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

Answer:

RAM

1. Volatile, temporary memory (only holds data while in use)
2. Very fast access time
3. Storage is generally limited randing from 1-64GB

SSD

1. Non-volatile, permanent memory (holds data without power)
2. Slower access time compared to RAM but faster than HDD
3. Storage is higher than RAM but generally less than HDD (more expensive per GB than HDD)

HDD

1. Non-volatile, permanent memory (holds data without power)
2. Slowest access time compared to RAM and SSD
3. Storage is highest and very cheap per GB

W2

With regards to disk drives, define

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

Answer:

1. Time taken to move drive's head to the correct track location on disk in which the data for read/write operations is located.
2. Time taken to rotate into the correct sector of the track in which the data for read/write operatiosn is located. Together with seek time it accounts for total time for read/write ops.
3. The time taken to transfer data from computer to memory after the location is identified. Usually measured in MB/s or GB/s.

W3

Explain the concepts of

- Logical block addressing
- Cylinder-head-sector addressing

Answer:

1. Used logical addressing rather than physical addressing. Uses specific numbers for blocks of memory, simplifying data access.
2. Uses physical address specified by 3d coordinate system of head, cylinder, and sector. More complex and less efficient than LBA.

W4

Define and list some benefits of

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

Answer: Ref: https://geeksforgeeks.org/difference-between-row-oriented-and-column-oriented-data-stores-in-dbms/# (https://geeksforgeeks.org/difference-between-row-oriented-and-column-oriented-data-stores-in-dbms/#)

1. Fixed-Length Reocrds

- These have the same size for all records
- Accessing particular records is easier since calculating offset if formulaic.
- Very inefficient since all records need to be length of longest record

2. Variable-length records

- Differen size for all records
- Accessing particular records is not as simple as fixed-length.
- More efficient and flexible since records need to only be as long as required

3. Row-oriented storage

- All records have a "row" in the database, traditional storage technique
- Easier to perform read/write/update operations
- Data-compression techniques less effective

4. Column-oriented storage

- All records are stored by columns.
- Compression techniques are more effective/
- Read/write/update ops are less efficient.

W5

Explain and list some differences between

- RAID 0
- RAID 1
- RAID 5

Answer: Reference: https://en.wikipedia.org/wiki/Standard_RAID_levels (https://en.wikipedia.org/wiki/Standard_RAID_levels)

1. RAID 0

- Data is split evenly into 2 or more discs without redundancy, parity.
- Any failure in one disc results in all data being lost (no fault tolerance)
- Uses capacity proportional to size of smallest disc.

2. RAID 1

- Data is replicated into 2 or more discs with redundancy but no parity.
- Provides fault tolerance due to multiple copies.
- Uses capacity equal to that of one disc with multiple copies.

3. RAID 5

- Essentially RAID 0 but with parity. Data is striped across multiple discs and parity info distributed.
- Provides fault tolerance since data can be rebuilt using this information.

- Uses capacity equal to all discs but 1 (that one being used for 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 (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) PRIMARY KEY,
    nameFirst    VARCHAR(50) NOT NULL,
    nameLast     VARCHAR(50) NOT NULL,
    birthYear    INT,
    birthCountry VARCHAR(100),
    deathYear    INT,
    deathCountry varchar(100)
);
```

```
 * 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

nameFirst	nameLast	personID
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 definer = root@localhost function generate_person_id(nameFirst VARCHAR(50), nameLast VARCHA
returns CHAR(9)
deterministic
begin
    declare last_substr VARCHAR(5);
    declare last_clean VARCHAR(50);
    declare first_substr VARCHAR(2);
    declare first_clean VARCHAR(50);
    declare id_num INT;
    declare id_prefix VARCHAR(7);
    declare id_pattern VARCHAR(8);
    declare result VARCHAR(9);

    set id_num = 0;

    set last_clean = replace(replace(replace(replace(replace(nameLast, '.', ''), '-', ''), ' ', '
    set last_substr = lower(substr(last_clean, 1, 5));

    set first_clean = replace(replace(replace(replace(replace(nameFirst, '.', ''), '-', ''), ' ',
    set first_substr = lower(substr(first_clean, 1, 2));

    set id_prefix = concat(last_substr, first_substr);
    set id_pattern = concat(id_prefix, '%');

    select count(*) into id_num from People
        where personID like id_pattern;

    set result = concat(id_prefix, LPAD(cast((id_num+1) as char), 2, '0'));

    return result;
end;
```

```
 * mysql+pymysql://root:***@localhost
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 set_person_id
    before insert
    on People
    for each row
begin
    set new.personID = generate_person_id(new.nameFirst, new.nameLast);
end;
```

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

Out[6]: []

In [7]:
```
%%sql

create trigger update_person_id
    before update
    on People
    for each row
begin
    if old.personID != new.personID then
        signal SQLSTATE '45000'
        set message_text = "Changing personID is not allowed!!!";
        set new.personID = old.personID;
    end if;

    if old.nameFirst != new.nameFirst or old.nameLast != new.nameLast then
        set new.personID = generate_person_id(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(50),
    in nameLast VARCHAR(50),
    in birthYear INT,
    in birthCountry VARCHAR(100),
    in deathYear INT,
    in deathCountry VARCHAR(100),
    out personID char(9)
)
begin
    set personID = generate_person_id(nameFirst, nameLast);

