Case Study

Now that you have learnt about functional dependencies (FD), you will encounter some algorithms to compute certain operations. Algorithms can be executed by a computer. So, that is exactly what we are going to do. To simplify your job, we will have quite a number of useful classes written in Python. This will also be useful to help you better understand normal forms (NF).

Files

You are provided with the following files from Canvas "`Files > Cases > FD`":

File Name	Description
`Types.py`	A Python class for attributes, relations, FD, and set of FDs.
`Util.py`	A collection of functions to simplify creation of classes.
`FD.py`	A collection of algorithms, this will be your main task.
`Tests.py`	A simple test cases, you are encouraged to add your own test cases.

Python

To use the file, you will need Python installed, preferably Python 3.11 and above. You can download Python at the following website: `https://www.python.org/downloads/`. Follow the instruction depending on your operating system:

- Windows: `https://docs.python.org/3/using/windows.html#installation-steps`
- MacOS: `https://docs.python.org/3/using/mac.html#getting-and-installing-python`

We would recommend

- **Uncheck** the box for "Install launcher for all users (recommended)".
- **Check** the box for "Add Python 3.XX to PATH".

There is no other additional libraries you need.

Classes

We will describe the basic classes that are provided in `Types.py`. Note that we will only talk about a simplified description by explaining the operations.

Attrs

This class encapsulates a set of attributes.

- Instantiation: `attr = Attrs('A', 'B', 'C', 'D')`
- Operations: *in the following operations, we assume* `attr1 = Attrs('A', 'B', 'C')` *and* `attr2 = Attrs('B', 'C', 'D')`.
  - `attr1 | attr2 == Attrs('A', 'B', 'C', 'D')`: *union*
  - `attr1 & attr2 == Attrs('B', 'C')` : *intersection*
  - `attr1 - attr2 == Attrs('A')` : *set difference*
- Relational: *in the following operations, we assume* `attr1 = Attrs('A', 'B', 'C')`, `attr2 = Attrs('B', 'C', 'D')`, *and* `attr = Attrs('A', 'B', 'C', 'D')`.
  - `(attr1 <= attr)  == True` : *subset or equal to*
  - `(attr1 < attr2)  == False`: *proper subset*
  - `(attr2 >= attr)  == True` : *superset or equal to*
  - `(attr2 > attr1)  == False`: *proper superset*
  - `(attr2 == attr1) == False`: *equal to*
  - `(attr2 != attr1) == True` : *not equal to*

  Note the following properties:

  - `not (a1 < a2)` is not equal to `a1 >= a2`
  - `not (a1 <= a2)` is not equal to `a1 > a2`
  - `not (a1 > a2)` is not equal to `a1 <= a2`
  - `not (a1 >= a2)` is not equal to `a1 < a2`
  - `not (a1 == a2)` is ***equal*** to `a1 != a2`
  - `not (a1 != a2)` is ***equal*** to `a1 == a2`
- We also provide the following operations that *abuses* Python operator overloading. We assume `attr = Attrs('A', 'B', 'C')`.
  - Non-Empty Powerset.
    * `+attr == [Attrs('A'), Attrs('B'), Attrs('C'), Attrs('A', 'B'), Attrs('A', 'C'), Attrs('B', 'C'), Attrs('A', 'B', 'C')]`
    * The order may be different.
  - Iteration.

```python
attr = Attrs('A', 'B', 'C')
for a in attr:
  print(a) # print 'A', 'B', 'C' but may be in different order
```

Rel

This class encapsulates a relation. The operations are similar to `Attrs` except that we have relation name. The name of the relation will follow the left operand.

FD

This class encapsulates a functional dependency. We can think of a functional dependency

`{A, B} -> {C, D}`

as a nested tuple

`(('A', 'B'), ('C', 'D'))`

We will call this the "key" comparison following the Python convention.

- Instantiation: `fd = FD(Attrs('A', 'B'), Attrs('C', 'D'))` to represent $AB \rightarrow CD$.
- Properties:
  - `fd.src == Attrs('A', 'B')`: *the source*
  - `fd.dst == Attrs('C', 'D')`: *the target/destination*
- Relational: *but note that less/greater than are not really that useful*
  - `fd1 == fd2`: *equal to*
  - `fd1 != fd2`: *not equal to*
  - `fd1 <= fd2`: *"key" comparison less than or equal to*
  - `fd1 <  fd2`: *"key" comparison less than*
  - `fd1 >= fd2`: *"key" comparison greater than or equal to*
  - `fd1 >  fd2`: *"key" comparison greater than*

Sigma

This class encapsulates a set of functional dependency.

- Instantiation:
  - `s0 = Sigma(FD(Attrs('A', 'B'), Attrs('C', 'D')))`
  - `s1 = Sigma(FD(Attrs('A', 'B'), Attrs('C')))`
  - `s2 = Sigma(FD(Attrs('A', 'B'), Attrs('D')))`
- Operations: *similar to set operations*
  - `s1 | s2`: *union*
  - `s1 & s2`: *intersection*
  - `s1 - s2`: *set difference*
- Relational: *set relational operation*
- We can also perform *iteration* (*using* `for`*-loop*) to get individual FDs.

Utilities

The above classes will be used as a basis for all the implementation. Please get yourself familiar with it. But since it is tedious to create those, we have provided some functions to simplify the their creation using `str` (*i.e.*, string). These functions are available in `Util.py`.

- `A(s)`: *instantiates* `Attrs`.
  - `s` is a `str` of all attributes without comma (*e.g.*, `A('ABC')`).
  - The attributes will be converted to *uppercase*.
- `R(s)`: *instantiates* `Rel`.
  - `s` is a `str` of the following format `'R(ABCD)'` (*e.g.*, `R('R(ABCD)')`).
    - `R` is the relation name.
    - `ABCD` is the attributes.
  - The attributes will be converted to *uppercase*.
- `F(s)`: *instantiates* `FD`.
  - `s` is a `str` of the following format `'AB -> CD'` (*e.g.*, `F('AB -> CD')`).
    - All whitespaces will be removed.
    - Must be separated with an "arrow" (*i.e.*, `'->'`).
  - The attributes will be converted to *uppercase*.
- `S(s)`: *instantiates* `Sigma`.
  - `s` is a `str` of the following format `'fd1; fd2; ...'` (*e.g.*, `S('AB -> CD; AB -> C; AB -> D')`).
    - All whitespaces will be removed.
    - FDs must be separated with a "semicolon" (*i.e.*, `';'`).
  - The attributes will be converted to *uppercase*.

Algorithm

As an illustration of how to use the classes, we provide the core algorithm for the course, namely the *attribute closure* algorithm. This algorithm compute the attribute closure of a given set of attributes with respect to a particular set of functional dependencies.

Pseudo-Code

Input: a set of attributes α, a set of functional dependencies Σ.

Output: a set of attributes that corresponds to $(\alpha)^+$ (*the attribute closure of* α).

1. Let the initial result $\theta = \alpha$ (*a copy of it*).
2. While (*there is an FD* $\beta \rightarrow \gamma$ *such that* $(\beta \subseteq \theta) \wedge (\gamma \nsubseteq \theta)$) then:
   - Add γ to θ (*i.e.*, $\theta := \theta \cup \gamma$).
3. Return θ.

Code

The translation from pseudo-code to code is by using *fix-point* algorithm. The essence of a fix-point algorithm is we look at the result from each iteration and if there is no changed in between, we stop.

We do this by recording the previous iteration result (*i.e.*, `prv`) and compute the current result (*i.e.*, `res`). Then we check if `prv != res` and stop if they are equal (*i.e.*, `prv == res`). Alternatively, we continue the `while`-loop if `prv != res` is still `True`.

```
# Attribute closure with respect to a set of FD sigma
# input
#   - attrs: Attrs
#   - sigma: Sigma
# output
#   - res: Attrs
def attribute_closure(attrs, sigma):
  res = attrs.copy()
  prv = Attrs()

  # this uses a fix-point algorithm
  while prv != res:
    prv = res.copy()
    for fd in sigma:                            # iteration
      if fd.src <= res and not(fd.dst <= res):  # subset
        res = res | fd.dst                      # union
  return res
```

Current Task

There will be **NO** solution given to the following tasks. We would highly recommend you try to implement them to have a better understanding of the algorithm covered in functional dependencies and normalization (*e.g.*, BCNF and 3NF).

1. Superkey algorithm `superkeys(rel, sigma)`.
   - `rel` is a relation `Rel`.
   - `sigma` is a set of FDs `Sigma`.
   - The output is a set of attributes `Attrs` corresponding to the superkeys of `rel` with respect to `Sigma`.
2. Key algorithm `keys(rel, sigma)`.
   - `rel` is a relation `Rel`.
   - `sigma` is a set of FDs `Sigma`.
   - The output is a set of attributes `Attrs` corresponding to the keys of `rel` with respect to `Sigma`.
3. Prime attribute algorithm `prime_attributes(rel, sigma)`.
   - `rel` is a relation `Rel`.
   - `sigma` is a set of FDs `Sigma`.
   - The output is a set of attributes `Attrs` corresponding to the keys of `rel` with respect to `Sigma`.

Future Task

1. FD projection algorithm `projection(attrs, sigma)`.
   - `attrs` is a set of attributes `Attrs`.
   - `sigma` is a set of FDs `Sigma`.
   - The output is a set of FD which is a projection of `sigma` on `attrs`.
2. BCNF check algorithm `check_bcnf(rel, r1, sigma)`.
   - `rel` is a relation `Rel`.
   - `r1` is a relation `Rel`.
   - `sigma` is a set of FDs `Sigma`.
   - The output is either `None` if there is no violation, or an FD `fd` corresponding to the violation.
     - Basically check if `r1` is in BCNF with respect to `sigma`, given that `r1` is a *decomposed schema* from the original relation `rel`.
3. 3NF check algorithm `check_3nf(rel, r1, sigma)`.
   - `rel` is a relation `Rel`.
   - `r1` is a relation `Rel`.
   - `sigma` is a set of FDs `Sigma`.
   - The output is either `None` if there is no violation, or an FD `fd` corresponding to the violation.
     - Basically check if `r1` is in 3NF with respect to `sigma`, given that `r1` is a *decomposed schema* from the original relation `rel`.
4. Lossless-join decomposition check algorithm `is_lossless(rel, rn, sigma)`.
   - `rel` is a relation `Rel`.
   - `rn` is a set of relation `Rel`.
   - `sigma` is a set of FDs `Sigma`.
   - The output `True` if the decomposition of `rel` to all `ri` in `rn` is a lossless-join decomposition, otherwise return `False`.
5. Dependency-preserving decomposition check algorithm `is_preserving(rel, rn, sigma)`.
   - `rel` is a relation `Rel`.
   - `rn` is a set of relation `Rel`.
   - `sigma` is a set of FDs `Sigma`.
   - The output `True` if the decomposition of `rel` to all `ri` in `rn` is a dependency-preserving decomposition, otherwise return `False`.

6. BCNF decomposition algorithm `decompose_bcnf(rel, sigma)`.
    - `rel` is a relation `Rel`.
    - `sigma` is a set of FDs `Sigma`.
    - The output is a set of relations corresponding to the BCNF decomposition of `rel` with respect to `sigma` such that:
        - The decomposition is a *lossless-join* decomposition.
        - Basically, CS2102 BCNF decomposition algorithm.
7. 3NF decomposition algorithm `decompose_3nf(rel, sigma)`.
    - `rel` is a relation `Rel`.
    - `sigma` is a set of FDs `Sigma`.
    - The output is a set of relations corresponding to the 3NF decomposition of `rel` with respect to `sigma` such that:
        - The decomposition is a *lossless-join* decomposition.
        - The decomposition is a *dependency-preserving* decomposition.
        - Basically, CS2102 3NF decomposition (*i.e.*, synthesis) algorithm.

Note

Since you are not allowed computers for final assessment, this exercise is only useful potentially for assignments. However, we believe that the exercise will allow you to have a better understanding of the problem. You may even double-check all the answers on past year paper.