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

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

Out[1]:

1
1

In [2]:
```
%%sql

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

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

Out[2]:
```
[]
```

In [3]:
```
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[3]:
```
[{'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

1. RAM, also called random access memory, is volatile storage, meaning it loses contents when power is switched off. Data retrieval speed is faster than that of SSDs and HDDs but its capacity is smaller than that of SSDs and HHDs. It is more expensive per unit storage compared to SSDs and HDDs.
2. Solid state drives are non-volatile storage devices that use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally. Its data retrieval speed is faster than HDDs but slower than RAM and has smaller capacity than HDDs but larger capacity than RAM. More expensive per unit storage than HDDs but cheaper than RAM
3. Hard drives are non-volatile storage devices that use spinning magnetic disks and heads to store and retrieve data. Its data retrieval speed is the slowest among all 3 but has the largest capacity usually. It has the cheapest per unit storage

W2

With regards to disk drives, define

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

1. Seek time is the time needed to move head to cylinder/track
2. Rotational latency time is the time needed to wait for sector to get under head
3. Transfer time/data transfer rate is the time needed to move data from disk to memory.

W3

Explain the concepts of

- Logical block addressing
- Cylinder-head-sector addressing

1. Logical block addressing is a method of disk addressing that assigns a unique identifier to a block with each address being (DeviceID,BlockID) with BlockID starting from 0. The disk controller and disk implementation translate the logical block address into the physical address of blocks.

2. Cylinder-head-sector addressing is a method of disk addressing that locates the data with the physical geometry of the disk. The address is represented by cylinder number, head number, and sector number.

W4

Define and list some benefits of

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

1. Fixed-length records are record where each of its attribute has a predetermined size. Accessing fixed-length records is often faster than accessing variable-length records and it is easier to predict required storage space.
2. Variable-length records are record where each of its attribute has varying size. It allows more flexible data storage since it stores data with varying sizes. It is also space efficient as it only uses space as needed.
3. Row-oriented storage stores all attributes of a single record adjacent to each other in terms of memory location. It is good for querying multiple columns of a single row which suits online transactional processing. It also aligns with the natural structure of relational databases which is easier for developers to understand. (https://www.sentinelone.com/blog/understanding-row-vs-column-oriented-databases/#:~:text=Row%2DOriented%20Databases&text=All%20columns%20in%20a%20single,transactional%20processing%20(OLTP)%2 (https://www.sentinelone.com/blog/understanding-row-vs-column-oriented-databases/#:~:text=Row%2DOriented%20Databases&text=All%20columns%20in%20a%20single,transactional%20processing%20(OLTP)%2
4. Column-oriented storage stores all values from the same attribute adjacent to each other in terms of memory location. It is best for analytical processing applications which requries frequent column-wise data access. It is better for compression too as compression algorithms run more efficiently on similar data. (https://www.sentinelone.com/blog/understanding-row-vs-column-oriented-databases/#:~:text=Row%2DOriented%20Databases&text=All%20columns%20in%20a%20single,transactional%20processing%20(OLTP)%2 (https://www.sentinelone.com/blog/understanding-row-vs-column-oriented-databases/#:~:text=Row%2DOriented%20Databases&text=All%20columns%20in%20a%20single,transactional%20processing%20(OLTP)%2

W5

Explain and list some differences between

- RAID 0
- RAID 1

- RAID 5

1. RAID 0 consists of striping, without mirroring or parity. RAID 0 is used to create fast disk as it distributes data evenly across multiple disks.
2. RAID 1 consists of data mirroring, without parity or striping. RAID 1 is used to create reliable disk as if one drive fails, data remains accessible from the mirrored drive.
3. RAID 5 consists of block-level striping with distributed parity. RAID 5 also creates reliable disk but because of parity information where if one drive fails, data can be reconstructed using parity information and data from the remaining drives.

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 (https://dev.mysql.com/doc/refman/8.0/en/year.html) is too small.
- Your table will be empty for the next few sections. We will insert data later.

In [4]:
```
%%sql
CREATE TABLE People (
    personID CHAR(9),
    nameFirst VARCHAR(20) not null,
    nameLast VARCHAR(20) not null,
    birthYear INT,
    birthCountry VARCHAR(20),
    deathYear INT,
    deathCountry VARCHAR(20),
    PRIMARY KEY (personID)
)
```

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

Out[4]:
```
[]
```

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)

nameFirst	nameLast	personID
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

In [5]:
```
%%sql
CREATE FUNCTION RemoveSpecialChar(s VARCHAR(20))
    RETURNS VARCHAR(20)
    READS SQL DATA
    BEGIN
    DECLARE result VARCHAR(20);
        SET result = REPLACE(REPLACE(REPLACE(REPLACE(s, '.', ''), '-', ''), ' ', ''),"'",'');
    RETURN result;
END;

CREATE FUNCTION GeneratePersonID(nameFirst VARCHAR(20), nameLast VARCHAR(20))
    RETURNS CHAR(9)
    READS SQL DATA
BEGIN
    DECLARE lastSubstr VARCHAR(5);
    DECLARE firstSubstr VARCHAR(2);
    DECLARE number INT;
    DECLARE paddedNumber VARCHAR(2);
    DECLARE finalID CHAR(9);

    -- Remove special char
    SET nameFirst = RemoveSpecialChar(nameFirst);
    SET nameLast = RemoveSpecialChar(nameLast);

    -- Get lastSubstr and firstSubstr
    SET lastSubstr = LEFT(LOWER(nameLast), 5);
    SET firstSubstr = LEFT(LOWER(nameFirst), 2);

    -- Get the next available number for the combination of lastSubstr and firstSubstr
    SELECT COALESCE(MAX(CAST(SUBSTRING(personID, LENGTH(personID) - 1,2) AS UNSIGNED)) + 1, 1) INTO number
    FROM People
    WHERE SUBSTRING(personID, 1, LENGTH(personID) - 2) = CONCAT(lastSubstr, firstSubstr);