    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 personID_old CHAR(9),
    in nameFirst_new VARCHAR(50),
    in nameLast_new VARCHAR(50),
    in birthYear_new INT,
    in birthCountry_new VARCHAR(100),
    in deathYear_new INT,
    in deathCountry_new VARCHAR(100),
    out personID_new CHAR(9)
)
begin
    declare last_substr VARCHAR(5);
    declare first_substr VARCHAR(2);
    declare last_substr_old VARCHAR(5);
    declare first_substr_old VARCHAR(2);

    set last_substr = left(lower(nameLast_new), 5);
    set first_substr = left(lower(nameFirst_new), 2);

    set last_substr_old = left(personID_old, 5);
    set first_substr_old = left(right(personID_old, 2+2), 2);

    if last_substr != last_substr_old or first_substr != first_substr_old then
        set personID_new = generate_person_id(nameFirst_new, nameLast_new);
    else
        set personID_new = personID_old;
    end if;

    update People
    set
        nameFirst = COALESCE(nameFirst_new, nameFirst),
        nameLast = COALESCE(nameLast_new, nameLast),
        birthYear = COALESCE(birthYear_new, birthYear),
        birthCountry = COALESCE(birthCountry_new, birthCountry),
        deathYear = COALESCE(deathYear_new, deathYear),
        deathCountry = COALESCE(deathCountry_new, deathCountry)
    where personID = personID_old;
```

```
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';

grant select on s24_hw3.* TO 'general_user'@'localhost';
grant execute on procedure s24_hw3.createPerson TO 'general_user'@'localhost';
grant execute on procedure s24_hw3.updatePerson TO 'general_user'@'localhost';

revoke insert, delete, update on s24_hw3.* from 'general_user'@'localhost';
```

```
 * mysql+pymysql://root:***@localhost
(pymysql.err.OperationalError) (1396, "Operation CREATE USER failed for 'general_user'@'localhos
t'")
[SQL: create user 'general_user'@'localhost' identified by 'dbuserdbuser';]
(Background on this error at: https://sqlalche.me/e/20/e3q8) (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

Answer: Assuming we have some way of knowing whether a person is a manager or a player, we can create two views based on People, one each for player and manager. If we can add another column in People table which says if a person is a player or manager then creating the two views with a simple "where" condition is easy. If we can't modify the people table, then we

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 [11]:
```
# 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[11]:

	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 [12]:
```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()

    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 [13]:
```
%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://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 [14]:
```
# 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 [15]:
```
# 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 [16]:
```
# 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 [17]:
```
# 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 [18]:
```
# 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, 'Changing personID is not allowed!!!')
[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 [19]:
```
# 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 [20]:
```
# Connect to database as general_user
%sql mysql+pymysql://general_user:dbuserdbuser@localhost/s24_hw3
```

In [21]:
```
# Checking if Select works
%sql select * from s24_hw3.People limit 10;
```

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

Out[21]:

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 [22]:
```
# Checking if Execture works
%sql call s24_hw3.createPerson('Sparsh', 'Binjrajka', 1999, 'UK', null, null, @newPersonID);

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

Out[22]:

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
binjrsp01	Sparsh	Binjrajka	1999	UK	None	None

In [23]:
```
# Checking if insert fails
%sql insert into People (nameFirst, nameLast, birthYear, birthCountry, deathYear, deathCountry) v
%sql select * from s24_hw3.People where People.personID = 'williro01';
```

```
 * 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 ('Robin', 'Williams' , 1945, 'USA' , null, null);]
(Background on this error at: https://sqlalche.me/e/20/e3q8) (https://sqlalche.me/e/20/e3q8))
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
0 rows affected.
```

Out[23]:

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry

In [24]:
```
# Checking if update fails
%sql update s24_hw3.People set deathYear = 2025 where nameFirst = 'Sparsh';
%sql select * from s24_hw3.People where People.nameFirst = 'Sparsh';
```

```
 * 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 deathYear = 2025 where nameFirst = 'Sparsh' ;]
(Background on this error at: https://sqlalche.me/e/20/e3q8) (https://sqlalche.me/e/20/e3q8))
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
```

Out[24]:

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
binjrsp01	Sparsh	Binjrajka	1999	UK	None	None

In [25]:
```
# Checking if update fails
%sql delete from s24_hw3.People where nameFirst = 'Sparsh';
%sql select * from s24_hw3.People where People.nameFirst = 'Sparsh';
```

```
 * 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 nameFirst = 'Sparsh' ;]
(Background on this error at: https://sqlalche.me/e/20/e3q8) (https://sqlalche.me/e/20/e3q8))
 * mysql+pymysql://general_user:***@localhost/s24_hw3
   mysql+pymysql://root:***@localhost
   mysql+pymysql://root:***@localhost/s24_hw3
1 rows affected.
```

Out[25]:

personID	nameFirst	nameLast	birthYear	birthCountry	deathYear	deathCountry
binjrsp01	Sparsh	Binjrajka	1999	UK	None	None

