User-centric Systems for Data Science **Prof. John Liagouris**
Fall Semester 2021 liagos@bu.edu

Assignment 2

Submission deadline: October 22 2021, 11:59 pm

In this assignment you will extend the operator library you built for Assignment 1 to support transparent provenance tracking. The resulting library will enable users to retrieve various types of provenance-related information for individual tuples, such as lineage, Where- and How-provenance. The last assignment task focuses on the concept of data responsibility, which we will discuss in Lecture 6.

For this and future assignments, you will need to have Python 3.7+, Pytest, and Ray installed in your machine (cf. Section 10 "Resources" for more information).

You must follow the code skeleton provided in the Gitlab repository. Inline comments will help you identify the parts of the code you need to fill in. Keep in mind that the assignment does not require writing much code: the logic of each data operator can be implemented in less than 20 LOC. Always keep your code simple and well documented.

We will be using a mix of real and synthetic data. Real data include movie ratings from a large Netflix dataset whereas friendship relationships between users are synthetic. The input data are available in the Gitlab repository. Make sure you understand the data format first (cf. Section 1 "Data schema"). You might also want to create a toy dataset of the same format to test your code easily.

1. Data schema

The data we will use for this assignment consist of two CSV files: `Friends` and `Ratings`. The former contains friendship relationships as tuples of the form `UserID1 UserID2`, denoting two users who are also friends. A user can have one or more friends and the friendship relationship is *symmetric*: if A is a friend of B, then B is also a friend of A and both tuples (`A B` and `B A`) are present in the file. `Ratings` contains user ratings as tuples of the form `UserID MovieID Rating`. For example, tuple `12 3 4` means that "the user with ID `12` gave `4` stars to the movie with ID `3`".

Hint #1: You can use Python's CSV reader to parse input files.

Hint #2: Consider encapsulating your Python tuples in `ATuple` objects (see code skeleton).

2. TASK I: Implement backward tracing (credits: 40/100)

The first task is to extend the operators you built in Assignment 1 with support for backward tracing. For each operator, you will have to implement a new method (in Python 3 syntax):

```
lineage(tuples: List[ATuple]) -> List[List[ATuple]]
```

that returns the lineage of the given list of `tuples`.

As discussed in Lecture 2, the lineage of an output tuple, let *t*, with respect to a query *q(D)* is the collection of input tuples that contributed to having the tuple *t* in the output of the query. Let `recommendation` be the output (movie id) of the second query from Assignment 1:

```
SELECT R.MID
FROM ( SELECT R.MID, AVG(R.Rating) as score
       FROM Friends as F, Ratings as R
       WHERE F.UID2 = R.UID
             AND F.UID1 = 'A'
       GROUPBY R.MID
       ORDERBY score DESC
       LIMIT 1 )
```

To successfully complete this task, you must implement a new method for `ATuple`:

```
lineage() -> List[ATuple]
```

so that you can retrieve the lineage of any `recommendation` as follows:

```
lineage = recommendation.lineage()
```

Calling `recommendation.lineage()` should internally call:

```
operator.lineage(tuples=[recommendation])
```

where `operator` is a handle to the operator that produced the tuple `recommendation` (i.e. the root operator of the query tree).

Example: Let's assume that the recommendation query returns a recommendation `r=ATuple(10)` (where `10` is the movie ID) to user `0` and this recommendation was based on the following input tuples:

```
(in Friends.csv)      (in Ratings.csv)

     0 1                   1 10 5

     0 4                   4 10 8

     0 18                  18 10 2
```

In this case, calling `r.lineage()` should return the following list of tuples as the lineage of `r`:

```
[(0, 1), (0, 4), (0, 18), (1, 10, 5), (4, 10, 8), (18, 10, 2))]
```

Hint #1: This task requires modifying each operator so that it maintains some intermediate data in memory (input-to-output mappings). Try minimizing the amount of intermediate data by leveraging knowledge about the operator logic.

Hint #2: Use the `track_prov` flag to enable and disable materialization of intermediate data.

Hint #3: Make sure the input to the `operator.lineage` method is a list of `tuples` all of which exist in the output of the respective operator.

Hint #4: Calling `lineage` on an operator with a list of `tuples` (recommendations) as input must return the lineage of these `tuples` as a list of lists:

```
[ [lineage_of_tuple_1], [lineage_of_tuple_2], ... ]
```

3. Task II: Implement Where-provenance query (credits: 10/100)

The second task is to extend your operators with support for Where-provenance (cf. Lecture 2). For each operator, you will have to implement a new method (in Python 3 syntax):

```
where(att_index: int, tuples: List[ATuple]) -> List[List[Tuple]]
```

that returns the Where-provenance of the attribute at index `att_index` for each tuple in `tuples`.

Let $t[a]$ be an attribute of a tuple t in the output of a query $q(D)$. As discussed in Lecture 2, the Where-provenance of $t[a]$ is the list of attributes of the input tuples whose values contributed to $t[a]$'s value. Let `prediction` be the output (average rating) of the first query from Assignment 1:

```
SELECT AVG(R.Rating)
FROM Friends as F, Ratings as R
WHERE F.UID2 = R.UID
      AND F.UID1 = 'A' AND R.MID = 'M'
```

To successfully complete this task, you must implement a new method for `ATuple`:

```
where(att_index: int) -> List[Tuple]
```

so that you can retrieve the Where-provenance of any 'likeness' `prediction` as follows:

```
where_from = prediction.where(att_index=0)
```

Calling `prediction.where(att_index=0)` should internally call:

```
operator.where(att_index=0, tuples=[prediction])
```

where `operator` is a handle to the operator that produced the tuple `prediction` (i.e. the root operator of the query tree). The result `where_from` should be a list of tuples of the form:

```
[ (input_filename, line_number, tuple, attribute_value) ]
```

Example: Let's assume that the prediction query returns a predicted rating `r=ATuple(5.0)` for movie `10` and this prediction depends on the average ratings in three input tuples:

```
1 10 5    (line 4 in Ratings.csv)

4 10 8    (line 12 in Ratings.csv)

18 10 2   (line 122 in Ratings.csv)
```

In this case, calling `r.where(att_index=0)` should return the following list of tuples:

```
[('Ratings.csv', 4, (1, 10, 5), 5), ('Ratings.csv', 12, (4, 10, 8), 8), ('Ratings.csv', 122, (18, 10, 2), 2)]
```

Hint #1: Consider building your solution upon the backward tracing functionality you implemented for TASK I.

Hint #2: Make sure the input to the `operator.where` method is a list of `tuples` all of which exist in the output of the respective operator and the given attribute `index` is valid.

Hint #3: Calling `where` on an operator with a list of `tuples` (predictions) as input must return the Where-provenance of the attributes at the given `index` as a list of lists:

```
[ [where_from_tuple_1], [where_from_tuple_2], ... ]
```

4. TASK III: Implement How-provenance query (credits: 30/100)

The third task is to extend your operators with support for How-provenance. In this case, each operator must generate and propagate provenance metadata (cf. Lecture 2) during query execution so that each output tuple has the complete provenance information stored in its metadata. To retrieve the How-provenance for a tuple, you must extend `ATuple` with a new method (in Python 3 syntax):

```
how() -> string
```

that returns the provenance metadata in the form of a string. The example below explains how you should format the result string. Let `recommendation` be the output (movie id) of the query:

```
SELECT R.MID
FROM ( SELECT R.MID, AVG(R.Rating) as score
        FROM Friends as F, Ratings as R
        WHERE F.UID2 = R.UID
              AND F.UID1 = 'A'
        GROUPBY R.MID
        ORDERBY score DESC
        LIMIT 1 )
```

When done with this task, you must be able to retrieve the How-provenance of a `recommendation` as follows:

```
how_prov = recommendation.how()
```

Example: Let's assume that the above query returns the recommendation `r=ATuple(10)` to user `0` based on the following input tuples:

```
(in Friends.csv)      (in Ratings.csv)

     0 1                   1 10 5

     0 4                   4 10 8

     0 18                  18 10 2
```

In this case, calling `r.how()` should return the following provenance metadata in the form of a string:

```
AVG( (f1*r1@5), (f2*r2@8), (f3*r3@2) )
```

where `fi,ri` are unique identifiers for tuples in `Friends` and `Ratings` respectively, `fi*ri` denotes co-existence of tuples `fi` and `ri` in the input, and `ri@c` means that "the actual rating in tuple with id `ri` is `c`" (you will need this information for TASK IV).

Hint #1: Use the `propagate_prov` flag to enable and disable metadata propagation.

Hint #2: Calling `how()` on a `ATuple` should only involve a simple lookup in the tuple metadata.

Hint #3: Consider generating unique identifiers for tuples in `Friends` (resp. `Ratings`) as `fi` (resp. `ri`), where `i` is the line number of the tuple in the file.

5. TASK IV: Retrieve most responsible tuples (credits: 20/100)

The fourth and final task is to implement a method that, given a movie `recommendation` computed by the query of TASK III, returns the input tuples whose responsibility ρ (cf. Lecture 6) for that particular `recommendation` is at least 50% ($\rho \geq 0.5$). More precisely, you will have to implement the following method (in Python 3 syntax):

```
responsible_inputs() ->  List[(ATuple, float)]
```

that returns the list of all tuples with responsibility (`float`) $\rho \geq 0.5$.

When done with this task, you should be able to retrieve all input tuples (from `Friends` and `Ratings`) that are responsible for a particular `recommendation` by at least 50% as follows:

```
tuples = recommendation.responsible_inputs()
```

Example: Let's assume that the recommendation query returns the recommendation `r=ATuple(10)` to user `0` based on the following input tuples:

```
(in Friends.csv)      (in Ratings.csv)

     0 1                   1 10 5

     0 2                   2 10 3

     0 3                   3 9 4

     0 4                   4 9 2
```

In this case, calling `r.responsible_inputs()` should return the following list of tuples:

```
[((0, 1), 0.5), ((0, 2), 0.5), ((0, 4), 0.5), ((1, 10, 5), 0.5), ((2, 10, 3), 0.5), ((4, 9, 2), 0.5)]
```

all of which have responsibility ρ=0.5.

Hint: Consider building your solution upon the How-provenance functionality you implemented for Task III and avoid re-executing the recommendation query multiple times.

6. Testing

You must have a simple test for each operator you implement and we strongly recommend using Pytest for this purpose. In case you are not familiar with Pytest, you might want to first spend some time reading the code snippets provided in the documentation.

All test functions must be added to the separate `tests.py` file provided with the code skeleton. Before submitting your solution, make sure the command `pytest tests.py` runs all your test functions successfully.

7. Logging

Logging can save you hours when debugging your program, and you will find it extremely helpful as your codebase grows in size and complexity. Always use Python's logger to generate messages and avoid using `print()` for that purpose. Keep in mind that logging has some performance overhead even if set to INFO level.

8. Git

You will use git to maintain your codebase and submit your solutions. If you are not familiar with git, you might want to have a look here. Note that well-documented code is always easier to understand and grade, so please make sure your code is clean, readable, and has comments.

Make sure your git commits have meaningful messages. Messages like "*Commit*" or "*Fix*", etc. are pretty vague and must be avoided. Always try to briefly describe what each commit is about, e.g. "*Add Join operator*", "*Fix provenance tracking in AVG operator*", etc., so that you can easily search your git log if necessary.

Each time you finish a task (including the related tests), we strongly recommend that you use a commit message "Complete *Task X*". You will find these commits very helpful in case you want to rollback and create new branches in your repository.

9. Deliverables

Each submission must be marked with a commit message "*Submit Assignment X*" in git. If there are multiple submissions with the same message, we will only consider the last one before the deadline. In case all your submission commits are after the deadline, we will only consider the last commit, however, **late submissions will be eligible for up to 50% of the original grade**.

Your submission must contain:

1. The code you wrote to solve the assignment tasks (in `assignment.py`)
2. The code you wrote for testing (in `tests.py`)

Before submitting your solution, always make sure your code passes all tests successfully.

10. Resources

- Explaining Collaborative Filtering Recommendations: https://tinyurl.com/y8otpqnp
- Python tutorial: https://docs.python.org/3/tutorial/
- Python logger: https://docs.python.org/3/library/logging.html
- Pytest: https://docs.pytest.org/en/stable/
- Ray documentation: https://docs.ray.io/en/latest/
- Git Handbook: https://guides.github.com/introduction/git-handbook/