CS107 / AC207

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 20

Thursday, November 11th 2021

Fabian Wermelinger
Harvard University

RECAP OF LAST TIME

- `python` internals:
  - Code objects
  - The interpreter and the evaluation loop
  - Frame objects
  - Generator objects
  - Traceback objects
- `python` lists and `numpy` arrays

OUTLINE

- Introduction to databases
- Data models
- `SQLite` and `python`

INTRODUCTION TO DATABASES

Why learn databases?

- SQL (**S**tructured **Q**uery **L**anguage) is very popular and will remain so for some time.
- You will therefore encounter databases in your career.
- Unlike Microsoft Excel/Access, SQL databases *scale*.
- A robust SQL database allows you to do work with terabytes of data, multiple related tables and thousands of columns.
- SQL integrates well with `python` or `R`.
- Basic knowledge of SQL beyond data analysis is helpful since it shows up in almost all web applications and content management systems.

MOTIVATION

- It is very hard to implement a database well. Understanding how they work is important.
- Data storage/management/wrangling are not just database concerns. Packages such as `pandas` and `dplyr` require a similar understanding.
- When you need to integrate a database system in your program, it is important that you are able to make decisions whether such a data management system performs well and can scale to large data:
  - You should understand the query performance of your transactions.
  - Is your application dominated by single transactions or do you need bulk processing of data? The former would rely on a transactional database. Analytical databases are often supported by hardware and are more suitable for the bulk data analysis. A comparison of transactional and analytical databases is given here.

WHAT KIND OF DATA ACCESS DO YOU NEED?

The answer to this question depends on your problem and the resources you have available.

Database Genre	*Examples*
Relational	SQL and its derivatives
Document oriented	CouchDB, MongoDB
Key-value	Riak, Memcached, leveldb
Graph oriented	Neo4J
Columnar	HBase

DATA MODELS

- Database management systems (DBMS) organize and structure data in a specific way. The access techniques and data structures used for this task are called the *data model* of the DBMS.
- One of the most influential data models (still widely used today) is the *relational data model*. It is the data model used by SQL. This data model has evolved from the concepts of some earlier data models (some in use today):

  File management systems

  All data is stored permanently in files on disk. This is not a data model because individual files *are unaware* of each other. It is still widely used today, an example is the Unix hierarchical file system.

  Hierarchical databases

  An important early data model to manage large lists of assembly parts (e.g. automobile industry) which is organized into assembly groups that decrease in size as we go down the hierarchy. E.g., a car is divided into engine, body and chassis which are in turn further divided into sub-assembly groups and so on. This introduces a parent/child relationship among assemblies and sub-assemblies. Similar to a tree structure.

  Network databases

  Extend hierarchical databases by allowing multiple parent/child relationships between data entries (called records). E.g., connect customer records and product records with order records.

RELATIONAL DATA MODEL

- The central element in the *relational data model* is a *table*.
- A table is a grid of rows and columns that store *data*.
- Each row holds a collection of columns, also called a tuple.
- Each column contains data of a *specified type*. Common data types are numbers, characters and dates.
- Multiple tables relate to each other via common values in columns. We call these identifiers *keys*.
- SQL is used to define the structure of a table and the relations among tables. It is further used to *extract* or *query* data from tables.
- In contrast to `python` (imperative language: describes *how* a program should accomplish a task), SQL is a *declarative language* describing *what* a program should accomplish (without the how).

RELATIONAL DATA MODEL

Example: student class enrollments

- ***Table 1:*** `student_enrollment`

student_id	class_id	class_section	semester
CHRISPA004	COMPSCI107	3	Fall 2021
DAVISHE010	COMPSCI107	3	Fall 2021
ABRILDA002	APCOMP207	40	Fall 2021
DAVISHE010	APCOMP207	40	Fall 2021
RILEYPH002	APCOMP207	40	Fall 2021

- ***Table 2:*** `students`

student_id	first_name	last_name	dob
ABRILDA002	Abril	Davis	1999-01-10
CHRISPA004	Chris	Park	1996-04-10
DAVISHE010	Davis	Hernandez	1987-09-14
RILEYPH002	Riley	Phelps	1996-06-15

- ***Table 3:*** `class_student_list`

class_id	first_name	last_name
COMPSCI107	Davis	Hernandez
COMPSCI107	Chris	Park
APCOMP207	Abril	Davis
APCOMP207	Davis	Hernandez
APCOMP207	Riley	Phelps

- Table 1 on the left shows student enrollments in two different classes.
- This table does not contain any details about students or classes.
- The *relation* for this data is established through the *unique keys* in the columns with attribute `student_id` and `class_id`.
- The relations between tables allows us to create new rows with data obtained from both tables, such as mapping class enrollments to student names.

RELATIONAL DATA MODEL

Example: student class enrollments

- A table for class information would work the same way with a `class_id` column and several other columns with data about the classes.
- Note how the different tables are organized such that they each contain a main *entity* that the database manages. The students' names and birth dates are stored in a separate table to reduce *redundant data*. *If a student signs up for multiple classes, we store the student data only once and save storage space on the disk*.
- This relation among tables and organization into main entities is a powerful feature of the relational data model.

KEY-VALUE MODEL

- A key-value data model uses a dictionary like data structure (hash table for fast look-up).
- It is not a relational data model. Instead, keys uniquely identify data records in a dictionary.
- Can be more flexible than the relational data model and follows more closely modern concepts like object oriented programming.
- Often require far less memory to store an equivalent relational database. This can result in better performance for certain applications.
- A popular example of a key-value database is redis.

DOCUMENT MODEL

- A document data model stores the data in nested records.
- In contrast to the relational data model, where the description of an object is obtained by multiple tables, the document model stores all data related to the object in one record.
- Example encodings for records (documents) are XML, YAML or JSON.
- A popular example of a document-oriented database is MongoDB.

WORKFLOW FOR (RELATIONAL) DATABASES

Components to a database:

1. ***Hardware:*** is where the data is physically stored. Databases are usually operated in a client/server model. The physical storage of the data usually happens on the server side. Clients communicate through user interfaces. Examples are web forms for order placement or the iTunes app.
2. ***Software:*** the main component that implements a data model with a corresponding API.
3. ***Data:*** the essence of a database. Metadata is description of data similar to the nutrition label on food products. Metadata is what is usually stored in a table for example. It may contain a reference to large data (e.g. a file path) if there is such an association.
4. ***Procedures:*** are general instructions to interact with the database such as the setup and installation of a DBMS, login and logout from the server, creating backups, etc.
5. ***Database access language:*** provides commands to access, insert, delete or update data stored in the database; other commands include creation and management of tables. Similar to the shell, a DBMS usually provides a command line interface or scripting facilities.

WORKFLOW FOR (RELATIONAL) DATABASES

Database access language:

- We need a language to help us easily *query* items in a database.
- It should provide simple *verbs* to describe what to do (*declarative* language).
- Pandas is a library for `python` that allows users to work with data structures and relational databases.
- The `dplyr` package offers data manipulation tools for the `R` programming language, including a tool set for the manipulation of relational databases.

WORKFLOW FOR (RELATIONAL) DATABASES

Database access language:

A notebook by T. Augspurger contains a tabular comparison of *verbage* between `pandas` and `dplyr`. The following table is a modification by D. Sondak:

Verb	*dplyr*	*pandas*	*SQL*
QUERY/SELECTION	`filter()` (and `slice()`)	`query()` (and `loc[]`, `iloc[]`)	SELECT WHERE
SORT	`arrange()`	`sort()`	ORDER BY
SELECT-COLUMNS/PROJECTION	`select()` (and `rename()`)	`[]` (`__getitem__`) (and `rename()`)	SELECT COLUMN
SELECT-DISTINCT	`distinct()`	`unique()`, `drop_duplicates()`	SELECT DISTINCT COLUMN
ASSIGN	`mutate()` (and `transmute()`)	`assign()`	ALTER/UPDATE
AGGREGATE	`summarise()`	`describe()`, `mean()`, `max()`	None, AVG(),MAX()
SAMPLE	`sample_n()` and `sample_frac()`	`sample()`	implementation dep., use RAND()
GROUP-AGG	`group_by` / `summarize`	`groupby` / `agg`, `count`, `mean`	GROUP BY
DELETE	?	`drop` /masking	DELETE/WHERE

SQLite AND python

From https://sqlite.org:

SQLite is a C-language library that implements a *small*, *fast*, self-contained, high-reliability, full-featured, *SQL database engine* (it does not require a separate server process). SQLite is the *most used database engine in the world*. SQLite is *built into all mobile phones* and most computers and comes bundled inside countless other applications that people use every day.

- We need a way of querying a relational database. There are many languages available, we will focus on SQL (Structured Query Language) because of its long lasting history and the arguments above.
- NoSQL databases are gaining in popularity. However, we will stick with relational databases in this class.

SQLite AND python

- We will use SQLite. The following are useful references:
  - SQLite homepage.
  - A thorough guide to SQLite database operations in python.
  - SQL tutorial.
- SQLite is built into python. It implements a standard database API for all databases called DB-API 2.0. It is specified in PEP 249; a brief introduction can be found here.
- There is an even higher level python API called SQLAlchemy.
- You can install SQLite on your system if you need it. Linux distros have built packages in their repositories or you can find pre-built packages at https://www.sqlite.org/download.html.
- The SQLite browser may be useful as well: https://sqlitebrowser.org/. Most Linux distros will have this tool in their repo as well.
- A command line interface is available as well: https://www.sqlite.org/cli.html. Linux distros will have this available as well.

SQLite AND python

The plan for the following slides:

- We will work through an example using the `sqlite3` package in `python`.
- This package will allow us to execute basic SQLite commands in `python` to build and manipulate our database.
- We will start by creating a SQL database and work up from there.
- ***Note:*** we could work with `pandas` alternatively to make our lives easier. We will work with SQLite commands directly to internalize the basic commands and use some features of `pandas` next week.

SQLite AND python

Core SQL commands:

Command	*Description*
`SELECT`	Select a table name
`INSERT`	Insert data into the table
`UPDATE`	Change data values in the table
`DELETE`	Delete data in the table

We can string these commands together to perform our basic operations on the database.

SQLite AND python

Structural SQL commands:

Command	*Description*
CREATE	Create a table in the database
DROP	Delete a table in the database
ALTER	Add, delete or modify columns in an existing table

These commands are used to modify the structure of a table in the database.

SQLite AND python

Starting a database:

- Our goal is to create a database with tables of presidential candidates and their contributors using data from 2008 (`candidates.txt`).
- We start by importing `sqlite3` :

```
1 import sqlite3
```

This package is shipped with the `python` standard library.

- The first thing we need to establish is a connection to the database of a given filename. A new database will be created if it does not exist.
- To perform operations on our database, we need to get a cursor object to it by calling the `.cursor()` method on the connection object.
- We need support for *foreign keys* in our database. We will see what those are in just a few slides. Support for foreign keys is disabled in SQLite by default, the following command enables it through the cursor object:

```
1 cursor.execute('PRAGMA foreign_keys=1')
```

SQLite AND python

Creating a table:

- We have established a connection to our database, which does not have any tables yet.
- We will create a table for the presidential candidates with the following columns: `id`, `first_name last_name`, `middle_initial`, `party`.
- The data type of the `id` column must be an integer and the rest shall be strings. We can simply achieve this with the following code:

```
cursor.execute('''CREATE TABLE candidates (
id INTEGER PRIMARY KEY NOT NULL,
first_name TEXT,
last_name TEXT,
middle_initial TEXT,
party TEXT NOT NULL)''')

db.commit()  # commit the changes to the database
```

SQLite AND python

What did we just do?

1. `cursor.execute()` runs the SQLite command, which we pass in as a string.
2. The `id` column is special: It contains integer values and is tagged as `PRIMARY KEY`. This means that values in this column are *unique* and cannot have `NULL` values (missing data or empty cell).

   A table can only have one `PRIMARY KEY`.
3. `first_name`, `last_name`, `middle_initial` are all columns with `TEXT` values *of unlimited length*.

   Note that SQL has other data types such as `VARCHAR(N)` and `CHAR(N)`. `VARCHAR(N)` allows variable text lengths up to `N` characters in length and `CHAR(N)` expects text of *exactly* `N` characters. You can also have `REAL/FLOAT` (8-byte floating point) or `BLOB` types for large binary data.
4. The `party` column is also of `TEXT` type and cannot have `NULL` as an additional constraint.

SQLite AND python

SQL conventions:

Note that we have followed a convention wherein *SQL commands are issued in capital letters* and *table fields are written in lowercase text*.

Commit your changes:

Always commit your changes to your database! If you fail to do that, you will lose them when you close the database. See the `.rollback()` method of a connection object.

```
1 db.commit()  # commit the changes to the database
```

SQLite AND python

Adding values to a table:

- Adding values into a table is achieved with the `INSERT` command:

```
1 cursor.execute('''INSERT INTO candidates
2 (id, first_name, last_name, middle_initial, party)
3 VALUES (?, ?, ?, ?, ?)''', (16, 'Mike', 'Huckabee', '', 'R'))
4
5 db.commit()  # commit the changes to the database
```

- Note the *declarative* style of the language. Commands express the "what" instead of the "how".

SQLite AND python

Adding values to a table:

- Adding values into a table is achieved with the `INSERT` command:

```
cursor.execute('''INSERT INTO candidates
(id, first_name, last_name, middle_initial, party)
VALUES (?, ?, ?, ?, ?)''', (16, 'Mike', 'Huckabee', '', 'R'))

db.commit()  # commit the changes to the database
```

- The " ? " in the declaration above have a special meaning. We could also use the `python` string `.format()` method to insert parameters. This would leave our code more vulnerable to SQL injection attacks. Two reasons why you should use the `?` placeholders are:

1. They leave the burden of correctly encoding and escaping data items to the database module.
2. They prevent injection of arbitrary SQL syntax into a database query. See https://bobby-tables.com/python for a guide on preventing SQL injection attacks.

SQLite AND python

SQL queries:

- Let us first add another two entries:

```
cursor.execute('''INSERT INTO candidates
(id, first_name, last_name, middle_initial, party)
VALUES (?, ?, ?, ?, ?)''', (32, 'Ron', 'Paul', '', 'R'))

cursor.execute('''INSERT INTO candidates
(id, first_name, last_name, middle_initial, party)
VALUES (?, ?, ?, ?, ?)''', (20, 'Barack', 'Obama', '', 'D'))

db.commit()  # commit the changes to the database
```

SQLite AND python

SQL queries:

- Getting all rows columns from a table:

```
cursor.execute("SELECT * FROM candidates")
rows = cursor.fetchall()
print(f'All rows and columns: got {len(rows)} rows')
for row in rows:
    print(row)
```

```
All rows and columns: got 3 rows
(16, 'Mike', 'Huckabee', '', 'R')
(20, 'Barack', 'Obama', '', 'D')
(32, 'Ron', 'Paul', '', 'R')
```

- Selecting specific content with `WHERE`:

```
cursor.execute("SELECT * FROM candidates WHERE first_name = 'mike'")
rows = cursor.fetchall()
print(f"Looking for 'mike': got {len(rows)} rows")
for row in rows:
    print(row)
```

```
Looking for 'mike': got 0 rows
```

Note the case-sensitive search. You need to use `LOWER()` or `UPPER()` to avoid case sensitivity.

SQLite AND python

SQL queries:

- Select a specific column:

```
cursor.execute("SELECT first_name FROM candidates")
rows = cursor.fetchall()
print(f"Looking for first_name: got {len(rows)} rows")
for row in rows:
    print(row)
```

```
Looking for first_name: got 3 rows
('Mike',)
('Barack',)
('Ron',)
```

SQLite AND python

Adding another table:

- We add another table to our database and introduce a few new SQL commands and ideas.
- The new table contains data on supporters and their contributions to each candidate. This will introduce a *relation* to our first table.

```
cursor.execute('''CREATE TABLE contributors (
id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
last_name TEXT,
first_name TEXT,
middle_name TEXT,
street_1 TEXT,
street_2 TEXT,
city TEXT,
state TEXT,
zip TEXT,
amount FLOAT(7,3),
date DATETIME,
candidate_id INTEGER NOT NULL,
FOREIGN KEY(candidate_id) REFERENCES candidates(id))''')
db.commit()  # commit the changes to the database
```

SQLite AND python

Adding another table:

- `AUTOINCREMENT`: the `id` for a new entry to the table will be automatically generated. You do not need to enter this manually. The counter starts at 1 and increments by 1.
- `FLOAT(7,3)`: floating point number with 7 digits, 3 of them after the decimal point.
- `DATETIME`: returns the date in the format `YYYY-MM-DD HH:MM:SS`.
- `FOREIGN KEY`: allows us to link the `contributors` table with the `candidates` table.

SQLite AND python

Adding another table:

Tables are related to one another by the data they contain. The relational data model uses *primary keys* and *foreign keys* to represent these relationships among tables.

- A *primary key* is a column or combination of columns in a table whose values uniquely identify each row of the table. A table has *only one primary key*.
- A *foreign key* is a column or combination of columns in a table whose values are a primary key value for some other table. A table *can contain more than one foreign key*, linking it to one or more other tables.
- A primary key/foreign key combination creates a parent/child relationship between the tables that contain them.

SQLite AND python

Adding another table:

In our example, we declare a child/parent relationship between the `candidate_id` attribute in the `contributors` table and the `id` attribute in the `candidates` table through the statement:

```
cursor.execute('''CREATE TABLE contributors (
id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
last_name TEXT,
first_name TEXT,
middle_name TEXT,
street_1 TEXT,
street_2 TEXT,
city TEXT,
state TEXT,
zip TEXT,
amount FLOAT(7,3),
date DATETIME,
candidate_id INTEGER NOT NULL,
FOREIGN KEY(candidate_id) REFERENCES candidates(id))''')
```

SQLite AND python

Adding contributors en masse:

- If you have a list of contributors you can add them all at once using the `cursor.executemany()` method (***note:*** we omit the `id` column):

```
contributors = [
("Agee", "Steven", "", "549 Laurel Branch Road", "", "Floyd", "VA",
    int(24091), 500.0, '2007-06-30', 16),
("Buck", "Jay", "K.", "1855 Old Willow Rd Unit 322", "", "Northfield", "IL",
    int(600932918), 200.0, '2007-09-12', 20),
("Choe", "Hyeokchan", "", "207 Bridle Way", "", "Fort Lee", "NJ",
    int(70246302), -39.50, '2008-04-21', 32),
]

cursor.executemany('''INSERT INTO contributors
(last_name, first_name, middle_name, street_1, street_2, city, state, zip, amount, date, candidate_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)''', contributors)
db.commit()

cursor.execute('SELECT last_name FROM contributors WHERE amount <= 200')
for row in cursor.fetchall():
    print(row)
```

```
('Buck',)
('Choe',)
```

SQLite AND python

Adding contributors with an `id` mismatch into the `candidates` table:

- Adding contributors with non-existent candidate identifier:

```
cursor.execute('''INSERT INTO contributors
(last_name, first_name, middle_name, street_1, street_2, city, state, zip, amount, date, candidate_id)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)''',
("Buckler", "Steve", "", "24351 Armada Dr.", "", "Dana Point", "CA", int(926291), 50, '2007-07-30', 34))
```

```
Traceback (most recent call last):
  File "/home/fabs/harvard/CS107/lecture20/presidential.py", line 121, in <module>
    main()
  File "/home/fabs/harvard/CS107/lecture20/presidential.py", line 112, in main
    cursor.execute('''INSERT INTO contributors
sqlite3.IntegrityError: FOREIGN KEY constraint failed
```

- The `traceback` is clear: there is no candidate with `candidate_id` equal to `34`.
- The foreign key *prevents* us from entering invalid data. The exception above is raised because we have set

```
cursor.execute('PRAGMA foreign_keys=1')
```

when we created our database at the beginning.

SQLite AND python

Summary:

- SQL is based on the *relational data model* that organizes the data in a database as a *collection of tables*.
- Each table has a table name that *uniquely* identifies it.
- Each table has one or more columns, which are arranged in a specific, left-to-right order.
- Each table has zero or more rows, each containing a single data value in each column.
- All data values in a given column *have the same data type* and are drawn from a set of legal values called the *domain of the column*.
- A primary key/foreign key combination creates a parent/child relationship between the tables that contain them.
- When done working with our `sqlite3` database, we close it with:

```
1 db.close()
```

RECAP

- Introduction to databases
- Data models
- `SQLite` and `python`