GoT Data Visualization

Data Loading

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

In [26]:
```
%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 [27]:
```
%%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 episodeEnd
    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
```

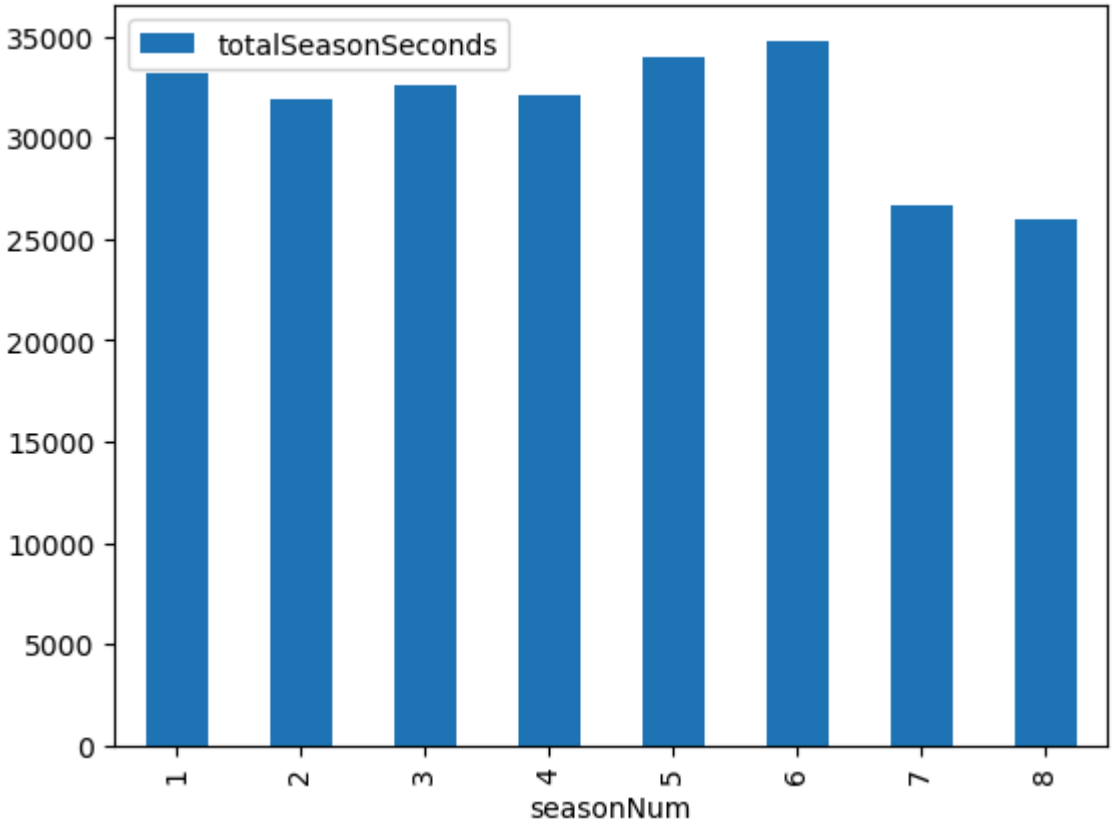
In [28]:
```
# You must show the SQL output

season_running_time = season_running_time.DataFrame()
season_running_time
```

Out[28]:

	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

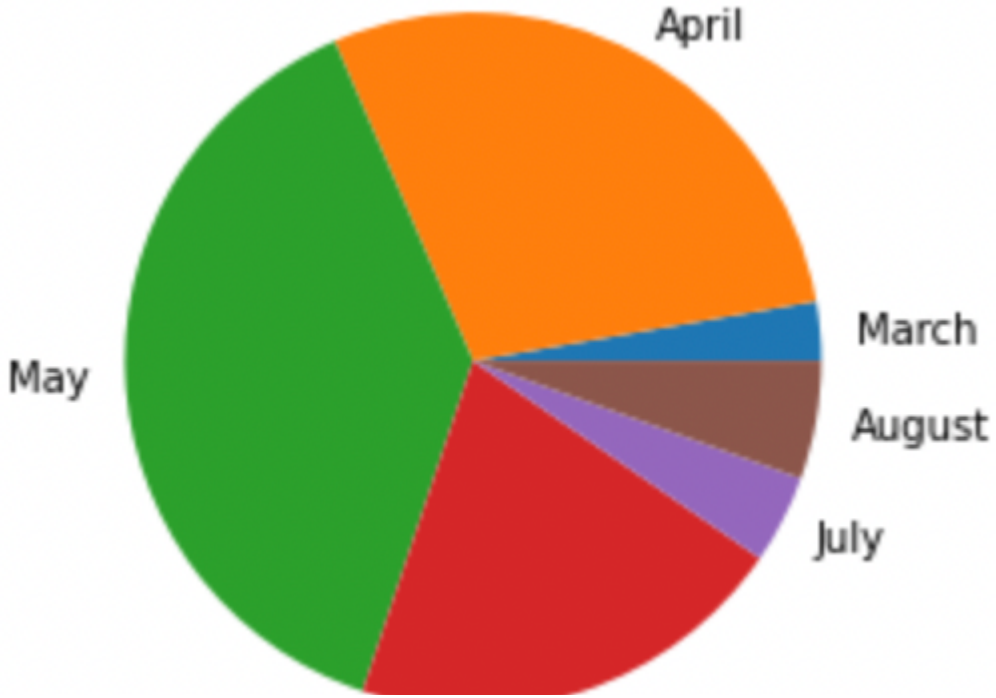
In [29]:
```
# You must show the visualization