    -- Pad number
    SET paddedNumber = LPAD(number, 2, '0');

    -- Get final personID
    SET finalID = CONCAT(lastSubstr, firstSubstr, paddedNumber);

    RETURN finalID;
END;
```

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

Out[5]: []

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 [6]:
```
%%sql
CREATE TRIGGER generate_personID_before_insert
BEFORE INSERT ON People
FOR EACH ROW
BEGIN
    SET NEW.personID = GeneratePersonID(NEW.nameFirst, NEW.nameLast);
END;
```

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

Out[6]: []

In [7]:
```
%%sql
CREATE TRIGGER immutable_personID
BEFORE UPDATE ON People
FOR EACH ROW
BEGIN
    DECLARE new_personID CHAR(9);

    -- Check if personID is modified
    IF OLD.personID != NEW.personID THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot modify personID directly';
    END IF;

    -- Check if nameFirst or nameLast is modified
    IF LEFT(LOWER(OLD.nameLast), 5) != LEFT(LOWER(NEW.nameLast), 5) OR LEFT(LOWER(OLD.nameFirst), 2) != LEFT(
        -- Get new personID
        SET NEW.personID = GeneratePersonID(NEW.nameFirst, NEW.nameLast);
    END IF;
END;
```

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

Out[7]:
```
[]
```

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 paramaters
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 [8]:
```
%%sql
CREATE PROCEDURE createPerson(IN nameFirst VARCHAR(20),
                              IN nameLast VARCHAR(20),
                              IN birthYear INT,
                              IN birthCountry VARCHAR(20),
                              IN deathYear INT,
                              IN deathCountry VARCHAR(20),
                              OUT personID CHAR(9)
                             )
BEGIN
    -- Get personID
    SET personID = GeneratePersonID(nameFirst, nameLast);

    -- Insert new person record
    INSERT INTO People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
    VALUES (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry);
END;
```

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

Out[8]:
```
[]
```

In [9]:
```
%%sql
CREATE PROCEDURE updatePerson(IN oldpersonID CHAR(9),
                              IN newNameFirst VARCHAR(20),
                              IN newNameLast VARCHAR(20),
                              IN newBirthYear INT,
                              IN newBirthCountry VARCHAR(20),
                              IN newDeathYear INT,
                              IN newDeathCountry VARCHAR(20),
                              OUT newPersonID CHAR(9)
                             )
BEGIN
    DECLARE oldNameFirst VARCHAR(20);
    DECLARE oldNameLast VARCHAR(20);
    SELECT Namefirst,NameLast INTO oldNameFirst,oldNameLast FROM People WHERE personID = oldpersonID;

    IF LEFT(LOWER(oldNameLast), 5) != LEFT(LOWER(newNameLast), 5) OR LEFT(LOWER(oldNameFirst), 2) != LEFT(LOW
        -- Get new personID from new nameFirst and nameLast
        SET newPersonID = GeneratePersonID(newNameFirst, newNameLast);
    ELSE
        -- no change
        SET newPersonID = oldpersonID;
    END IF;

    -- Update the person record if not null
    UPDATE People
    SET
        nameFirst = IF(newNameFirst IS NOT NULL, newNameFirst, nameFirst),
        nameLast = IF(newNameLast IS NOT NULL, newNameLast, nameLast),
        birthYear = IF(newBirthYear IS NOT NULL, newBirthYear, birthYear),
        birthCountry = IF(newBirthCountry IS NOT NULL, newBirthCountry, birthCountry),
        deathYear = IF(newDeathYear IS NOT NULL, newDeathYear, deathYear),
        deathCountry = IF(newDeathCountry IS NOT NULL, newDeathCountry, deathCountry)
    WHERE personID = oldpersonID;
END;
```

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

Out[9]:
```
[]
```

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 [10]:
```
%%sql
CREATE USER 'general_user'@'localhost' identified by 'dbuserdbuser'
```

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

Out[10]:
```
[]
```

In [11]:
```
%%sql
GRANT SELECT ON * TO 'general_user'@'localhost';
```

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

Out[11]:
```
[]
```

In [12]:
```
%%sql
GRANT execute ON procedure createPerson TO 'general_user'@'localhost';
GRANT execute ON procedure updatePerson TO 'general_user'@'localhost';
```

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

Out[12]:
```
[]
```

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

We can first add an attribute to the `people` table to indicate if a person is player or manager. Then create a view for player and manager each with a where clause and include all attributes.

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 [13]:
```
# 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[13]:

	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 [14]:
```python
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

    # 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()
    #print("result:",result)
    sql_conn.commit()
    cur.close()
    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 [15]:
```
%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()]
#print("expected_ids:", expected_ids)

for i, (p, e_id) in enumerate(zip(people_df.to_dict(orient="records"), expected_ids)):
    #print(p)
    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://root:***@localhost
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 reseting your data.

In [16]:
```
# 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 [17]:
```
# 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://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   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 [18]:
```
# Update birth country and year
%sql call updatePerson(@ruthID, null, null, 1895, "USA", 1948, "USA", @ruthID)
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://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   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 [19]:
```
# 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://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   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 [20]:
```
# 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://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
(pymysql.err.OperationalError) (1644, 'Cannot modify personID directly')
[SQL: update people set personID = "dff9" where personID = "ruthba01"]
(Background on this error at: https://sqlalche.me/e/20/e3q8) (https://sqlalche.me/e/20/e3q8))
Success
```

In [21]:
```
# 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://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
   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
```

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 [25]:
```
# Connect to database as general_user
%sql mysql+pymysql://general_user:dbuserdbuser@localhost/s24_hw3
```

In [26]:
```
%sql select * from People limit 10
```

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

Out[26]:

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
abernte01	Ted	Abernathy	1921	USA	2001	USA
abernte02	Ted	Abernathy	1933	USA	2004	USA
abreujo01	Jose	Abreu	1987	Cuba	None	None
abreujo02	Joe	Abreu	1913	USA	1993	USA
adamsau01	Austin	Adams	1986	USA	None	None
adamsau02	Austin	Adams	1991	USA	None	None
adamsbo01	Bob	Adams	1952	USA	None	None
adamsbo02	Bob	Adams	1907	USA	1970	USA
adamsbo03	Bob	Adams	1901	USA	1996	USA
adamsbo04	Bobby	Adams	1921	USA	1997	USA

