Assignment #8

👀 Click on this link: https://classroom.github.com/a/DmNcLsP8 (https://classroom.github.com/a/DmNcLsP8) to accept this assignment in GitHub classroom. This will create your homework repository. Clone your new repository.

In this homework, you'll:

1. Use window functions and views
2. Analyze query performance and work with indexes
3. Use SQLAlchemy to interact with a relational database

Part 1: Importing Data

Download the zip file linked to from the top of this kaggle project on Olympic history (https://www.kaggle.com/datasets/heesoo37/120-years-of-olympic-history-athletes-and-results). You may need to log in to kaggle to download this file. If you do not have an account to download the dataset, please reference the pinned post regarding this homework on the course forum.

The two files contained in the archive (zip file) are:

- `athlete_events.csv`
- `noc_regions.csv`

These files contain 120 years of Olympic history: from Athens 1896 to Rio 2016. The license on the analysis and dataset is CC0. Note, however, that the dataset is a result of a screen scrape of a site that had been taken down and revived again recently. The newest version of the site does not have a license.

To import this data:

1. Download the csvs into the root directory of your repository
2. Create a database called `homework08`
3. In your `homework08` database, create two tables to hold the data from the csvs
    - use the `create.sql` script provided in the repository
    - the sql in `create.sql` drops the tables that are about to be created if they exist (`DROP TABLE IF EXISTS tablename;`)... dropping `athlete_event` first, followed by `noc_region`
    - the create statements are written so that:
        - each table matches the file name, but singular (`athlete_event` and `noc_region`)
        - there is an artificial primary key for `athlete_event` called `athlete_event_id` (this is different from `id` , which is included in the original dataset)
        - all columns in the files are represented in the tables
        - the column names are lowercase
4. Use `copy` or `\copy` to import data for each table
    - again, do this in your `homework08` database
    - remember that `\copy` allows relative paths, but must be written on a single line
    - ⚠️ **when importing, treat the value `NA` as `null`**
    - for example, using `\copy` might look like this:
        - `\copy noc_region from data/noc_regions.csv with csv null as 'NA' header`
        - `\copy athlete_event (id, name, sex, age, height, weight, team, noc, games, year, season, city, sport, event, medal) from data/athlete_events.csv with csv null as 'NA' header`

Part 2: Views 👀 and Windows 🪟

1. Write a View

In `queries.sql` , write the number and topic of this query in an sql comment. Write your sql query below the comment. The query should be based on the requirements below. The output of the query does not have to be included.

Requirements

Create a view that will help you in writing any of the 3 following queries (2 - 4). Read through the specifications of the next 3 queries. Based on those requirements, write a view that you'll use later.

For example, your view can:

- automatically join the two tables based on `noc`
- only show rows where a medal exist
- group to count medals
- any combination of the above

You can name the view whatever you like.

Example Output

In your commandline `psql` client, running `\dv` will show your view.

2. Use the Window Function, `rank()`

In `queries.sql` write the number and topic of this query in an sql comment. Write your sql query below the comment. The query should be based on the requirements below. The output of the query does not have to be included.

Requirements

Show the top **3 ranked regions** for each **fencing** 🤺 event based on the number of **total gold medals** 🥇 that region had for that fencing event:

- display the following columns:
  1. the `region` as specified by the noc_region table (use `noc` to associate the two tables)
     - note that some `athlete_event` rows don't have associated values in `noc_region` based on `noc`
     - consequently, you'll have to use a join that gets all rows from `athlete_event`
     - in theses cases where `region` ends up as null
     - use `Singapore` as the region name for `noc` value, `SGP`
     - for all other `noc` values that lead to a null region name, use the value in `team` instead
     - specifically, `ROT` is `Refugee`, `TUV` is `Tuvalu`, and `UNK` is `Unknown`
  2. `event`
  3. number of gold medals as `gold\_medals`
  4. rank as `rank`
- sort by the event and rank ascending

Example Output

region	event	gold_medals	rank
France	Fencing Men's Foil, Individual	10	1
Italy	Fencing Men's Foil, Individual	9	2
Russia	Fencing Men's Foil, Individual	2	3
Poland	Fencing Men's Foil, Individual	2	3
rows omitted for brevity			
Russia	Fencing Women's epee, Team	8	1
China	Fencing Women's epee, Team	4	2
Romania	Fencing Women's epee, Team	4	2

3. Using Aggregate Functions as Window Functions

In `queries.sql` write the number and topic of this query in an sql comment. Write your sql query below the comment. The query should be based on the requirements below. The output of the query does not have to be included.

Requirements

Show the rolling sum of medals per region, per year, and per medal type.

Example Output

- display the following columns:
  1. the `region` as specified by the noc_region table (use `noc` to associate the two tables)
     - note that some `athlete_event` rows don't have associated values in `noc_region` based on `noc`
     - consequently, you'll have to use a join that gets all rows from `athlete_event`
     - in theses cases where `region` ends up as null
     - use `Singapore` as the region name for `noc` value, `SGP`
     - for all other `noc` values that lead to a null region name, use the value in `team` instead
     - specifically, `ROT` is `Refugee`, `TUV` is `Tuvalu`, and `UNK` is `Unknown`
  2. `year`
  3. `medal`
  4. count of that kind of medal for that year for that region as `c`
  5. sum as the total number of that kind of metal for that region
- hint: use the aggregate function, `sum` (we've seen `sum` before, but used with `GROUP BY`), for this
- hint: the running sum for each year for every region and medal combo is required, consequently, you'll want to sum for that region and medal combination (use this to determine what to partition by)
- sort by `region`, `year` and `medal` ascending

region	year	medal	c	sum
Afghanistan	2008	Bronze	1	1
Afghanistan	2012	Bronze	1	2
Algeria	1984	Bronze	2	2
Algeria	1992	Bronze	1	3
Algeria	1992	Gold	1	1
Algeria	1996	Bronze	1	4
Algeria	1996	Gold	2	3
Algeria	2000	Bronze	3	7
Algeria	2000	Gold	1	4
Algeria	2000	Silver	1	1
Algeria	2008	Bronze	1	8
Algeria	2008	Silver	1	2
Algeria	2012	Gold	1	5
Algeria	2016	Silver	2	4
rows omitted for brevity				

4. Use the Window Function, `lag()`

In `queries.sql` write the number and topic of this query in an sql comment. Write your sql query below the comment. The query should be based on the requirements below. The output of the query does not have to be included.

Requirements

Show the height of every gold medalist for pole valut events, along with the height of the gold medalist for that same pole value event in the previous year.

- display the following columns:
  1. `event`
  2. `year`
  3. `height`

4. the height of last year's gold medalist as `previous_height`

- hint: use the lag (https://www.postgresql.org/docs/14/functions-window.html) function to do this
- note that the previous year may have a null value (for example, if it's the first gold medal for an event, then a previous year would not have a value)
- sort by the event and year ascending

event	year	height	previous_height
Athletics Men's Pole Vault	1906	170	NULL
Athletics Men's Pole Vault	1908	178	170
Athletics Men's Pole Vault	1908	170	178
Athletics Men's Pole Vault	1912	188	170
Athletics Men's Pole Vault	1920	172	188
rows omitted for brevity			
Athletics Women's Pole Vault	2000	172	NULL
Athletics Women's Pole Vault	2004	174	172
Athletics Women's Pole Vault	2008	174	174
Athletics Women's Pole Vault	2012	183	174
Athletics Women's Pole Vault	2016	173	183

Part 3: EXPLAIN / ANALYZE and Indexes

In `explain_analyze.sql` , before each sql statement, in a commen, write the number and task description for each task below:

1. Drop an existing index:
   - in case you've created an index already, add the following query at the beginning of your file:
   - `drop index if exists athlete_event_name_idx;`
2. Write a simple query:
   - write a query to find all rows that contain the athlete `Michael Fred Phelps, II` (use the `name` column)
     - display all columns
   - no specific sort order is required
3. Using `EXPLAIN ANALYZE` :
   - Rewrite your query, but prefix with `EXPLAIN ANALYZE` to show how much time it takes to run your query
   - include the output of the query in a comment beneath your `EXPLAIN ANALYZE` statement
   - for example, it may look like (the execution time will vary):

```
Gather  (cost=1000.00..8079.36 rows=3 width=133) (actual time=64.665..68.857 rows=30 loops=
1)
  Workers Planned: 2
  Workers Launched: 2
  ->  Parallel Seq Scan on athlete_event  (cost=0.00..7079.06 rows=1 width=133) (actual tim
e=35.356..40.256 rows=10 loops=3)
  Filter: (name = 'Michael Fred Phelps, II'::text)
  Rows Removed by Filter: 90362
Planning Time: 4.632 ms
Execution Time: 71.158 ms
```

4. Add an index:
   - write a query to add an index to the `name` column of the `athlete_event` table
   - ⚠️ make sure to name your index `athlete_event_name_idx`
5. Verifying improved performance:
   - repeat your `EXPLAIN ANALYZE` query from (3)
   - again, include the output of the query in a comment beneath your `EXPLAIN ANALYZE` statement

- this time, the output should look like this – note that a `Seq Scan` is **not used**, and instead, an `Index Scan` is used (meaning that the query planner used the index to find the rows more quickly):

```
 Index Scan using athlete_event_name_idx on athlete_event  (cost=0.42..16.42 rows=3 width=1
33) (actual time=0.531..0.552 rows=30 loops=1)
Index Cond: (name = 'Michael Fred Phelps, II'::text)
 Planning Time: 11.278 ms
 Execution Time: 0.868 ms
```

6. Ignoring an index:
    - write a query using the `name` column in the `where` clause
    - try to come up with *some* other operation or filter so that the index is not used (that is, an `Index Scan` is not used)

Part 4: SQLAlchemy

In this part, you'll use SQLAlchemy to write to and read from postgres. You'll be using the existing database from previous parts, `homework08`, that contains your two tables, `athlete_event` and `noc_region`.

👀 The majority of the setup required for SQLAlchemy is already present in the file, `olympics.py`.

1. Configuration
    - to start, define the username, password, host, and database name in a file so that you can use it to connect to your database without hardcoding your database credentials in your source code
    - create a file called `config.ini` in the root of your repository
    - ⚠️ **DO NOT ADD IT TO YOUR REPOSITORY** (a `.gitignore` file is present in your repository already, so git should already exclude `config.ini` from its operations)
    - `config.ini` is read by the in boilerplate code in `olympics.py` to configure the database connection
    - add the following text to your `config.ini` file, replacing the `username` and `password` fields as necessary

```
[db]
username=yourusername
password=yourpassword
host=localhost
database=homework08
```

    - if you do not have a user with a password, you can try creating a new user: `CREATE ROLE newusername WITH SUPERUSER CREATEDB CREATEROLE LOGIN PASSWORD 'newpassword'`
2. Define two classes, `AthleteEvent` and `NOCRegion`
    - see the slides on the SQLAlchemy ORM (../slides/py-db/sql-alchemy-relationships.html?print)
    - allow access related tables in your class definitions
        - you can use the `noc` field in `athlete_event` as a foreign key reference to the `noc_region` table (even without an actual foreign key constraint placed there)
        - represent multiple `athlete_event` children in `noc_region` by using a property, `athlete_events` (plural) in your `NOCRegion` class along with the `relationship` function * `athlete_events = relationship("AthleteEvent", back_populates="noc_region")`
        - do the same from your `AthleteEvent` class so that its associated `noc_region` can be accessed via a property with the same name
        - `noc_region = relationship("NOCRegion", back_populates="athlete_events")`
        - see more details on backpopulation in the docs (https://docs.sqlalchemy.org/en/14/orm/basic_relationships.html)
    - add `__str__` and `__repr__` methods that returning a string containing the values for:
        - `name`
        - `noc`
        - `season`
        - `year`
        - `event`
        - `medal`

3. Insert a new record into the `athlete_event` table using your `AthleteEvent` class by using the following data from the 2020 Summer Olympics for Men's Street Skateboarding:
    - name: Yuto Horigome
    - age: 21
    - team: Japan
    - medal: Gold
    - year: 2020
    - season: Summer
    - city: Tokyo
    - noc: JPN
    - sport: Skateboarding
    - event: Skatboarding, Street, Men
4. Using your class(es) above, perform the following search
    - find all rows in `athlete_event`
    - that have an `noc` as `JPN`
    - ...a `year` that's greater than or equal to `2016`
    - ...and a `medal` that's `'Gold'`
    - use the `query` and `filter` functions (https://docs.sqlalchemy.org/en/14/orm/query.html#sqlalchemy.orm.Query.filter) to do this
    - within the filter function, pass in arguments that will be treated as conditions (for example `AthleteEvent.noc == 'JPN')
    - multiple conditions as arguments will be joined using the logical operator, `AND`
    - (check the docs for more details on filter (https://docs.sqlalchemy.org/en/14/orm/query.html#sqlalchemy.orm.Query.filter))
    - iterate through the result
    - print out the `name`, `region` (from `noc_region`), the `event,` year `,` `and` season` result = session.query(AthleteEvent).filter(AthleteEvent.noc == 'JPN', AthleteEvent.year >= 2016, AthleteEvent.medal == 'Gold')