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

Out[29]: <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 [30]:
```
%%sql

episodes_per_month <<

select month_, count(*) as episode_count
from(
    select *, month(episodeAirDate) as month_
    from episodes_basics) q
group by month_
```

```
   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
```

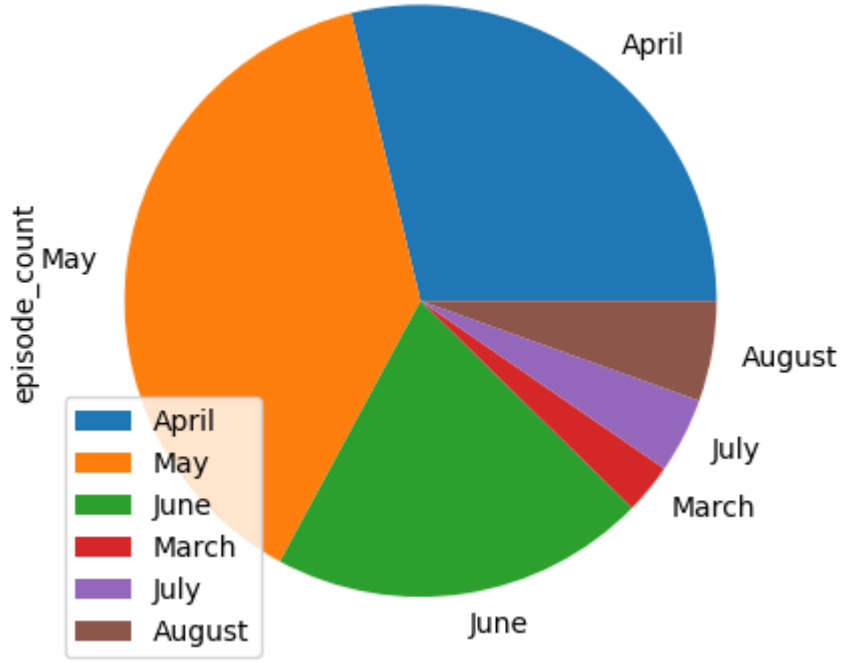
In [31]:
```
# SQL output

episodes_per_month = episodes_per_month.DataFrame()
episodes_per_month
```

Out[31]:

	month_	episode_count
0	4	21
1	5	28
2	6	15
3	3	2
4	7	3
5	8	4

In [32]:
```
# TODO: visualization
number_to_month_map = {1: 'January', 2: 'February', 3: 'March', 4: 'April', 5: 'May', 6: 'June',
                       8: 'August', 9: 'September', 10: 'October', 11: 'November', 12: 'December'

episodes_per_month['month_'] = episodes_per_month['month_'].map(number_to_month_map)
episodes_per_month.index = episodes_per_month['month_']
episodes_per_month.plot.pie(y='episode_count')
```

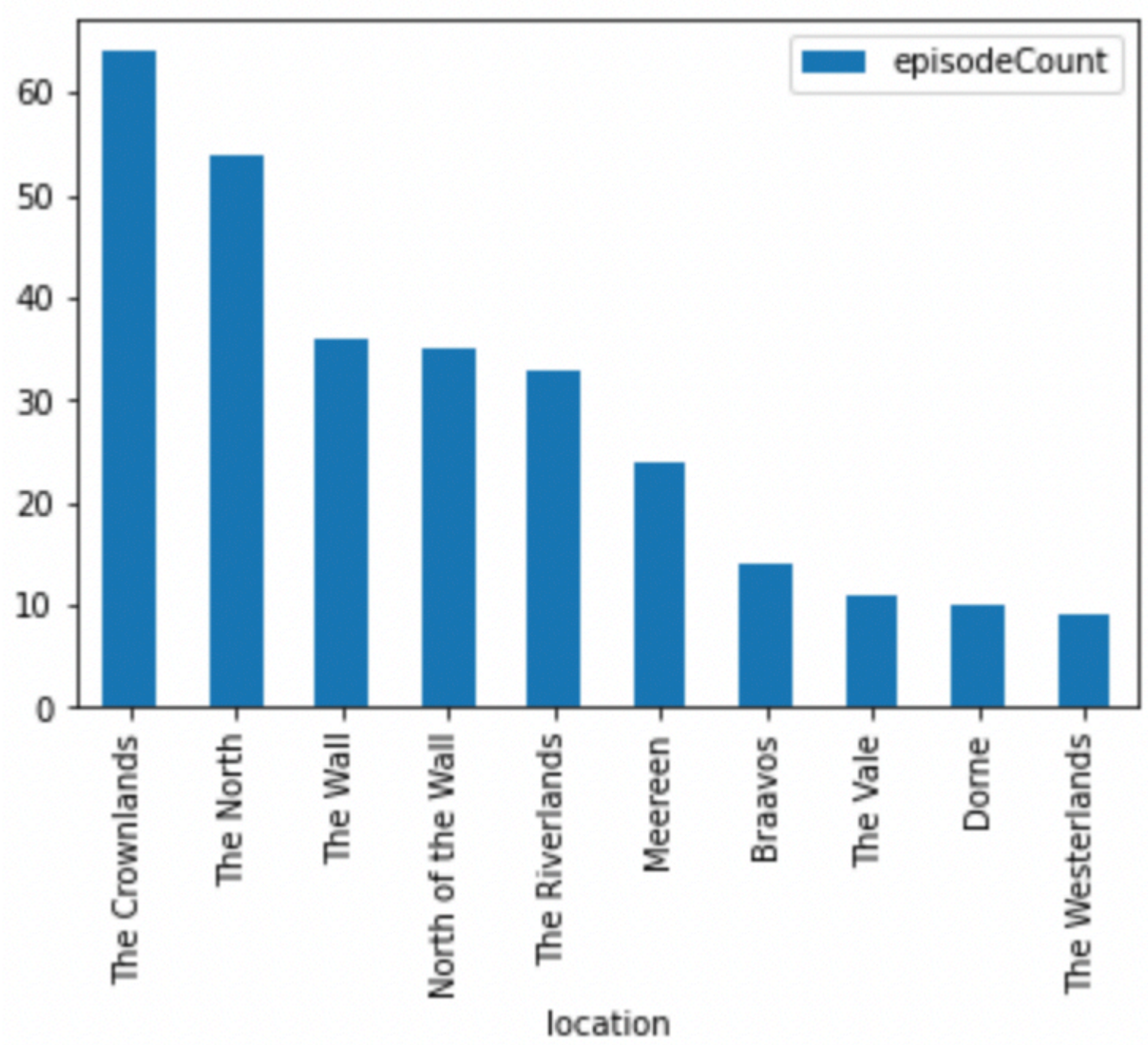
Out[32]:
```
<Axes: ylabel='episode_count'>
```