In [27]:
```
%sql CALL createPerson('test', 'test', 1900, 'USA', 2000, 'USA', @test);
```

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

Out[27]:
```
[]
```

In [28]:
```
%%sql
INSERT INTO People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
    values ('test', 'test', 1900, 'USA', 2000, 'USA');
```

```
 * 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 People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry)
    values ('test', 'test', 1900, 'USA', 2000, 'USA');]
(Background on this error at: https://sqlalche.me/e/20/e3q8) (https://sqlalche.me/e/20/e3q8))
```

In [29]:
```
%%sql
UPDATE People
SET
    nameFirst = 'test',
    nameSecond = 'test';
```

```
 * 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 People
SET
    nameFirst = 'test',
    nameSecond = 'test';]
(Background on this error at: https://sqlalche.me/e/20/e3q8) (https://sqlalche.me/e/20/e3q8))
```

In [30]:
```
%%sql
DELETE FROM People
WHERE nameFirst = 'test'
```

```
 * 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 People
WHERE nameFirst = 'test']
(Background on this error at: https://sqlalche.me/e/20/e3q8) (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 [31]:
```
%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 (https://github.com/donald-f-ferguson/W4111-Intro-to-Databases-Spring-2024/blob/main/examples/process_got/GoT_Processing.ipynb) helpful
  - You may also find the Pandas docs (https://pandas.pydata.org/docs/reference/frame.html) 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 [32]:
```
%%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;
```

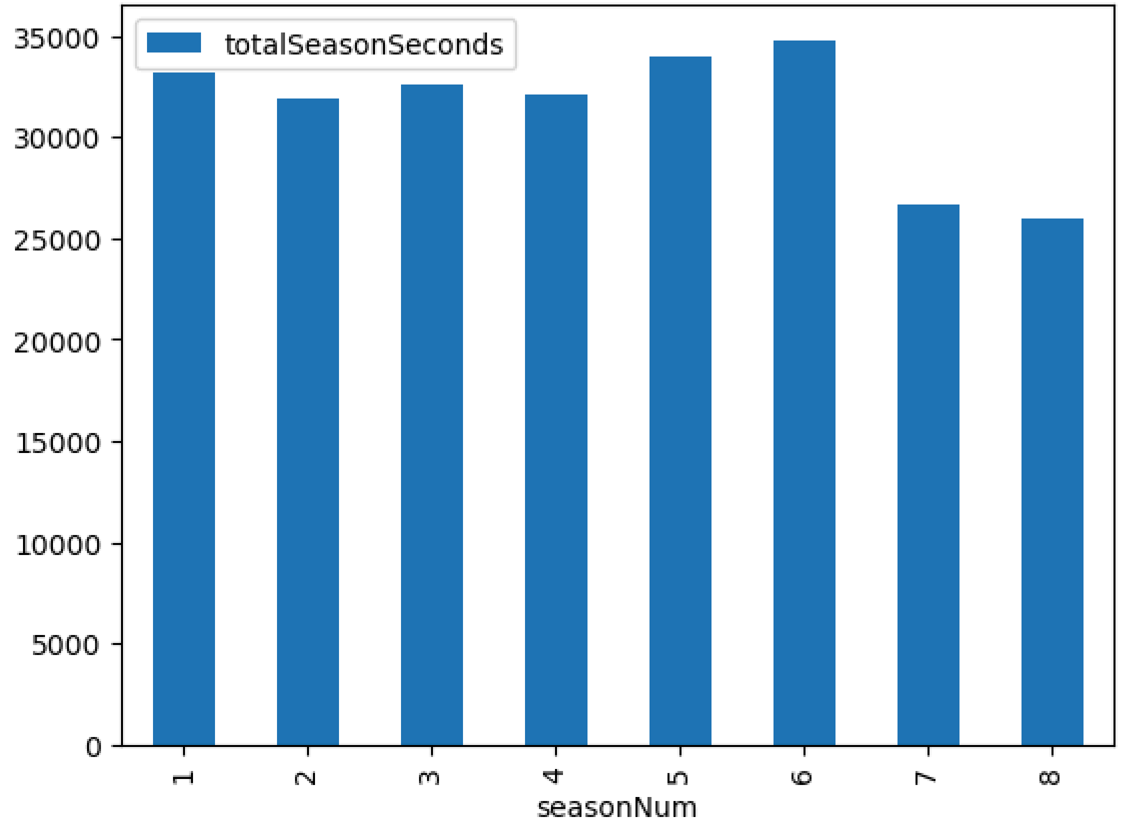
```
   mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
 * mysql+pymysql://root:***@localhost/s24_hw3
8 rows affected.
Returning data to local variable season_running_time
```

In [33]:
```
# You must show the SQL output

season_running_time = season_running_time.DataFrame()
season_running_time
```

Out[33]:

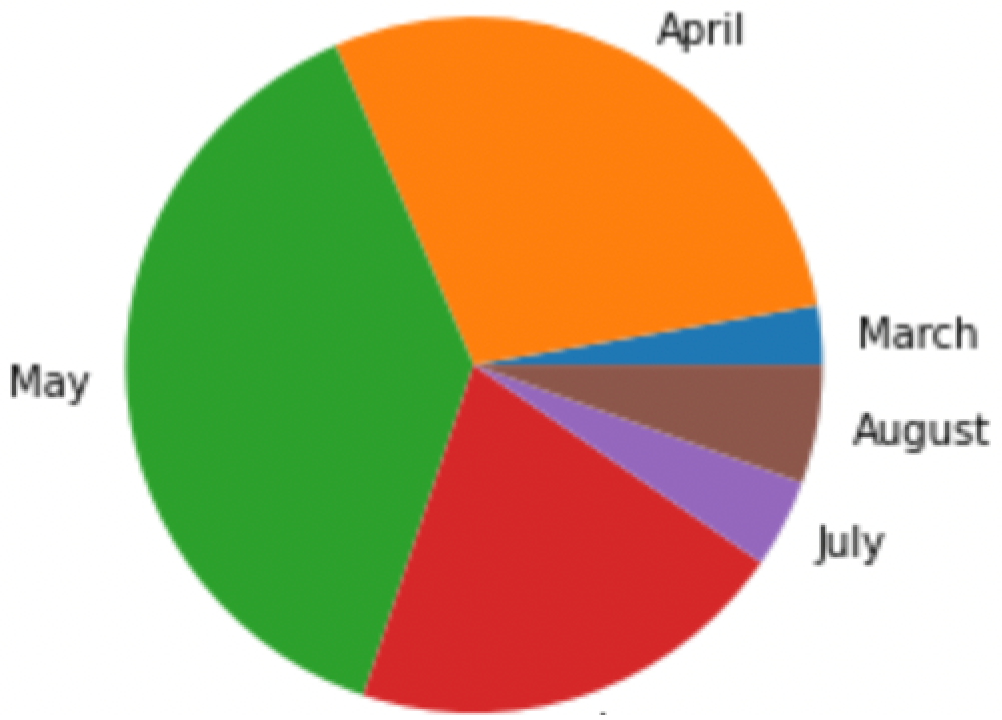
	seasonNum	totalSeasonSeconds	totalRunningTime
0	1	33143	0 days 09:12:23
1	2	31863	0 days 08:51:03
2	3	32541	0 days 09:02:21
3	4	32100	0 days 08:55:00
4	5	34003	0 days 09:26:43
5	6	34775	0 days 09:39:35
6	7	26675	0 days 07:24:35
7	8	25922	0 days 07:12:02

```
In [34]: # You must show the visualization

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

Out[34]: <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 [35]:
```
%%sql

episodes_per_month <<

SELECT
    DATE_FORMAT(episodeAirDate, '%M') AS Month,
    count(date_format(episodeAirDate, '%M')) AS "Episode Count"
FROM
    episodes_basics
GROUP BY
    Month
LIMIT 10
```

```
   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
```

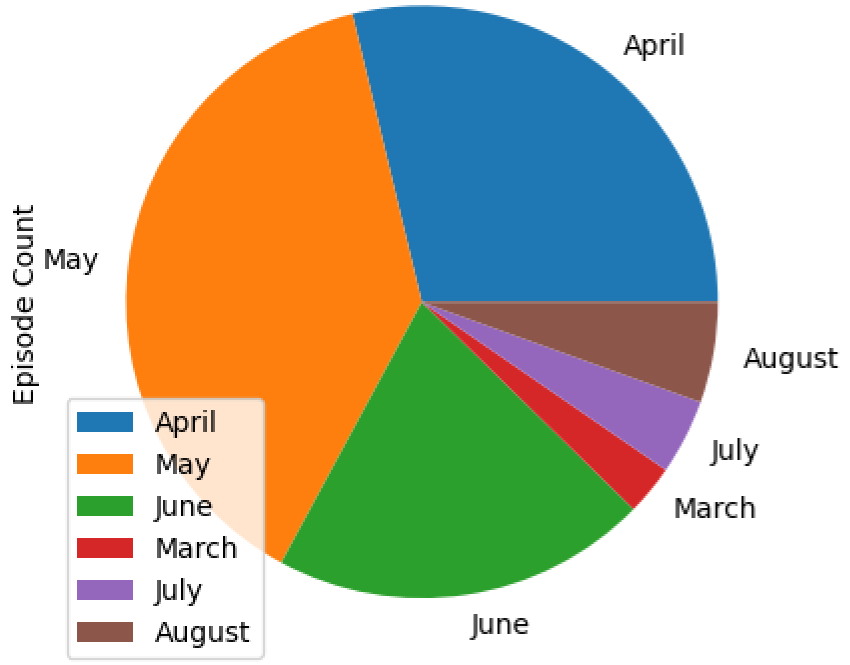
In [36]:
```
# SQL output

