running_time.DataFrame()  
season_running_time
```

```
Out[ ]: 

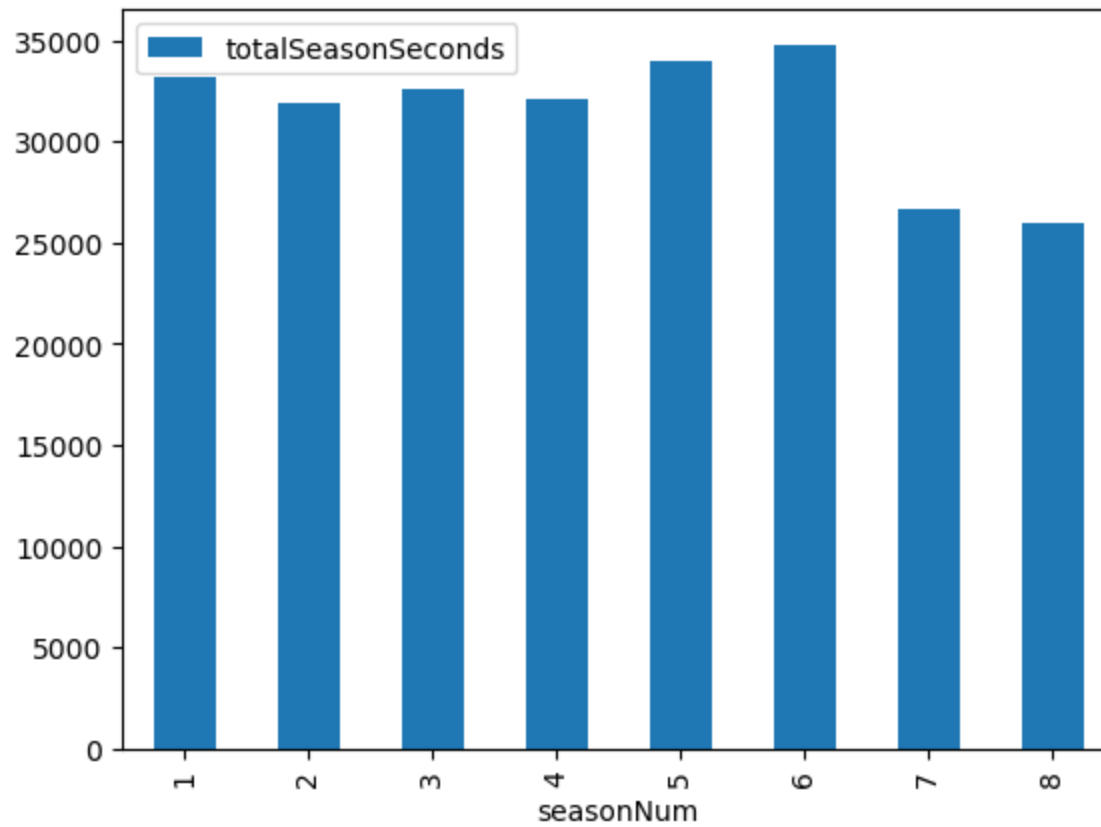
|   | seasonNum | totalSeasonSeconds | totalRunningTime |
|---|-----------|--------------------|------------------|
| 0 | 1         | 33143              | 0 days 09:12:23  |
| 1 | 2         | 31863              | 0 days 08:51:03  |
| 2 | 3         | 32541              | 0 days 09:02:21  |
| 3 | 4         | 32100              | 0 days 08:55:00  |
| 4 | 5         | 34003              | 0 days 09:26:43  |
| 5 | 6         | 34775              | 0 days 09:39:35  |
| 6 | 7         | 26675              | 0 days 07:24:35  |
| 7 | 8         | 25922              | 0 days 07:12:02  |


```