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:



In [33]:
```
%%sql

location_episode_count <<

select sceneLocation, count(distinct seasonNum, episodeNum) as episode_count
from episodes_scenes
group by sceneLocation
order by episode_count 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
```

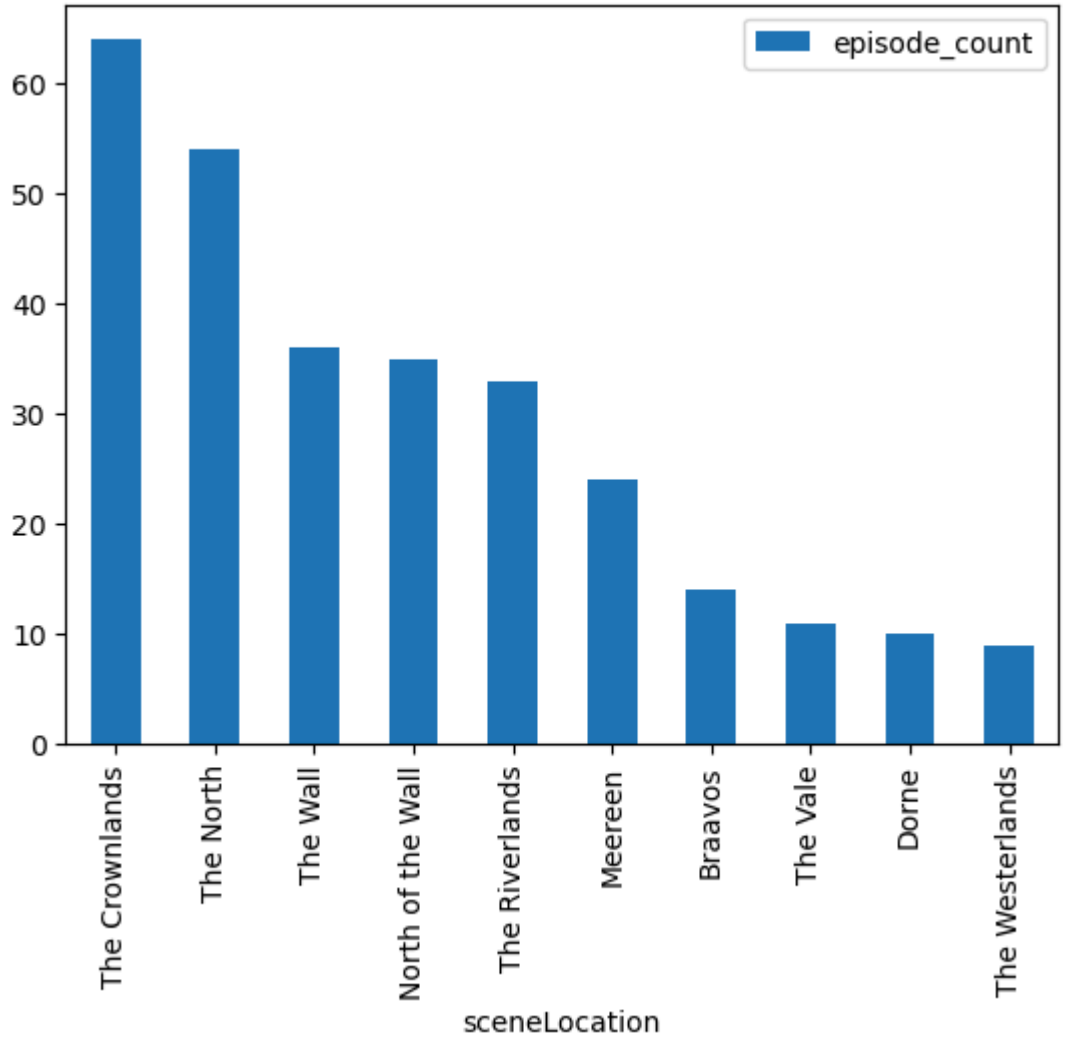
In [34]:
```
# SQL output

location_episode_count = location_episode_count.DataFrame()
location_episode_count
```