episodes_per_month = episodes_per_month.DataFrame()
episodes_per_month
```

Out[36]:

	Month	Episode Count
0	April	21
1	May	28
2	June	15
3	March	2
4	July	3
5	August	4

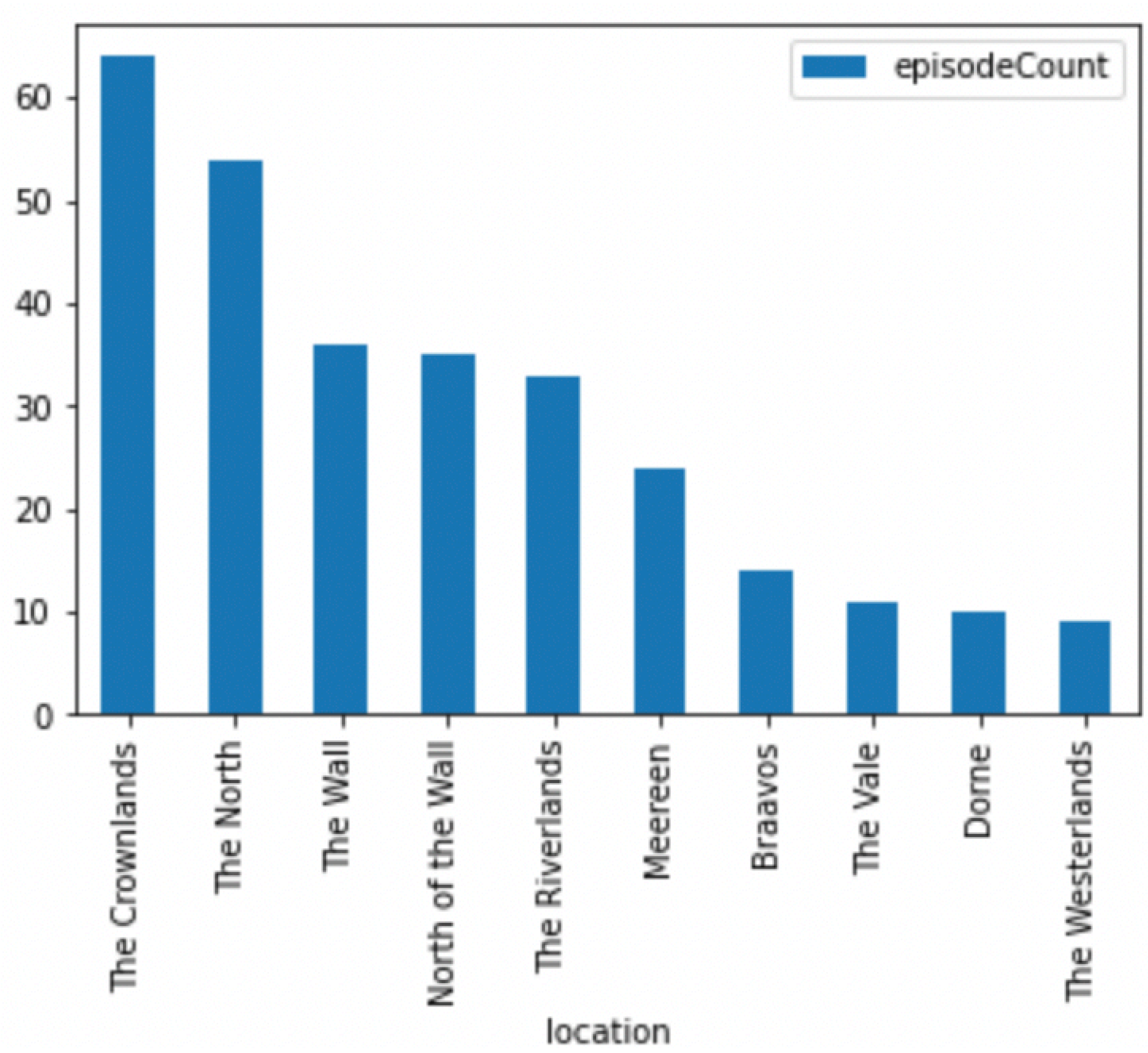
In [37]:
```
# TODO: visualization
episodes_per_month.set_index("Month", inplace=True)
episodes_per_month.plot.pie(x="Month", y="Episode Count")
```

Out[37]:
```
<Axes: ylabel='Episode 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:


episodeCount
60
50
40
30
20
10
0
The Crownlands
The North
The Wall
North of the Wall
The Riverlands
Meereen
Braavos
The Vale
Dorne
The Westerlands
location

In [47]:
```
%%sql

location_episode_count <<
    SELECT
        sceneLocation,
        count(*) AS episodeCount
    FROM
    (SELECT DISTINCT sceneLocation, seasonNum, episodeNum FROM episodes_scenes) tmp
    GROUP BY
        sceneLocation
    ORDER BY
        episodeCount desc
    LIMIT 10
```

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

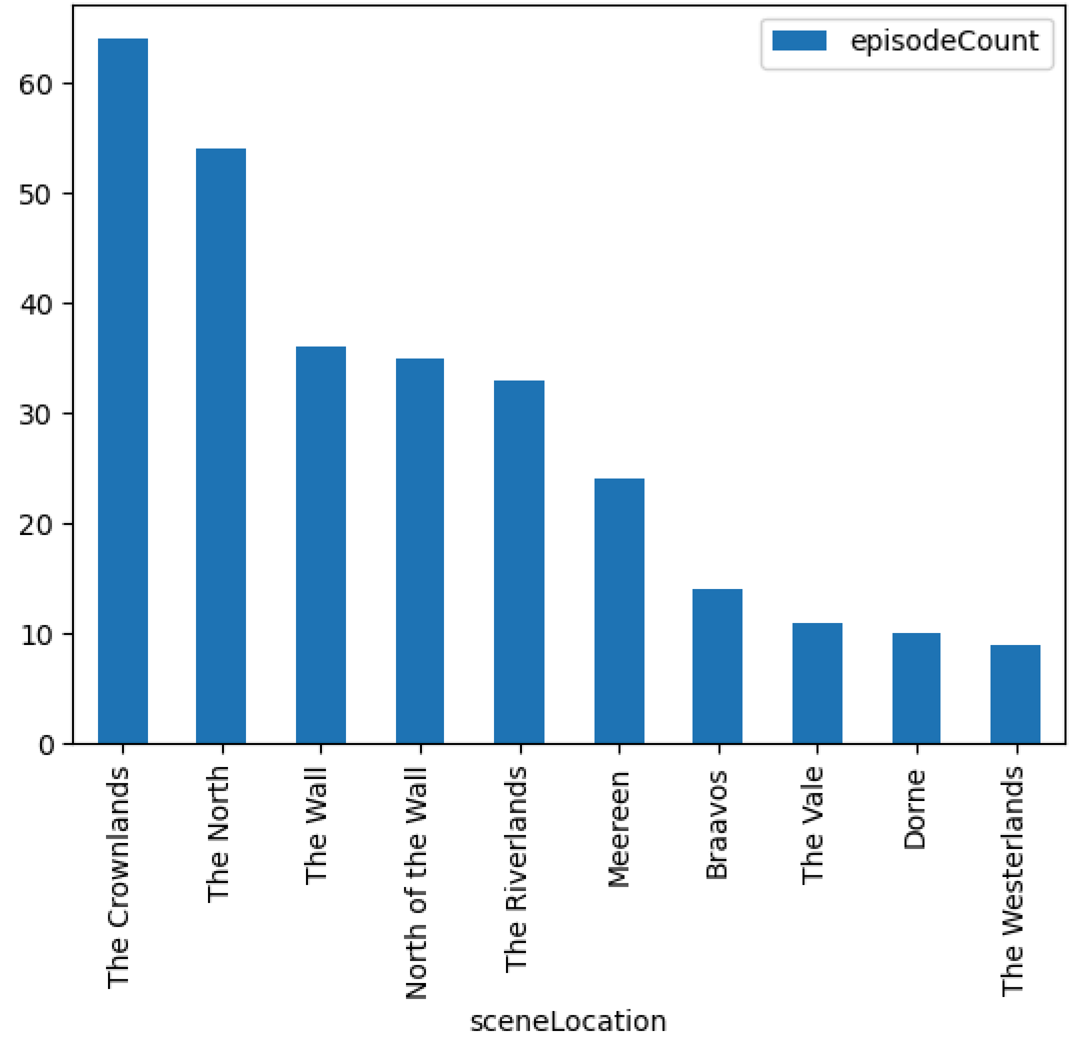
In [48]:
```
# SQL output

location_episode_count = location_episode_count.DataFrame()
location_episode_count
```