```
In [ ]: # You must show the visualization
```

```
season_running_time[["seasonNum", "totalSeasonSeconds"]].plot.bar(  
    x="seasonNum", y="totalSeasonSeconds"  
)
```

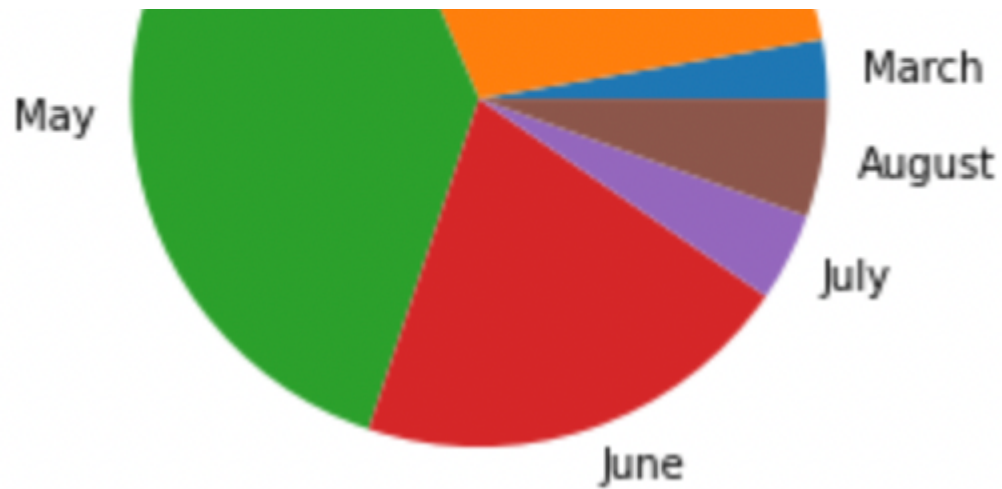
```
Out[ ]: <Axes: xlabel='seasonNum'>
```



DV1

- Create a pie chart showing the proportion of episodes aired in each month (regardless of year)
- You should use the `episodes_basics` table
- As an example, your pie chart may look like this:





In []: %%sql

```
episodes_per_month <<
SELECT COUNT(*) AS Episodes_Count,
       CASE SUBSTR(episodeAirDate, 6, 2)
         WHEN '01' THEN 'January'
         WHEN '02' THEN 'February'
         WHEN '03' THEN 'March'
         WHEN '04' THEN 'April'
         WHEN '05' THEN 'May'
         WHEN '06' THEN 'June'
         WHEN '07' THEN 'July'
         WHEN '08' THEN 'August'
         WHEN '09' THEN 'September'
         WHEN '10' THEN 'October'
         WHEN '11' THEN 'November'
         WHEN '12' THEN 'December'
         ELSE 'Unknown'
       END AS Month_Name
FROM episodes_basics
GROUP BY Month_Name
```

```
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
6 rows affected.
Returning data to local variable episodes_per_month
```

```
In [ ]: # SQL output
```

```
episodes_per_month = episodes_per_month.DataFrame()
episodes_per_month
```

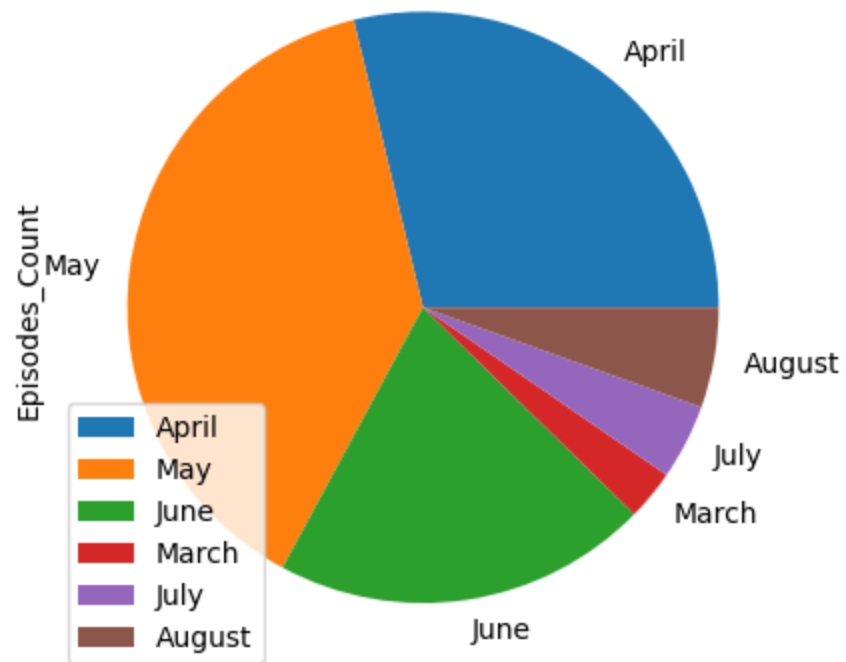
```
Out[ ]:
```

	Episodes_Count	Month_Name
0	21	April
1	28	May
2	15	June
3	2	March
4	3	July
5	4	August

```
In [ ]: # TODO: visualization
```

```
episodes_per_month.plot(
    kind="pie", y="Episodes_Count", labels=episodes_per_month["Month_Name"]
)
```

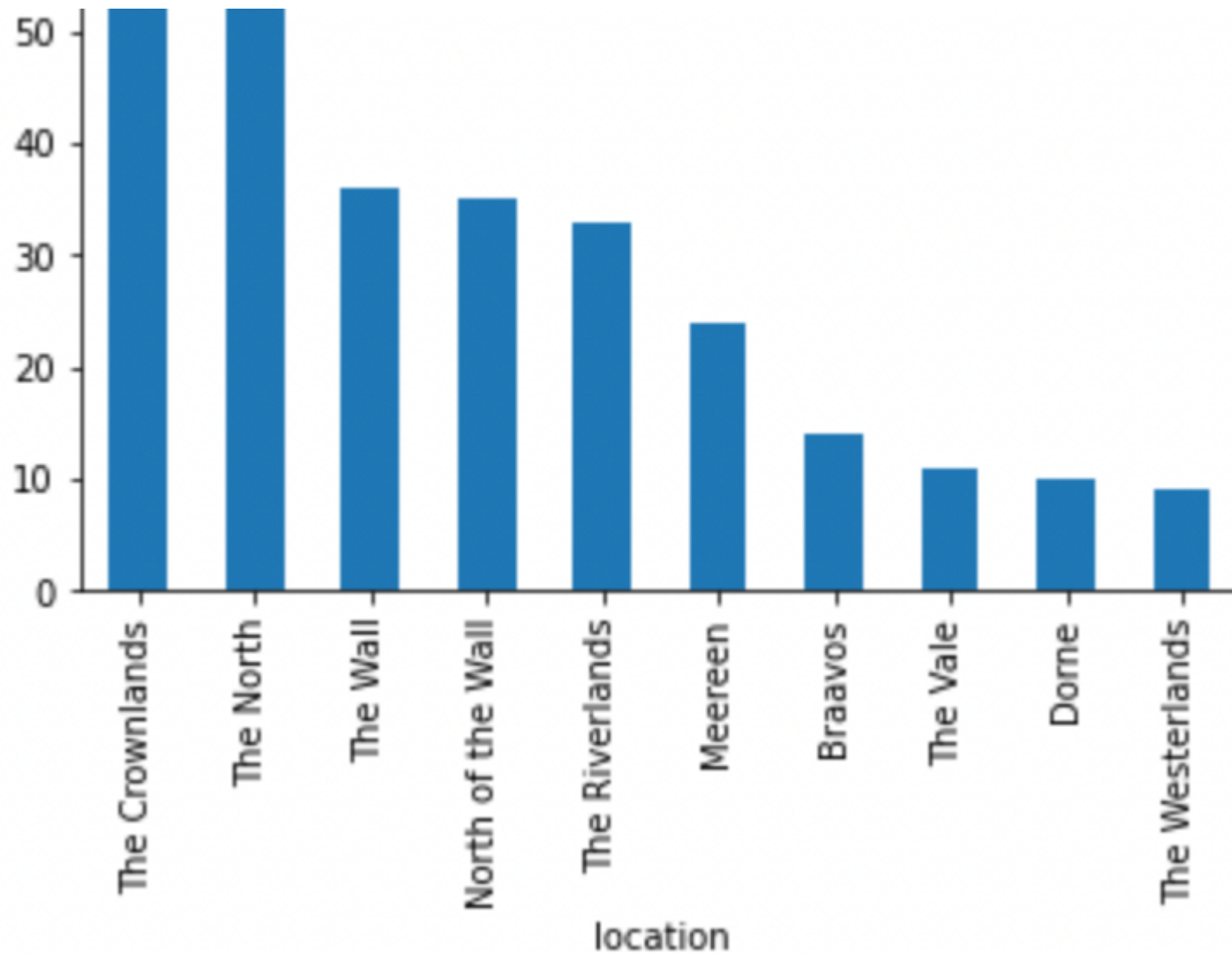
```
Out[ ]: <Axes: ylabel='Episodes_Count'>
```



DV2

- Create a bar chart showing the number of episodes that every location (not sublocation) appeared in
 - You are counting the number of episodes, not scenes. If a location appeared in multiple scenes in a single episode, that should increment your count only by one.
 - You should order your chart on the number of episodes descending, and you should only show the top 10 locations
- You should use the `episodes_scenes` table
- As an example, your bar chart may look like this:





In []: `%%sql`

```
location_episode_count <<
SELECT COUNT(DISTINCT CONCAT(seasonNum, ",", episodeNum)) as episodeCount,
       sceneLocation
FROM episodes_scenes
GROUP BY sceneLocation
ORDER BY episodeCount desc;
```

```
mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
26 rows affected.
Returning data to local variable location_episode_count
```

In []: *# SQL output*

```
location_episode_count = location_episode_count.DataFrame()
location_episode_count
```

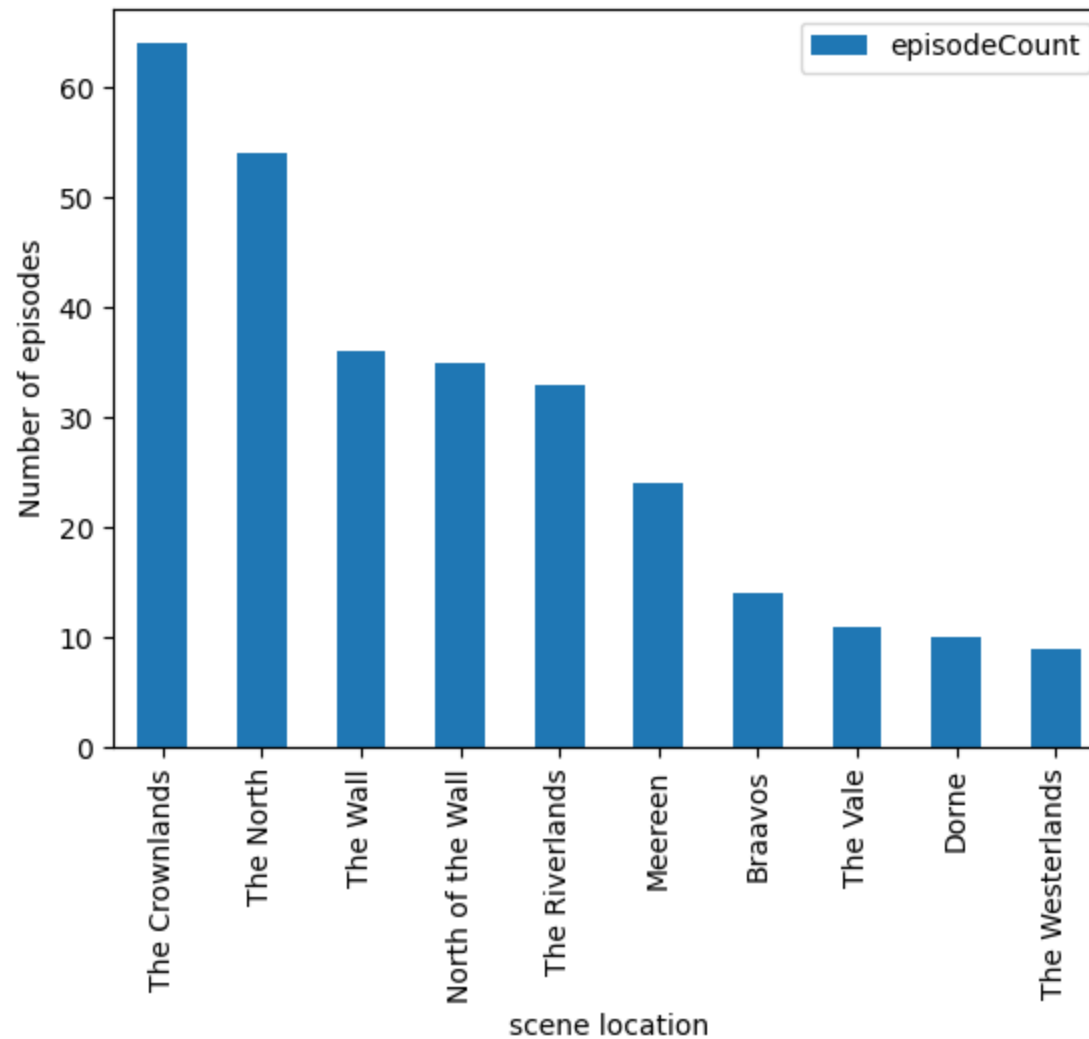

Out of 1:

	episodeCount	sceneLocation
0	64	The Crownlands
1	54	The North
2	36	The Wall
3	35	North of the Wall
4	33	The Riverlands
5	24	Meereen
6	14	Braavos
7	11	The Vale
8	10	Dorne
9	9	The Westerlands
10	8	The Reach
11	7	The Iron Islands
12	6	Qarth
13	6	The Dothraki Sea
14	6	Vaes Dothrak
15	5	The Red Waste
16	5	Yunkai
17	4	The Narrow Sea