Out[34]:

	sceneLocation	episode_count
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

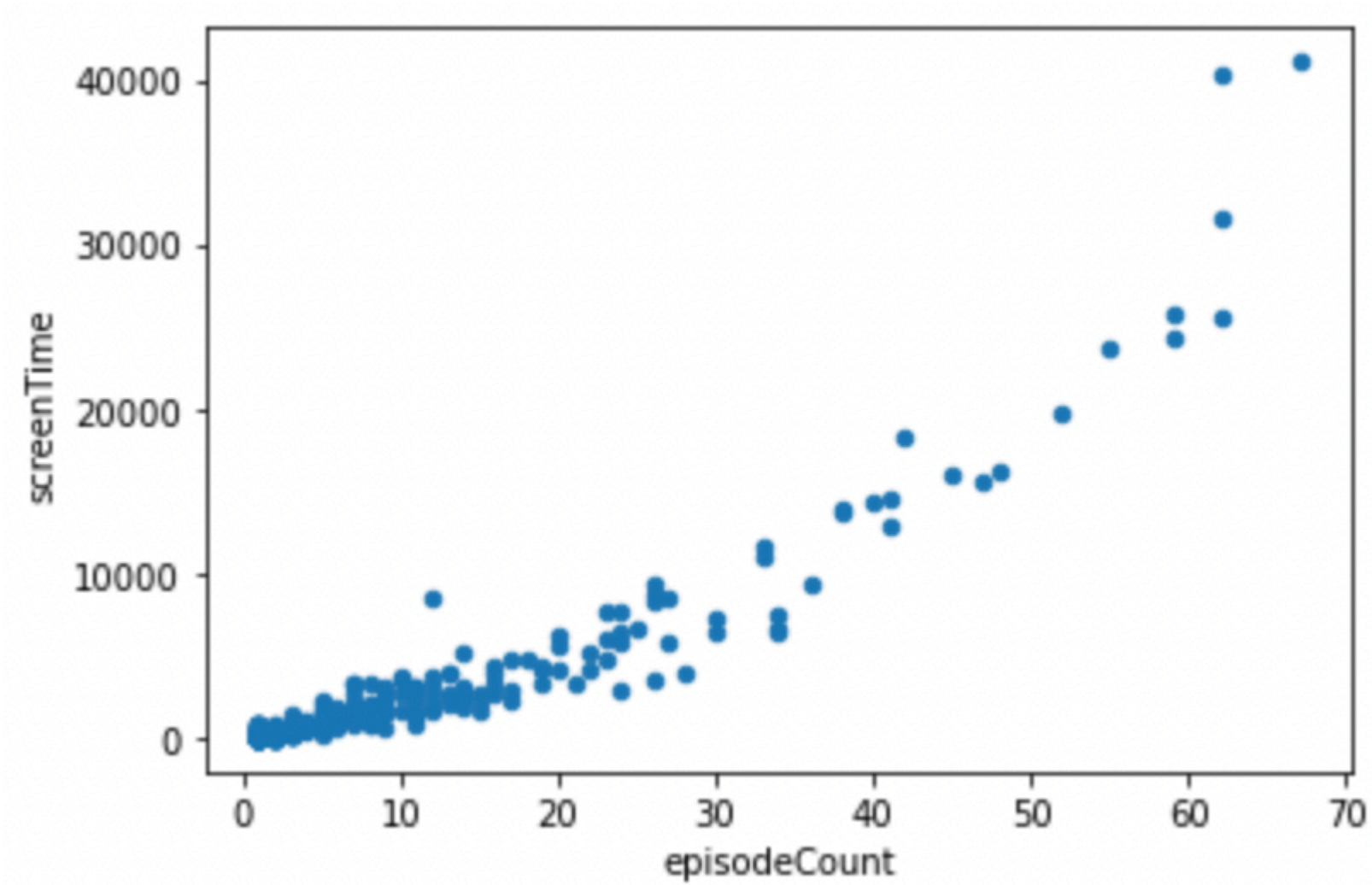
In [35]:
```
# TODO: visualization
location_episode_count[['sceneLocation', 'episode_count']].plot.bar(x='sceneLocation', y='episode_
```

Out[35]: <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 [36]:
```
%%sql

episode_count_screen_time <<

select characterName, count(distinct seasonNum, episodeNum) as episodeCount,
        sum(time_to_sec(sceneEnd) - time_to_sec(sceneStart)) as screenTime
from(
    select characterName, e.episodeNum as episodeNum, e.seasonNum as seasonNum, sceneStart, scene
    from episodes_characters e left join episodes_scenes es
        on e.episodeNum = es.episodeNum and e.seasonNum = es.seasonNum and e.sceneNum = es.sceneN
group by 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
```

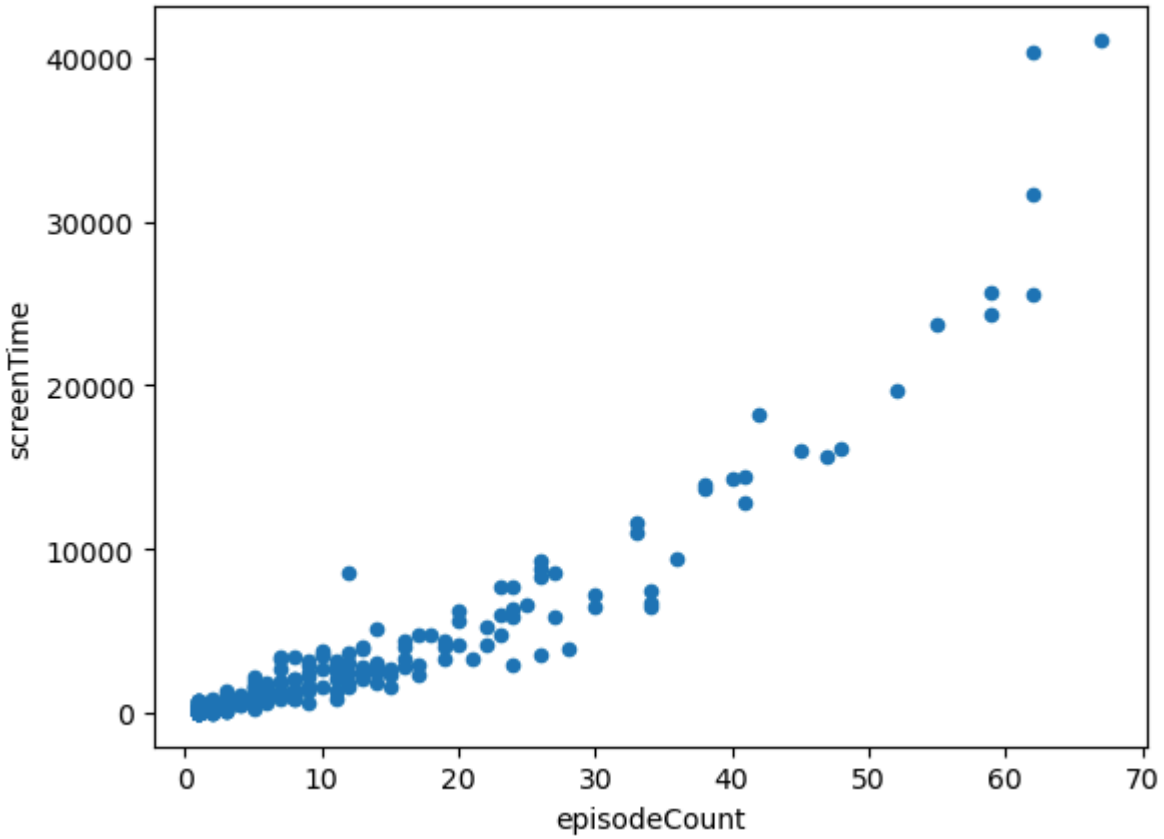
In [37]:
```
# SQL output
# Output is big, so just show first 10 rows

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

Out[37]:

	characterName	episodeCount	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 [38]:
```
# TODO: visualization

episode_count_screen_time.plot.scatter(y='screenTime', x='episodeCount')
```

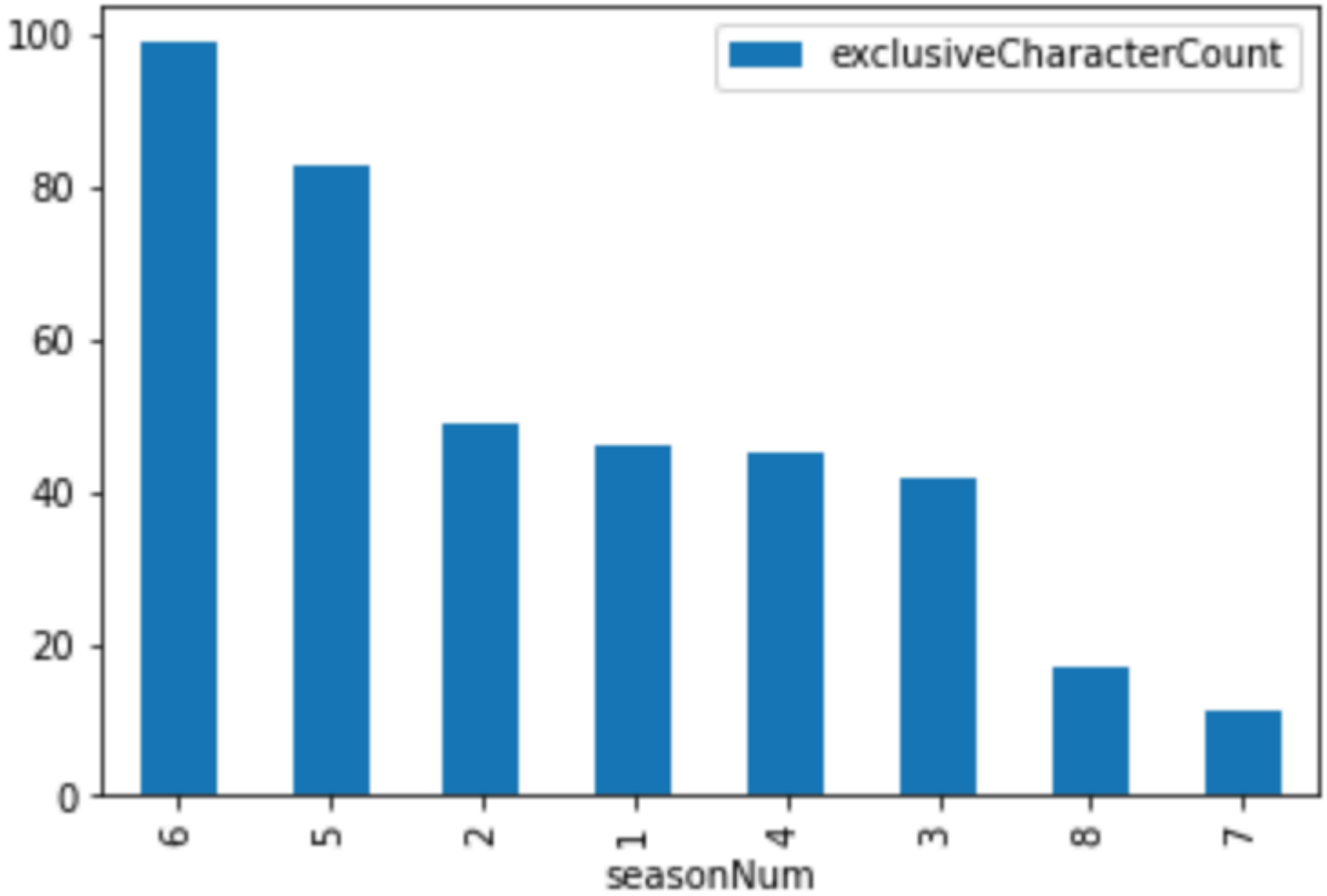
Out[38]: <Axes: xlabel='episodeCount', ylabel='screenTime'>



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 [39]:
```
%%sql

season_exclusive_characters <<

with one as (
    select characterName, count(distinct seasonNum) as season_count
    from episodes_characters
    group by characterName)
select seasonNum, count(distinct one.characterName) as exclusiveCharacterCount
from episodes_characters e left join one on e.characterName = one.characterName
where season_count = 1
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
```

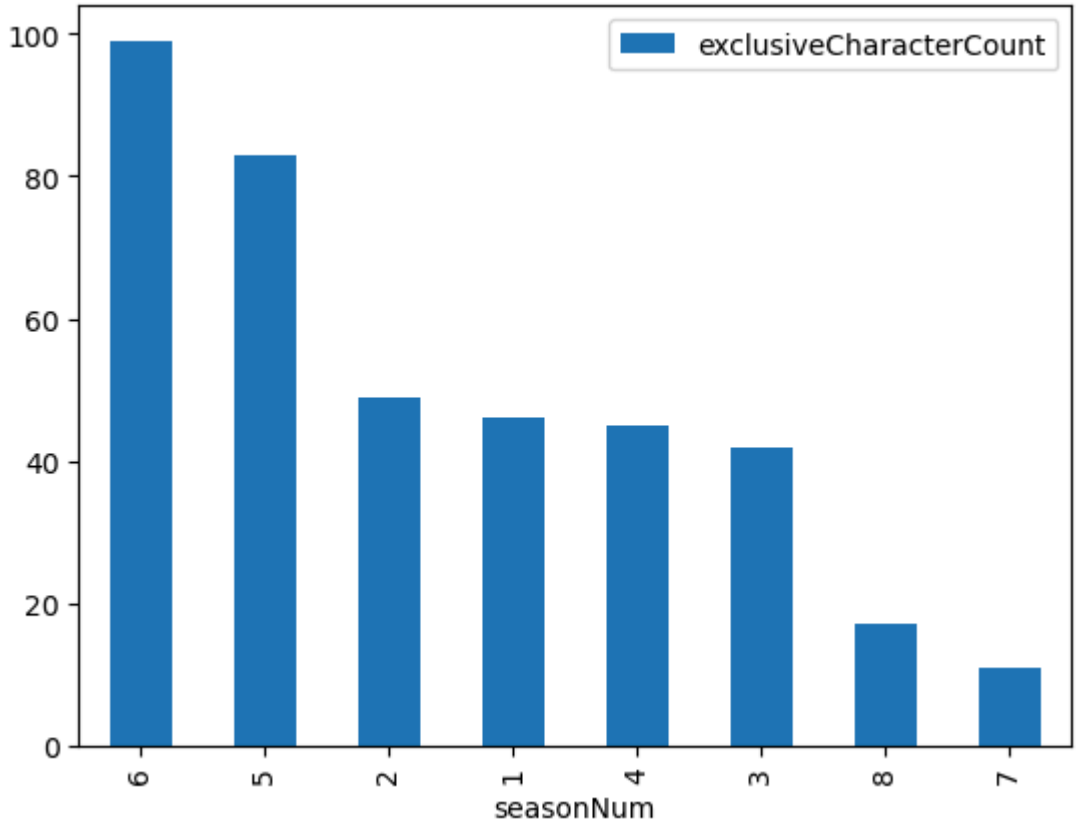
In [40]:
```
# SQL output

season_exclusive_characters = season_exclusive_characters.DataFrame()
season_exclusive_characters
```

Out[40]:

	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 [41]:
```
# TODO: visualization

season_exclusive_characters.plot.bar(y='exclusiveCharacterCount', x='seasonNum')
```

Out[41]: <Axes: xlabel='seasonNum'>



In [ ]: