Activities 8: ORM and database queries

Table of contents

Coding activities 8: Working with classes and databases part 2

Theme: Using Python to work with data

This week returns to working with classes and databases and focuses on the SQLModel library as you will use this in COMP0034.

Note that for coursework 2 you can use SQLAlchemy rather than SQLModel if you prefer. SQLAlchemy arguably has a steeper learning curve, however some students may already have experience of this in which case use it. SQLModel uses SQLAlchemy.

You may not need all the insert, select, update and query methods to complete the coursework. You will likely need insert to add data in order to do the testing.

Instructions (8-0-instructions.md) are in the docs/8_classes_database_2 folder:

Next activity (8-01-methods.md)

1. Adding methods to classes

Python classes

A method is a function defined inside a class that operates on instances of that class (objects). It usually takes self as its first parameter, which refers to the instance calling the method.

You can add as many methods as you like.

An example of a method (`register_athlete`):

```
class ParalympicEvent:
    def __init__(self, name, sport, classification):
        self.name = name
        self.sport = sport
        self.classification = classification
        self.athletes = []

    def register_athlete(self, athlete_name):
        """ Register the athlete with the event

        Args:
            athlete_name: A string representing the name of the athlete
        """
        self.athletes.append(athlete_name)
```

To use the method on an instance of the class:

```
ev = ParalympicEvent(name="Boccia Pairs", sport="Boccia",
        classification="BC4")
ev.register_athlete(athlete_name="Alison Levine")
```

Static and class methods

Class methods

Class methods are used to access or modify class-level data.

Class methods start with the decorator `@classmethod`.

Their first parameter is `cls`. This refers to the class itself.

For example:

```
class ParalympicEvent:
    def __init__(self, name, sport, classification):
        self.name = name
        self.sport = sport
        self.classification = classification
        self.athletes = []

    @classmethod
    def from_dict(cls, data):
        """Creates an event from a dictionary"""
        return cls(data['name'], data['sport'], data['classification'])
```

Static methods

Static methods are utility functions that don't need access to class or instance data.

They use the decorator `@staticmethod`.

They do not require either `self` or `cls` as a parameter.

For example:

```
class ParalympicEvent:
    def __init__(self, name, sport, classification):
        self.name = name
        self.sport = sport
        self.classification = classification
        self.athletes = []

    @staticmethod
    def is_valid_classification(classification):
        """Checks if the classification is within a valid range"""
        return 1 <= classification <= 10
```

Dunder, double underscore, methods

Dunder methods are special methods in Python that begin and end with double underscores ``__. They let you define how your objects behave with built-in Python operations like printing, addition, iteration, and more.

Common dunder methods and their uses

- `__init__` Object initialization `obj = MyClass()`
- `__str__` String representation (for print) `print(obj)`
- `__repr__` Official string representation `repr(obj)`
- `__len__` Length of object `len(obj)`
- `__getitem__` Indexing `obj[key]`
- `__setitem__` Item assignment `obj[key] = value`
- `__eq__`, `__lt__`, Comparisons (==, , etc.) `obj1 == obj2`

- `__iter__`, `__next__` Iteration `for x in obj:` `__call__` Callable objects `obj()` `__enter__`, `__exit__` Context managers (with) `with obj:`

You saw examples of the `__init__` in week 5. You may have spotted examples of `__str__` or `__repr__` too where custom code has been used to override the default implementation.

For example:

```
class ParalympicEvent:
    """ Represents a Paralympic event """

    def __init__(self, name, sport, classification):
        self.name = name
        self.sport = sport
        self.classification = classification
        self.athletes = []

    def __str__(self):
        """User-friendly string representation"""
        return f"{self.name} ({self.sport}, Class
        {self.classification}) with {len(self.athletes)} athlete(s)"

    def __repr__(self):
        """Unambiguous representation for debugging"""
        return (f"ParalympicEvent(name='{self.name}',
        sport={self.sport}, "
                f"classification={self.classification},
        athletes={self.athletes})")
```

To use:

```
ev = ParalympicEvent(name="Boccia Pairs", sport="Boccia",
        classification="BC4")
ev.register_athlete(athlete_name="Alison Levine")
repr(ev)
print(ev)
```

SQLModel classes

The SQLModel tutorial only shows models with attributes. In most cases SQLModel is being used to map to database tables.

