This week, you will learn how to process a denormalised CSV file, transform it into a set of normalised relational database tables, and generate an equivalent MongoDB document structure.

To get started, refer to the **students_enrollment.csv** file provided. The dataset includes the following columns:

```
_id,name,age,program,houseNo,street,town,course_name,course_instructor,
course_credits,hobbies
```

The sample data are structured as shown below:

```
123456,Alice Johnson,22,Computer Science,45,High Street, Birmingham, Data
Analytics, Dr. Smith,30,"Traveling, Reading"

123456,Alice Johnson,22,Computer Science,45,High Street, Birmingham, Machine
Learning, Prof. Johnson,25,"Traveling, Reading"
```

There are several valid ways to normalise this dataset and define the associated tables for a MySQL implementation.

For example, you might create the following tables:

1. Student
- **student_id** (PK)
- Name
- Age
- Program
- **address_id** (FK → Address.address_id)

2. Address
- **address_id** (PK)
- houseNo
- Street
- town

3. Course
- **course_id** (PK)
- course_name
- course_instructor
- course_credits

4. Hobby
- **hobby_id** (PK)
- hobby_name

5. Enrollment
- **enrollment_id** (PK)
- **student_id** (FK → Student.student_id)
- **course_id** (FK → Course.course_id)

6. StudentHobby
- **student_hobby_id** (PK)
- **student_id** (FK → Student.student_id)
- **hobby_id** (FK → Hobby.hobby_id)

Download the **`CSVProcessing.py`** file. This script takes the **`students_enrollment.csv`** file as input and generates six output files: **Student.csv**, **Address.csv**, **Course.csv**, **Enrollment.csv**, **Hobby.csv**, and **StudentHobby.csv,** corresponding to the tables described on the previous page.

```
10  # Sequential counters
11  address_counter = 1
12  course_counter = 1
13  enrollment_counter = 1
14  hobby_counter = 1
15  student_hobby_counter = 1
```

These counters are used to generate sequential unique IDs (i.e., primary keys) for five of the tables, excluding the Student table. For example, Address IDs follow the pattern A1, A2, A3, …, while Course IDs follow C1, C2, C3, …. **Why don't we need a counter for the Student table?**

```
17  # Containers for unique table entries
18  addresses = {}
19  students = {}
20  courses = {}
21  enrollments = []
22  hobbies = {}
23  student_hobbies = []
```

Curly braces {} create a **dictionary**, which is a key-value mapping.
Purpose: To store unique items and allow fast lookup.
Key idea: We want to check if an item already exists before adding it, so we use a key to look it up quickly.

Square brackets [] create a list, which is an ordered collection of items.
Purpose: To store all entries, including duplicates, in the order they are processed. We don't need a key lookup here because each row corresponds to a relationship (like enrollment or student-hobby mapping) and must be stored in order.

```
38          # ---------- Address table ----------
39          addr_key = f'{row["houseNo"]}|{row["street"]}|{row["town"]}'
40          if addr_key not in addresses:
41              addresses[addr_key] = {
42                  "address_id": f"A{address_counter}",
43                  "houseNo": row["houseNo"],
44                  "street": row["street"],
45                  "town": row["town"]
46              }
47              address_counter += 1
48
49          address_id = addresses[addr_key]["address_id"]
```

1. **Line 39:** Create a unique key for each address
2. **Lines 40-46:** Check if the address already exists (to ensure no duplicate addresses)
   - If not, create a new dictionary with:
     - address_id (sequential ID, e.g., A1, A2…)
     - House number, street, town
   - Store in addresses dictionary
3. **Line 47:** increment the counter for next address
4. **Line 49:** Retrieve the address_id for foreign key in the Student table

Key idea: Use a dictionary with a composite key to maintain unique addresses and assign sequential IDs.

```python
# ---------- Student table ----------
if student_id not in students:
    students[student_id] = {
        "student_id": student_id,
        "name": name,
        "age": age,
        "program": program,
        "address_id": address_id
    }
```

Check if the student already exists (to prevent duplicate students)

- If not, create a new student record
- Store relevant information in a dictionary:
  - student_id → primary key
  - name, age, program → student details
  - address_id → foreign key linking to Address table
- Store in students dictionary

```python
# ---------- Course table ----------
course_key = f'{row["course_name"]}|{row["course_instructor"]}|{row["course_credits"]}'
if course_key not in courses:
    courses[course_key] = {
        "course_id": f"C{course_counter}",
        "course_name": row["course_name"],
        "course_instructor": row["course_instructor"],
        "course_credits": row["course_credits"]
    }
    course_counter += 1

course_id = courses[course_key]["course_id"]
```

Apply the same approach used for generating the Address table. Refer to the explanation on the previous page for guidance.

```
74          # ---------- Enrollment table ----------
75          enrollments.append({
76              "enrollment_id": f"E{enrollment_counter}",
77              "student_id": student_id,
78              "course_id": course_id
79          })
80          enrollment_counter += 1
```

Create a new enrollment record

- enrollment_id → unique sequential ID for the enrollment (E1, E2, …)
- student_id → foreign key linking to Student table
- course_id → foreign key linking to Course table
- Increment the counter to ensure sequential enrollment IDs

```
84          if student_id not in processed_hobby_students:
85              processed_hobby_students.add(student_id)
86
87              hobby_list = [h.strip() for h in row["hobbies"].split(",")]
88
89              for hobby in hobby_list:
90                  # Create hobby if new
91                  if hobby not in hobbies:
92                      hobbies[hobby] = {
93                          "hobby_id": f"H{hobby_counter}",
94                          "hobby_name": hobby
95                      }
96                      hobby_counter += 1
97
98                  # Create Student-Hobby mapping
99                  student_hobbies.append({
100                     "student_hobby_id": f"SH{student_hobby_counter}",
101                     "student_id": student_id,
102                     "hobby_id": hobbies[hobby]["hobby_id"]
103                 })
104                 student_hobby_counter += 1
```

Line 84-85: It ensures each student's hobbies are processed only once by tracking which student IDs have already been handled.

Line 87: Split comma separated string into individual hobbies. Converts "Running, Cooking" → ["Running", "Cooking"]
Lines 91-96: Unique hobby records are created in the same way as unique student records, but with sequential hobby_id values assigned for each new hobby.

Lines 99-104: Map students to hobbies (Many-to-Many)

- Creates a record in the Student–Hobby join table
  - student_id → FK to Student table
  - hobby_id → FK to Hobby table
  - Each student can have multiple hobbies, and each hobby can belong to multiple students

```
109     def write_csv(filename, fieldnames, rows):
110         with open(OUT_DIR / filename, "w", newline='', encoding="utf-8") as f:
111             writer = csv.DictWriter(f, fieldnames=fieldnames)
112             writer.writeheader()
113             writer.writerows(rows)
```

A function to write to a CSV file.

- **Line 110**: Open a CSV file for writing. Creates or overwrites the file in the OUT_DIR folder.
- **Line 111:** Create a CSV writer using column headers
- **Line 112:** Write the header row
- **Line 113:** Write all data rows

```
115 write_csv("Address.csv",
116           ["address_id", "houseNo", "street", "town"],
117           list(addresses.values()))
118
119 write_csv("Student.csv",
120           ["student_id", "name", "age", "program", "address_id"],
121           list(students.values()))
122
123 write_csv("Course.csv",
124           ["course_id", "course_name", "course_instructor", "course_credits"],
125           list(courses.values()))
126
127 write_csv("Enrollment.csv",
128           ["enrollment_id", "student_id", "course_id"],
129           enrollments)
130
131 write_csv("Hobby.csv",
132           ["hobby_id", "hobby_name"],
133           list(hobbies.values()))
134
135 write_csv("StudentHobby.csv",
136           ["student_hobby_id", "student_id", "hobby_id"],
137           student_hobbies)
```

The **`write_csv`** function is then invoked six times, each call generating one of the corresponding CSV files for the normalised tables.

The simplest MongoDB document structure for this dataset is shown below. It relies solely on embedding for related data. However, you may also design alternative structures that combine embedding with referencing, depending on the requirements of your application.

```
{
  "_id": 123456,
  "name": "Alice Johnson",
  "age": 22,
  "program": "Computer Science",
  "address": {
    "houseNo": "45",
    "street": "High Street",
    "town": "Birmingham"
  },
  "courses": [
    {"name": "Data Science", "instructor": "Dr. Smith", "credit": 30},
    {"name": "Machine Learning", "instructor": "Prof. Johnson", "credit": 25}
  ],
  "hobbies": ["Traveling", "Reading"]
}
```

Following the steps demonstrated in the Week 5 lab, you can read the CSV file and generate the corresponding JSON document. The **`CSV2MongoDocument.py`** script provided on Canvas will perform this conversion for you. The code in this script should be straightforward to follow, assuming you have already understood the examples explained earlier in this document.

Download the **ecommerce_orders.csv** file from Canvas. This file contains around 10 orders placed by different customers. Your task is to normalise this dataset and adapt the previously provided Python script to generate multiple CSV files corresponding to the required tables. You should also modify the script to produce a JSON file containing the MongoDB documents. You may use the suggested tables and document structure as a guide.

1. Customers

- customer_id (PK)
- customer_name
- house_no
- Street
- town

2. Orders

- order_id (PK)
- order_date
- customer_id (FK)
- payment_method

3. Products

- product_id (PK)
- product_name
- Category
- unit_price

4. OrderItems

- order_item_id (PK)
- order_id (FK)
- product_id (FK)
- quantity


```
{
  "_id": 1001,
  "order_date": "2025-01-12",
  "payment_method": "Credit Card",
  "customer": {
    "customer_id": "C001",
    "customer_name": "John Smith",
    "address": {
      "house_no": "12",
      "street": "Maple Road",
      "town": "London"
    }
  },
  "items": [
    {
      "product_name": "Wireless Mouse",
      "category": "Electronics",
      "unit_price": 15.99,
      "quantity": 2
    },
    {
      "product_name": "Laptop Stand",
      "category": "Electronics",
      "unit_price": 29.99,
      "quantity": 1
    }
  ]
}
```