Databases — Part 2

The objective of this sequence of works is to understand the concept of databases and to handle it with the SQL language: we will manipulate this language thanks to the `sqlite3` module in Python.

In the last work, we have seen that a relational database consists of multiple tables, that have relations between them. We have seen how to mentally represent those tables, and how to read them "by hand". This week, we will learn the basics of SQL (Structured Query Language — please don't write or say "SQL language", this would be redundant), a language designed to manipulate those databases.

We have seen examples of SQL last week, namely how to create a table (`CREATE TABLE`), how to add data into it (`INSERT INTO`), how to update it (`UPDATE`) and how to make simple queries (`SELECT`). This week, we will define with more details those operations.

As we have seen, a table consists of different rows, each row representing a record. A record consists of different fields: there is one field per column of the table. The fields of a table have a name and a type. The data types that we will manipulate in this course are:

- numeric data (integers, floating-point numbers)
- textual data (strings)
- date and time
- booleans (true / false)

Choosing the right data type for a field is relatively straightforward on small examples. Just note that when the numbers at hand can be really big or really small, one has to be cautious: for example unlike `Python`, there are several data types to represent integers (depending on the range of values expected). However, we will not dive into the details of this different types.

1 Creating a table

Figure 1 shows an example from previous week to understand how to create new tables.

First, the users table is created. This table contains three fields: a field named "identifier", which is an integer, and which is the primary key. The "ASC" keyword tells SQL to automatically manage the ids in ascending order, starting at 1. Thus, you do not need to manage the identifiers, and you do not need to make sure that they are all different. SQL does it for you. When updating the table (see next section), this field is automatically set. There are also two fields named "surname" and "first_name" which will contain the different names of the users.

Then, the genres table is created. This table contains two fields: a field named "identifier" and a field named "genre" (which will contain the textual representation of the different literary genres: Poetry, Novel, Theater...). This table is similar to the previous one.

Then, we create the books table. As in the genre tables, the identifiers are automatically managed by SQL. Each book has then an author and a title (text). Then comes the difficult part of this table: the genre field. This field is an integer, and has to be a reference to the genres table. We see that because of the constraint we have put on this field: the constraint, named "fk_genre" (foreign key: genre), tells SQL that the genre field from this table has to be a value from the identifier field of the genres table. This creates a relation between the books table and the genres table.

2 Store data into tables

Storing a new record into a table can be made thanks to the following command:

```
INSERT INTO table(field1, field2, ...) VALUES (value1, value2, ...);
```

To add our two first users in our library, we will execute:

```sql
CREATE TABLE IF NOT EXISTS genres(
    identifier INTEGER PRIMARY KEY,
    genre TEXT NOT NULL
);
```

```sql
CREATE TABLE IF NOT EXISTS users(
    identifier INTEGER PRIMARY KEY,
    surname TEXT NOT NULL,
    first_name TEXT NOT NULL
);
```

```sql
CREATE TABLE IF NOT EXISTS books(
    identifier INTEGER PRIMARY KEY,
    author TEXT NOT NULL,
    title TEXT NOT NULL,
    genre INTEGER NOT NULL,
    CONSTRAINT fk_genre FOREIGN KEY(genre) REFERENCES genres(identifier)
);
```

Figure 1: Creating the genres and books table with SQL.

```sql
INSERT INTO users(surname,first_name) VALUES ('Smith', 'Alice');
INSERT INTO users(surname,first_name) VALUES ('Johnson', 'Bob');
```

As we have seen before, there is no need to fill the "identifier" field, it is automatically incremented and stored (Alice has id 1, Bob has id 2).

Exercise 1

A database contains two tables whose structure is the following:

- users: surname, first_name, age
- books: title, author, price

Give the request to:

1. Add the user "Elizabeth Brown", 20 years old
2. Add the book "Le rouge et le noir", which costs 4.99€.

3 Read data from tables

Reading data is a little more difficult than storing because it is important to select data before accessing it. You must then write filters to do so. Select one or more fields to read is easy. Selecting which records you are interested in is more difficult: it is this part of the request that will require a little thinking. The command to read data, without filtering, is:

```sql
SELECT field1, field2, ... FROM table;
```

This command will select all the records (all the rows) of the table, and only keep the fields mentionned. It is possible to select all the fields at once with the special character * :

```sql
SELECT * FROM table;
```

To filter the records we wish to access, you must add the filter at the end of the request. The easiest filters are those based on the value of a field (equal, more than, less than...). It is also possible to combine different filters with logical operators like "and", "or", "not"... The general syntax is:

```sql
SELECT fields FROM table WHERE conditions;
```

For instance, to select all the poetry books (genre 1) in the library, we will ask the following request:

```sql
SELECT * FROM books WHERE genre=1;
```

To get all the books written by Shakespeare:

```sql
SELECT * FROM books WHERE author='Shakespeare';
```

WARNING: The equality test on strings is an exact test (everything has to be equal: diacritics, capital letters...).

REMARK: It is possible to use the operators <, >, <=, >=, ! =..., in addition to the equal operator. When we are not sure of the exact spelling of a field, it is also possible to use the `LIKE` operator. For example, to search for all the books whose author start with a "S":

```
SELECT * FROM books WHERE author LIKE 'S%';
```

Instead of using the "%" character, that matches any string, whatever its length is, it is possible to use the "_" character, to match exactly one character. Thus, if one wishes to search for all the books whose title is exactly 4 characters, one will use 4 times the underscore:

```
SELECT * FROM books WHERE title LIKE '____';
```

As a last example, to search for all the books written by Hugo that are not poetry:

```
SELECT * FROM books WHERE author='Hugo' AND genre!=1;
```

Exercise 2

We take the same database than in the previous exercise. Write one request per question, to give:

1. All the books.
2. All the users with the surname Jones.
3. All the books written by Hugo.
4. All the books that cost less than 5€.
5. All the books that cost less than 5€ and those that cost more than 12€.
6. All the users whose first name starts with Tyler- (for example, Tyler-John or Tyler-Jay).

Sometimes, there are just too many answers, and one wishes to get the answers by block, instead of having all of them. The `LIMIT` keyword is there for this purpose. To get the list of the 10 first users of the library:

```
SELECT * FROM users LIMIT 10;
```

To get the 10 next users:

```
SELECT * FROM users LIMIT 10,10;
```

Then, it could be really useful to sort the results. For example, if one wishes to sort the results by surname, in ascending order ("ASC", use "DESC" for descending order), and get only the 20 first results:

```
SELECT * FROM users ORDER BY surname ASC LIMIT 20;
```

Exercise 3

We take the same database than in the two previous exercises. Write one request per question, to give:

1. The 5 youngest users.
2. The 10 most expensive books.
3. The 10 books that follow (by price).

4 Updating tables

Reading and writing is a nice first step, but it is impossible to manage big databases without considering the possibility to modify them. The `UDPATE` command allows this. As with the `SELECT` command, it is possible to modify one record or multiple ones. The syntax is the following, where the conditions can be the same as in the previous section:

```
UPDATE table SET field1=..., field2=... WHERE conditions;
```

Let us take as example the table from last week, see Table 1. If one wishes to update the record of the 1st borrowing to add a return date, one would execute:

```
UPDATE borrowings SET Date_To='2021-03-02' WHERE Identifier=1;
```

Identifier	Borrower_id	Book_id	Date_From	Date_To
1	4	2	01/01/2021	—
2	1	4	02/01/2021	08/01/2021
3	4	5	04/01/2021	16/01/2021

Table 1: Work13 — Exercise 1: The 3 borrowings.

Exercise 4

We take the same database than in the three previous exercises. A bug was found in the database: William Wordsworth was misspelled "Wordsword". What is the request we have to write to remove this bug?

5 Erase data

The last "basic" operation is the erasure of a record. Beware that an erased data cannot be recovered (unless one has a copy of the data). It is important to use this command with great care. In particular, do not forget to add a `WHERE` in the command, else, the full table is erased! The syntax is the following:

```
DELETE FROM table WHERE conditions;
```

For example, if one wishes to remove the user with identifier 12, one will write:

```
DELETE FROM users WHERE identifier=12;
```

6 Joins

Before closing this overview of the SQL language, we must see probably the most interesting operation on tables: the join. Usually, the way we have created the tables implies that, for us to retrieve pertinent information, we have to use information from multiple tables to get our results.

Of course, it is possible to perform multiple basic operations and to manually look through the results, but this require a lot of time and/or programming skills. But it would be a shame to use a SQL database without using it to perform this work we call a join.

A possible way to see the joins is to use the same `SELECT` command as before, but to select the fields from multiple tables, and to test conditions on those fields. The general syntax would be:

```
SELECT field1, field2, field3, ... FROM table1, table2, ... WHERE field3=field4
    ...;
```

There is one issue when executing a `SELECT` command on multiple tables: how does it work when multiple tables have a field that has the same name? In our example, all our tables have a field "identifier". How will we distinguish in which table we want our identifier field to be tested? To do that, we must use "table.identifier" to indicate that we want to use the identifier field of a given table. In our running example of the library, if we want to print the title, the author and the literary genre of all the books in our library, we can write the following query:

```
SELECT title, author, genres.genre FROM books, genres WHERE books.genre=genres.
    identifier;
```

In this request, there is no need to specify in which table the title and author fields appear (no field named title or author appear in the genres table). For the genre, we do not want to print the identifier (present in the books table), we want the textual representation of the genre (present in the genres table).

Of course, it is possible to write more complex joins, for example by using three tables or more.

Exercise 5

We take a database a little more evolved than in the previous exercises. This database contains the following tables:

- users: id, surname, first_name, age
- authors: id, surname, first_name
- genres: id, genre
- books: id, title, author_id, price, genre_id

(in each table, the id field is an integer identifier; in the books table, the author_id is linked to the authors table and the genre_id is linked to the genres table)

Write one request per question, to give:

1. The titles of the books written by Victor Hugo.
2. All the books in the poetry genre (id 1, as in previous examples).
3. All the books in the poetry genre written by Victor Hugo.
4. You have heard that the author of some books of this library has just opened an account in the library to borrow books. Give the list of all his or her books.