UE 803 - Data Science

Session 5: Data storage (part 2)

Yannick Parmentier - Université de Lorraine / LORIA

Introduction

- Recall: querying data using an expressive (enough) and **efficient query language + program**
- **2 main options**:
    - *noSQL* (aka *distributed*) databases → **flexibility**
    - *SQL* (aka *relational*) databases → **expressivity** (high-level query language)

Outline

1. Relational DBMS

2. Structured Query Language

3. Introducing the SQLite RDBMS

4. Introducing the SQLalchemy Python library

Relational database management systems

Structure & properties

Relational databases

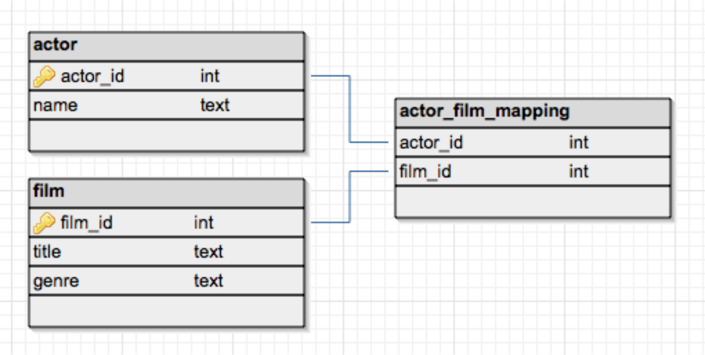
- First proposed by Codd (1970)
- Are made of 2 components:
  - an underlying **model** (data structure *definition*)
  - a set of **tables** (data *values*)
- **Tables** are (technically) somewhat similar to *spreadsheets*, where:
  - **rows** (aka *records*) are identified uniquely
  - **columns** (aka *fields*) are typed

Relational databases (continued)

- **Tables** are (conceptually) of two types :
  - *entities*
  - *relations* (between entities)

- Example:

  (model)



actor	
actor_id	int
name	text

film	
film_id	int
title	text
genre	text

actor_film_mapping	
actor_id	int
film_id	int

(values)

actor_id	name
1	John Wayne
2	Steve McQueen
3	Humphrey Bogart

Tables of an RDBMS

- Mandatory *internal* unique record identifier
  ➡ **primary key**

  - may be *atomic* (made of a single field) or not
  - may be *automatically incremented* or not
  - is such that **fields** which are not part of a primary key can be **deduced** from it

- Field(s) bound to *external* identifier(s)
  ➡ **foreign key(s)**

- **Golden rules**: non-null primary keys, atomic field values, non-primary key fields cannot be infered from subparts of a primary key, etc.

  ➡ See "A Simple Guide to Five Normal Forms in Relational Database Theory" by W. Kent (1982) [link]

Designing relational databases

- **Data model design** requires *analyzing workflows* and processes

  → **methodology!**

- Various refinement steps of the model (normalization) to ensure:
  - **data non-redundancy**
  - **data integrity**
  - **data accuracy**

Distributed *vs* relational databases

Distributed *vs* relational databases

- **noSQL**
  - *dynamicity*
    (cf unstable data)
  - *flexibility*
    (no admin)
  - *scalability*
  - *distributability*
  - *efficiency*
    (fast reads/**writes**)

Distributed *vs* relational databases

- **noSQL**
  - *dynamicity*
    (cf unstable data)
  - *flexibility*
    (no admin)
  - *scalability*
  - *distributability*
  - *efficiency*
    (fast reads/**writes**)

- **SQL**
  - *stability*
    (cf stable data)
  - *expressivity*
    (cf complex model)
  - *replicability*
  - *high configurability*
  - *efficiency*
    (cf fast **reads**)

Queying relational databases

Introducing SQL

Introducing SQL

- **Declarative** Domain-Specific Language (DSL)
- Primarily used to **insert/extract information** to/from databases' **tables**
- **ANSI/ISO normalization** since 1986 (revised in 2016)
- **Used in many RDBMS** (PostgreSQL, Oracle, MySQL, ...)
- **Various available interpreters** depending on the RDBMS (SQL+, TOAD, Squirrel SQL, ...)

SQL is typed

- Built-in data types:
  - **Character strings** (variable size): `VARCHAR2(n)`
  - **Binary strings**: `VARBINARY(n)`
  - **Boolean** values: `BOOLEAN`
  - **Numerical** values: `INTEGER | FLOAT | DECIMAL(precision, scale)`
  - **Temporal** values (date times): `DATE | TIME | TIMESTAMP`
  - **Empty** value: `NULL`

An SQL query

Selecting (filtering) records:

```
SELECT <columns> [AS <names>]

FROM <table> [<alias>]

[JOIN <table> ON <column> | NATURAL JOIN <table>]

[WHERE <predicate on rows>]

[GROUP BY <columns> [HAVING <predicate on groups>]]

[ORDER BY <columns> [DESC | ASC]]
```

NB: SQL is not case sensitive!

Selecting data

- List all actors whose name starts with "W" in alphabetical order:

```sql
SELECT *
 FROM  actor
 WHERE name LIKE 'W%'
 ORDER BY name;
```

- Result (answer to the query) returned by the system:

```
id_actor  name
--------- -----------
1         Wayne, John
```

Selecting data (continued)

- List all actors together with the number of movies they play in:

```
SELECT name, count(*) AS nb_movies
 FROM actor a, actor_film_mapping play_in
 WHERE a.actor_id = play_in.actor_id
 GROUP BY name;
```

- Result (answer to the query) returned by the system:

```
name                   nb_movies
---------------------- ---------
Wayne, John            2
```

Selecting data (continued)

- **Subqueries** (appearing in the constraints of the WHERE clause):

```
SELECT isbn,
       title,
       price
 FROM  Book
 WHERE price < (SELECT AVG(price) FROM Book)
 ORDER BY title;
```

- **Derived tables** (appearing in the FROM/JOIN clause):

```
SELECT b.title, b.price, sales.items_sold, sales.company_nm
FROM Book b
  JOIN (SELECT SUM(Items_Sold) Items_Sold, Company_Nm, ISBN
        FROM Book_Sales
        GROUP BY Company_Nm, ISBN) sales
  ON sales.isbn = b.isbn
```

Adding/removing/changing data

Inserting / **deleting** / **updating** records:

```sql
INSERT INTO <table>
 (<column1>, <column2>, <column3>)
 VALUES
 (<val_col1>, <val_col2>, <val_col3>);


DELETE FROM <table>
 WHERE <column> = <value>;


UPDATE <table>
 SET <column1> = <value1>
 WHERE <column2> = <value2>;
```

SQL for implementing a data model

Creating / altering / deleting tables:

```
CREATE TABLE <table> (
 column1 <type1>,
 column2 <type2>,
 column3 <type3> [NOT NULL],
 PRIMARY KEY (<column1>, <column2>)
);


ALTER TABLE <table> ADD <column4> <type4> [NOT NULL];


DROP TABLE <table>;
```

SQL for controlling transactions

Committing and rollbacking:

```
CREATE TABLE tbl_1(id int);

 INSERT INTO tbl_1(id) VALUES(1);
 INSERT INTO tbl_1(id) VALUES(2);

COMMIT;

UPDATE tbl_1 SET id=200 WHERE id=1;

SAVEPOINT id_1upd;

UPDATE tbl_1 SET id=1000 WHERE id=2;

ROLLBACK to id_1upd;

 SELECT id from tbl_1;
```

Implementing relational databases

Introducing SQLite

SQLite

[Download link]

- **lightweight** RDBMS:
  - self-contained (developed in ANSI-C)
  - serverless
  - zero-configuration
  - transactional
  - uses dynamic typing (unlike other RDBMS)
  - can work with in-memory databases (for fast access)

Using SQLite directly in a terminal

```
$ sqlite file.db
SQLite version 2.8.17
Enter ".help" for instructions
sqlite> .help
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in a text format
.echo ON|OFF           Turn command echo on or off
.exit                  Exit this program
(...)
.output FILENAME       Send output to FILENAME
.output stdout         Send output to the screen
.read FILENAME         Execute SQL in FILENAME
.schema ?TABLE?        Show the CREATE statements
.separator STRING      Change separator string for "list" mode
.show                  Show the current values for various settings
.tables ?PATTERN?      List names of tables matching a pattern
.timeout MS            Try opening locked tables for MS milliseconds
.width NUM NUM ...     Set column widths for "column" mode
```

Using SQLite in a graphical environment

- SQLiteBrowser (https://sqlitebrowser.org/)

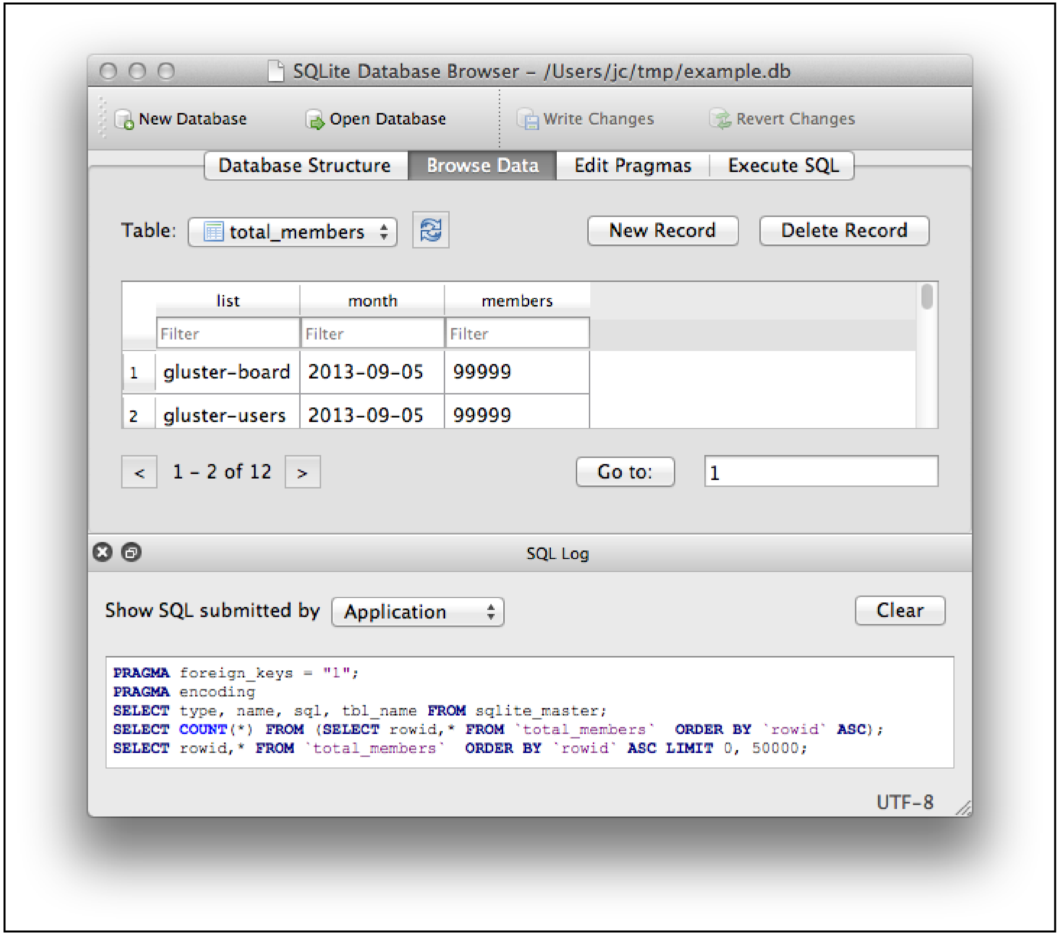


Table: total_members

	list	month	members
1	gluster-board	2013-09-05	99999
2	gluster-users	2013-09-05	99999

1 – 2 of 12

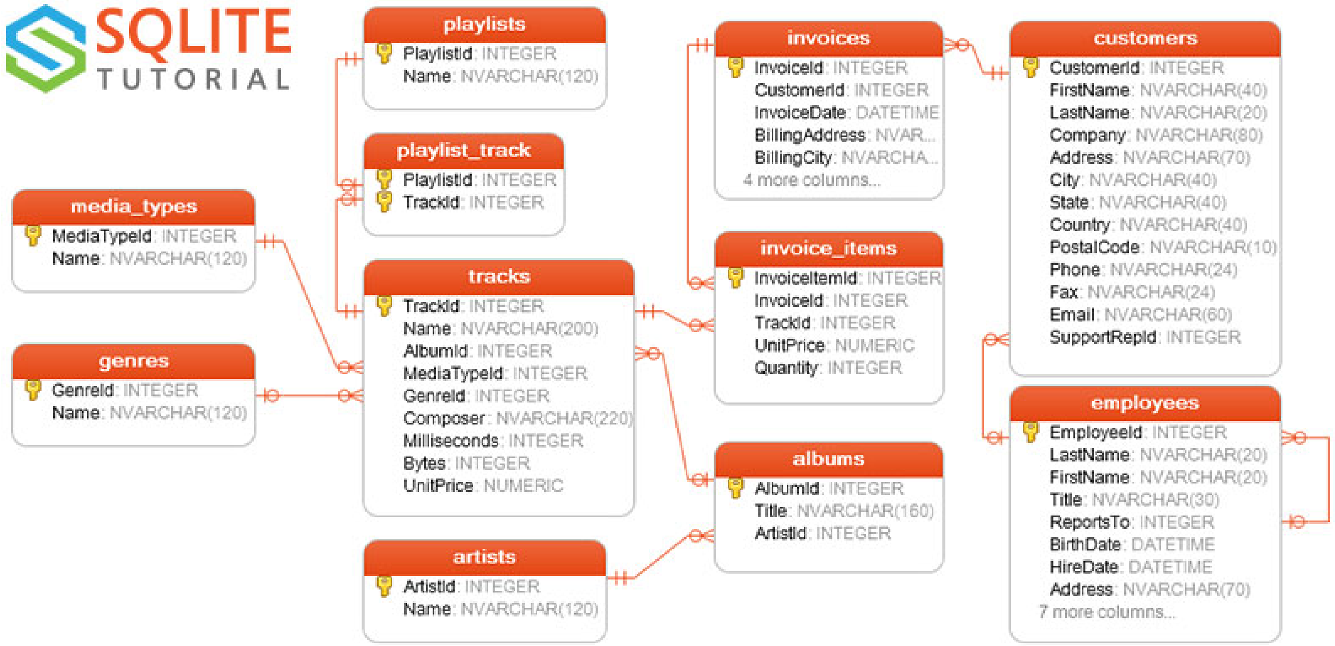
Go to: 1

Show SQL submitted by: Application

```
PRAGMA foreign_keys = "1";
PRAGMA encoding
SELECT type, name, sql, tbl_name FROM sqlite_master;
SELECT COUNT(*) FROM (SELECT rowid,* FROM `total_members`  ORDER BY `rowid` ASC);
SELECT rowid,* FROM `total_members`  ORDER BY `rowid` ASC LIMIT 0, 50000;
```

Introducing SQLite: official tutorial

a) **Data Model**



playlists
PlaylistId: INTEGER
Name: NVARCHAR(120)

playlist_track
PlaylistId: INTEGER
TrackId: INTEGER

media_types
MediaTypeId: INTEGER
Name: NVARCHAR(120)

genres
GenreId: INTEGER
Name: NVARCHAR(120)

tracks
TrackId: INTEGER
Name: NVARCHAR(200)
AlbumId: INTEGER
MediaTypeId: INTEGER
GenreId: INTEGER
Composer: NVARCHAR(220)
Milliseconds: INTEGER
Bytes: INTEGER
UnitPrice: NUMERIC

artists
ArtistId: INTEGER
Name: NVARCHAR(120)

invoices
InvoiceId: INTEGER
CustomerId: INTEGER
InvoiceDate: DATETIME
BillingAddress: NVAR...
BillingCity: NVARCHA...
4 more columns...

invoice_items
InvoiceItemId: INTEGER
InvoiceId: INTEGER
TrackId: INTEGER
UnitPrice: NUMERIC
Quantity: INTEGER

albums
AlbumId: INTEGER
Title: NVARCHAR(160)
ArtistId: INTEGER

customers
CustomerId: INTEGER
FirstName: NVARCHAR(40)
LastName: NVARCHAR(20)
Company: NVARCHAR(80)
Address: NVARCHAR(70)
City: NVARCHAR(40)
State: NVARCHAR(40)
Country: NVARCHAR(40)
PostalCode: NVARCHAR(10)
Phone: NVARCHAR(24)
Fax: NVARCHAR(24)
Email: NVARCHAR(60)
SupportRepId: INTEGER

employees
EmployeeId: INTEGER
LastName: NVARCHAR(20)
FirstName: NVARCHAR(20)
Title: NVARCHAR(30)
ReportsTo: INTEGER
BirthDate: DATETIME
HireDate: DATETIME
Address: NVARCHAR(70)
7 more columns...

b) **Sample Data Values** [Download link]

SQL with Python

Introducing SQLalchemy

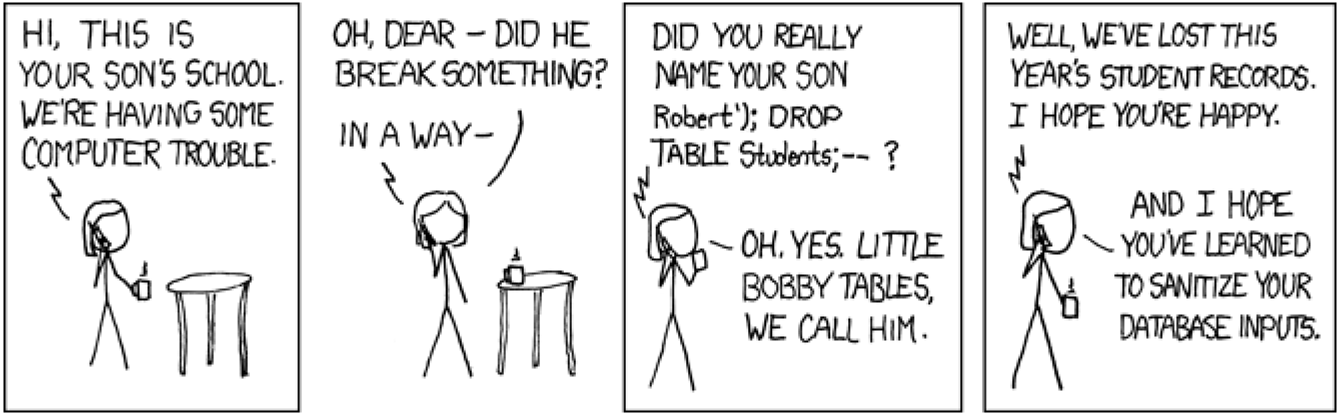
Introducing SQLalchemy

- **How to execute SQL queries from Python code ?**
  - Answer #1: using the *built-in python* *sqlite3 API*

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor()

# Create table
c.execute("""CREATE TABLE stocks
    (date text, trans text, symbol text, qty real, price real)""")

# Insert a row of data
c.execute("""INSERT INTO stocks
    VALUES ('2006-01-05','BUY','RHAT',100,35.14)""")

# Save (commit) the changes and then close the connection
conn.commit()
conn.close()
```

Introducing SQLalchemy (continued)

➡ danger: SQL injection!


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

(from https://xkcd.com/327/)

Introducing SQLalchemy (continued)

- **How to execute SQL queries from Python code ?**
  - Answer #2: using a *Object Relational Mapper*

- Python library (API🔗) providing a **high-level** interface to relational databases
  - **Tables** are mapped to Python *classes*
  - **Records** are mapped to Python *objects*
  - **Mappings** are done **semi-automatically** using SQLalchemy's `declarative_base` objects

Introducing SQLalchemy (continued): implementing a model

- How to create entity tables ?

```
#file base.py
from sqlalchemy import Column,ForeignKey,Integer,String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship
from sqlalchemy import create_engine,PrimaryKeyConstraint

Base = declarative_base()

class Actor(Base):
  __tablename__ = 'actor'
  # Each column is also a python instance attribute
  actor_id = Column(Integer, primary_key=True)
  name     = Column(String(250), nullable=False)

class Film(Base):
  __tablename__ = 'film'
  film_id = Column(Integer, primary_key=True)
  title   = Column(String(250), nullable=False)
  gender  = Column(String(250))
```

Introducing SQLalchemy (continued)

- How to create relation tables ?

```
#file base.py (continued)

class Actor_film_mapping(Base):
  __tablename__ = 'actor_film_mapping'
  actor = relationship(Actor)
  film  = relationship(Film)
  actor_id = Column(Integer, ForeignKey('actor.actor_id'))
  film_id  = Column(Integer, ForeignKey('film.film_id'))
  __table_args__ = (
    PrimaryKeyConstraint('actor_id', 'film_id'),
  )

# Create an engine that stores data in the local
# directory's example.db file.
engine = create_engine('sqlite:///example.db')

# Create all tables in the engine. This is equivalent
# to "Create Table" statements in raw SQL.
Base.metadata.create_all(engine)
```

Introducing SQLalchemy (continued): using a model

- How to use the generated tables ?

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from base import Actor, Film, Actor_film_mapping, Base

engine = create_engine('sqlite:///example.db')
# Bind the engine to the metadata of the Base
# class from base.py
Base.metadata.bind = engine

DBSession = sessionmaker(bind=engine)
session = DBSession()

# Insert an actor in the actor table
actor1 = Actor(name='Smith, John')
session.add(actor1)
session.commit()
#(...) session.rollback() # if needed
```

Introducing SQLalchemy (continued)

- How to select all movies where an actor plays ?

```python
from base import Actor, Film, Actor_film_mapping, Base
from sqlalchemy.orm import sessionmaker
from sqlalchemy import create_engine

engine = create_engine('sqlite:///example.db')

Base.metadata.bind = engine
DBSession = sessionmaker(bind = engine)
session = DBSession()

actor1 = session.query(Actor).first()
print(actor1.name) #'Smith, John'

session.query(Actor_film_mapping)\
.filter(Actor_film_mapping.actor_id == actor1.actor_id).all()
[<base.Actor_Film_mapping object at 0x2ee3cd0>]
```

See the full Query API 🔗.

Exercise Sheet #6 - SQLite

Exercise Sheet #7 - SQLalchemy

Thank you!

Slideshow created using remark.