Activities 4: Database design

Table of contents

Activities 4: Relational database design and creation

Themes: Designing applications & Using Python to work with data

Setup

Pre-requisites

1. Have forked the COMP0035 tutorials repository, cloned it to your computer, and set up a project with a virtual environment within your IDE (VS Code or PyCharm).
2. Updated the forked repo in your GitHub account see (/docs/2_pandas/2-0-instructions.md)

New: Enable mermaid support

The activities use a tool called Mermaid (https://mermaid.js.org/syntax/entityRelationshipDiagram.html) to define and display database diagrams. To view the diagrams in the markdown you need to have this tool installed.

For PyCharm, Go to PyCharm | Settings | Plugins then search for the Mermaid plugin (https://plugins.jetbrains.com/plugin/20146-mermaid).

For VS Code, go to Code | Settings | Extensions and find the Mermaid chart extension (https://marketplace.visualstudio.com/publishers/MermaidChart).

Complete the activities

There are many activities this week, less in weeks 4 and 5. You can spread out the activities over future weeks.

Activity instructions are in the docs/3_database folder, including:

1. Introduction to database design (lecture recap) (4-01-database-design.md)
2. Introduction to ERD (lecture recap) and ERD drawing tools (4-02-erd-intro.md)
3. Conceptual database design (4-03-conceptual-design.md)
4. Logical design to 1NF (4-04-logical-design-1nf.md)
5. Logical design to 2NF (4-05-logical-design-2nf.md)
6. Logical design to 3NF (4-06-logical-design-3nf.md)
7. Logical design - constraints (4-07-logical-design-constraints-data.md)
8. Logical design - referential integrity (4-08-logical-design-constraints-fk.md)
9. Logical design activity (4-09-logical-design-activity.md)
10. Physical design - SQLite schema (4-10-physical-design-structure.md)
11. Physical design - Python to create SQLite database structure (4-11-physical-design-create-db.md)
12. Database design - next steps (4-16-next-steps.md)
13. SQL INSERT and SELECT intro (4-12-sql-add-data.md)
14. Add data to tables with no FK (4-13-insert-no-fk.md)
15. Select data (4-14-select-query.md)
16. Add data to tables with an FK (4-15-insert-with-fk.md)
17. *Optional* Normalisation and application code (4-17-normalisation-tradeoff.md)- A brief intro to the trade-off between database normalisation and query design

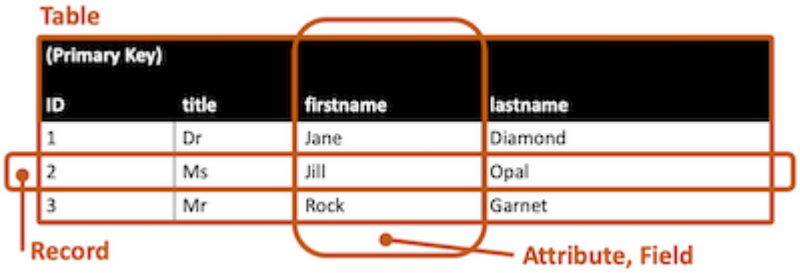
1. Introduction to database design (recap)

This is a brief recap of the lecture material.

Relational database

A **relational database** is a method of structuring data as **tables** associated to each other by shared attributes:

- a row corresponds to a unit of data called a record
- a column corresponds to an attribute (or field) of that record
- a table contains these rows and columns



Relational databases typically use **Structured Query Language (SQL)** to define, manage, and search data.

Relational database design

Database design is the organization of data according to a database model. The designer determines what data must be stored and how the data elements interrelate.

Connolly & Begg describe three stages of database design, summarised as:

1. Conceptual database design
    - Goal: Create a high-level data model that captures the data requirements.
    - Output: An Entity-Relationship (ER) model.
    - Focus: What data should be stored, and the relationships between data items.
2. Logical database design
    - Goal: Translate the conceptual model into a logical model that can be implemented in a specific type of database (e.g., relational).
    - Output: A relational schema with tables, attributes, primary/foreign keys, and constraints. Includes normalization to reduce redundancy and improve integrity. Can also be drawn as an ERD.
    - Focus: How the data will be structured logically, independent of any specific database management system (DBMS).
3. Physical database design

    Goal: Optimize the logical model for performance and storage on a specific DBMS. Output: A physical schema detailing file structure, indexing, and more. The full definition in Connolly & Begg's book goes beyond what is covered in this course. Focus: How the data will be stored and accessed efficiently. **For COMP0035, physical design is interpreted as the creation of the schema in an SQLite database.**

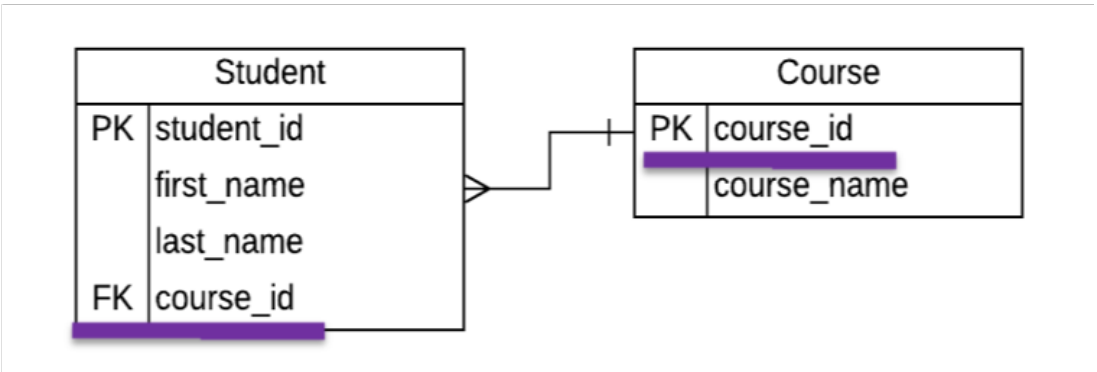
Normalisation

Normalisation is the process of structuring data into tables in a way that avoids update anomalies; and reduces data redundancy and the resulting data storage.

Update anomalies can occur when the same value is in multiple tables and needs to be updated or deleted in both; issues can occur if not all occurrences are identified and updated.

The process of normalisation divides larger tables into smaller tables and links them using relationships between key attributes.

- **Primary key (PK)**: a column (or combination of columns) guaranteed to be unique for each record and cannot be NULL
- **Foreign key (FK)**: a column in table A storing a primary key value from table B



0-1. Keys

A course is taken by many students, each student is enrolled on only one course

There are different levels of normalisation. It is not always desired, or necessary, to achieve 6th normal form, for the purposes of this course (and many applications) 3rd normal form is typically sufficient.

This course focuses on the general principles of normalization rather than the formal relational theory (refer to either of the database textbooks in the reading list if you wish to understand this topic in more detail).

Design your database so that:

- Each column/row intersection has only one entry.
- Each row and column in a table is unique.
- Each table has a primary key (a unique identifier).
- Tables with relationships are linked by primary/foreign key relationships.
- If you delete a record you don't also lose a value that may still be needed.
- Where there is a many-to-many relationship you will need to add a new table between the two tables so forming two one-to-many relationships.

Next activity (4-02-erd-intro.md)

2. Entity Relationship Diagram (ERD) - recap

This is a brief recap of the lecture material.

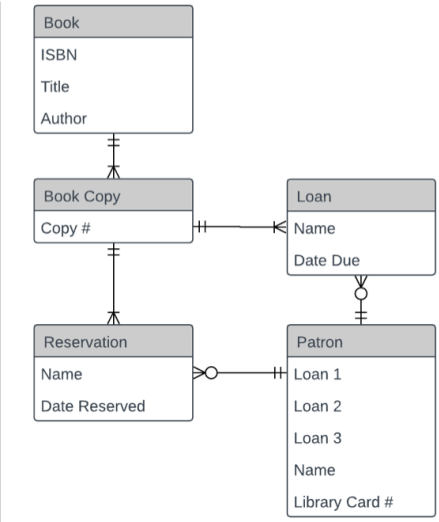
An **ERD** is a structural diagram used in database design. An ERD is a visual representation of the relationships between entities in a database. It's an important tool in database design and helps to illustrate how data is interconnected.

ERDs are used to plan and design databases, ensuring that the structure is logical and efficient. They help database designers and developers understand the data requirements and relationships.

It contains different symbols and connectors that represent:

- **Entities**: These are objects or concepts that can have data stored about them. For example, in a university database, entities might include Student, Course, and Professor.
- **Attributes**: These are the details or properties of an entity. For instance, a Student entity might have attributes like StudentID, Name, and DateOfBirth.
- **Relationships**: These show how entities are related to each other. For example, a Student might enroll in a Course, or a Professor might teach a Course.
- **Cardinality**: This indicates the number of instances of one entity that can be associated with instances of another entity. Common cardinalities include one-to-one, one-to-many, and many-to-many.

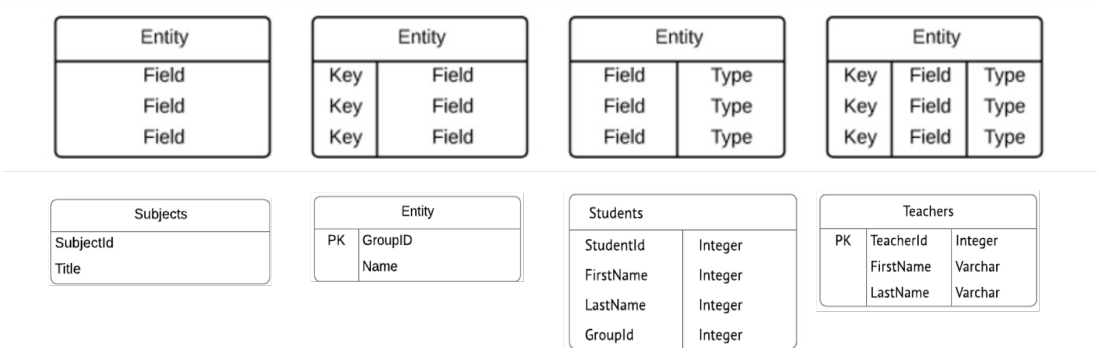
An example:



0-1. ERD

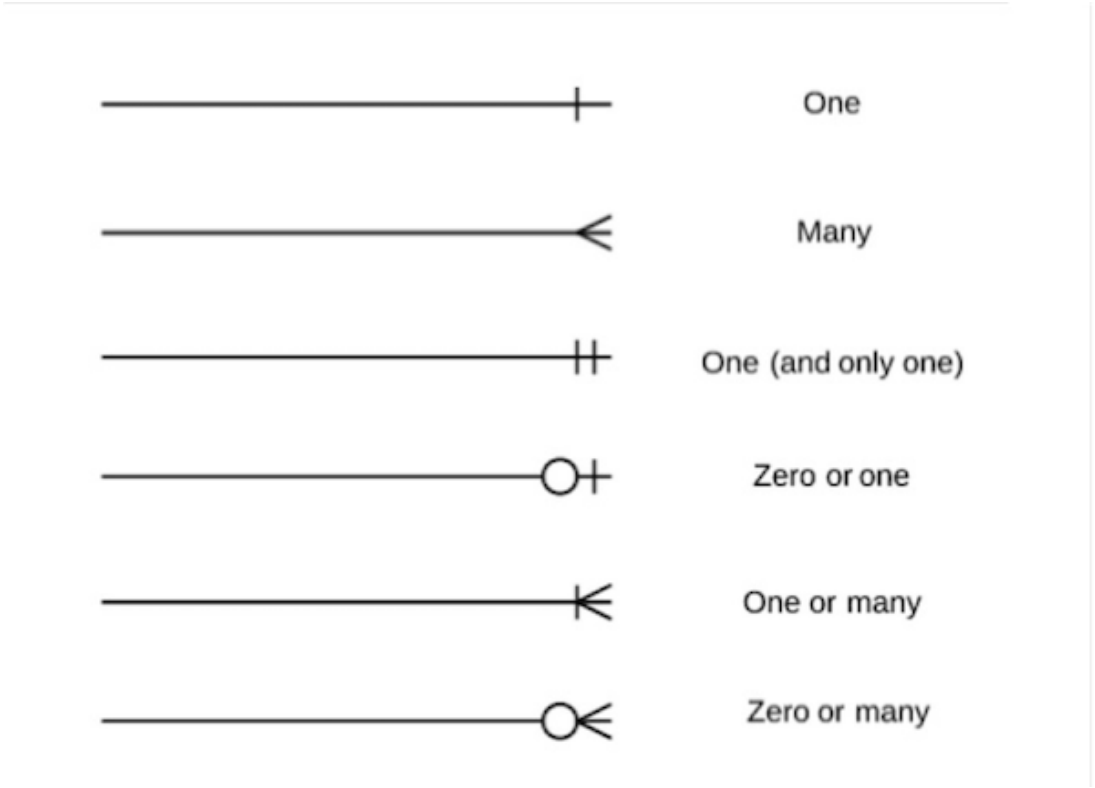
As the ERD is refined, further detail can be added to it. The ERD diagram can be used at different stages of the database design process.

The following diagram illustrates entities and attributes at different levels of detail:



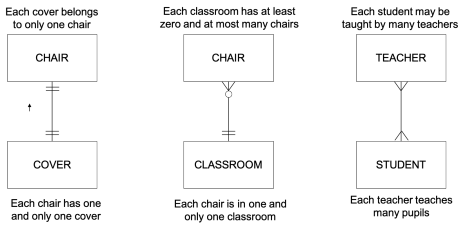
0-1. ERD entity symbols with differing level of detail

The relationships between entities can be represented with Crow's Foot notation, or other notations such as Chen's notation.



0-1. Crow's foot notation

Entity relationships may be one-to-one, one-to-many, many-to-many, these are shown as:



0-1. Relationship examples

Relational databases do not typically implement many-to-many relationships. During the process of normalisation:

- one-to-one: Tables are associated through the primary key in each table
- one-to-many: The primary key attribute in the one must be listed as an additional attribute (foreign key) in the many. The tables are associated by the similar attributes.

- many-to-many: A new table is always created and the primary key attributes of the original tables are made attributes of the new table. These are often combined to form a composite primary key – or a new surrogate primary key is created.

ERD drawing tools

At the time of writing the following were free online tools to use but may require you to create an account.

- draw.io (https://app.diagrams.net)
- Visual Paradigm online (https://online.visual-paradigm.com/diagrams/templates/entity-relationship-diagram/)
- Diagrams.net (https://app.diagrams.net/)
- LucidChart (https://lucid.app/pricing/lucidchart?anonId=0.87353863184e1c9485e&sessionDate=2022-12-05T10%3A16%3A15.712Z&sessionId=0.c6a8888f184e1c9485&createAccount) Free account is now limited to 3 diagrams; there is an ERD crow's foot template available.

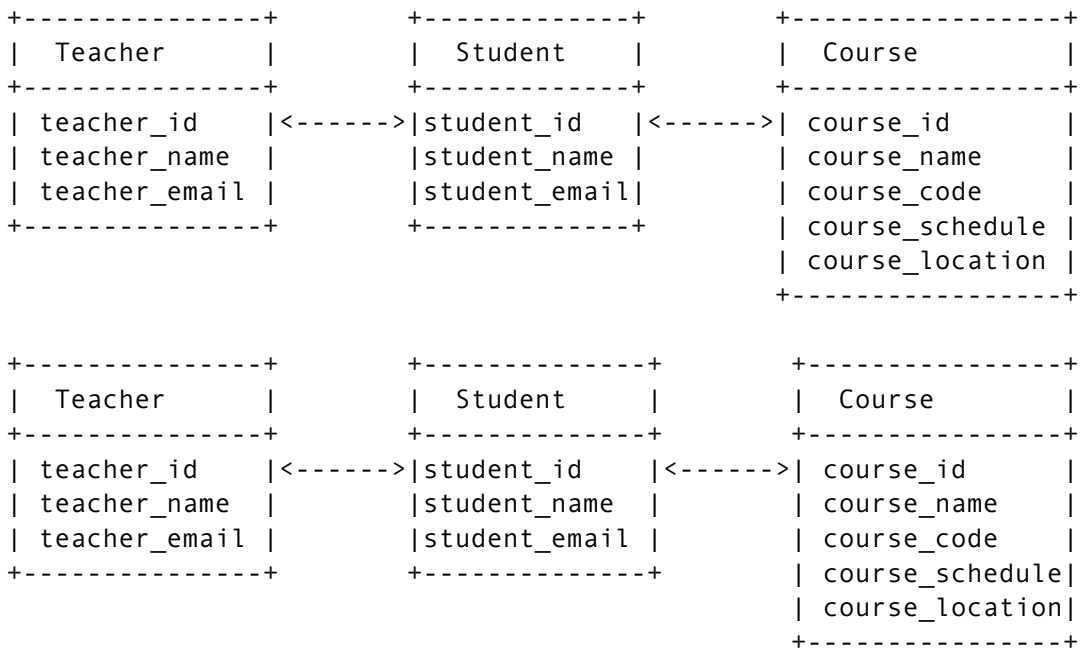
There are also coding tools, such as Mermaid (https://mermaid.js.org/intro/getting-started.html#_3-using-mermaid-plugins) (plugins for PyCharm and VS Code) that can be used to generate diagrams.

genAI and ERD generation

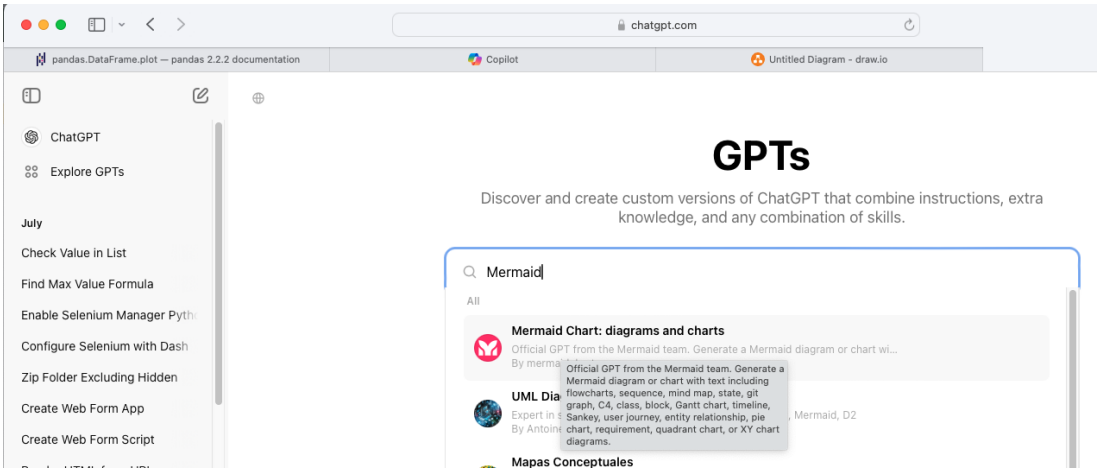
Tools use as chatGPT and CoPilot can give suggestions and draw diagrams based on an uploaded data file.

In copilot (https://copilot.microsoft.com) try entering the prompt `Draw an ERD for the data in the attached csv file` and attach the `student_data.csv` file to it.

Note that the format of the diagram below given by copilot *is not* a good enough format for the coursework as it does not follow an appropriate notation and lacks some of the necessary details.



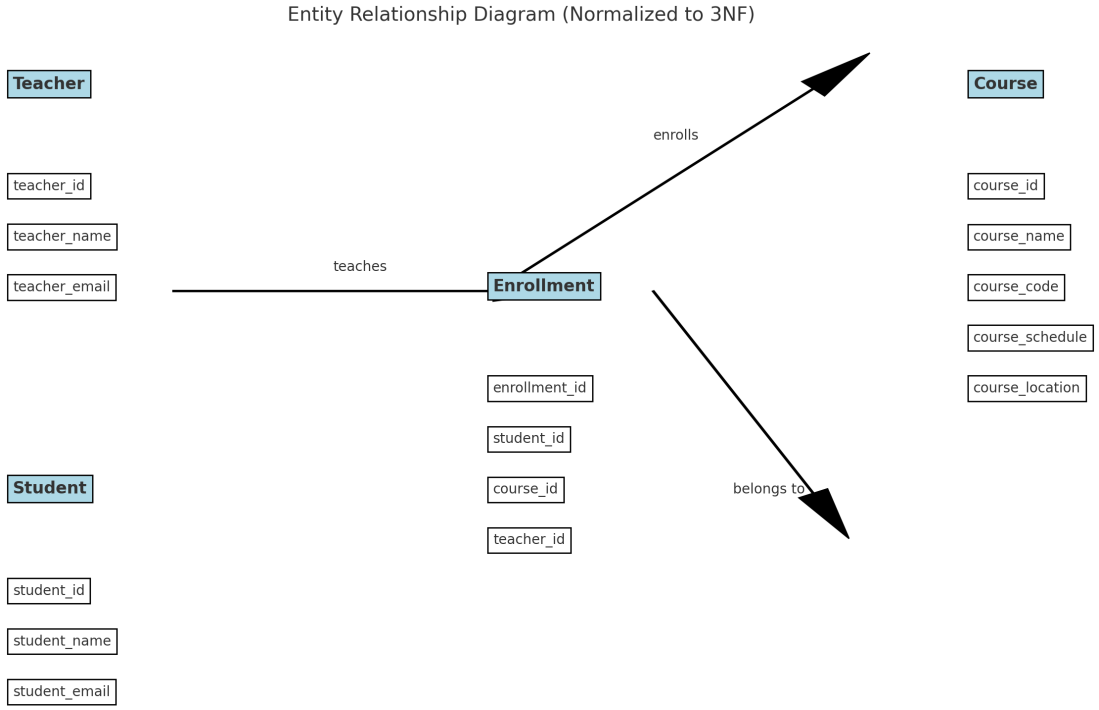
In chatGPT try going to 'Explore GPTs' and then search for Mermaid which is a diagramming tool (or Lucid). The free level of chatGPT only allows you a few prompts per day so if you've already used them for today then you won't be able to do this:



0-1. Find Mermaid GPT

In the Mermaid or Lucid GPT try `Draw an entity relationship diagram that is normalised to 3NF for the attached data`

On previous occasions this has returned the mermaid syntax for the ERD as well as a well formatted ERD. However on this occasion the diagram format is not good enough for the coursework as it does not use a recognised ERD notation.



0-1. Mermaid GPT drawn ERD

This gave the following response:

The provided data contains the following columns:

- record_id: A unique identifier for each record.
- teacher_name: The name of the teacher.
- teacher_email: The teacher's email.
- student_name: The name of the student.
- student_email: The student's email.
- course_name: The name of the course.
- course_code: The unique code for the course.
- course_schedule: The schedule of the course (days and time).

- course_location: The location where the course is held.

Analysis for Normalization:

- 1NF (First Normal Form): The data is in 1NF since all columns contain atomic values (no repeating groups or multivalued attributes).
- 2NF (Second Normal Form): To achieve 2NF, the table must first be in 1NF, and all non-key attributes should depend on the entire primary key. The current data appears to have a partial dependency (e.g., student details depending only on student name, teacher details on teacher name, etc.).
- 3NF (Third Normal Form): To achieve 3NF, the table must be in 2NF, and there should be no transitive dependency (i.e., no non-key attribute should depend on another non-key attribute).

Normalization Approach:

Entities:

- Teacher: teacher_id, teacher_name, teacher_email
- Student: student_id, student_name, student_email
- Course: course_id, course_name, course_code, course_schedule, course_location
- Enrollment: This links students to courses and teachers. enrollment_id, student_id, course_id, teacher_id

Let me now draw an ER diagram based on this normalized structure.

Here is the Entity Relationship (ER) Diagram, normalized to Third Normal Form (3NF). It includes four entities:

- Teacher: Attributes are teacher_id, teacher_name, and teacher_email.
- Student: Attributes are student_id, student_name, and student_email.
- Course: Attributes are course_id, course_name, course_code, course_schedule, and course_location.
- Enrollment: This serves as the relationship table linking students to courses and teachers. It includes enrollment_id, student_id, course_id, and teacher_id.
- Each entity contains only the relevant attributes, eliminating redundancies. The relationships between students, teachers, and courses are managed through the Enrollment entity.

Next activity (4-03-conceptual-design.md)

3. Conceptual database design

In this activity you will complete the conceptual design stage for a database based on the `paralympics_raw.csv` and `npc_codes` data files.

To view the mermaid code in this markdown file as diagrams in your IDE, you need to install the Mermaid plugin for your IDE (https://mermaid.js.org/intro/getting-started.html#_3-using-mermaid-plugins).

The conceptual design can be hand-drawn on paper; it does not have to be digital.

Conceptual design

Goal: Create a high-level data model that captures the data requirements.

Output: An Entity-Relationship (ER) model.

Focus: What data should be stored, and the relationships between data items.

Step 1 List the attributes

Refer to activity 2.02 pd.read_excel (../2_pandas/2-02-pandas-df.md) and activity 2.03 pandas.columns (../2_pandas/2-03-pandas-describe.md) for guidance.

You may want to create a new package folder in 'src/activities' for the database work e.g. 'src/activities/db'.

1. Create a python module e.g. database_design.py
2. You can either import and use the code in `from activities.starter.starter_db import read_raw_excel` or copy it to your own module. Use the code to:
    - Read the Excel version of the raw data file `data/paralympics_all_raw.xlsx`, create one dataframe for the `sheet_name="events"` and another for `sheet_name="country_codes"`
    - Print information about the data such as columns, datatypes and unique values of columns

A description the attributes

Brief description of the attributes for reference.

Games:

- 'type': Type of Paralympics (summer or winter)
- 'year': Year of the event
- 'country': Country where the event was held
- 'host': Host city
- 'start': Start date of the event
- 'end': End date of the event
- 'disabilities_included': A list of strings, the disability categories included in the games. One of: 'Spinal injury', ' Amputee', 'Vision Impairment' 'Les Autres', 'Cerebral Palsy', 'Intellectual Disability', 'Cerebral'
- 'countries': Number of participating countries
- 'events': Number of events within the event (e.g. Women's 400m Freestyle - S9, Men's 100m Butterfly - S14, etc.)
- 'sports': Number of sports included in the event

- ‘participants_m’: Number of male participants
- ‘participants_f’: Number of female participants
- ‘participants’: Total number of participants

Country code:

- ‘Code’: National Paralympic Committee (NPC) code for the competing team
- ‘Name’: Name of the country
- ‘Region’: NPC definitions of regions. One of: ‘Asia’, ‘Europe’, ‘Africa’, ‘America’, ‘Oceania’, or can be empty if member type is construct
- ‘SubRegion’: NPC definitions of subregions. One of: ‘South, South-East’, ‘North, South, West’, ‘North’, ‘West, Central’, ‘Carribean, Central’, ‘South’,‘East’, ‘Oceania’, ‘West’,‘Central, East’, or empty if MemberType is construct
- ‘MemberType’: Type of the competing team, one of: ‘country’, ‘team’, ‘dissolved’, ‘construct’
- ‘Notes’: Any text notes on the record.

List potential entities

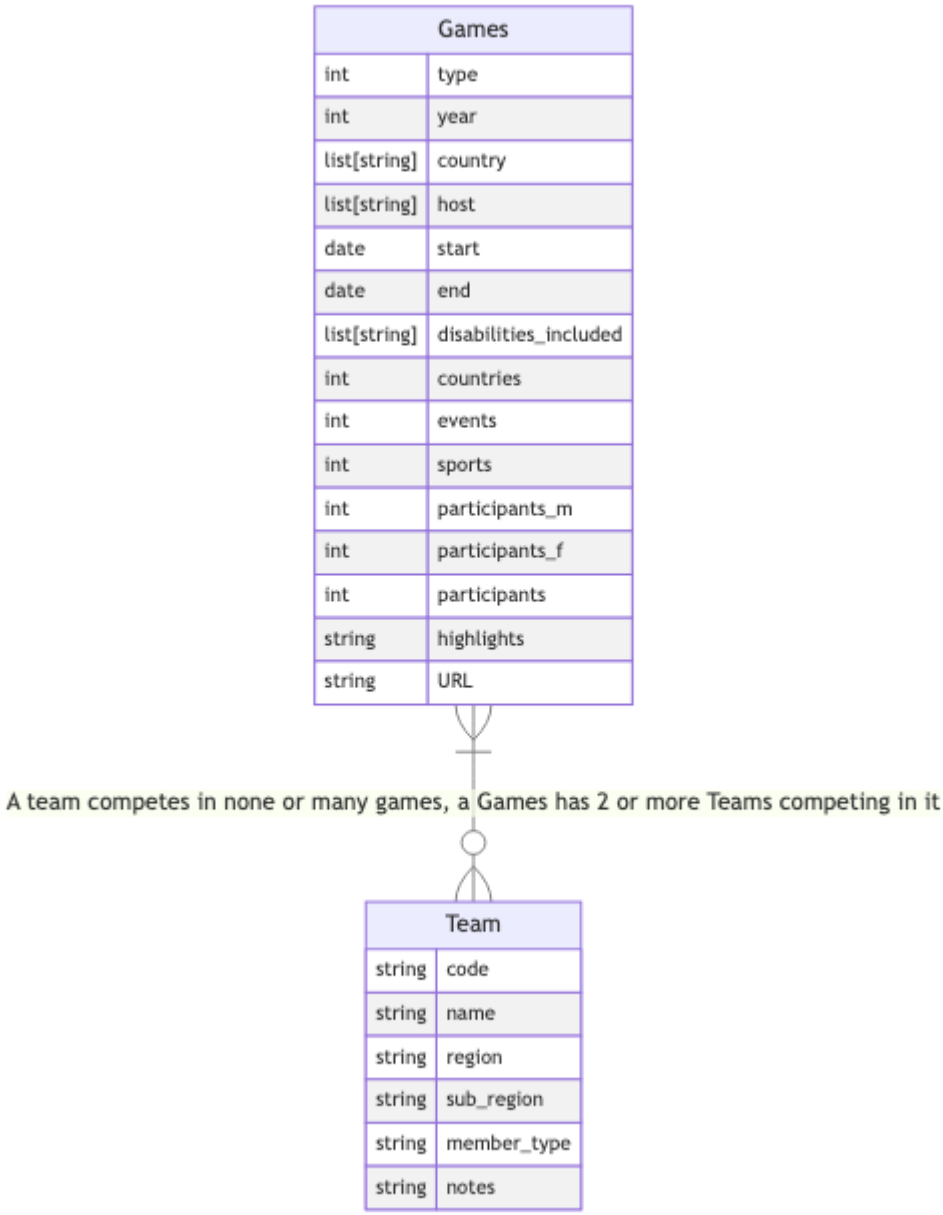
The data lists details about paralympic games. ‘ParalympicGames’, or ‘Games’, is an entity.

You also have details about paralympic country codes for each paralympics team. ‘Team’, is another entity.

There appears to be a relationship between Games host and Team given that they are linked by country name. However, the relationship is more accurately that one Games has many Teams competing in it, and a Team may compete in many Games.

You may have noticed that in 1984 there were two host cities in two countries: Stoke Mandeville, UK and New York, USA

The diagram below uses the Mermaid tool which draws ER diagrams in a given format. Other tools may use a different format. The detail you want to show is the entities, their attributes and any relationships between the entities.



This is now the conceptual design, it shows the entities, their attributes and any relationships between those entities.

Next activity (4-04-logical-design-1nf.md)

4. Logical database design to 1NF

This activity takes the conceptual database design and applies principles of normalisation to a level that is broadly 3NF.

Logical database design

- Goal: Translate the conceptual model into a logical model that can be implemented in a specific type of database (e.g., relational).
- Output: A relational schema with tables, attributes, primary/foreign keys, and constraints. Includes normalization to reduce redundancy and improve integrity. Can also be drawn as an ERD.
- Focus: How the data will be structured logically, independent of any specific database management system (DBMS).

Normalisation

Normalisation is a concept in database design aimed at ensuring that the data is stored in a structured and efficient manner. It aims to reduce redundancy (e.g. duplicated data) and improve integrity (e.g. avoid errors being introduced when data is added, updated or deleted).

As previously discussed, this module does not enforce a formal proof of normalisation. Instead, you are encouraged to consider the principles below and their impact on the design or your database.

You will be designing a database for use in a web application, and are often trade-off's between database efficiency and coding efficiency. A more complex database structure may be more efficient in terms of data redundancy and ensure greater data integrity; however it could make querying that data to use in an application more complex. This topic is not covered in the teaching material, you are encouraged to read wider than the course materials here to understand the implications for your later coursework in this module and the next.

The rest of this activity walks through applying the principles of the first the levels of normalisation.

First normal form (1NF)

A table is in 1NF if the intersection of every column and record contains only one value.

The key aspects of 1NF are:

- **Atomicity**: Every column contains only atomic (indivisible) values. This means that each field contains a single value, not a set of values or a list.
- **Uniqueness**: Each column must have a unique name.
- **No Repeating Groups**: There should be no repeating groups or arrays. Each row must be unique and identifiable by a **primary key**.

A **primary key (PK)** is an attribute (column) containing unique values so it can be used to uniquely identify a given row.

A **composite primary key** is where a combination of two or more columns are used to uniquely identify a row.

Review the contents of the conceptual design and the dataframes against the above checklist. Note any issues as these will need to be addressed, e.g.:

❌ Atomicity. Fail. The row for 'summer 1984' contains lists for host and city as the event was held in two locations. 'disabilities_included' also contains lists.

✅ Uniqueness. Pass. Each column has a unique name.

✅ No Repeating Groups: Pass. A row could be uniquely identified by a combination of the 'type' and 'year' attributes (if the New York/Stoke Mandeville event were split into two rows). When two or more attributes uniquely identify a row, this is called a **composite primary key**.

Activity: Identify primary keys (1NF)

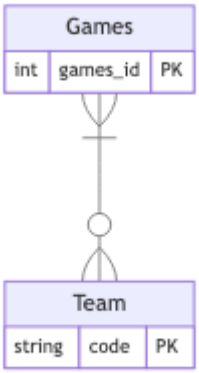
Each row in each table must be uniquely identified.

In Team, the three-letter code uniquely identifies the rows. The code is the **primary key** attribute.

In Games, there is no single unique attribute. A combination of 'type' and 'year' could be used as a **composite primary key**.

Integer primary keys are generally considered more efficient for indexing, and easier to refence in programme code than a composite key, so add a new attribute `games_id`, auto-incrementing integer i.e. starts at 1 and the database management system typically automatically increases it for each subsequent row.

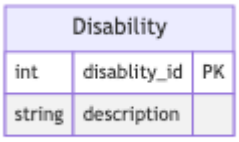
You now have these primary keys (other attributes removed for simplicity):



Activity: Create new tables to remove the list values (1NF)

The row for Stoke Mandeville and New York has lists in 'country' and 'host'. The 'disabilities_included' also contains lists. These break the principle of **atomicity** so must be resolved.

Create a new table for Disability. It should have a primary key and the string description of the disability e.g.



The relationship between Games and Disability can be described as a Games can provide events for many Disability categories; and each Disability category can be in many Games. This is a **many-to-many** relationship.

One solution to the 'country' and 'host' could be to split the data into two rows, one for Stoke Mandeville and one for New York. This is problematic as you do not have separate data on the events, participants, countries etc. for the two separate elements of the event.

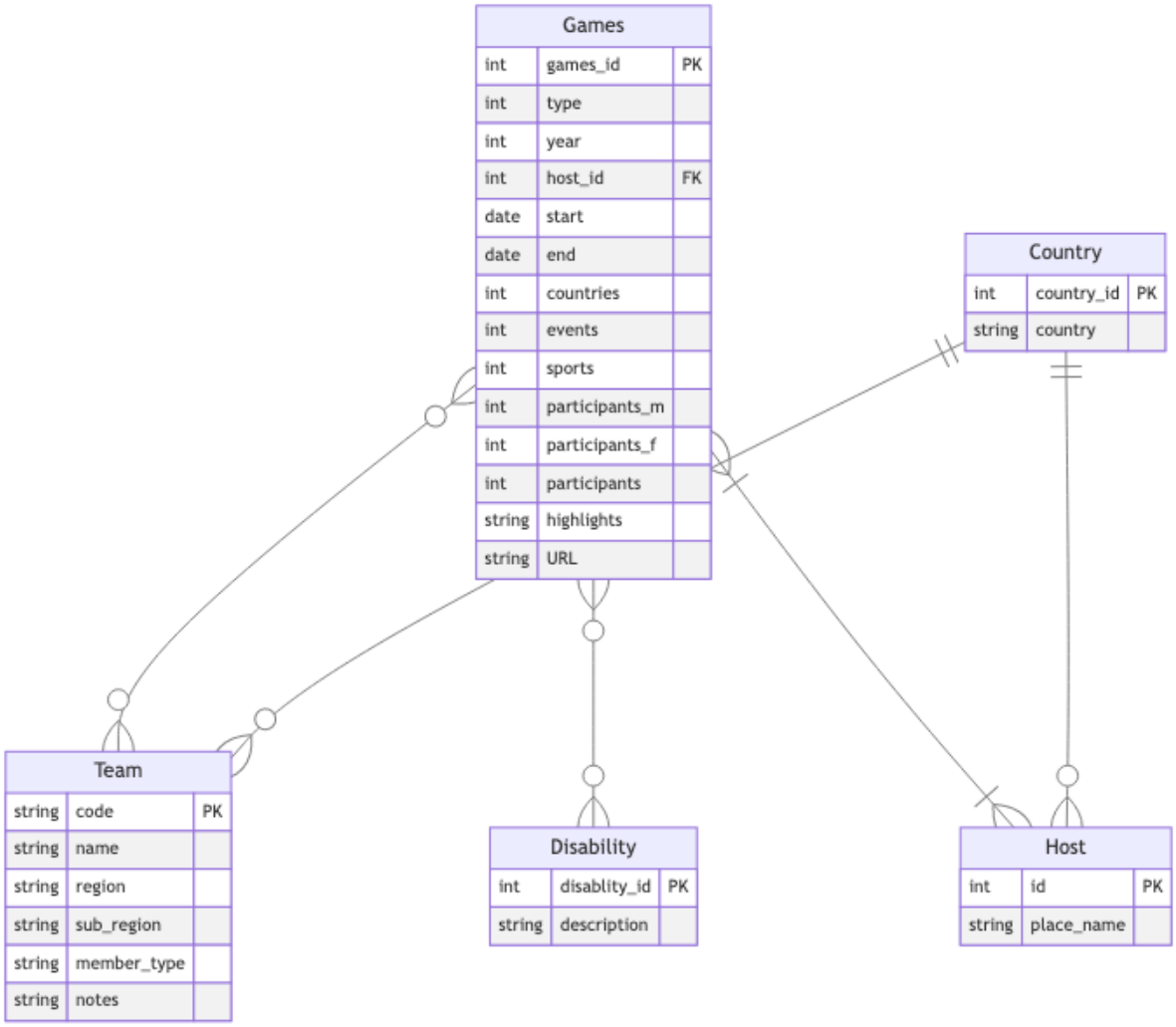
Another solution is to treat 'country' and 'host' as separate entities, i.e. create a new table for each of these. The reason for separating them into two further tables rather than one, is that one country can have many host cities. For example, USA has hosted events in Salt Lake City, New York and Los Angeles.

The relationship between Host and Country can be now described as: "A Country has one or many hosts, a Host is in one and only one Country". This is a **one-to-many** relationship.

Each of the new tables needs a unique attribute as primary key, **PK**. To form the relationship with the other table, you place the PK of the table on the one side of the relationship in the many side of the relationship where is becomes a Foreign Key, **FK**.

The relationship between the Games and Host can now be described as: "A host can host one or many Games, and a Games can be hosted by one or more Host city". This is a **many-to-many** relationship.

These tables now look like this:



Resolve the many-to-many relationships

However, there are still issues with design with repeating values for the many-to-many relationships.

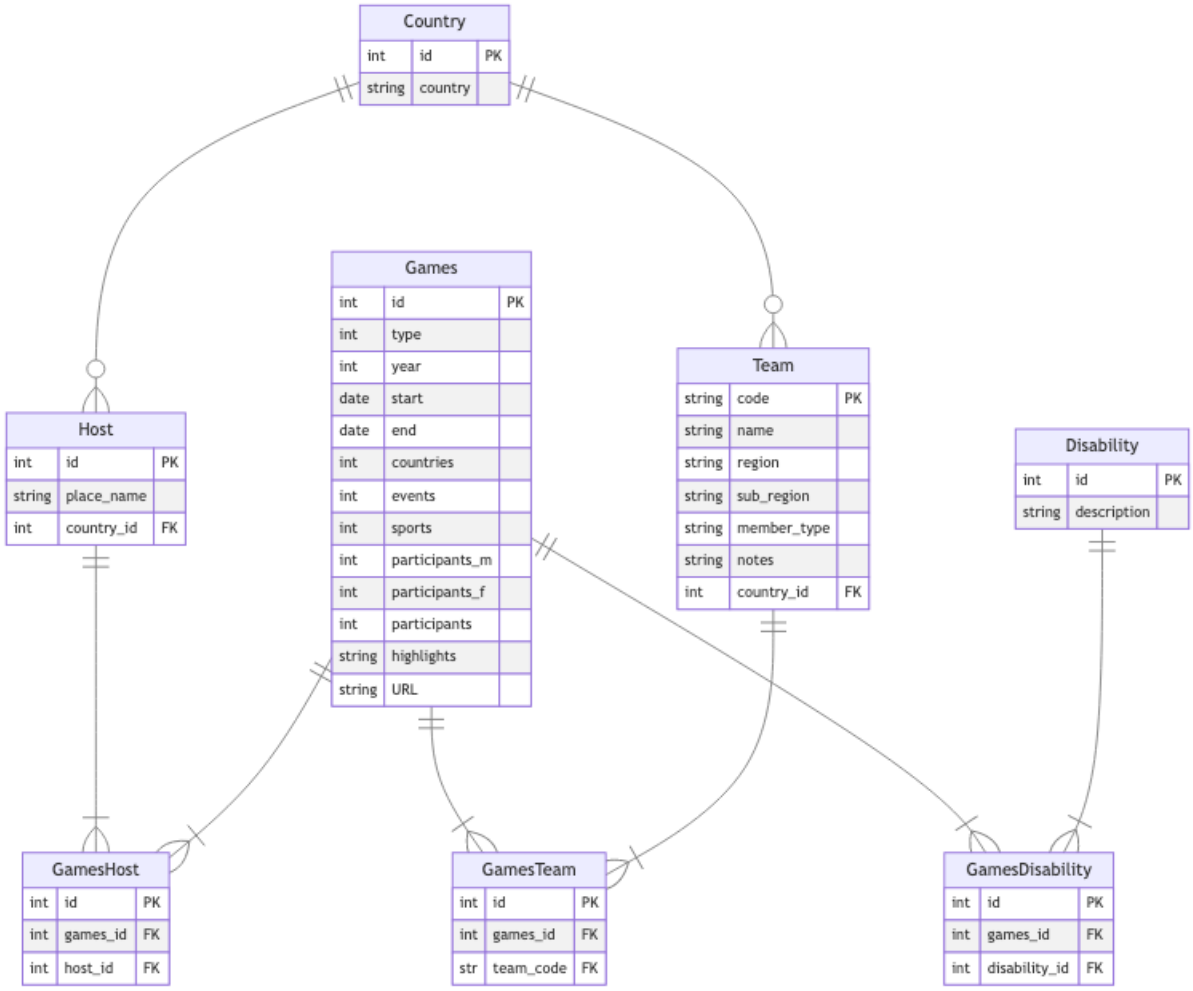
A relational database does not support the many-to-many relationship so you need to add another table that 'joins' or links the two tables. This joining table is sometimes called an 'association' table. It will have the PK of both tables as FK attributes; often these are the only two attributes in the table and can be combined as a composite primary key.

The many-to-many relationships that need to be addressed:

- Games : Team
- Games : Disability
- Games : Host

This can be resolved by adding 3 new tables:

- GamesTeam
- GamesDisability
- GamesHost



Next activity (4-05-logical-design-2nf.md)

5. Logical database design to 2NF

Second normal form (2NF)

Second normal form (2NF) is when a table that is already in 1NF and in which the values of each non-primary-key attribute (column) can be worked out from the values in all the attributes (columns) that make up the primary key.

To determine 2NF you need to know the concept of a **functional dependency**.

A **functional dependency** indicates how attributes relate to one another. A functional dependency exists when one attribute (or a set of attributes) uniquely determines another attribute. In other words, if you know the value of one attribute, you can determine the value of another attribute. There is a formal notation for this which we will not use in this course.

For tables with a single-column primary key, 2NF is automatically satisfied if the table is in 1NF. This is because there can't be partial dependencies when there's only one column in the primary key.

The key aspects of 2NF are:

- 1NF is met
- No partial dependencies, where a non-key attribute is dependent on only a part of a composite primary key. Every non-key attribute must depend on the entire primary key, not just a part of it.

Consider a table storing information about courses and the instructors who teach them:

CourseID	***InstructorID***	***InstructorName***	***CourseName***
101	1	Smith	Math
102	2	Johnson	Science
103	1	Smith	Algebra

In this table, CourseID and InstructorID together form the composite primary key.

InstructorName depends only on InstructorID, not on the combination of CourseID and InstructorID.

To convert this table to 2NF, we need to remove the partial dependency by creating two separate tables:

Courses Table:

CourseID	***CourseName***
101	Math
102	Science
103	Algebra

Instructors Table:

InstructorID	***InstructorName***
1	Smith
2	Johnson

Now, each non-key attribute is fully dependent on the primary key of its respective table, satisfying 2NF.

Is the paralympics ERD in 2NF?

The tables in the paralympics ERD already satisfy 2NF. We already replaced the 'type' and 'year' composite primary key with a single primary key called 'id'.

You could map the functional dependencies and demonstrate 2NF, though it isn't needed in this case:

Games

- PK: id
- Functional dependencies: id → type, year, host_id, start, end, countries, events, sports, participants_m, participants_f, participants, highlights, URL
- Each game is uniquely identified by id, and all other attributes depend on it.

Team

- PK: code
- Functional dependencies: code → name, region, sub_region, member_type, notes, country_id
- Each team has a unique code, which determines its other attributes.

Host

- PK: code
- Functional dependencies: code → host
- Each host is uniquely identified by a code.

Disability

- PK: id
- Functional Dependencies: id → description
- Each disability has a unique ID and a description.

Country

- PK: id
- Functional Dependencies: id → country, team_code
- Each country has a unique ID, which determines its name and associated team code.

GamesTeam

- PK: id
- Functional Dependencies: id → games_id, team_id
- This is a junction table for the many-to-many relationship between Games and Team. The surrogate key id determines the foreign keys.
- Additionally: {games_id, team_id} → id (this composite key could also uniquely identify the row)

GamesDisability

- PK: id
- Functional Dependencies:
    - id → games_id, disability_id
    - {games_id, disability_id} → id

GamesHost

- PK: id
- Functional Dependencies:
    - id → games_id, host_id
    - {games_id, host_id} → id

Next activity (4-06-logical-design-3nf.md)

6. Logical database design to 3NF

Third normal form (3NF)

Third normal form (3NF) is a table that is already in 1NF and 2NF, and in which the values in all non-primary-key columns can be worked out from only the primary key column(s) and no other columns.

The key aspects of 3NF are:

- It is in Second Normal Form (2NF)
- No Transitive Dependencies: All non-prime attributes (attributes that are not part of any candidate key) must be directly dependent on the primary key. There should be no transitive dependency, where a non-prime attribute depends on another non-prime attribute.

Consider this table:

StudentID (PK)	***CourseID (PK)***	***InstructorID***	***InstructorName***
1	101	1	Smith
2	102	2	Johnson
3	103	1	Smith

StudentID and CourseID together form the composite primary key.

InstructorName is dependent on InstructorID, not directly on the composite key.

To convert this table to 3NF, we need to remove the transitive dependency by creating separate tables:

StudentsCourses Table:

StudentID (PK)	***CourseID (PK)***	***InstructorID***
1	101	1
2	102	2
3	103	1

Instructors Table:

InstructorID (PK)	***InstructorName***
1	Smith
2	Johnson
1	Smith

Each non-primary attribute is directly dependent on the primary key of its respective table and so satisfies 3NF.

Is the paralympics ERD in 3NF?

- Games: All attributes (e.g., type, year, host_id, etc.) depend directly on id. No transitive dependencies (e.g., host_id is a foreign key, but host details are stored in the Host table).
- Team: Attributes like name, region, sub_region, etc. depend directly on code. country_id is a foreign key, not a derived attribute.
- Host: host depends directly on code.
- Disability: description depends directly on id.

- Country: country, team_code depend directly on id. team_code is a foreign key, not a derived attribute.
- GamesTeam, GamesDisability, GamesHost: These are association tables with surrogate keys (id) and foreign keys. No non-key attributes, so no transitive dependencies.

The design meets 3NF. All non-key attributes depend only on the primary key, and there are no transitive dependencies.

You could add further tables to the design if you wished; for example create tables for the region, sub_region and member_type values that are used in the Team table.

Next activity (4-07-logical-design-constraints-data.md)

7. Identifying constraints

Now that you have a normalised table structure, this next step clarifies any constraints on the attributes.

Constraints include:

- validation of the data values for an attribute, e.g. allowable values, whether null is allowed
- validation on foreign keys e.g. what to do in the child table when a row in the parent table is updated or deleted.

Keys are also constraints. You have already identified Primary Key and Foreign Key constraints in the previous activities.

SQL data constraints

The following are common constraints that can be noted for an attribute in a SQL database:

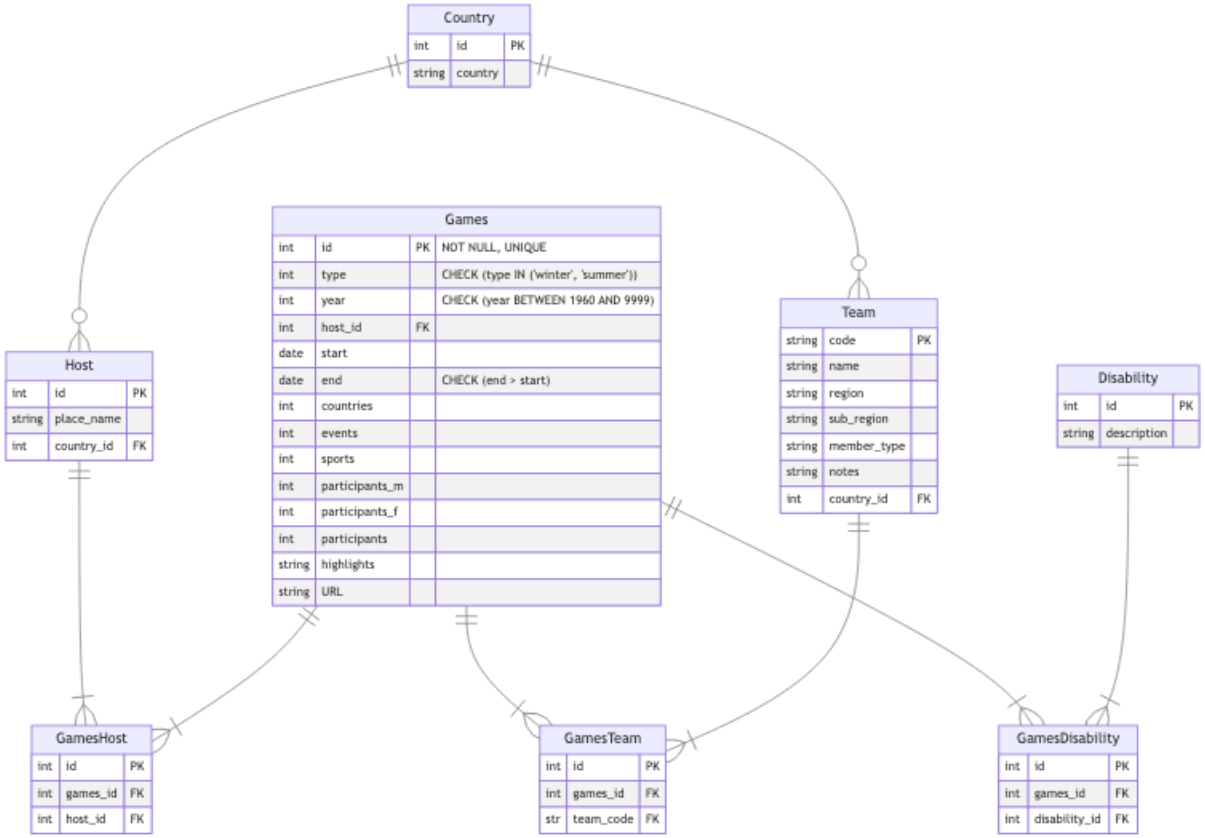
- PRIMARY KEY: A combination of NOT NULL and UNIQUE. Uniquely identifies each row in a table.
- FOREIGN KEY: Ensures that values in a column (or a group of columns) match values in another table's column(s), maintaining referential integrity.
- CHECK: Ensures that all values in a column satisfy a specific condition.
- DEFAULT: Sets a default value for a column if no value is specified.
- NOT NULL: Ensures that a column cannot have a NULL value.
- UNIQUE: Ensures that all values in a column are different. This is required for a single PK field!

Activity: Determine the data constraints for the attributes in the paralympics data

1. Add the constraints to the ERD for the Team table. The Games table is done for you as an example. Note that many DBMSs automatically treat the PK field as `NOT NULL`, and `UNIQUE` so you do not need to specify this.

   e.g. for the Team table:

   - 'code': Must be 3 letters capitalised
   - 'name': Required.
   - 'region': One of: 'Asia', 'Europe', 'Africa', 'America', 'Oceania', or can be empty if member type is construct
   - 'sub_region': One of: 'South, South-East', 'North, South, West', 'North', 'West, Central', 'Carribean, Central', 'South','East', 'Oceania', 'West','Central, East', or empty if MemberType is construct
   - 'member_type': Type of the competing team, one of: 'country', 'team', 'dissolved', 'construct'
   - 'notes': Any text notes on the record.



Next activity (4-08-logical-design-constraints-fk.md)

8. Identifying foreign key constraints

ON UPDATE and ON DELETE actions

In relational databases, foreign key constraints help maintain data integrity when changes occur. This is referred to as referential integrity.

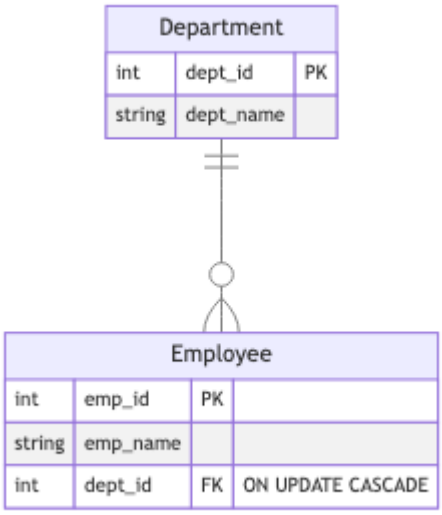
The constraints cover what action the DBMS should take ON UPDATE and/or ON DELETE. If an action is not explicitly specified, it defaults to "NO ACTION".

Update constraints

Constraint	***Action***
ON UPDATE CASCADE	Ensures that when a value in the parent table (the table being referenced) is updated, all corresponding values in the child table (the table with the foreign key) are automatically updated to match the new value. This is useful in scenarios where the primary key of a parent table might change, and you want to ensure that all related records in the child table reflect this change to maintain referential integrity.
ON UPDATE SET NULL	Sets the foreign key in the child table to NULL if the referenced key in the parent table is updated.
ON UPDATE RESTRICT	Prevents the update on the parent table if there are matching rows in the child table.
ON UPDATE NO ACTION	Similar to RESTRICT, it prevents the update but allows the action to be deferred until the end of the transaction.
ON UPDATE SET DEFAULT	Sets the foreign key in the child table to a default value if the referenced key in the parent table is updated.

Example of "ON UPDATE CASCADE"

Consider two tables: employees and departments. The employees table has a foreign key that references the departments table. If the dept_id in the departments table is updated, the corresponding dept_id in the employees table will also be updated automatically.



Delete constraints

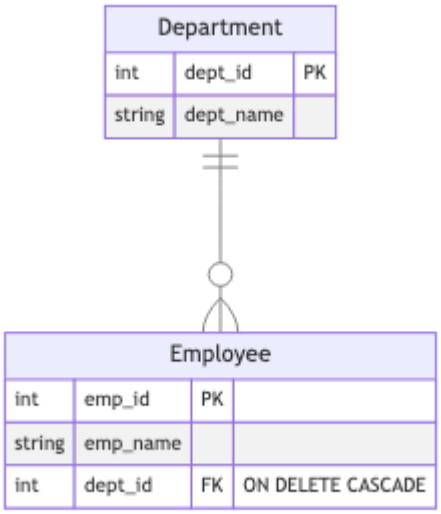
DELETE constraints help maintain data integrity when rows are deleted from a table.

These are the main DELETE constraints:

Constraint	***Action***
ON DELETE CASCADE	Automatically deletes all rows in the child table that have a foreign key reference to the deleted row in the parent table. Useful when you want to ensure that no orphaned records remain in the child table after a parent record is deleted.
ON DELETE SET NULL	Sets the foreign key in the child table to NULL when the referenced row in the parent table is deleted. Useful when you want to keep the child records but remove the reference to the deleted parent record.
ON DELETE RESTRICT	Prevents the deletion of a row in the parent table if there are matching rows in the child table. Useful when you want to ensure that no parent record is deleted if it has associated child records.
ON DELETE NO ACTION	Similar to RESTRICT, it prevents the deletion of a parent row if there are matching child rows, but the enforcement can be deferred until the end of the transaction. Useful when you want to defer the integrity check until the transaction is complete.
ON DELETE SET DEFAULT	Sets the foreign key in the child table to a default value when the referenced row in the parent table is deleted. Useful when you want to keep the child records and set a default reference when the parent record is deleted.

Example of 'ON DELETE CASCADE'

If a department is deleted, all employees in that department will also be deleted.



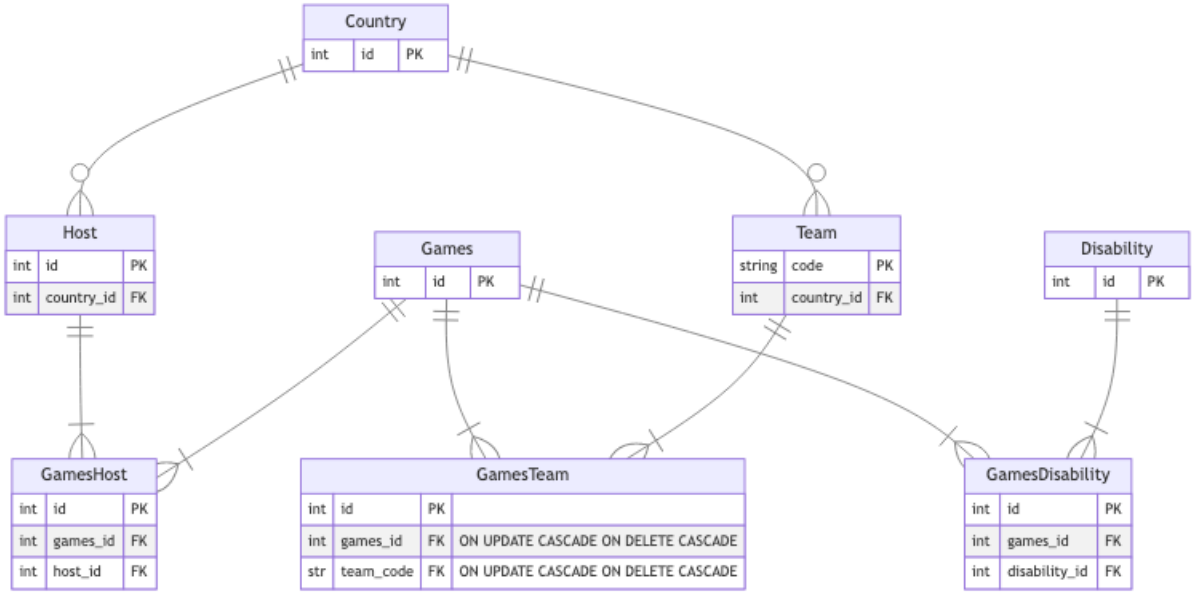
Apply foreign key constraints to the paralympics tables

All non-key attributes have been removed for this exercise just to make it easier to see the attributes you need for the constraints.

The constraints are described as:

- Games ||–|{ GamesTeam: If a Games is deleted, the GamesTeam record should also be deleted. On update, also update.
- Team ||–|{ GamesTeam: If a Games is deleted, the GamesTeam record should also be deleted. On update, also update.
- Games ||–|{ GamesDisability: If a Games is deleted, the GamesDisability record should also be deleted. On update, also update.
- Disability ||–|{ GamesDisability: If a Disability is deleted, the GamesDisability record should be deleted. On update, also update.
- Host ||–|{ GamesHost: If a Host is deleted, delete the GamesHost. On update, also update.
- Games ||–|{ GamesHost: If a Games is deleted, delete the GamesHost. On update, also update.
- Country ||–o{ Host: If a country is updated, update the host. If a country is deleted, set the host to NULL.
- Country ||–o{ Team: If a country is updated, update the team. If a country is deleted, set the team to NULL.

Constraints are applied to the FK attributes. For example: `int games_id FK "ON UPDATE CASCADE, ON DELETE CASCADE"`



Next activity (4-09-logical-design-activity.md)

9. Logical design activity

The previous activities covered a lot of theory which could make this process some complicated. However, applying the principles of normalisation for this module does require you to explicitly prove each stage of the normalisation.

Try to create a normalised design for the data below.

Take the table as shown below and consider how to split it into further tables so that:

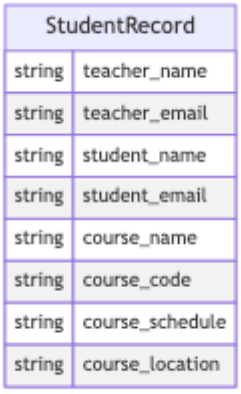
- Repeated/duplicated data is avoided.
- Each column/row intersection has only one entry.
- Each column name is unique.
- Each row has a unique identifier, i.e., a primary key for each table.
- Tables with relationships are linked by primary key:foreign key relationships.
- Where there is a many-to-many relationship, add a new table between the two tables so forming two one-to-many relationships.

Draw the design however you wish e.g., on paper, using a sketch app, in PowerPoint, using a diagramming app, writing the mermaid syntax in a markdown file, etc.

The data looks like this:

teacher_name	***teacher_email***	***student_name***	***student_email***	***course_name***
John Smith	john.smith@school.com	Alice Brown	alice.brown@school.com	Mathematics
John Smith	john.smith@school.com	Bob Green	bob.green@school.com	Mathematics
Jane Doe	jane.doe@school.com	Alice Brown	alice.brown@school.com	Physics
Jane Doe	jane.doe@school.com	Charlie White	charlie.white@school.com	Physics
Mark Taylor	mark.taylor@school.com	Bob Green	bob.green@school.com	Chemistry

This has been drawn as a conceptual design, i.e. one that has not been normalised, as follows:



Using gen-AI tools to generate a design

As a learning point it would be better to work out the design yourself, and then use gen-AI to compare its design to your own. You would then have a basis for understanding whether what the AI produced looks reasonable. It was clear in last year's coursework submissions where students simply used AI without checking the result as the designs often had issues!

To use AI (https://m365.cloud.microsoft/chat/) try copy/paste the prompt:

```
Given the following markdown format table of data, design a database
structure that is normalised to 3NF.

| teacher_name | teacher_email          | student_name  |
student_email            | course_name | course_code | course_schedule
| course_location |
|:-------------|:----------------------|:--------------|:-------------------------|:
| John Smith   | john.smith@school.com  | Alice Brown   |
alice.brown@school.com   | Mathematics | MATH101     | Mon-Wed-Fri 9am
| Room 101        |
| John Smith   | john.smith@school.com  | Bob Green     |
bob.green@school.com     | Mathematics | MATH101     | Mon-Wed-Fri 9am
| Room 101        |
| Jane Doe     | jane.doe@school.com    | Alice Brown   |
alice.brown@school.com   | Physics     | PHYS201     | Tue-Thu 11am
| Room 202        |
| Jane Doe     | jane.doe@school.com    | Charlie White |
charlie.white@school.com | Physics     | PHYS201     | Tue-Thu 11am
| Room 202        |
| Mark Taylor  | mark.taylor@school.com | Bob Green     |
bob.green@school.com     | Chemistry   | CHEM301     | Mon-Wed 2pm
| Room 303        |
```

Next activity (4-10-physical-design-structure.md)

10. Physical design: defining an SQLite schema in Python

Physical design

Goal: Optimize the logical model for performance and storage on a specific DBMS.

Output: A physical schema detailing file structure, indexing, and more.

Focus: How the data will be stored and accessed efficiently.

Connolly and Begg's definition of physical design goes beyond what is covered in this course. **For COMP0035** physical design is interpreted as the creation of the schema in an SQLite database.

For this activity you will use the `sqlite3` library which is part of the base Python installation so you do not need to explicitly install it in your virtual environment. Later in the course, `sqlmodel` or `sqlalchemy` will be used instead of `sqlite3`; these use a different approach so `sqlite3` is suggested for coursework 1.

Python is not required to create an SQLite database; independent tools such as DBBrowser lite, and tools integrated in the IDE that can create SQLite databases. However, for the coursework assessment you must create the database by defining a SQL schema and implement that as an SQLite file using Python code.

Using SQL

Structured Query Language or SQL provides a syntax for creating relational database. SQLite is a variant of SQL, and there are some differences.

- SQLite documentation (https://www.sqlite.org/docs.html)
- general SQL references (https://www.w3schools.com/sql/)

The general structure of a SQLite statement to create two related tables (https://www.sqlite.org/lang_createtable.html) is:

```
CREATE TABLE table_name_1
(
    column_name_1 data_type PRIMARY KEY,
    column_name_2 data_type NOT NULL,
    column_name_3 data_type NOT NULL UNIQUE,
    FOREIGN KEY (column_name_1) REFERENCES table_name_2 (column_name_1)
);

CREATE TABLE table_name_2
(
    column_name_1 data_type PRIMARY KEY,
    column_name_2 data_type
);
```

SQL keywords are not case-sensitive, writing 'create table' has the same effect as 'CREATE TABLE'. By convention the SQL keywords are upper case and adhering to this makes your code more readable by others.

The ; at the end of a SQL statement is required by some database management systems and is used to separate SQL statements.

SQLite data types

SQLite uses a dynamic type system, which is different from the static type systems used in other DBMS such as PostgreSQL or MySQL.

SQLite does not enforce strict data types for columns. Instead, it uses a concept called type affinity, which means each column has a preferred type, but it can still store values of other types.

Type affinity is determined by the declared type of the column (e.g., INTEGER, TEXT, REAL, etc.). SQLite will attempt to convert values to the column's affinity when inserting data.

SQLite maps declared types to one of five storage classes:

- NULL: The value is a NULL.
- INTEGER: A signed integer.
- REAL: A floating-point number.
- TEXT: A text string.
- BLOB: A binary large object.

Note there is no DATE. Dates can be stored as TEXT (e.g. '2025-09-19 10:30:00'), INTEGER or REAL. TEXT is generally used unless you have a specific reason to use the other types.

For more refer to the SQLite documentation (https://www.sqlite.org/datatype3.html).

The main thing to consider is:

> Prefer the 5 SQLite data types when you define the schema, if you use other data types then SQLite will attempt to map these to its types. Dates are typically stored as text in the format '2025-09-19 00:00:00'

Activity: Write SQL statements to define the tables

You can write SQL statements as Python strings.

However, IDEs typically have tools that support checking of SQL written in SQL files so this approach is suggested.

By default, SQLite does not enforce FK unless specifically configured. Add `PRAGMA foreign_keys = ON;` to the start of the sql statements to enable this.

1. Copy the starter file paralympics_schema_starter.sql (../../src/activities/starter/paralympics_schema_starter.sql) to your module. Rename it to remove `_starter`.

2. Write SQL statements that define the tables, their keys and any constraints. Some tables are defined for you so you see the syntax. The starter file has the ERD in comments for ease of reference.

   NB:

   - Store dates as TEXT.
   - Table creation order matters. You cannot add an FK if the table it refers to has not yet been created. Start with the tables that don't have any FK.
   - Keep table names to all lowercase, no spaces or underscores (not important now but will make it easier in later activities!)

Next activity (4-11-physical-design-create-db.md)

11. Create the paralympics database

In this activity you will create an SQLite database using Python code and the SQL schema created in the last activity.

The general approach when using sqlite3 is:

1. Create a connection: You start by connecting to a SQLite database file. If the file doesn't exist, it will be created.
2. Create a cursor: A cursor is an object that allows you to execute SQL commands and fetch results from the database.
3. Execute SQL commands: Using the cursor, you can run SQL statements like creating tables, inserting data, or querying records.
4. Commit the transaction: If you've made changes such as creating tables or inserting data, you need to commit them to save the changes permanently.
5. Close the connection: Finally, you close the connection to free up resources and ensure everything is properly saved and shut down.

Complete the following steps:

1. Create a new function in your database module to create the database structure e.g. `create_db(sql_scipt_path, db_path)` which takes the .sql file location as an argument. You may also want to pass the file path to the database file. It should create a sqlite database but does not return anything from the function.

2. Use `sqlite3` which does not need to be installed, add `import sqlite3`.

3. Create a connection to the file. The general is: `connection = sqlite3.connect(db_path)`

4. Create a cursor. The syntax is: `cursor = connection.cursor()`

5. Execute the sql commands using the cursor object. The general syntax is `cursor.execute("YOUR_SQL_STATEMENT")` however, in this case rather than executing a single SQL statement you want to execute a series of statements contained in a your SQL script file, so instead use `cursor.executescript(path_to_sql_script)`

6. Commit the changes using the connection object. The syntax is: `connection.commit()`

7. Close the connection. The syntax is `connection.close()`

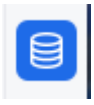
8. Define the sql file path and the database file path as variables, e.g. in `main`. Call the function in your code passing the variables as arguments. Run your code to create the database.

   NB: There is no mandatory file extension for sqlite files. By convention though they typically use `.db` or `.sqlite` so `paralympics.db` or `paralympics.sqlite`.

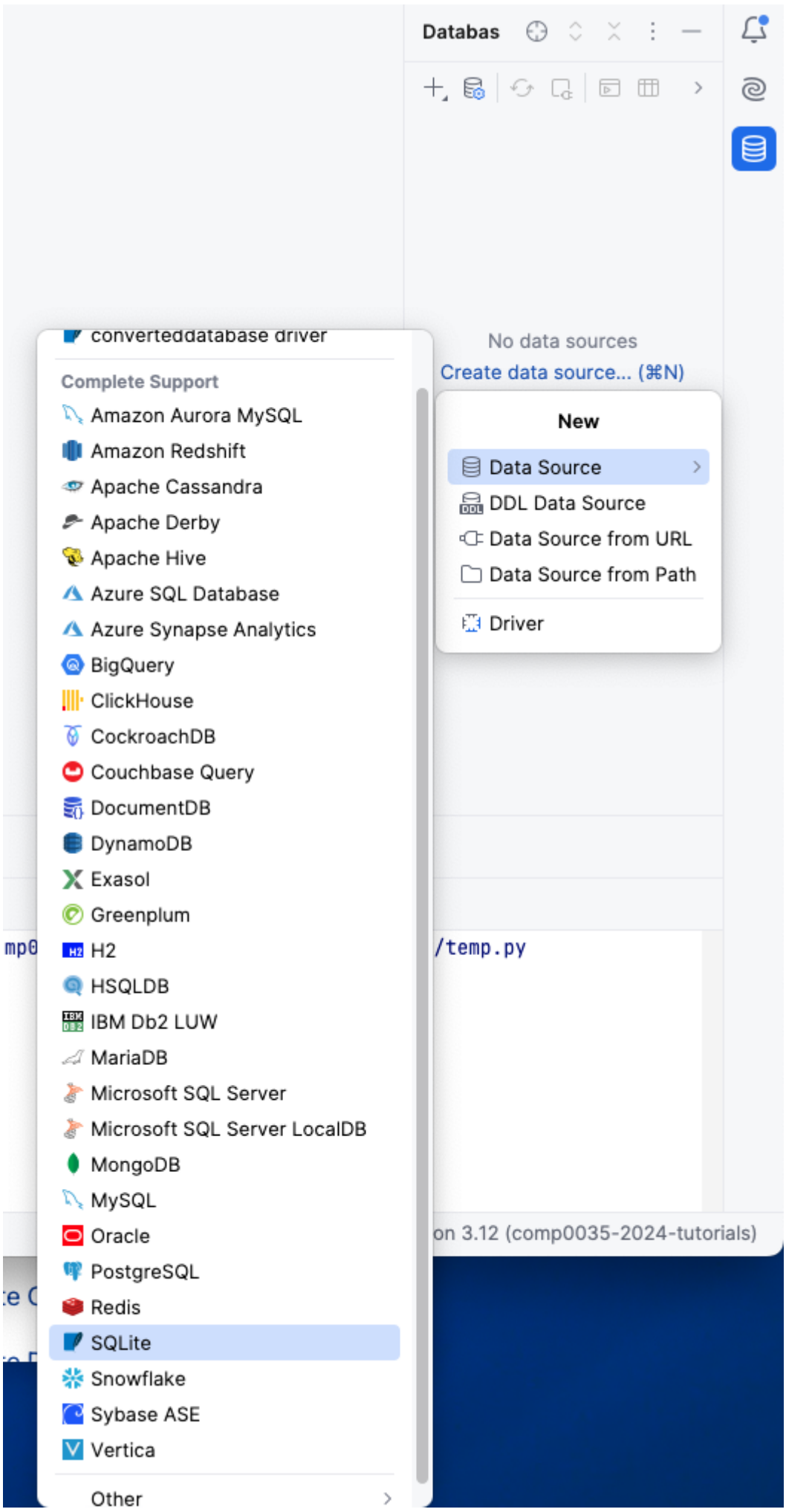
View the database structure in VS Code or PyCharm

If you added a VS Code extension for viewing SQLite databases, you should be able to open the database file from the project file list in VS Code by clicking on it. You should then see the table structure.

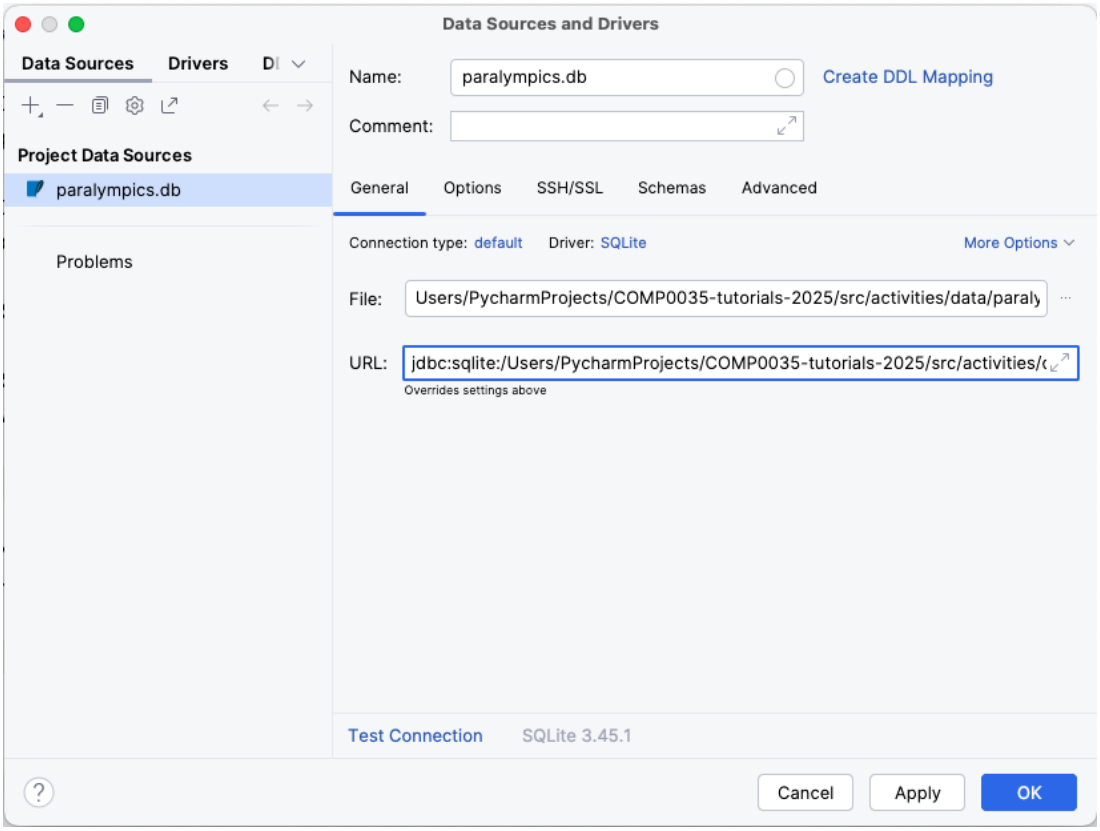
In PyCharm Professional (not available in the standard version) there is a database pane in the IDE.

Select it by clicking on the icon 

Select options to add a new data source with the type SQLite

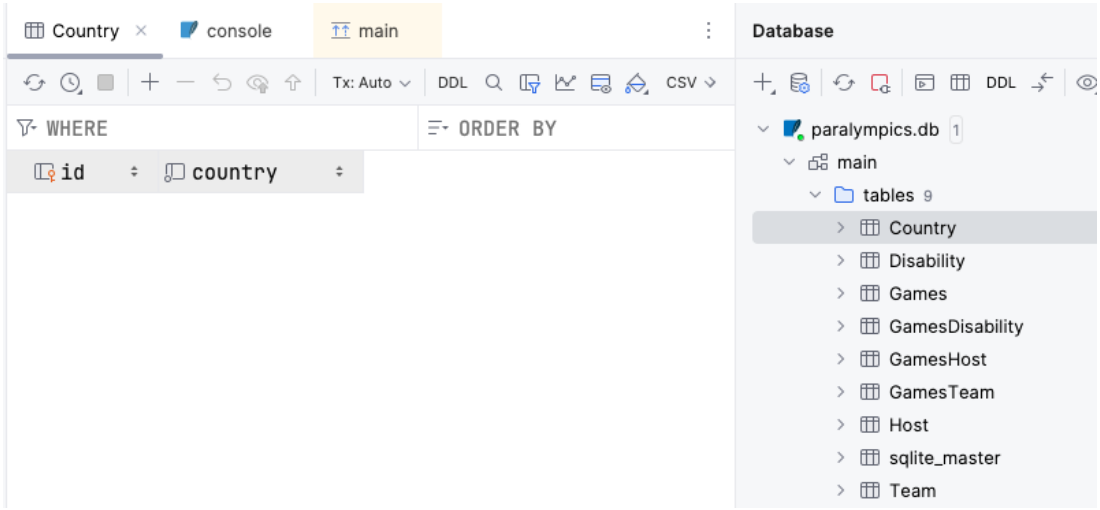

Databas
No data sources
Create data source... (⌘N)
New
Data Source
DDL Data Source
Data Source from URL
Data Source from Path
Driver
Complete Support
Amazon Aurora MySQL
Amazon Redshift
Apache Cassandra
Apache Derby
Apache Hive
Azure SQL Database
Azure Synapse Analytics
BigQuery
ClickHouse
CockroachDB
Couchbase Query
DocumentDB
DynamoDB
Exasol
Greenplum
H2
HSQLDB
IBM Db2 LUW
MariaDB
Microsoft SQL Server
Microsoft SQL Server LocalDB
MongoDB
MySQL
Oracle
PostgreSQL
Redis
SQLite
Snowflake
Sybase ASE
Vertica
Other
/temp.py
on 3.12 (comp0035-2024-tutorials)

Select the location of the database file. If the driver at the bottom of the screen does not show as SQLite then there will be an option to install it instead which you should select.



0-1. pyc-config-data-source.png

You can then expand the view to look at the tables, and double-click on a table to open the view of the data added to the table.



0-1. pyc-db-view.png

Activity (optional): Create a database for the student records

Use the student_schema.sql (../../src/activities/starter/student_schema.sql) and your function to create a database from script to create the student records database. Hopefully you wrote your function in a way that it can create any database from any script.

Next activity (4-12-sql-add-data.md)

12. Introduction to SQL queries to add data to the database

So far you have used SQL statements to create tables.

To add data to those tables you need to learn the basics of the SELECT and INSERT statements.

- INSERT is used to add new rows to a table.
- SELECT is used to find values row a table.

To add data to the tables that do not have any FK attributes you can simply INSERT values into the rows.

To add data to tables that have an FK attribute, then you need to query the parent table to find the value of the PK field for the relevant row and then use this value when you insert the new row in the child table.

For example, imagine you have a table for owners and a table for their pets. An owner can have more than one pet. To add a new owner, Zavian, and his pet, Frodo, you would need to:

- INSERT a row into the owners for Zavian.
- SELECT the id of the row where name=Zavian, assume the result is 3.
- Now you can INSERT a row for Champion where the owner_id is 3.

id	***name***
1	Fred
2	Skylar

id	***pet_name***	***owner_id***
1	Champion	1

The next two activities walk through the basic skills for INSERT and SELECT.

SQL references and tutorials

The following references have more details:

- SQLite SELECT reference (https://www.sqlite.org/lang_select.html)
- SQLite SELECT tutorial (https://www.sqlitetutorial.net/sqlite-select/) gives examples.
- SQLite INSERT reference (https://www.sqlite.org/lang_insert.html)
- SQLite INSERT tutorial (https://www.sqlitetutorial.net/sqlite-insert/) gives examples.

There are plenty of other reference sites available if you search.

Next activity (4-13-insert-no-fk.md)

13. INSERT data to a table that does not have a foreign key

INSERT syntax

The core syntax of a SQL query to insert one row of values into a table is:

```
INSERT INTO table_name (column_name_1, column_name_1)
VALUES (value1, value2);
```

If the table only has those two columns, then you can omit the column names like this:

```
INSERT INTO table_name
VALUES (value1, value2);
```

To insert multiple rows with values:

```
INSERT INTO table_name (column_name_1, column_name_1)
VALUES (value1, value2),
       (value1, value2),
       (value1, value2);
```

The above allow you to specify the names of the columns to insert the values, as well as the values.

You could create queries this way, save them in a .sql script and execute the script as you did for the database creation.

This approach would be cumbersome where you have many columns and/or many values to insert.

Using sqlite3 to insert

Insert a single row

To insert a single row, the code structure is that used to create the database structure just with a different query.

```
connection = sqlite3.connect(db_path)  # Create a connection to the
        database using sqlite3
cursor = connection.cursor()  # Create a cursor object to execute SQL
        commands
cursor.execute('PRAGMA foreign_keys = ON;')  # Enable foreign key
        constraints for sqlite

# Define the SQL INSERT query
insert_sql = 'INSERT INTO student (student_name, student_email) VALUES
        ("Harpreet", "harpreet@school.com")'

cursor.execute(insert_sql)  # Execute the insert query
connection.commit()  # Commit the changes
connection.close()  # Close the connection
```

In an application you are more likely to add values generated from variables.

sqlite3 supports the use of **parameterised queries** using the generic syntax:
`cursor.execute("INSERT INTO table_name VALUES(?, ?, ?)", data)`

- Use one `?` for each of the columns in the table
- `data` is the values for each of the columns e.g. `("some_string", 7, 12.87)` and must match the same order as the columns in the table

If you only want to insert a value or values for a subset of columns then specify the column names, e.g.
`cursor.execute("INSERT INTO table_name (column_name2, column_name3) VALUES(?, ?)", data)`

Parameterised queries offer benefits, a key one of which is that they separate SQL code from user input as the data is treated as values. This helps prevent SQL injection attacks (more on this next term).

An example of a single row insert:

```
value_str = "string1"
value_int = 12
value_float = 12.01
cursor.execute("INSERT INTO table_name VALUES(?, ?, ?)", (value_str,
        value_int, value_float))

# The same as above, written without the variable names
cursor.execute("INSERT INTO table_name VALUES(?, ?, ?)", ("string1", 12,
        12.01))

# Using specific columns only
cursor.execute("INSERT INTO table_name (name, age) VALUES(?, ?, ?)",
        ("string1", 12))
```

Insert multiple rows

To insert a multiple rows instead of one, use a list of values and `cursor.executemany()`

```
data = [
    ("string1", 12, 12.01),
    ("string2", 76, 33.07),
    ("string3", 45, 76.56),
]
cursor.executemany("INSERT INTO table_name (name, age, score)
        VALUES(?, ?, ?)", data)
```

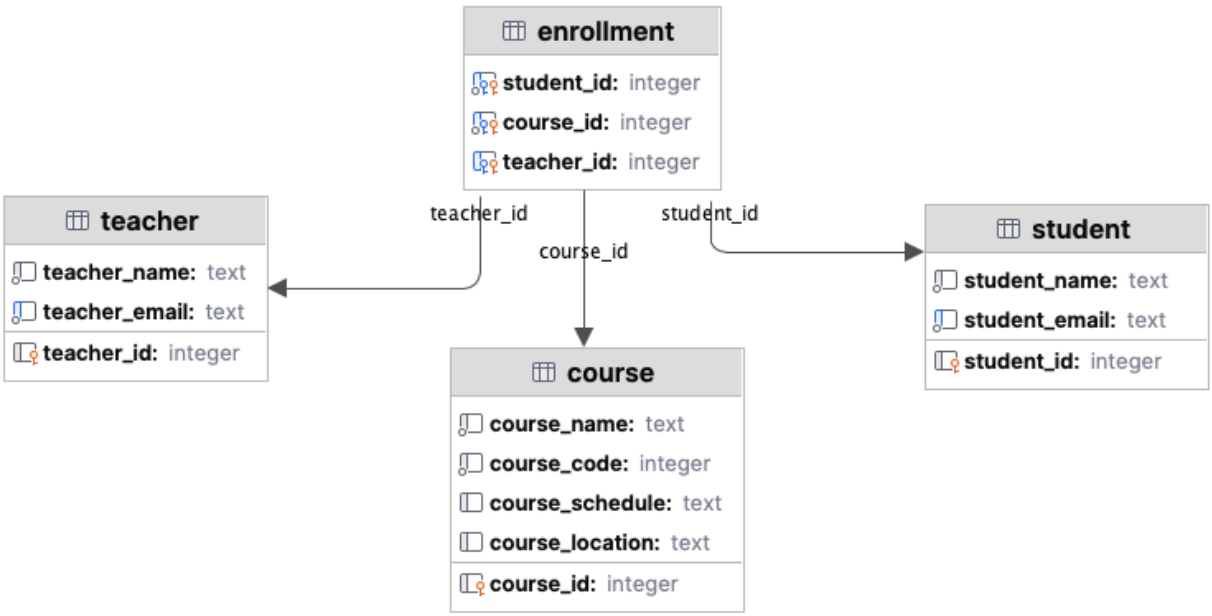
Activity: Insert a row into the student database

Using one of the methods above, insert values into the teacher, course or student database.

Choose some sample data from below:

```
teacher_name,teacher_email,student_name,student_email,course_name,course_code,course_s
John Smith,john.smith@school.com,Alice
Brown,alice.brown@school.com,Mathematics,MATH101,Mon-Wed-Fri 9am,Room
101
John Smith,john.smith@school.com,Bob
Green,bob.green@school.com,Mathematics,MATH101,Mon-Wed-Fri 9am,Room 101
Jane Doe,jane.doe@school.com,Alice
Brown,alice.brown@school.com,Physics,PHYS201,Tue-Thu 11am,Room 202
Jane Doe,jane.doe@school.com,Charlie
White,charlie.white@school.com,Physics,PHYS201,Tue-Thu 11am,Room 202
Mark Taylor,mark.taylor@school.com,Bob
Green,bob.green@school.com,Chemistry,CHEM301,Mon-Wed 2pm,Room 303
```

The database schema is as follows, note that the course table should really be split further but doesn't affect this activity:



Activity: Insert multiple rows using the data from a dataframe

For this activity, read the data into a pandas dataframe and use values in parameterised queries.

1. Write the code for the teacher, student and course tables. An extract of the approach for the student table:

```
# Code to create the connection and cursor is omitted from the
        excerpt, you will need it

data_path = resources.files(data).joinpath("student_data.csv")
df = pd.read_csv(data_path)

# Enable foreign key constraints for sqlite
cursor.execute('PRAGMA foreign_keys = ON;')

# Define the SQL insert statements for the parameterised queries
student_sql = 'INSERT INTO student (student_name, student_email)
        VALUES (?, ?)'

# Create dataframe with the unique values for the columns needed
        for the student table (database add the PK automatically)
student_df = pd.DataFrame(df[['student_name',
        'student_email']].drop_duplicates())

# Get the values as a list rather than pandas Series. The
        parameterised query expects a list.
student_data = student_df.values.tolist()

# Use `executemany()` with a parameterised query to add the values
        to the table.
cursor.executemany(student_sql, student_data)
```

2. Add the code for the teacher and student tables

Next activity (4-14-select-query.md)

14. Introduction to SQL queries

SQL syntax

The core syntax of a query to find one or more rows or values from a table is:

```
SELECT column_names
FROM table_name
WHERE some_condition;
```

To select all columns and rows from the `student` table in the sample.db (../../src/activities/data/sample.db) database you can use the '*' instead of writing all the column names. The SQL looks like this:

```
SELECT *
FROM student;
```

To select one or more rows meeting a given condition, add a WHERE clause (https://www.sqlitetutorial.net/sqlite-where/).

This example selects one result. It finds the student_id column from the student table where the value in the student_name column is 'Bob Green':

```
SELECT student_id
FROM student
WHERE student_name = "Bob Green";
```

This example selects several results. It finds the teacher_email and teacher_name of the teachers with teacher_id of 1 or 2:

To run these using sqlite3 python code:

```
import sqlite3
from pathlib import Path

# Create a SQL connection to our SQLite database and a cursor
db_path = Path(__file__).parent.parent.joinpath('data_db_activity',
        'student_normalised.db')
con = sqlite3.connect(db_path)
cur = con.cursor()

# Select all rows and columns from the student table
cur.execute('SELECT * FROM student')
rows = cur.fetchall()  # Fetches more than 1 row

# Select the student_id column
cur.execute('SELECT student_id FROM student WHERE student_name="Alice
        Brown"')
row = cur.fetchone()  # Fetches the first result

# Close the connection
con.close()
```

- `fetchall()` returns a list of tuples, each tuple contains field values of a row.
- `fetchone()` returns a row as a tuple.
- `fetchmany(size)` returns a specified number of rows as tuples.

Accessing values from the result row

The rows are returned as tuples. To access specific values from a tuple you need to use list style notation to access the element (NB: lists values start at 0 and not 1).

For example, with a single result:

```
# From the student table, find the student_id column, where the value
        in the name column is "Bob Green"
cur.execute('SELECT student_id FROM student WHERE student_name="Bob
        Green"')
row = cur.fetchone()  # Fetches the first result only
print(row)  # Prints the tuple containing the student_id
print(row[0])  # Access the value of the first column in the result,
        i.e. prints the student_id
```

For example, with a multiple row result:

```
cur.execute('SELECT teacher_name, teacher_email FROM teacher WHERE
        teacher_id in (1, 2)')
rows = cur.fetchall()  # Fetches all rows from the result
# Iterate the rows and print each row
for row in rows:
    print(row)
    # Iterate the items in the row and print each item
    for item in row:
        print(item)
# Print only the value of the first item in the first row
print(rows[0][0])
```

Activity: Practice select queries

1. Open the example_sql_query.py (../../src/tutorialpkg/sample_code/example_sql_queries.py).
2. Run `sample_insert_queries()` to see the results of the examples above.
3. Add your own query code to the sample queries code:

- Find all rows and columns for the courses table
- Find the course code for 'Chemistry'
- Find all course where the schedule includes Monday

The following references may be useful: - SQLite SELECT reference (https://www.sqlite.org/lang_select.html) - SQLite SELECT tutorial (https://www.sqlitetutorial.net/sqlite-select/)

Next activity (4-15-insert-with-fk.md)

15. Insert data to tables with foreign keys

In the previous activities you used INSERT to add data to the student, teacher and course tables; and SELECT to retrieve values from the data that was added.

This activity combines SELECT and INSERT to add rows to the tables with foreign keys.

To add an enrollment record for a row in the csv file you need to find the id values from teacher, student, and course for that row.

You could take the row values and run three separate SELECT queries to find the relevant ids from the teacher, student, and course tables.

You could shorten the SQL needed using a nested SQL query.

An example of the syntax:

```
INSERT INTO employees (emp_name, dept_id)
SELECT 'Alice Johnson', dept_id
FROM departments
WHERE dept_name = 'Computer Science';
```

Activity: use nested query to insert data into the enrollment table

The following is a paramaterised query that could be used to get the values or the enrollment table.

```
INSERT INTO enrollment (student_id, course_id, teacher_id)
VALUES ((SELECT student_id FROM student WHERE student_email = ?),
        (SELECT course_id FROM course WHERE course_name = ? AND
        course_code = ?),
        (SELECT teacher_id FROM teacher WHERE teacher_email = ?))
```

1. Delete the current data from the tables. Your code would fail database constraints if you try to add the same data again. You could drop and recreate the tables, or the following deletes all rows from all the tables. `python def delete_rows(db_path):   conn = sqlite3.connect(db_path)   cur = conn.cursor() cur.execute("SELECT name FROM sqlite_master WHERE type='table' AND name NOT LIKE 'sqlite_%';")   table_names = [row[0] for row in cur.fetchall()]   for table_name in table_names: cur.execute(f"DELETE FROM {table_name}")   conn.commit() conn.close()`
2. Add code to your function where you added the data to also add the enrollment data.

```
enrollment_insert_sql = """
                        INSERT INTO enrollment (student_id,
        course_id, teacher_id)
                        VALUES ((SELECT student_id FROM student
        WHERE student_email = ?), \
                                (SELECT course_id FROM course
        WHERE course_name = ? AND course_code = ?), \
                                (SELECT teacher_id FROM teacher
        WHERE teacher_email = ?)) \
                        """
for _, row in df.iterrows():
    cursor.execute(
        enrollment_insert_sql,
        (
            row['student_email'],
            row['course_name'],
            row['course_code'],
            row['teacher_email'],
        )
    )
```

3. Run the code again to create the add the data to the database. You should now have data in all the tables

Activity: Add data to the paralympics database

Code to add data is given in `starter/add_data_paralympics.py`. You may need to modify it to suit your schema.

The general approach used in this activity was:

1. Use pandas to load the data from .csv
2. Define the file path to the database
3. Create a connection to the database using sqlite3
4. Create a sqlite3 cursor that will be used to execute the SQL
5. Enable foreign key support
6. Execute the SQL using sqlite3 to create the database structure
7. Use sqlite3 and SQL queries to write code to add the data from the dataframe to the database
8. Close the connection

16. Further practice

1. Create an ERD for a data set with more entities and attributes

   These activities used a the 'paralympic games' and 'team codes' data in the paralympics_all_raw.xlsx (../../src/activities/data/paralympics_all_raw.xlsx) file.

   There is a further worksheet with the medal tables. Design additional tables to store this data, then update the schema and add the tables to the database file.

2. Find online tutorials such as:

   - Visual Paradigm: explains the concepts and provides an example (https://www.visual-paradigm.com/guide/data-modeling/what-is-entity-relationship-diagram/;WWWSESSIONID=1EAF6AF9532B727E5D05D2601FFF1B66.www1)
   - Freecodecamp video tutorial (https://www.youtube.com/watch?v=ztHopE5Wnpc) on database design
   - GURU99 Entity Relationship (ER) Diagram Model with DBMS Example (https://www.guru99.com/er-diagram-tutorial-dbms.html)

3. See database links in the Reading List for COMP0035.

Apply the knowledge to the coursework

- Read the coursework 1 specification so you know what it asks you to produce. It does not ask for every step that has been demonstrated in these activities.
- Apply the knowledge from these activities to your coursework data.

17. (Optional) Normalisation and application development

Normalisation helps keep your data clean and consistent, but can make your queries more complex and sometimes slower. The right balance depends on your application's needs.

Why normalise a SQL database

- Reduces data redundancy: Normalisation reduces or eliminates duplicate data, saving storage and making updates easier.
- Improves data integrity: By organizing data into related tables, it is easier to enforce consistency and avoid anomalies e.g., update, insert, delete anomalies.
- Easier maintenance: Changes to data structures or values may be simpler and less error-prone.

Potential disadvantages of normalisation

- Complex queries: Highly normalised databases often require more complex queries to retrieve related data. Tables need to be joined together in the queries.
- Performance overhead: JOIN operations can slow down read-heavy applications, especially with large datasets.
- Design complexity: Designing a fully normalised schema can be more time-consuming and may require deeper understanding of the data and its relationships.

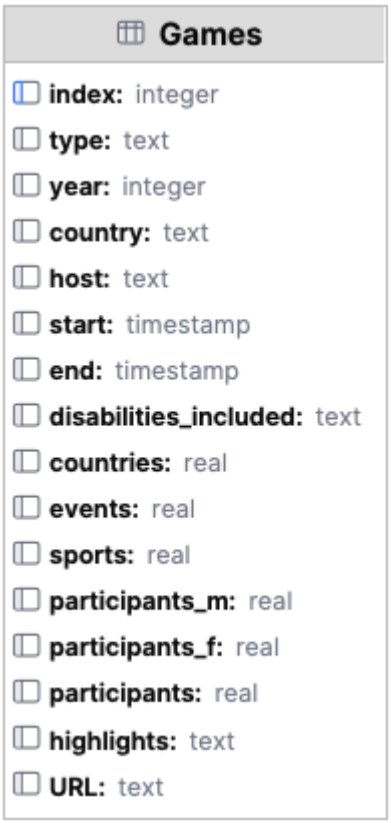
Example using the paralympics database

Two versions of the paralympics database have been created, one that has been normalised and one that has not.

The code in compare_queries.py (../../src/activities/starter/compare_queries.py) can be run to show the time taken to run each query.

Un-normalised database

para-not-normalised.sqlite (../../src/activities/data/para-not-normalised.sqlite)



0-1. Paralmpics database un_normalised

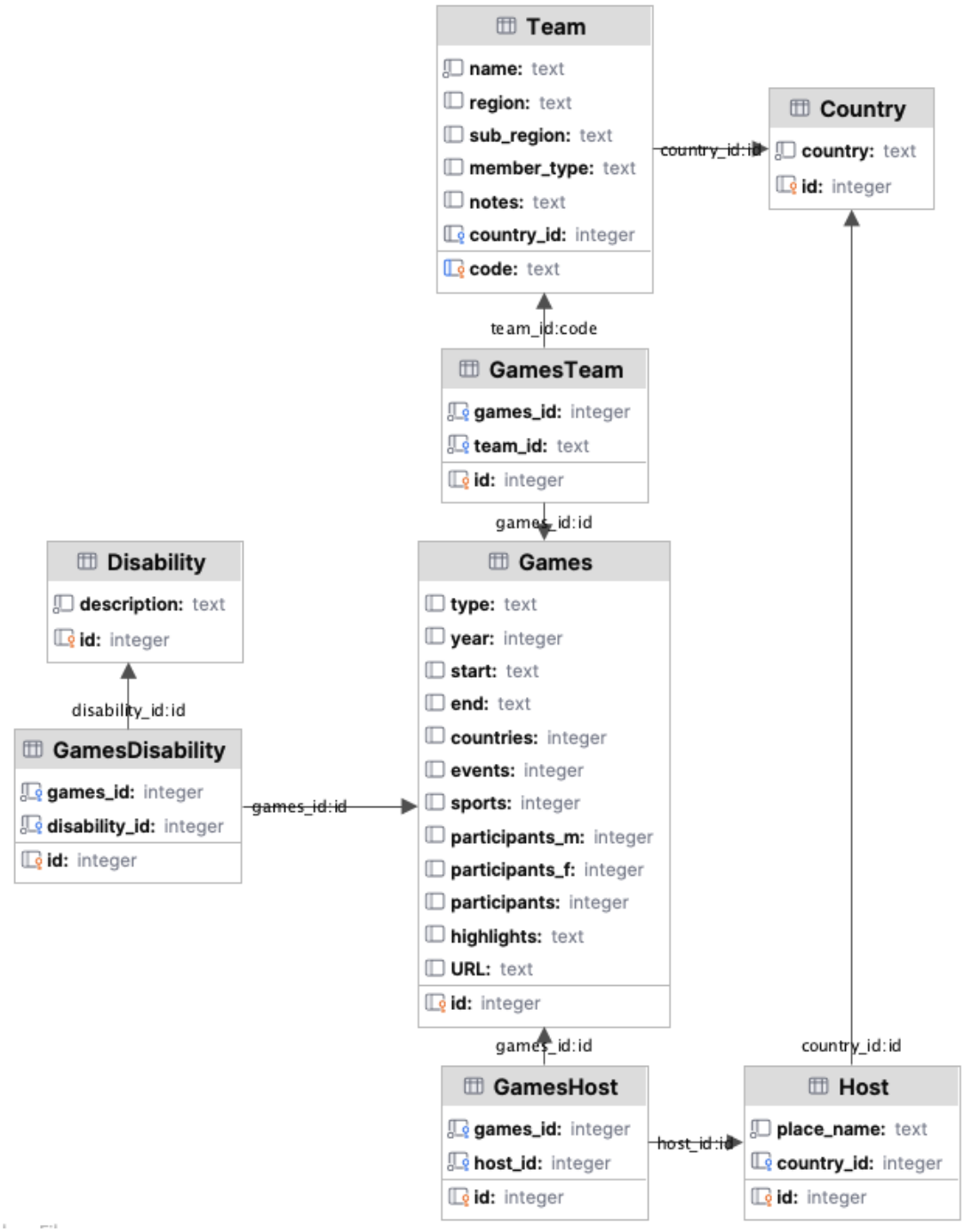
Query to find all details of the 2012 London Games:

```
SELECT *
FROM Games
WHERE type = 'summer'
  AND year = 2012
```

Queries timed on 3 executions: 0.000085 seconds, 0.000090 seconds, 0.000128 seconds

Normalised database

paralympics-normalised.db (../../src/activities/data/para-normalised.db)



0-1. Paralmpics database normalised

Query to find all details of the 2012 London Games:

```
SELECT g.type,
       g.year,
       c.country,
       h.place_name                       AS host,
       g.start,
       g.end,
       GROUP_CONCAT(d.description, ', ') AS disabilities_included,
       g.countries,
       g.events,
       g.sports,
       g.participants_m,
       g.participants_f,
       g.participants,
       g.highlights,
       g.URL
FROM Games g
         JOIN GamesHost gh ON g.id = gh.games_id
         JOIN Host h ON gh.host_id = h.id
         JOIN Country c ON h.country_id = c.id
         LEFT JOIN GamesDisability gd ON g.id = gd.games_id
         LEFT JOIN Disability d ON gd.disability_id = d.id
WHERE h.place_name = 'London'
  AND g.year = 2012
GROUP BY g.id, c.country, h.place_name
```

Queries timed on 3 executions: 0.000124 seconds, 0.000122 seconds, 0.000123 seconds