LEARN CODING

ale66

github.com/ale66/learn-coding

STRUCTURED DATA

Structured data is the gist of *institutions*

Data organised in *tables* is structured

Silent conventions make such tables easy to recognise and work with

SPREADSHEETS

- broadly support working with struc. data on a personal level
- fail to separate data and code: impossible to maintain!

RELATIONAL DB MANAGEMENT SYSTEMS

- software that creates *isolated*, permanent databases: data is safe
- One RDBMS can host hundreds of databases, which remain independent from each other
- we create, access and modify data only via the SQL language

STRUCTURED DATA

- Table organisation can be cumbersome: see the *publishing company* example
- A single table shall only represent either
  - a single class of homogenous objects, e.g., writers, catalog entries…
  - a record of the relationship between two objects that are described in the db

THE RELATIONAL DATA MODEL

- columns (attribute) names shall be unique in each table
- rows may in principle be repeated.
- special columns called keys use ids to make rows unique
- A *primary key* is an 'ID' column that is usually specified for search

SQL TABLES

Careful planning of the table:

```
1  CREATE TABLE mywriters (
2      writer_id INTEGER(3) PRIMARY KEY,
3      writer_fname varchar(20) NOT NULL,
4      writer_lname varchar(20) NOT NULL,
5      writer_gender 'M' or 'F',
6      writer_birthyear INTEGER(4)
7  );
```

A table may have a lifespan of decades!

QUERYING TABLES

SQL naturally expressess queries on structured data:

```sql
SELECT gender, SUM(no_of_books), AVG(no_of_books)

FROM writers

GROUP BY gender;
```

QUERYING TABLES, 1

```sql
SELECT City, PostalCode, CustomerName
FROM Customers
WHERE Country = 'Germany' AND City = 'Berlin';
```

`SELECT` defines the format of the output, columns and their order and names

QUERYING TABLES, 2

```sql
SELECT City, PostalCode, CustomerName
FROM Customers
WHERE Country = 'Germany' AND City = 'Berlin';
```

SELECT defines the format of the output, columns and their order and names

FROM defines the table(s) to be queried

QUERYING TABLES, 3

```sql
SELECT City, PostalCode, CustomerName
FROM Customers
WHERE Country = 'Germany' AND City = 'Berlin';
```

`SELECT` defines the format of the output, columns and their order and names

`FROM` defines the table(s) to be queried

`WHERE` defines the conditions to be met *by the rows*

Column names act as variables

THE CONCEPT OF CURSOR

an invisible finger that iteratively will point at each row of the table

The column names-variables take up the values of the row pointed at by the cursor

QUERYING TABLES, 4

```sql
SELECT gender, SUM(no_of_books), AVG(no_of_books)

FROM writers

GROUP BY gender;
```

GROUP BY divides up the table in several minitables with the same value of the grouping variable

HAVING selects the minitables according to a condition

QUERYING TABLES, 4

```sql
SELECT gender, SUM(no_of_books), AVG(no_of_books)

FROM writers

GROUP BY gender

HAVING SUM(no_of_books) > 10;
```

it male/female writers are not prolific enough we exclude them from the output

CURSOR

SQL executes row-by-row (mostly)

Cursor: an invisible finger pointing at the current row

Python can create cursors on tables and make them execute operations

KEY

Forces rows to be unique

Exists even in paper archive

```
1 CREATE TABLE students(
2     student_id: PRIMARY KEY int(6),
3     ...
4 );
```

FOREIGN KEY

Key values used in another table

Supports the concept of *normalization:*

- each cell has an atomic value
-
- each table is strictly about either
  - a stand-alone class of objects: students, modules, lectures
  - a relationship, possibly with multiple instances, between said entities

FK EXAMPLE, 1

We include a FK column in order to access extra data from an external table

```
1  CREATE TABLE students(
2      student_id: PRIMARY KEY int(6),
3      ...
4      // navigate to extra personal info
5      nino: varchar(16) REFERENCES residents(nino)
6  );
```

only inserts and updates involving a *valid* Nat'l insurance numbers are allowed

POSTGRESQL

```
1 CREATE TABLE students(
2     student_id: PRIMARY KEY int(6),
3     ...
4     cod_fis: varchar(16),
5     CONSTRAINT fiscal
6       FOREIGN KEY(cod_fis)
7       REFERENCES residents(cf)
8 );
```

FK EXAMPLE, 2

A table recording relationship instances: the exam

```
1 CREATE TABLE exams(
2     exam_date: date,
3     mark: int(2),
4     student_name: int(6) REFERENCES students(student_id),
5     module_name: int(6) REFERENCES modules(module_id),
6     programme: varchar(6) REFERENCES programmes(prog_id)
7 );
```

Only inserts and updates involving three *valid* keys are allowed

POSTGRESQL

An example with the Italian fiscal code (16 characters)

```
1  CREATE TABLE students(
2      student_id: PRIMARY KEY int(6),
3      ...
4      cod_fis: varchar(16) REFERENCES residents(cf),
5      CONSTRAINT fiscal
6        FOREIGN KEY(cod_fis)
7        REFERENCES residents(cf)
8  );
```

SQL AT WORK

IEEE Spectrum: Top Programming languages 2023:

SQL is the most requested language in job advertisements [worldwide]

MANY TABLES, MANY RELATIONWHIPS

- DBs are designed for the long-term
- Entity-Relationship diagrams: domain knowledge meets IT
- for maintainance, the more tables the better
- for querying, SELECTs become complex

JOINS

- Follows up the Foreign keys to create a giant summary table
- Row Selection follows
- column selection to improve readability

```
1  WHERE  ATable.fk = AnotherTable.key
```

```sql
SELECT ATable.col, AnotherTable.anotherCol
FROM   ATable, AnotherTable
WHERE  ATable.fk = AnotherTable.key
AND    ATable.col > 10
AND   AnotherTable.aThirdCol = 0
;
```

WITH ALIASES

```sql
SELECT T.col, AT.anotherCol
FROM   ATable as T, AnotherTable as AT
WHERE  T.fk = AT.key
AND    T.col > 10
AND    AT.aThirdCol = 0
;
```

WITH COL. RENAMING

```sql
SELECT T.col as Name, AT.anotherCol as Income
FROM   ATable as T, AnotherTable as AT
WHERE  T.fk = AT.key
AND    AT.anotherCol > 10
AND    AT.aThirdCol = 0
;
```

EXAMPLE QUERIES

See again the E-R project for the Film studios

```sql
CREATE TABLE Cities (
    city_id VARCHAR(30) PRIMARY KEY,
    city_name VARCHAR(30) NOT NULL,
    city_country VARCHAR(30) NOT NULL,
    city_population INT(8)
);
```

```sql
CREATE TABLE Actors (
  actor_id VARCHAR(30),
  actor_name VARCHAR(30) NOT NULL,
  actor_age INT(3),
  actor_gender VARCHAR(1) NOT NULL,
  born_in VARCHAR(30),
  PRIMARY KEY (actor_id),
  FOREIGN KEY (born_in) REFERENCES Cities(city_id)
);
```

```sql
CREATE TABLE Worked_in (
  actor VARCHAR(30),
  film VARCHAR(30),
  salary INT(8),
  PRIMARY KEY (actor_id, film_id),
  FOREIGN KEY (actor) REFERENCES Actors(actor_id),
  FOREIGN KEY (film) REFERENCES Films(film_id)
);
```

```sql
CREATE TABLE Films(
    film_id : PRIMARY KEY int(6),
    premiere : date,
    budget : int(10)
    ...
);
```

Which actor is coming from a big city?

```sql
SELECT Actors.actor_name, Cities.city_name
FROM   Actors, Cities
WHERE  Actors.birth_city = Cities.city_id
  AND  Cities.city_population > 1000000;
```

Are there pairs of actors who are from the same city?

We need two cursors running over the same table

```sql
SELECT A.actor_name, B.actor_name, A.birth_city
FROM   Actors AS A, Actors AS B
WHERE  A.birth_city = B.birth_city;
```

Select all pairs of actors who acted in the same film

We need `Worked_in`

```sql
SELECT A.actor_name, B.actor_name, W.film
FROM   Actors AS A, Actors AS B, Worked_in AS C, Worked_in AS D
WHERE  C.actor = A.actor_id
  AND  D.actor = B.actor_id
  AND  C.film = D.film
;
```

Has anyone ever acted in a film taking place in their own birthplace?

[we need to record the places where each film took place]

```
1 SELECT ...
2 FROM   ...
3 WHERE  ...
4 ;
```