However, this is still a Python class so all the above method types can be applied.

When you inherit SQLModel in the class, you inherit its methods:

SQLModel inherits from Pydantic's BaseModel giving: - `.dict()` – Convert model to dictionary - `.json()` – Convert model to JSON string - `.parse_obj()` – Create model from a dictionary - `.copy()` – Create a copy of the model - `.validate()` – Validate data against the model

Activity: Add a method to a class in week 3

1. Open one of your SQLModel `models.py` classes from week 5.
2. Try adding a `__repr__` and a method of your choice to the class.
3. Create an instance of the class then call the methods on that object.

Next activity (8-02-relationships.md)

2. Relationships in SQLModel

One-to-many

You already know that to achieve this you add the primary key attribute from the parent table as a foreign key attribute to the child table.

To do this in SQLModel you define the foreign key field like this example from the SQLModel documentation:

```
from sqlmodel import Field, Session, SQLModel


class Team(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str = Field(index=True)
    headquarters: str


class Hero(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str = Field(index=True)
    secret_name: str
    age: int | None = Field(default=None, index=True)

    team_id: int | None = Field(default=None, foreign_key="team.id")
```

SQLModel (and SQLAlchemy) has an advantage over pure SQL as it allows you to navigate the relationship from both child to parent AND parent to child by defining a `Relationship()` attribute in both classes.

This is the updated example from the SQLModel documentation:

```
from sqlmodel import Field, Relationship, SQLModel


class Team(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str = Field(index=True)
    headquarters: str

    heroes: list["Hero"] = Relationship(back_populates="team")


class Hero(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str = Field(index=True)
    secret_name: str
    age: int | None = Field(default=None, index=True)

    team_id: int | None = Field(default=None, foreign_key="team.id")
    team: Team | None = Relationship(back_populates="heroes")
```

Notice that "Hero" has quote marks in `heroes: list["Hero"] = Relationship(back_populates="team")`. This is because Hero is defined after Team, so you would get an error warning in your IDE if you try to enter it as `list[Hero]`. Adding the `""` makes the interpreter see it as a string. More on this here (https://sqlmodel.tiangolo.com/tutorial/relationship-attributes/type-annotation-strings/#about-the-string-in-listhero).

When you have a relationship defined in this way, and you append an object to the List attribute, then when you add and commit, SQLModel makes sure that the changes to both tables are made. See SQLMOdel documentation (https://sqlmodel.tiangolo.com/tutorial/relationship-attributes/create-and-update-relationships/#include-relationship-objects-in-the-many-side).

Cascade delete/update

Updates are handled using the `back_populates` argument in the Relationship.

Cascade delete constraints on the relationships require more knowledge to implement in SQLModel. This is sign-posed for students who want to read further (https://sqlmodel.tiangolo.com/tutorial/relationship-attributes/cascade-delete-relationships/); but not covered in the course activities.

Many to many

As you saw in the normalisation activity in week 3, many-to-many relationships are resolved by creating a new table between the tables to join or link them.

The `Relationship()` can still be defined on the two tables that have the many-to-many, note the extra parameter `link_model`.

The example from the SQLModel documentation:

```
from sqlmodel import Field, Relationship, SQLModel


class HeroTeamLink(SQLModel, table=True):
    team_id: int | None = Field(default=None, foreign_key="team.id",
        primary_key=True)
    hero_id: int | None = Field(default=None, foreign_key="hero.id",
        primary_key=True)


class Team(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str = Field(index=True)
    headquarters: str

    heroes: list["Hero"] = Relationship(back_populates="teams",
        link_model=HeroTeamLink)


class Hero(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    name: str = Field(index=True)
    secret_name: str
    age: int | None = Field(default=None, index=True)

    teams: list[Team] = Relationship(back_populates="heroes",
        link_model=HeroTeamLink)
```

Activity: Review the relationships for the student database

1. Open models.py (../../src/activities/starter/db_wk8/models.py)
2. Find the example of a one-to-many relationship between course and location
3. Find the example of a many-to-many relationship between course, teacher and student in the enrollment table - this has a 3-way link table!

Next activity (8-03-insert.md)

3. Insert data into a single table

Recap from week 5

Activity 5.7 (../5_classes_orm/5-06-sqlmodel-add-data.md) introduced how to add data to a single table.

Inserting data into the paralympics database is complex and beyond what you would need in your coursework so this activity returns to the student database example.

To recap, the SQL syntax to insert is like:

```
INSERT INTO tablename (colname1, colname2)
VALUES ("Something", 123.5);
```

You can insert values for all columns, or only named columns.

When using this with sqlite3 you can use parameterised queries to insert one or more values from variables.

The SQLModel equivalents are `add()` (https://sqlmodel.tiangolo.com/tutorial/insert/) for a single row and `add_all()` for multiple rows.

The steps when using SQLModel:

```
from sqlmodel import Session, create_engine
# 1. Create the engine which creates the connection to the database
sqlite_file_name = "database.db"
sqlite_url = f"sqlite:///{sqlite_file_name}"
engine = create_engine(sqlite_url, echo=True)

# 2. Create one or more rows to add
teacher = Teacher(name="Freida", email="freida@myorg.com")

# 3. Create a session
with Session(engine) as session:
    # 4. Add the rows (or rows if add_all)
    session.add(teacher)
    # 5. Commit the new row
    session.commit()
```

Activity: Generate the student database

The `app.py`, `database.py` and `models.py` from week 5 have been modified for the student database in activities/starter/db_wk8 (../../src/activities/starter/db_wk8).

1. Copy the files to your own package as you will edit the files. Check the imports are correct, you may need to update these.
2. Create the database using the code in `app.py`

Add data to a table without relations

src/activities/data/student_data.csv (../../src/activities/data/student_data.csv) has the raw data.

It has these fields:
`teacher_name,teacher_email,student_name,student_email,course_name,course_code,course_sche`

The class in models.py for the teacher has this structure:

```
class Teacher(SQLModel, table=True):
    id: int | None = Field(default=None, primary_key=True)
    teacher_name: str
    teacher_email: str
```

To insert values into the database you could create an object for each row as there are only a few rows; and then add and commit.

```
teacher = Teacher(teacher_name="Ani Sarana",
        teacher_email="as@school.com")
engine = database.engine

with Session(engine) as session:
    session.add(teacher)
    session.commit()
```

In practice, you will have many rows.

There are other ways to read data from a .csv file, however as you know pandas DataFrame then use that.

You can then iterate the rows and use the '.to_dict()' method of the Teacher class that is inherited from pydantic to get the values for each row.

```
import pandas as pd


def add_teacher_data():
    data_path = resources.files(data).joinpath("student_data.csv")
    cols = ["teacher_name", "teacher_email"]
    df = pd.read_csv(data_path, usecols=cols)

    # 2. Convert DataFrame rows to SQLModel instances
    teachers = []
    for _, row in df.iterrows():
        record = models.Teacher(**row.to_dict())
        teachers.append(record)

    # 3. Insert into database
    with Session(engine) as session:
        session.add_all(teachers)
        session.commit()
```

Activity: Add the teacher data to the database

Edit the starter files to add a teacher to the database.

The SQLModel documentation examples add this method to `app.py`. I added it to `database.py`. You choose where you think is most appropriate.

Next activity (8-04-insert-multiple.md)

4. Insert data into related tables

If you completed the exercise in week 3 to add data to the related tables you may remember the steps:

1. Add rows of data to the table with no relationships
2. Query to find the `id` of the parent row, then add the related child row in the other table using this `id` as the foreign key

The same approach is taken here to add the students, courses, teachers, locations and enrollments:

```
def add_all_data():
    """Adds data from CSV to each table using pandas DataFrame to
        filter the data. """
    df = pd.read_csv(data_path)

    # Find the unique location rows then create Location objects from
        these
    locations = df["course_location"].unique()
    loc_objects = []
    for loc in locations:
        location = Location(room=loc)
        loc_objects.append(location)

    # Find the unique student rows then create Student objects from
        these
    rows = df.drop_duplicates(subset=['student_name'])
    stu_objects = []
    for _, row in rows.iterrows():
        student = Student(student_name=row["student_name"],
        student_email=row["student_email"])
        stu_objects.append(student)

    # Find the unique teacher rows then create Teacher objects from
        these
    # You may have already added teacher data, in which case exclude
        this section
    rows = df.drop_duplicates(subset=['teacher_name'])
    teacher_objects = []
    for _, row in rows.iterrows():
        teacher = Teacher(teacher_name=row.teacher_name,
        teacher_email=row.teacher_email)
        teacher_objects.append(teacher)

    # Find the unique coutse rows then create Course objects from these
    rows = df.drop_duplicates(subset=['course_name'])
    course_objects = []
    for _, row in rows.iterrows():
        course = Course(course_name=row.course_name, course_code=row.course_code,
        course_schedule=row.course_schedule)
        course_objects.append(course)

    with Session(engine) as session:
        # Add the objects to the individual tables. Note there are no
        primary or foreign key values at this stage.
        # Once the objects are added to the database, the primary key
        value will be created
        session.add_all(loc_objects)
        session.add_all(stu_objects)
        session.add_all(teacher_objects)
        session.add_all(course_objects)
        session.commit()


        # Create and add the enrollment objects and add the location FK
        to the courses
```

```
for _, row in df.iterrows():
    # Find the ids of the rows
    location = session.exec(select(Location).where(Location.room
== row["course_location"])).first()
    s_id = session.exec(select(Student.id).where(Student.student_email
== row["student_email"])).first()
    c_id = session.exec(select(Course.id).where(Course.course_code
== row["course_code"])).first()
    t_id = session.exec(select(Teacher.id).where(Teacher.teacher_email
== row["teacher_email"])).first()
    # Update the course with the location using the
relationship attribute
    course.location = location
    # Create the new enrollment for the row
    enrollment = Enrollment(student_id=s_id, course_id=c_id,
teacher_id=t_id)
    session.add_all([course, enrollment])
    session.commit()
```

Note that there are other ways to do the above, I used this approach as it is consistent with code you have seen in the course to date.

Activity: Add data to the database

1. Add a function to add all the data. You could add this to `app.py` as shown in the SQLModel tutorial. I created a new module `queries.py` instead for this and the following activities.

2. Use/adapt the code above to insert all the rows.

3. Run app.py to create the database and add the rows.

Note: If you run app.py repeatedly the rows will be appended to the tables. You could drop all the tables e.g., using `SQLModel.metadata.drop_all(engine)`; or you could add queries to check if a table is empty before adding rows.

Next activity (8-05-select.md)

5. Select rows

In activity 3.14 the SQL select was introduced:

```
SELECT column_names
FROM table_name
WHERE some_condition
ORDER BY column_name -- ASC or DESC
LIMIT 1 -- or OFFSET
```

WHERE, ORDER BY and LIMIT/OFFSET are optional.

This activity introduces the SQLModel equivalent, `select()` (https://sqlmodel.tiangolo.com/tutorial/select)

Select (read) rows with SQLModel

As for the previous activities, you first create a Session(). This has been covered so is omitted here.

1. Create a `select` statement e.g. `statement = select(Games)`

   Often you want to select based on a condition using WHERE, e.g.: `WHERE name = 'Clare'`, `WHERE score > 25`, etc.

   In SQLModel append `.where()` to `select()`: `select(Hero).where(Hero.name == "Dave")` NB: uses `==` **not** `=`

   You can limit the number of results returned using LIMIT using `.limit()`, e.g. `select(Hero).where(Hero.name == "Dave").limit(2)` to get the first 2 results.

   Offset works the same way, so to skip the first 2 results: `select(Hero).offset(2)`

2. Execute the statement and capture the results in a variable.

3. Access the results so that you can use them in your code. The results may be one row, multiple rows, None, or an error.

   Options include:

   - `.first()` The first result of the query, or None if no results are found.
   - `.all()` A list of all results that match the query.
   - `.one_or_none()` One result or None, but raises an error if more than one result is found.
   - `.fetchmany(size=N)` A list of up to N results from the query.

Activity: select rows from the students database

Add code to select and print the results:

1. Select teacher where teacher_name == "Mark Taylor"
2. Select the names only for all the students

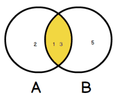
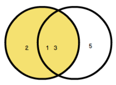
Select with related tables

To select rows from tables that are related, you need to join the tables together.

SQL uses JOIN and specifies the common column between the tables to join on e.g.,

```
SELECT hero.id, hero.name, team.name
FROM hero
         JOIN team
              ON hero.team_id = team.id
```

When using join, consider how you want to join the tables. SQLite supports:

Join type	***Venn diagram***	***Description***
INNER JOIN (https://www.sqlitetutorial.net/sqlite-inner-join/)		Selects all rows from both tables to appear in the result if and only if both tables meet the conditions specified in the ON clause.
LEFT JOIN (https://www.sqlitetutorial.net/sqlite-left-join/)		Select results include: Rows in table A (left table) that have corresponding rows in table B. Rows in the table A table and the rows in the table B filled with NULL values in case the row from table A does not have any corresponding rows in table B.
CROSS JOIN (https://www.sqlitetutorial.net/sqlite-cross-join/)		Produces a Cartesian product of two tables; multiplying each row of the first table with all rows in the second table if no condition introduced with CROSS JOIN.

RIGHT OUTER JOIN and FULL OUTER JOIN are not supported in SQLite.

SQLModel syntax varies depending on what result you want from the join. Some examples:

1. Select all Heros and their Teams: `statement = select(Hero, Team).where(Hero.team_id == Team.id)` gets the Hero and Team in the result for each Hero. Uses inner join.
2. Select all Heros and their Teams: `statement = select(Hero, Team).join(Team)` does the same as 1!
3. Select all Heros where they are in the team 'A': `statement = select(Hero).join(Team).where(Team.name == "Preventers")`. This is used where you want to find only the Hero attributes but based on a condition that is only in the associated table. Uses inner join.
4. Select all Heros and their Team even if they don't have a team (i.e. is null): `statement = select(Hero, Team).join(Team, isouter=True)` This is a LEFT OUTER JOIN.

You can join multiple tables together so long as there are keys that relate them. For example, find the Teachers who teach Physics:

```
select(Teacher.teacher_name).join(Enrollment).join(Course).where(Course.course_name
== "Physics")
```

Activity: Select data from joined tables

Add code to select and print the results:

1. Select all Students with their name and email that are enrolled on the Physics course, order by name descending.
2. Select all Courses that Student with id 1 is enrolled in.

Next activity (8-06-delete.md)

6. Delete rows

SQL

The SQL syntax for deleting rows is:

```
DELETE
FROM tablename
WHERE condition
```

Executing DELETE in SQL without specifying a condition will delete all rows from a table!

Delete one or more rows

The SQLModel equivalent of SQL DELETE is: `session.delete()`

To use this, you first locate the row or rows to be deleted using `select()` and then pass the results to `delete()`.

An example with the statements from the documentation:

```
with Session(engine) as session:
    # select the row
    statement = select(Hero).where(Hero.name == "Spider-Youngster")
    results = session.exec(statement)
    # create the object
    hero = results.first()
    # Delete and commit
    session.delete(hero)
    session.commit()
```

Delete from tables with relationships

If you did not set any constraints for 'ON DELETE' in the classes, then using SQL no via SQLModel by default SQL takes no action on the related records when the parent is deleted.

According to the SQLModel documentation the default is to SET NULL.

By setting `cascade_delete=True` in a Relationship(), SQLModel automatically deletes the related records when the parent one is deleted.

Example from the SQLModel documentation:

```
heroes: list["Hero"] = Relationship(back_populates="team",
cascade_delete=True)
```

If you also intend to interact with database directly, outside SQLModel, then `ondelete` must also be configured in the Field().

```
team_id: int | None = Field(default=None, foreign_key="team.id",
ondelete="CASCADE")
```

For more details, refer to the cascade delete relationships documentation (https://sqlmodel.tiangolo.com/tutorial/relationship-attributes/cascade-delete-relationships/#cascade-delete-relationships).

Activity: Delete rows

Add code to delete the following. Print before and after to show the results. Print the Enrollment table as well.

1. Delete the Teacher with name “John Smith”
2. Delete the Enrollment for Student with student_id 1 from Course with Course_id 1

Next activity (8-07-update.md)

7. Update rows

To update one or more columns in a row the SQL syntax looks like this:

```
UPDATE hero
SET age=16
WHERE name = "Spider-Boy"
```

In SQLModel you select the row, or rows, as objects, update the attributes and commit the changes.

Example from the SQLModel documentation (https://sqlmodel.tiangolo.com/tutorial/update):

```
def update_heroes():
    with Session(engine) as session:
        statement = select(Hero).where(Hero.name == "Spider-Boy")
        results = session.exec(statement)
        hero = results.one()
        hero.age = 16
        session.add(hero)
        session.commit()
```

Activity: Update students rows

Add code to update records, print the row before and after to check the results:

1. Update the course code for the Mathematics course to MATH102
2. update all teacher email addresses with the new domain @newschool.com. For this you also need Python string .replace() `teacher_email.replace("@school.com", "@newschool.com")`

Next activity (8-08-quality.md)

8. Code quality

Code quality was covered in week 4.

This activity is a prompt to remind you.

Docstrings and exception handling have been omitted from the activities this week to keep focus on the queries but should be considered when you write your coursework code.

Some SQLModel specific considerations:

1. Structure - refer to SQLModel specific guidance (https://sqlmodel.tiangolo.com/tutorial/code-structure/?h=structure)

2. Docstrings for classes - refer to the guidance for the style you are using e.g. Google style (https://google.github.io/styleguide/pyguide.html#384-classes)

3. Linting

   When you create the database you import models, yet the import is not explicitly used in the Python code. The linter will warn about this. It is an example of a situation you can ignore.

   Try `flake8 --max-line-length=100 src/activities/starter/db_wk8/database.py` in the terminal pane.

4. Exception handling in SQLModel

   SQLModel extends pydantic and SQLAlchemy so you would be handling Exceptions from these packages. Some of the common ones you might experience:

   - pydantic: ValidationError
   - sqlalchemy: IntegrityError, SQLAlchemyError (base class for sqlalchemy errors), OperationalError, see documentation (https://docs.sqlalchemy.org/en/20/core/exceptions.html)

   Have a look at the following tutorials on Exception handling for these:

   - Deal with Common Types of SQLAlchemy Exceptions for Running SQL Queries in Python (https://plainenglish.io/blog/deal-with-common-types-of-sqlalchemy-exceptions-for-running-sql-queries-in-python-9ec8db)
   - Pydantic Error Handling (https://docs.pydantic.dev/latest/errors/errors/)

Activity: Add exception handling

There are no tutor solutions to this activity.

Uncomment and run each of the functions in main in error_code.py (../../src/activities/starter/db_wk8/error_code.py).

Identify the exceptions raised then try to handle them.

Logging errors

A complementary technique to exception handling is to log errors as well to help with debugging. This is not taught until COMP0034 though you may want to investigate if you have time.

Optional: Paralympics database queries with SQLModel

Activity updated 27/11/25 In the tutorials we will demonstrate the use of SQLModel for a few of the queries.

Activity 1: SQLModel queries for the paralympics database

The code is in src/activities/starter/db_wk8/paralympics (../../src/activities/starter/db_wk8/paralympics)

```
db_wk8_demo
    |_ app.py              # Runs the query code
    |_ database.py         # Creates the database
    |_ models.py           # SQLModel classes for the paralympics data
    |_ query_service.py    # Class that has the queries
    |_ data
          |_ paralympics.db  # Version of the database for this
activity, created using the models
```

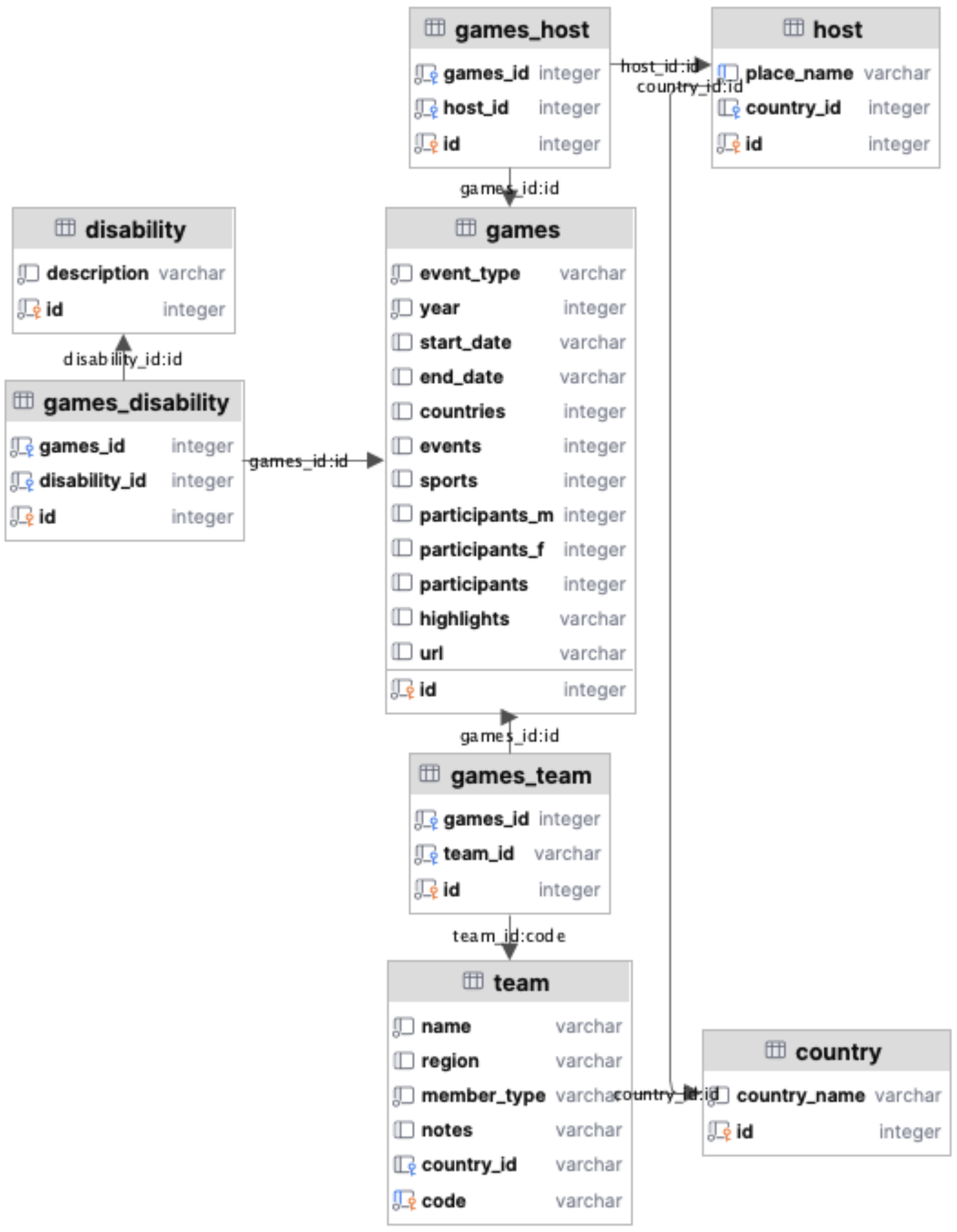
The tutorial demo will cover:

- adding relationships to a SQLModel class
- adding methods to a SQLModel class
- basic CRUD queries for a single table
- select queries with joins

The queries are:

1. List all Paralympics (games) with their year and type.
2. List all winter Paralympics (games) with host name(s) and year.
3. Find all disabilities recorded in the database.
4. Get all Paralympics (Games) that took place after the year 2000.
5. Find all teams from a specific region (e.g. Oceania).
6. List all hosts located in a specific country (e.g., 'Italy') and the year they held the Paralympics.
7. Show all Paralympics (games) along with their host city and host country.
8. List all disabilities associated with each Paralympics (games).
9. Find all teams that participated in a specific Paralympics (game) (e.g., Tokyo 2016).
10. Find all the Paralympics that have competitors who are 'Amputees'
11. Update all instances of the disability 'Les Autres' to 'Other'

The database structure is as follows:

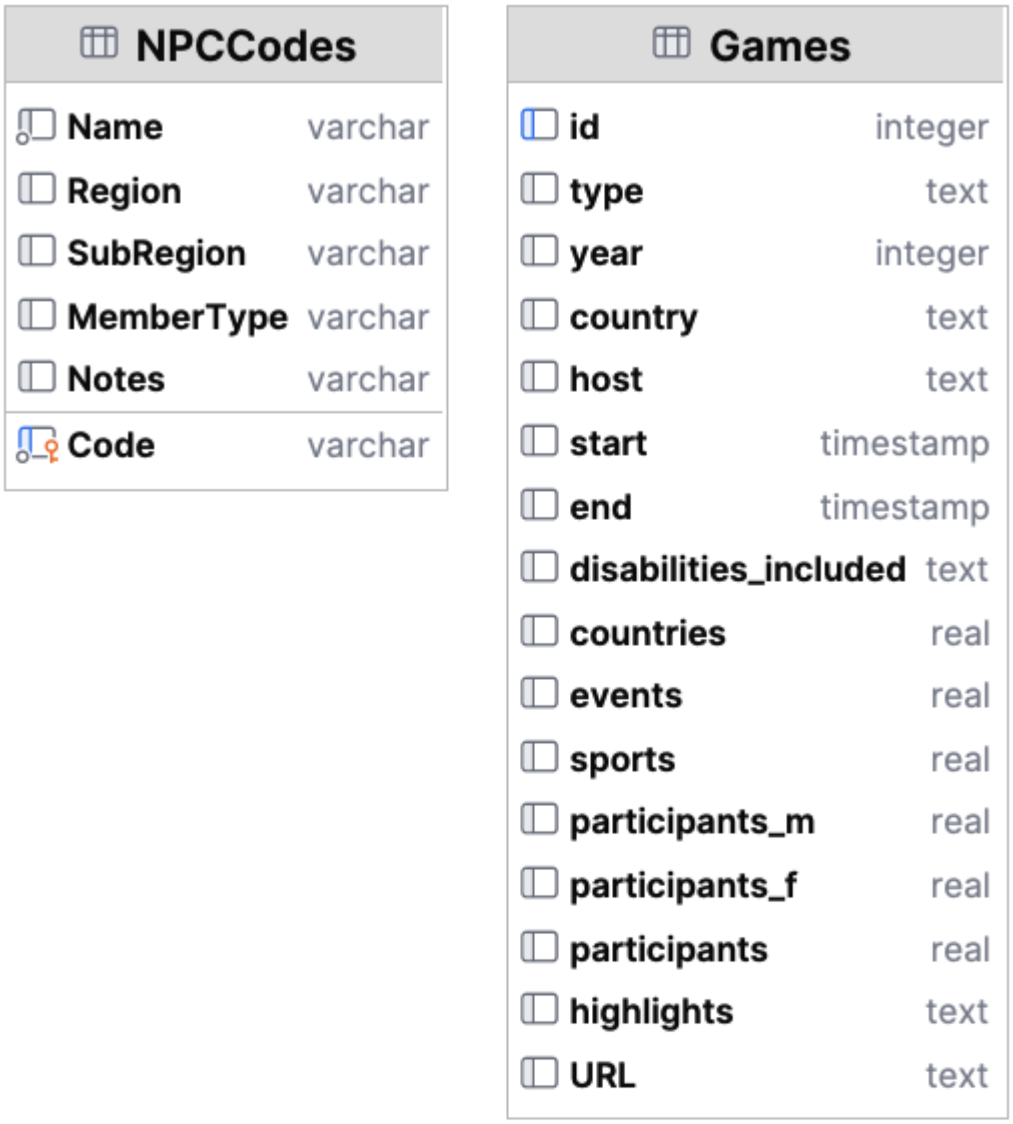


Activity 2: Differences between more/less normalised databases

This activity uses an extreme version of the paralympics database that has not been normalised to allow you to experience the differences in query complexity.

Using a database that has not been normalised **not recommended**, please do not do this in your coursework. Normalisation is important to reduce redundancy and increase data integrity.

The database structure is as follows:



NPCCodes	
Name	varchar
Region	varchar
SubRegion	varchar
MemberType	varchar
Notes	varchar
Code	varchar

Games	
id	integer
type	text
year	integer
country	text
host	text
start	timestamp
end	timestamp
disabilities_included	text
countries	real
events	real
sports	real
participants_m	real
participants_f	real
participants	real
highlights	text
URL	text

Queries

Write the same queries, this time for the database that has not been normalised. Try to do at least one update query, and one select query that has joins, to experience the differences. Reflect on the implications for query complexity and data integrity.

There is starter code in query_comparison.py (../../src/activities/starter/db_wk8/comparison/queries.py)

NB: Normalisation aims to reduce redundancy and increase data integrity. Select queries in this exercise may seem quicker and simpler using the database that has not been normalised, however, using a database that is not normalised is **not recommended**, please do not do this in your coursework.