Out[48]:

	sceneLocation	episodeCount
0	The Crownlands	64
1	The North	54
2	The Wall	36
3	North of the Wall	35
4	The Riverlands	33
5	Meereen	24
6	Braavos	14
7	The Vale	11
8	Dorne	10
9	The Westerlands	9

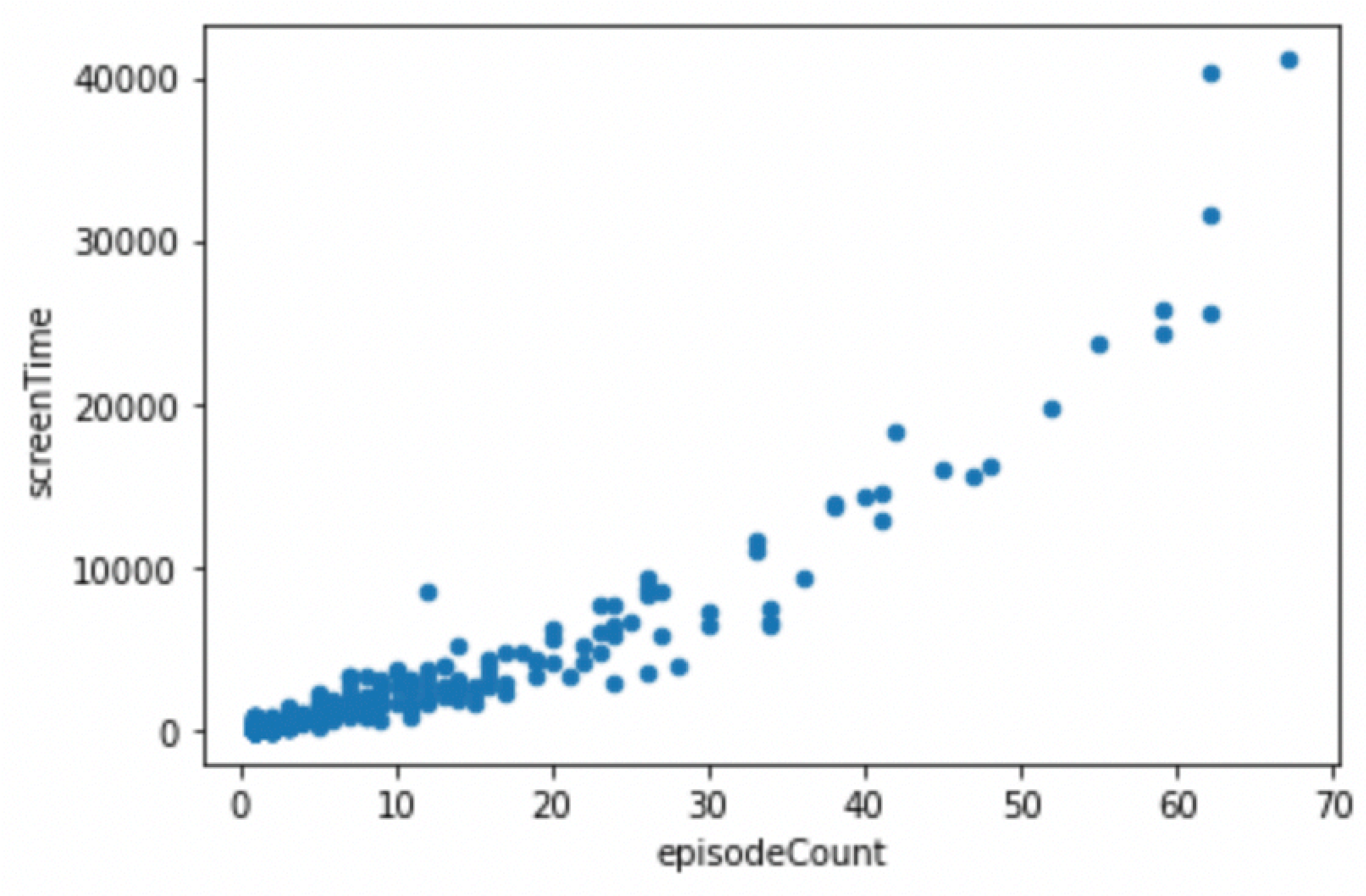
In [49]:
```
# TODO: visualization
location_episode_count.plot.bar(x='sceneLocation', y='episodeCount')
```

Out[49]:
```
<Axes: xlabel='sceneLocation'>
```



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 [41]:
```
%%sql

episode_count_screen_time <<
SELECT
    two.characterName AS characterName,
    COUNT(*) AS episodeCount,
    SUM(two.Screentime) AS ScreenTime
FROM
    (SELECT
        one.seasonNum AS seasonNum,
        one.episodeNum AS episodeNum,
        one.characterName AS characterName,
        SUM(one.sceneLength) AS ScreenTime
    FROM
        (SELECT
            es.seasonNum AS seasonNum,
            es.episodeNum AS episodeNum,
            es.sceneNum AS sceneNum,
            ec.characterName AS characterName,
            TIME_TO_SEC(TIMEDIFF(es.sceneEnd,es.sceneStart)) AS sceneLength,
            es.sceneLocation AS sceneLocation
        FROM episodes_scenes es
            INNER JOIN episodes_characters ec ON es.seasonNum=ec.seasonNum
        WHERE
            es.seasonNum=ec.seasonNum AND
            es.episodeNum=ec.episodeNum AND
            es.sceneNum=ec.sceneNum) one
    GROUP BY
        seasonNum, episodeNum, characterName) two
GROUP BY
    characterName
```

```
   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
```

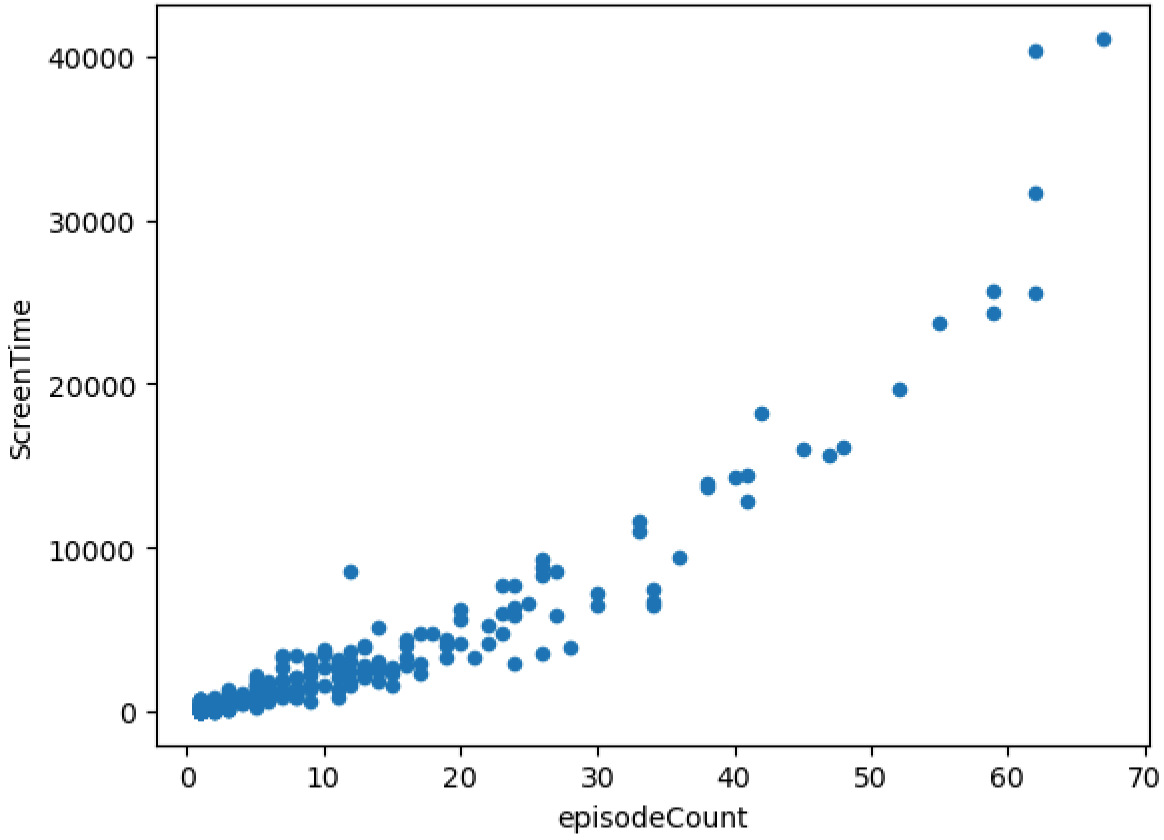
In [42]:
```
# SQL output
# Output is big, so just show first 10 rows

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

Out[42]:

	characterName	episodeCount	ScreenTime
0	Gared	1	362
1	Waymar Royce	1	306
2	Will	1	763
3	Wight Wildling Girl	2	51
4	White Walker	15	1557
5	Jon Snow	62	40365
6	Bran Stark	40	14346
7	Robb Stark	23	7721
8	Eddard Stark	12	8604
9	Catelyn Stark	26	9297

In [43]:
```
# TODO: visualization
episode_count_screen_time.plot.scatter("episodeCount","ScreenTime")
```

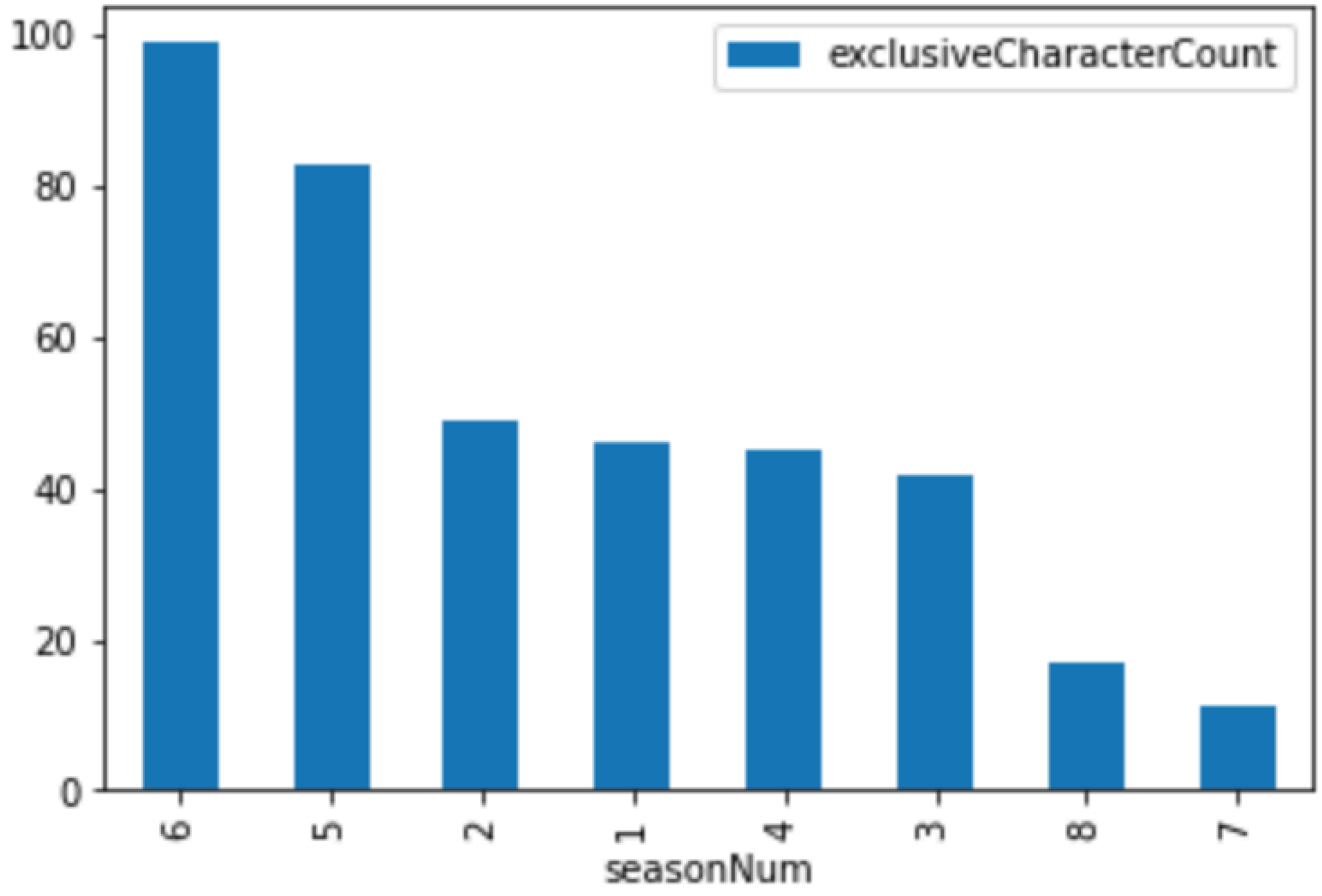
Out[43]:
```
<Axes: xlabel='episodeCount', ylabel='ScreenTime'>
```



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 [44]:
```
%%sql

season_exclusive_characters <<
SELECT
    two.seasonNum AS seasonNum,
    count(*) as exclusiveCharacterCount
FROM
(SELECT MAX(one.seasonNum) as seasonNum, count(*) AS exclusiveCharacterCount FROM
(select DISTINCT seasonNum, characterName from episodes_characters) one
GROUP BY
    characterName
HAVING
    exclusiveCharacterCount = 1) two
GROUP BY
    two.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
```

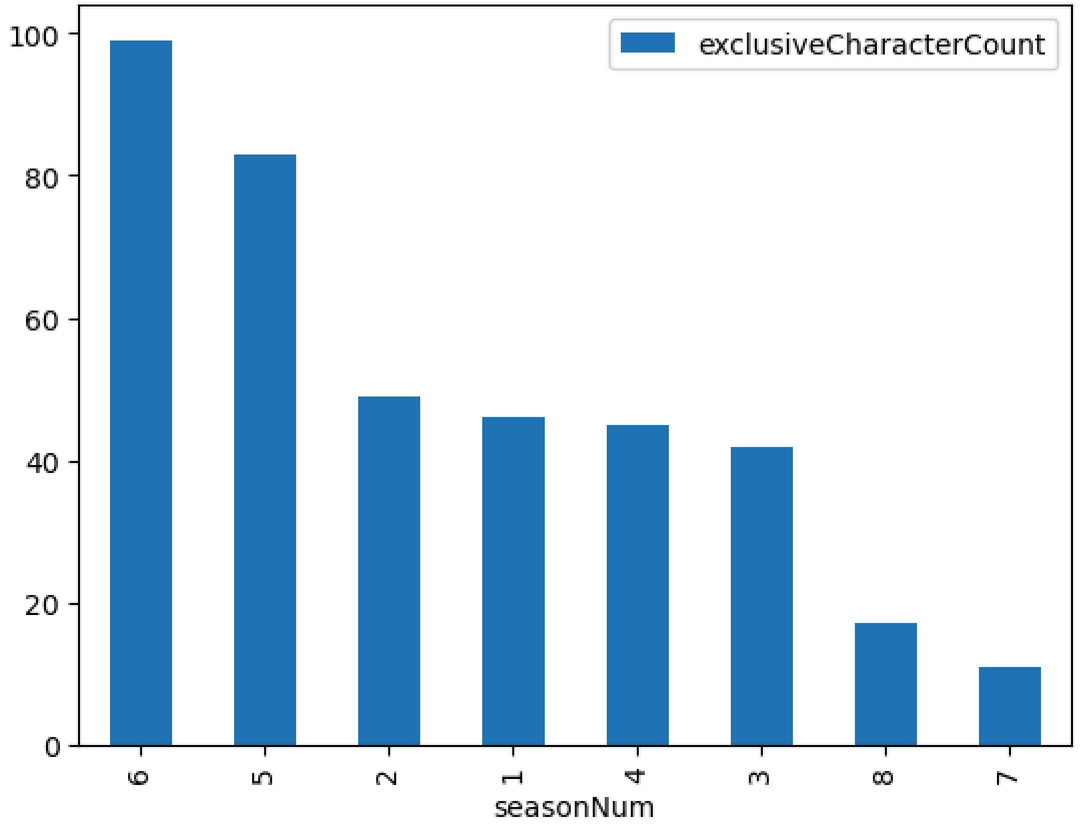
In [45]:
```
# SQL output

season_exclusive_characters = season_exclusive_characters.DataFrame()
season_exclusive_characters
```

Out[45]:

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

In [46]:
```
# TODO: visualization
season_exclusive_characters.plot.bar(x='seasonNum', y='exclusiveCharacterCount')
```

Out[46]:
```
<Axes: xlabel='seasonNum'>
```



In [ ]: