Assignment 2 Coursework3 Report

The report details the implementation of the two distinct parts of the assignment – database and program design. Database was designed based on the instructions provided and subset of multiomics data from 'Personal aging markers and ageotypes revealed by deep longitudinal profiling' (*https://www.nature.com/articles/s41591-019-0719-5*).

Part1: Database design

Database Context

The study investigated the multiomics of 106 individuals. If known, the characteristics of each subjects were noted, such as their race, sex, SSPG level, VMI, age and whether they were insulin resistant (IR) or sensitive (IS). Each Subject also got an unique SubjectID (*Figure 1*: "*Subject*" entity). Every subject included in the study has to donate a sample, hence every sample had to be linked to one subject.

Each subject donated a sample during their visit, identified with a visit ID. The sample got a unique ID, consisting of the corresponding SubjectID and VisitID (*Figure 1*: "*Sample*" entity). The transcriptome, proteome and metabolome were explored in the sample (*Figure 1*: "*Transcriptome*", "*Proteome*", "Peak" entities, respectively). The abundance of each transcript, protein and metabolite ID was noted per sample in the corresponding -omics table. Not every sample had to be examined for a specific omics, but every entry in the omics table had to be matched with one sample.

Additionally, a table with metabolites, their KEGG, HMDB, chemical class and pathways was provided (*Figure 1*: "Annotation" entity). Due to the nature of the metabolomics measurement, each identified peak could correspond to more than one metabolite. If the record was present in the table, it has to have at least one metabolite name and one peak ID ie. those entries could not have been left empty.

Object and relationship identification

The chosen object for the database were matched with real-life objects: Subject, Sample, Peak (Metabolome), Transcriptome, Proteome, Annotation. The relationships between entites were as follows:

- Each Subject had a one-to-many relationship with Sample – each subject donates at least one sample. VisitID constitutes an attribute of that relationship, as it characterizes each instance when the sample is donated.
- Each Sample is involved in a binary one-to-one relationship with Peak, Transcriptome and Proteome – each Sample has only one omics profile.
- Each Peak is involved in a many-to-many relationship with Annotation – each is matched with at least one metabolite.

Many-to-many relationship of Annotation and Peak is showed in a separate table PeakAnnotated.

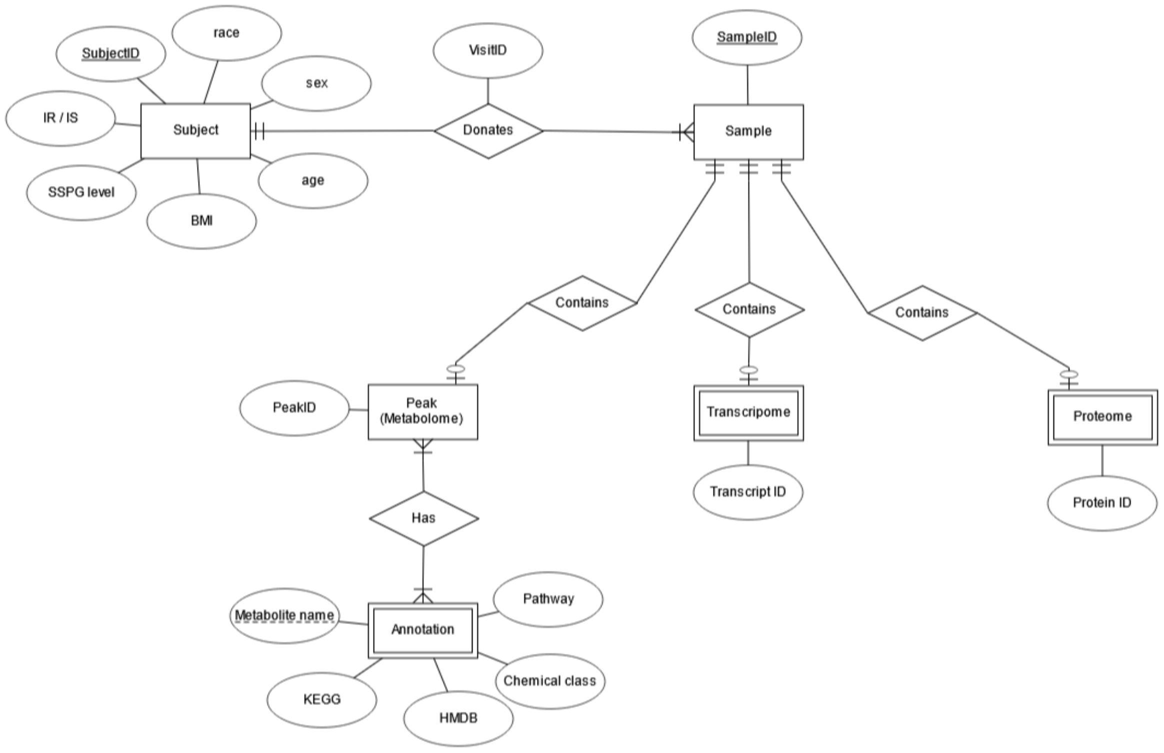

Figure 1. An Entity Relationship diagram for the database. Each Subject entity has an unique SubjectID, and optional attributes. Each Subject is linked to with their Sample and their relationship is characterised by a VisitID. Samples are identified by an unique SampleID. Each Sample is linked to at most one Peak (Metabolome), Transcriptome and Proteome entities. Each peak in the metabolome table gets assigned an unique PeakID. Each peak is matched with one or more metabolites, denoted in the Annotation table. Each TranscriptID and ProteinID corresponds to a transcript/protein name from the table headers and are attributes of the Transcriptome/Proteome. Samples Crow's foot notation, made with ERDPlus.

Primary keys of entities

All the unique attributes are Primary Keys of the table:

- SubjectID identifies Subject,
- SampleID identifies Sample,
- PeakID identifies Peak
- PeakID with Metabolite name together identify Annotation (composite primary key)

Tables in the database

The tables created based on the ERD in Figure 1 are shown in Figure 1a. However, due to the limitations of the assignment, not all the columns were loaded (Figure 2b, explained further in *part 7b* of the report).

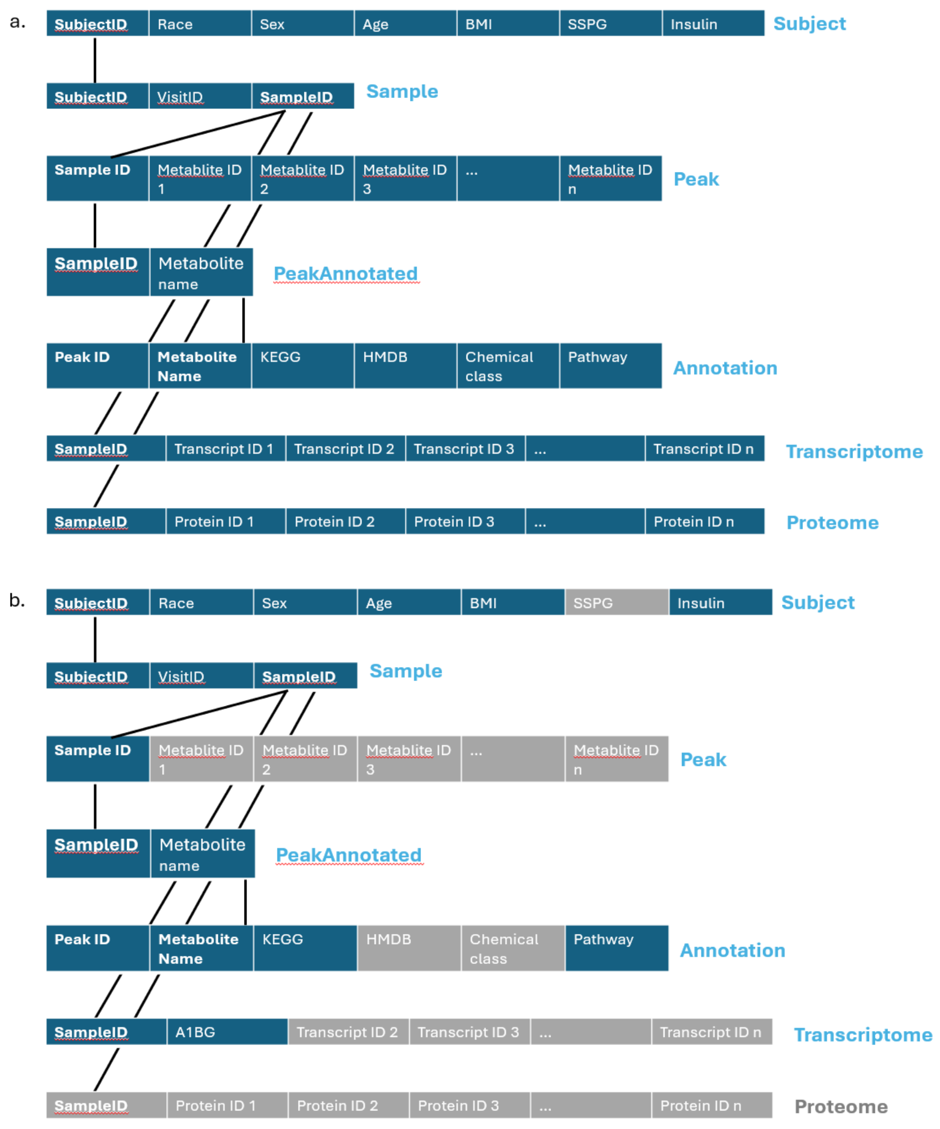

Figure 2. Tables within the schema of the database. The names of the tables are denoted next to the tables, with unaccessed columns marked by grey background in Fig.2b. The Foreign Keys are marked with black, solid lines.

Part2: Program design

The entirety of the code was wrote in Python 3.12.8 in Visual Studio Code.

1. Code organisation

a. File organisation

The main part of the program resides in the file "CW3032206_main.py". Program can be fully run from this script, as long as the supporting files are located in the same directory. The file "CW3032206_database_methods.py" encompasses the class DataManager with methods to create and manipulate the database. The remaining supporting files correspond to the accessed entities within the schema (Annotation, Peak, Sample, Subject, Transcriptome). The main python file also contains less in-depth functions, that increase the readability of the code, and functions for answering queries from Part B of the assignment.

b. Representing entities

Each assessed entity is represented by a class with attributes corresponding to the ERD on Figure 1. Each class has two methods within it – for gathering the data from the input file and for returning the attributes of an object as a list. The naming convention for the functions is, respectively: “gather_” + *name of the object;* “give_” + *name of the object*.

The purpose was mostly organisational – with a well-structured class, the code objects can be easily related to the EDR. Furthermore, object-oriented programming of the database allows for scaling up the code to load the entirety of information from the input files.

c. Representing relationships

The relationships are represented in tables that serve as input as columns with foreign keys. The attribute of the relationship between Subject and Sample – VisitID – had to be parsed from the SampleID to appear in the table. The parsing has been performed with *LoadData_SAMPLE()* function:

```
def LoadData_SAMPLE(SampleID_list):

        SampleID_UniqueList = Sample.make_unique(SampleID_list)

        sql = "INSERT INTO SAMPLE VALUES(?, ?, ?)"

        param = []

        for item in SampleID_UniqueList:

                Sample.SampleID = item

                item = item.split("-")

                Sample.SubjectID, Sample.VisitID = item

                row = [Sample.SubjectID, Sample.VisitID, Sample.SampleID]

                param.append(row)

        database.db_insertmany(sql, param)
```

d. Data normalisation and validation

Data acquired from the file is checked against the set criteria before loading into the database. Depending on the variable (shown in section 1), it has to be of a specific type, within a set range, precision or length. The *@getter* decorator within an object’s class ensures that the attributes assigned to every entity are fulfilling the type and value criteria. If they do not, a ValueError is raised. Such manipulation normalises inputs to the database and protects data integrity.

2. Acquiring data from source files

The source of the data were .csv and .tsv files, in which the entries are separated by commas and tabs, respectively. The functions of acquiring the data were included in the class of each entity. The input for the functions was provided in the main python file as a variable with input path and a *"with open(input) as source:"* statement preceding the *gather_* function. An example of the function gathering Subject data from the csv file is as follows:

```
def gather_subject(input):

        line = next(input)

        for line in input:

                list = ["NA", "Unknown", "unknown", "None", "NULL"]

                line = line.rstrip().split(",")

                new_line = [x if x not in list else None for x in line]

                SubjectID, Race, Sex, Age, BMI, SSPG, Insulin = new_line

                yield Subject(SubjectID, Race, Sex, Age, BMI, SSPG, Insulin)
```

The function assigns attributes to the entity objects. Subsequently, the *give_* function is used to return the object attributes as list, later used for data entry to the database.

a. Handling missing data

The missing information in the database could have been described in many ways, considered were options: "NA", "Unknown", "unknown", "None", "NULL". All were replaced by None, functioning in Python as an empty entry. After loading to the database, None is read by SQL as NULL. Conversions were implemented inside of the *gather_* functions inside of each entity class using list comprehensions.

```
def gather_subject(input):

        line = next(input)

                for line in input:

                        list = ["NA", "Unknown", "unknown", "None", "NULL"]

                        line = line.rstrip().split(",")

                        new_line = [x if x not in list else None for x in line]

                        SubjectID, Race, Sex, Age, BMI, SSPG, Insulin = new_line

                        yield Subject(SubjectID, Race, Sex, Age, BMI, SSPG, Insulin)
```

b. Handling duplicated metabolite names in Annotation

The repeated metabolite names in the Annotation table had a suffix of a numeral 1-5 in brackets. In the *gather_* function for the Annotation class, those were removed. Hence, the metabolite names were considered the same.

```
def gather_annotation(input):

        line = next(input)
```

```
for line in input:
    list = ["NA", "Unknown", "unknown", "None", "NULL"]
    line = line.rstrip().split(",")
    new_line = [x if x not in list else None for x in line]
    PeakID, MetaboliteName, KEGG, HMDB, ChemicalClass, Pathway = new_line
    MetaboliteName = MetaboliteName.replace("(1)", "").replace("(2)", "").replace("(3)",
    "").replace("(4)", "").replace("(5)", "")
    yield Annotation(PeakID, MetaboliteName, KEGG, HMDB, ChemicalClass, Pathway)
```

3. Methods for interfacing with the database

The methods needed to interact with the database are provided in the "CW3032206_database_methods.py" in the class DataManager. The class takes the database file (.db) as an input. Each function in the class requires the module sqlite3 and the creation of connection with the database and a cursor object.

a. Create database

The function *create_tables(self, entity)* creates one of the tables in the database schema of the name of the table as an input. Names of the tables are provided as the script iterates through a list in the main python file. Each names matches a specific sql query hard-coded in the function, that is then executed.

b. Input single or multiple rows to the database

Functions db_insert(self, sql, params) and db_insertmany(self, sql, params) take as an input a sql query and a list of parameters – rows to be inputted to the database. The number of columns parsed for each row must match the number of columns in the database. The first function inserts only one row, while the latter inserts many. The sql query uses a placeholder, a method favoured due to the increased protection from the sql query attack.

c. Query

Function db_query(self, sql) executes a sql query and returns it's results as a list or a tuple.

d. Verifying the presence of a table

Function db_istable(self, name) is used when creating tables, to verify whether a said table already exists. The required name stands for the name of the function. The result of the query is different than None if the table already exists.

4. Functions for the main implementation

a. Functions for inputting data from the file to the database

Due to the manipulation of the data unique to the specific tables, the functions are separate for each entity. The naming convention for the function is *LoadFunction_* +

Table name. The functions source the specific *gather_* and *give_* functions, as well as *db_insertmany().*

b. Functions for answering questions from part B of the assignment

Function define_query(question) takes the number of the query from the command line as an input and returns the sql command that is needed for that question.

Function answer_query(sql) performs the query using db_query() and returns the result as a formatted output with separated rows and columns.

5. Error handling: *logging*

The module ***logging*** was used to handle errors and warnings in a nicer fashion.

6. Command-line interface: *arg-parse*

In order to run the program from the command line, the **arg-parse** module is needed. Only one command line input is mandatory – the database file (.db), provided as the positional argument at the end. There are three optional arguments, provided as flags: "—createdb", "—loaddb", "—querydb". The presences of those strings in the command line determines whether the script runs database creation functions, loading or querying. Furthermore, the last flag stores it's value – a number of the query that is supposed to be run. An exemplary command line prompt is "python CW3032206_main.py --createdb –loaddb –querydb=2 assignment2.db", that would run the python script "CW3032206_main.py", creating, populating the database and performing the query number 2.

7. Implementation notes

a. Issue when re-running loading data into PeakAnnotated table

When loading data is re-run for an already created database, an unexpected "sqlite3.OperationalError" is raised that is not caused by an absence of the PeakAnnotated table, but by the database being locked. I have not managed to solve this problem before the submission deadline.

b. Note on database completion

In accordance with the instructions for the assignment, only data pertaining to the queries was loaded into the database. The backbone of the methods, classes and functions remains the same. However, due to the constraints on the command line input, the selection conditions had to be hard-coded into the script. In order for the classes to be universally used and for the database to be loaded fully, those conditions have to be erased from the script.

8. SQL commands

DDL CREATE:

"SUBJECT"

```
CREATE TABLE SUBJECT(
        SubjectID VARCHAR(255),

        Race VARCHAR(255),

        Sex VARCHAR(255),

        Age INT,

        BMI INT, SSPG INT,

        Insulin VARCHAR(255) CHECK (Insulin in ('IR','IS', NULL)),

        PRIMARY KEY (SubjectID)

        )
```

"SAMPLE"

```
CREATE TABLE SAMPLE(

            SubjectID VARCHAR(255),

            VisitID VARCHAR(255),

            SampleID VARCHAR(255),

            PRIMARY KEY (SampleID),

            FOREIGN KEY (SubjectID) REFERENCES SUBJECT(SubjectID)

        )
```

"PEAK"

```
CREATE TABLE PEAK(

        SampleID VARCHAR(255),

        metabolite1 INT,

        PRIMARY KEY (SampleID)

        )
```

"PeakAnnotated"

```
CREATE TABLE PeakAnnotated(

            SampleID VARCHAR(255),

            MetaboliteName VARCHAR(255),

            PRIMARY KEY (SampleID, MetaboliteName),

            FOREIGN KEY (SampleID) REFERENCES PEAK(SampleID),

            FOREIGN KEY (MetaboliteName) REFERENCES ANNOTATION(MetaboliteName)

        )
```

"ANNOTATION"

```
CREATE TABLE ANNOTATION(
        PeakID VARCHAR(255),
        MetaboliteName VARCHAR(255),
        KEGG VARCHAR(255),
        HMDB VARCHAR(255),
        ChemicalClass VARCHAR(255),
        Pathway VARCHAR(255),
        PRIMARY KEY (PeakID, MetaboliteName),
        FOREIGN KEY (PeakID) REFERENCES PEAK(PeakID)
     )
```

"TRANSCRIPTOME"

```
CREATE TABLE TRANSCRIPTOME(
     SampleID VARCHAR(255),
     A1BG VARCHAR(255),
     PRIMARY KEY (SampleID)
     )
```

QUERIES:

question == "1"

```
   SELECT SubjectID, Age
    FROM SUBJECT
    WHERE Age > 70
```

question == "2"

```
   SELECT SubjectID
    FROM SUBJECT
    WHERE Sex = "F"
    AND BMI BETWEEN 18.5 AND 24.9
    ORDER BY BMI DESC
```

question == "3"

```
   SELECT VisitID
```

```
      FROM SAMPLE

      WHERE SubjectID ="ZNQOVZV"
```

question == "4":

```
      SELECT DISTINCT SubjectID

      FROM SUBJECT

      WHERE Insulin = "IR" AND SubjectID IN (

      SELECT SubjectID

      FROM SAMPLE

      WHERE SAMPLE.SampleID IN (

      SELECT SampleID

      FROM PEAK

      WHERE PEAK.metabolite1 IS NOT NULL));

      """
```

question == "5"

```
     SELECT DISTINCT KEGG

      FROM ANNOTATION

      WHERE PeakID IN ("nHILIC_121.0505_3.5", "nHILIC_130.0872_6.3", "nHILIC_133.0506_2.3",
"nHILIC_133.0506_4.4")
```

question == "5a":

```
     SELECT DISTINCT KEGG

      FROM ANNOTATION

      WHERE PeakID = "nHILIC_121.0505_3.5"
```

question == "5b":

```
     SELECT DISTINCT KEGG

      FROM ANNOTATION

      WHERE PeakID = "nHILIC_130.0872_6.3"
```

question == "5c"

```
      sql = """SELECT DISTINCT KEGG

      FROM ANNOTATION
```

```
    WHERE PeakID = "nHILIC_133.0506_2.3"
```

question == "5d"

```
    sql = """SELECT DISTINCT KEGG
    FROM ANNOTATION
    WHERE PeakID = "nHILIC_133.0506_4.4"
```

question == "6":

```
   SELECT MIN(Age) as SmallestAge
    MAX(Age) as BiggestAge
    AVG(Age) as AverageAge
    FROM SUBJECT;
```

question == "7":

```
   SELECT COUNT(PeakID) AS AnnotationNumber, Pathway
    FROM ANNOTATION
    WHERE AnnotationNumber >=10
    GROUP BY Pathway
    ORDER BY AnnotationCount DESC"""
```

question == "8":

```
    sql = """SELECT MAX(A1BG)
    FROM TRANSCRIPT
    WHERE SampleID IN(
    SELECT SampleID
    FROM SAMPLE
    WHERE SubjectID = ZOZOW1T)
```

question == "9"

```
   SELECT Age, BMI
    FROM SUBJECT
    WHERE SubjectID NOT IN (
    SELECT SubjectID
    FROM SUBJECT
    WHERE Age IS NULL OR BMI IS NULL)
```