18	4	The Stormlands
19	3	Astapor
20	3	Pentos
21	3	Volantis
22	2	The Summer Sea

	episodeCount	sceneLocation
23	2	The Sunset Sea
24	2	Valyria
25	1	The Shivering Sea

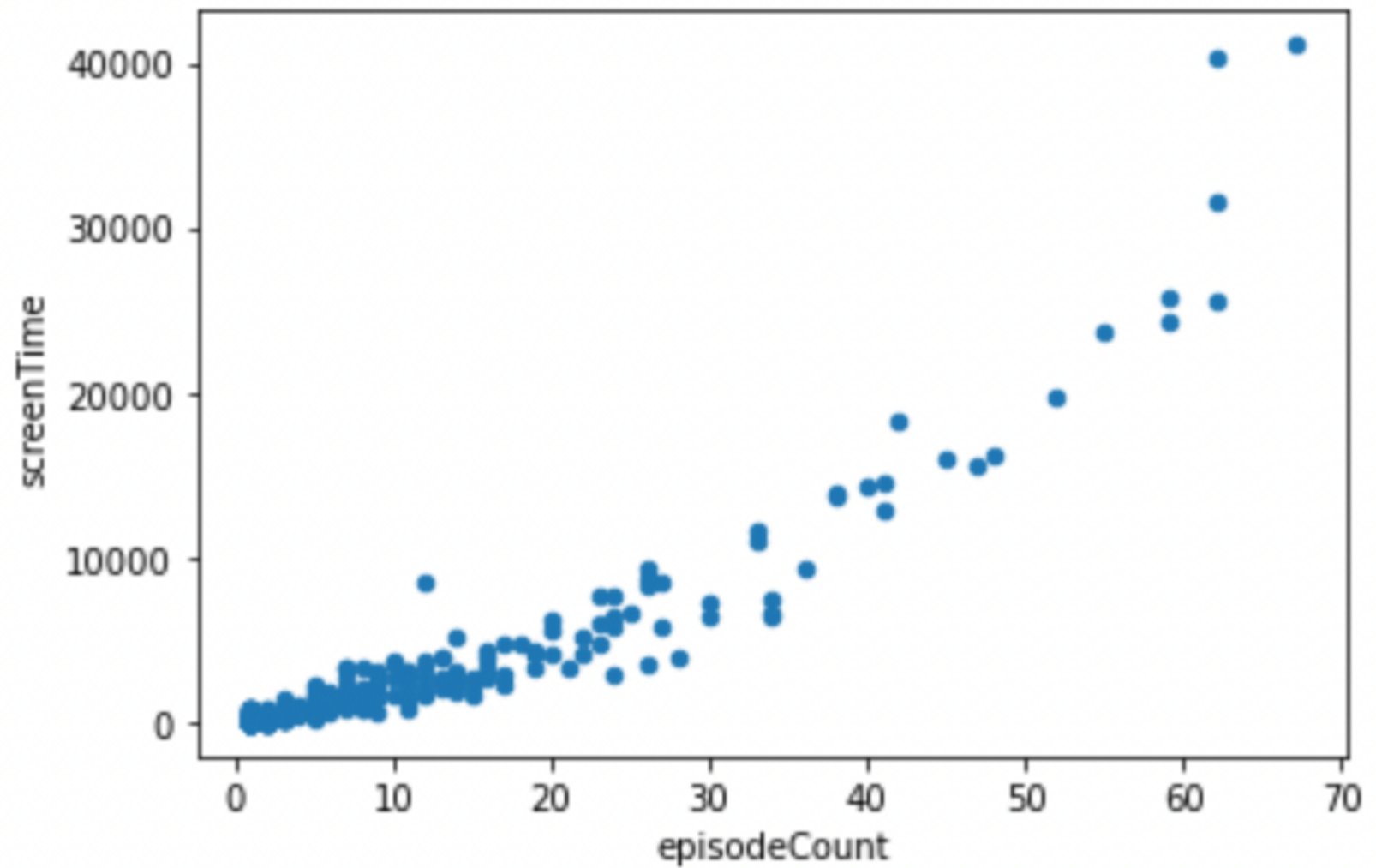
```
In [ ]: # TODO: visualization
ax = location_episode_count.head(10).plot(
    kind="bar", xlabel="scene location", ylabel="Number of episodes"
)
ax.set_xticklabels(
    location_episode_count["sceneLocation"].head(10), rotation=90, ha="center"
)
```

```
Out[ ]: [Text(0, 0, 'The Crownlands'),
Text(1, 0, 'The North'),
Text(2, 0, 'The Wall'),
Text(3, 0, 'North of the Wall'),
Text(4, 0, 'The Riverlands'),
Text(5, 0, 'Meereen'),
Text(6, 0, 'Braavos'),
Text(7, 0, 'The Vale'),
Text(8, 0, 'Dorne'),
Text(9, 0, 'The Westerlands')]
```



DV3

- Create a scatter plot showing the relationship between the number of episodes (not scenes) a character appears in and their screen time (in seconds)
 - A character's screen time is the sum of the time lengths of all the scenes that the character appears in
- You should use the `episodes_characters` and `episodes_scenes` tables
- As an example, your scatter plot may look like this:



```
In [ ]: %%sql
```

```
episode_count_screen_time <<
```

```
SELECT
```

```
    c.characterName,
```

```
    COUNT(DISTINCT CONCAT(s.episodeNum, ',', s.seasonNum)) AS numEpisodes,
```

```
    SUM(TIME_TO_SEC(TIMEDIFF(s.sceneEnd, s.sceneStart))) AS screenTime
```

```
FROM
```

```

    episodes_characters AS c
INNER JOIN
    episodes_scenes AS s ON c.sceneNum = s.sceneNum
                        AND c.episodeNum = s.episodeNum
                        AND c.seasonNum = s.seasonNum

GROUP BY
    c.characterName
order by screenTime desc;

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
577 rows affected.
Returning data to local variable episode_count_screen_time
577 rows affected.
Returning data to local variable episode_count_screen_time

```

```

In [ ]: # SQL output
        # Output is big, so just show first 10 rows

        episode_count_screen_time = episode_count_screen_time.DataFrame()
        episode_count_screen_time.head(10)

```

```

Out[ ]:

```

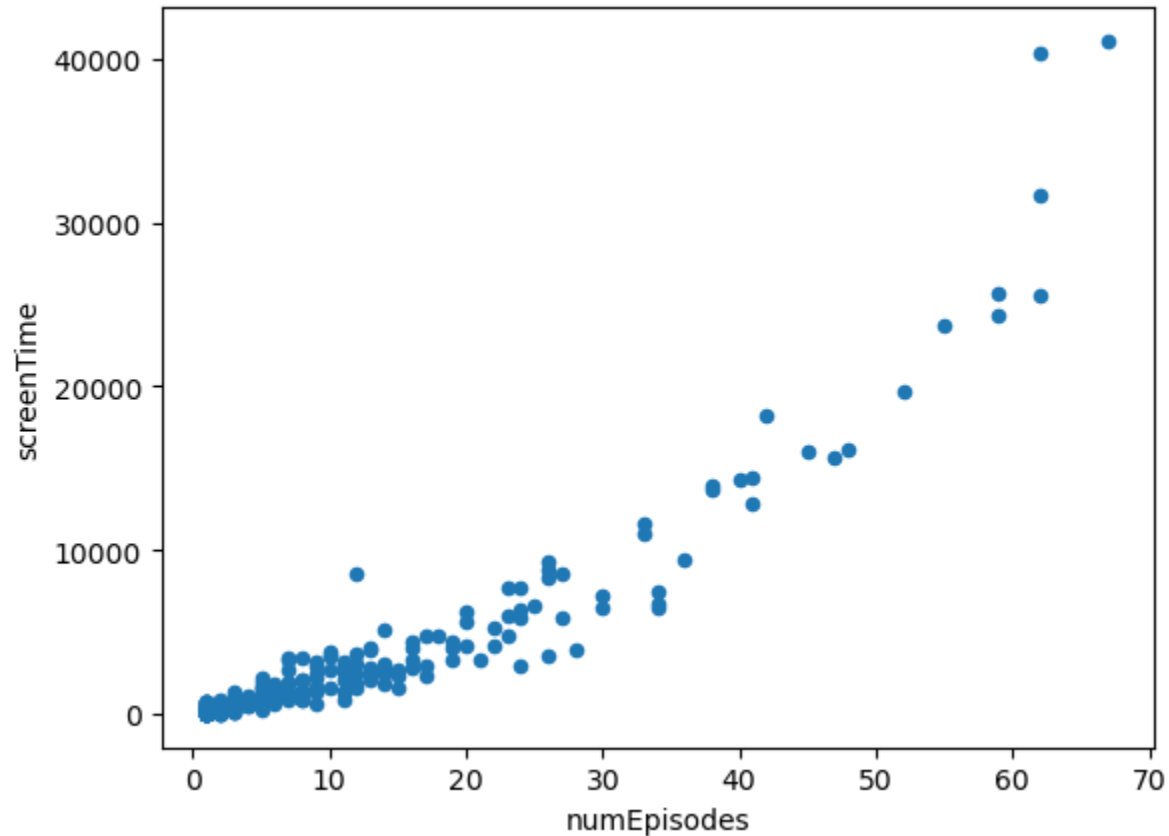
	characterName	numEpisodes	screenTime
0	Tyrion Lannister	67	41104
1	Jon Snow	62	40365
2	Daenerys Targaryen	62	31694
3	Sansa Stark	59	25705
4	Cersei Lannister	62	25522
5	Arya Stark	59	24315
6	Jaime Lannister	55	23675
7	Jorah Mormont	52	19653
8	Davos Seaworth	42	18185
9	Samwell Tarly	48	16118

```
In [1]: # TODO: visualization
```

```
ax = episode_count_screen_time.plot.scatter(x="numEpisodes", y="screenTime")
```

```
ax
```

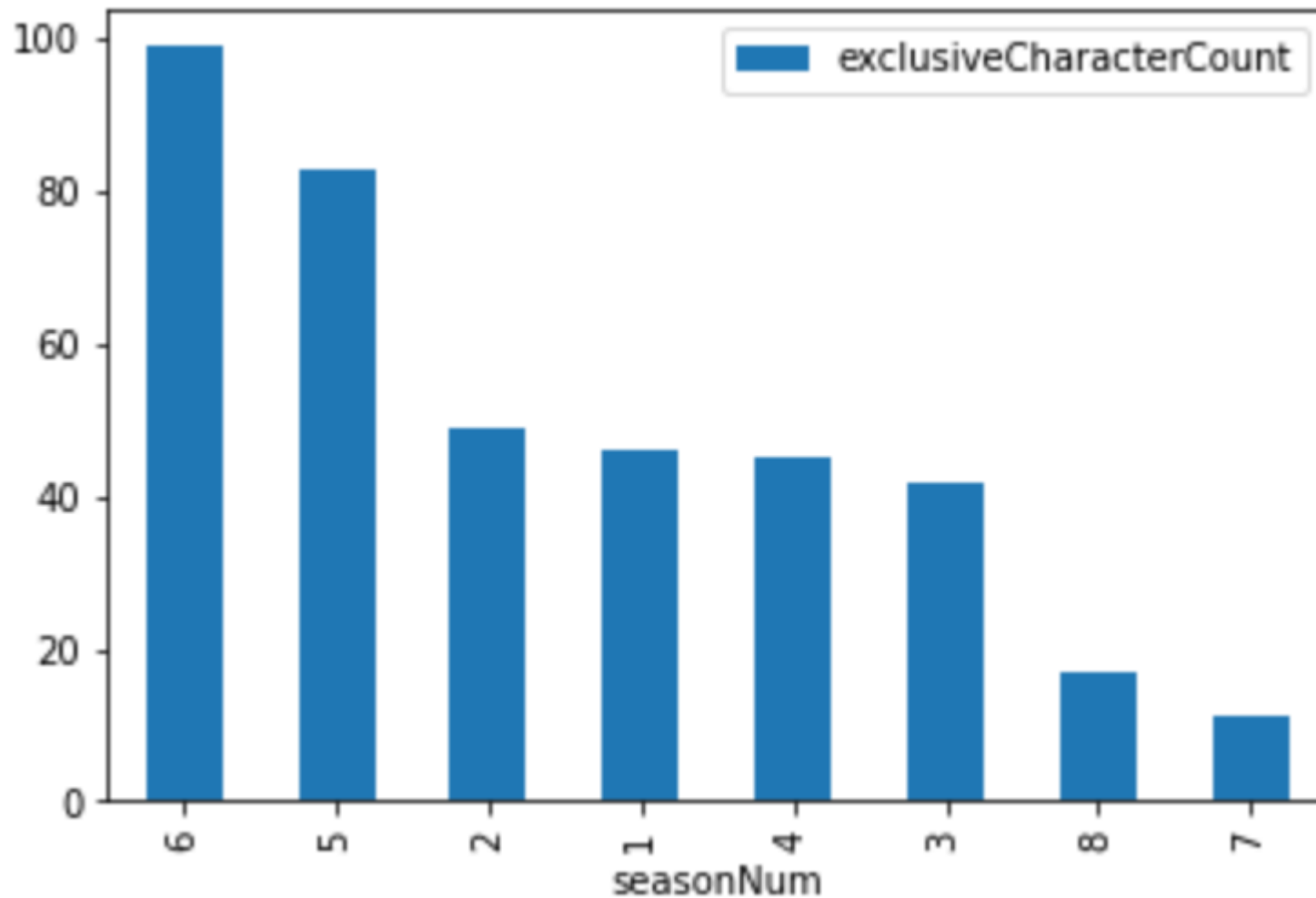
```
Out[1]: <Axes: xlabel='numEpisodes', ylabel='screenTime'>
```



DV4

- Create a bar chart showing the number of exclusive characters in each season
 - An exclusive character is a character that appeared in only that season, no other season
 - You should order your chart on the number of exclusive characters descending

- You should use the `episodes_characters` table
 - You can assume `characterName` is unique across all characters. That is, a single name is one unique character.
- As an example, your bar chart may look like this:



```
In [ ]: %%sql
season_exclusive_characters <<
SELECT COUNT(characterName) AS exclusiveCharacterCount, seasonNum
```

```

FROM (
    SELECT characterName, MAX(seasonNum) AS seasonNum, COUNT(DISTINCT seasonNum) AS num_seasons
    FROM episodes_characters
    GROUP BY characterName
    HAVING num_seasons = 1
) AS cNsNns
GROUP BY seasonNum
ORDER BY exclusiveCharacterCount desc;

```

```

mysql+pymysql://general_user:***@localhost/s24_hw3
mysql+pymysql://root:***@localhost
* mysql+pymysql://root:***@localhost/s24_hw3
8 rows affected.
Returning data to local variable season_exclusive_characters

```

```

In [ ]: # SQL output

season_exclusive_characters = season_exclusive_characters.DataFrame()
season_exclusive_characters

```

```

Out[ ]:

```

	exclusiveCharacterCount	seasonNum
0	99	6
1	83	5
2	49	2
3	46	1
4	45	4
5	42	3
6	17	8
7	11	7

```

In [ ]: # TODO: visualization
ax = season_exclusive_characters["exclusiveCharacterCount"].plot(
    kind="bar", xlabel="seasonNum"
)

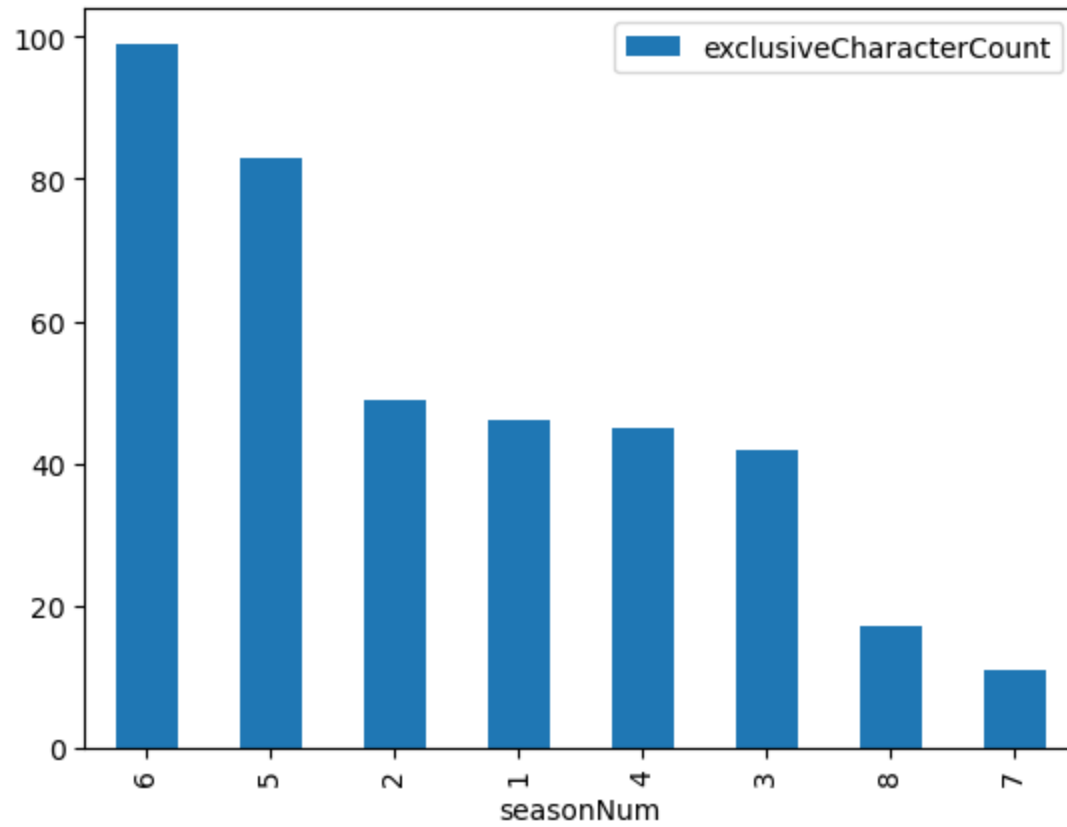
ax.set_xticklabels(season_exclusive_characters["seasonNum"], rotation=90, ha="center")

```



```
ax.legend()
```

Out[]: <matplotlib.legend.Legend at 0x7fb1baaa4220>



In []: