

SYSTEMS DEVELOPMENT FOR COMPUTATIONAL SCIENCE

LECTURE 21

Fabian Wermelinger

Harvard University

CS107 / AC207

Thursday, November 10th 2022

LAST TIME

- Python internals:
 - Code objects and bytecode
 - The interpreter and the evaluation loop
 - Frame objects
 - Generator objects
- Why are Python built-in lists slow and NumPy arrays fast?

TODAY

Main topics: *Databases, SQL and SQLite*

Details:

- Introduction to databases
- Data models
- Structured query language (SQL)
- Python and SQLite

INTRODUCTION TO DATABASES

Why learn databases?

- *Last time:* Python is the worlds most popular programming language in 2022, but → "employers love to see SQL skills as well.."
- SQL (Structured Query Language) 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.

<i>Database Genre</i>	<i>Examples</i>
Relational	SQL and its derivatives (PostgreSQL , MariaDB , MySQL and others)
Document oriented	CouchDB , MongoDB
Key-value	Riak , Memcached , leveldb , Amazon DynamoDB
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 (or *record*).
- 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**

1	student_id	class_id	class_section	semester
2	-----	-----	-----	-----
3	CHRISPA004	COMPSCI107	3	Fall 2021
4	DAVISHE010	COMPSCI107	3	Fall 2021
5	ABRILDA002	APCOMP207	40	Fall 2021
6	DAVISHE010	APCOMP207	40	Fall 2021
7	RILEYPH002	APCOMP207	40	Fall 2021

- **Table 2: students**

1	student_id	first_name	last_name	dob
2	-----	-----	-----	-----
3	ABRILDA002	Abril	Davis	1999-01-10
4	CHRISPA004	Chris	Park	1996-04-10
5	DAVISHE010	Davis	Hernandez	1987-09-14
6	RILEYPH002	Riley	Phelps	1996-06-15

- **Table 3: class_student_list**

1	class_id	first_name	last_name
2	-----	-----	-----
3	COMPSCI107	Davis	Hernandez
4	COMPSCI107	Chris	Park
5	APCOMP207	Abril	Davis
6	APCOMP207	Davis	Hernandez
7	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](#) or Amazon's [Dynamo](#) and the more recent [DynamoDB](#) systems.

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 is on the server side. Clients communicate through user interfaces. Examples are web forms for order placement on Amazon 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:

- To access the database a language is required to easily *query* items.
- 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.
- SQL is the most commonly used database access language for relational database management systems (RDBMS).

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:

<i>Verb (action)</i>	<i>dplyr</i>	<i>pandas</i>	<i>SQL</i>
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 as well as bundled inside countless other applications that people use every day.
- The language we use to query a relational database is SQL. 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 in the following.* Some 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 a 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

SQLite example in Python: `presidential.py`

- We will work through an example using the `sqlite3` package in Python.
- The `sqlite3` 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 (which is probably easier). We will work with SQLite commands directly to internalize the basic commands and use some features of Pandas in the exercise next week.

SQLITE AND PYTHON

Core SQL commands:

<i>Command</i>	<i>Description</i>
SELECT	Select a table name
INSERT	Insert data into the table
UPDATE	Change data values in the table
DELETE	Delete data in the table

- Basic commands to operate on a database.
- They are expressed as verbs → actions.
- In `sqlite3` we have to write them as strings.

SQLITE AND PYTHON

Structural SQL commands:

<i>Command</i>	<i>Description</i>
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 are basic commands used to modify the structure of a table in the database.

SQLITE AND PYTHON

Starting a database:

- The goal is to create a database with tables of presidential candidates and their contributors using data from 2008 (candidates.txt).

- Start by importing `sqlite3`:

```
1 import sqlite3
```

→ the 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 achieve this with the following code:

```
1 cursor.execute('''CREATE TABLE candidates (  
2 id INTEGER PRIMARY KEY NOT NULL,  
3 first_name TEXT,  
4 last_name TEXT,  
5 middle_initial TEXT,  
6 party TEXT NOT NULL)''')  
7  
8 db.commit() # commit the changes to the database
```

SQLITE AND PYTHON

What did we just do?

1. `cursor.execute('string')` 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 be** NULL as an additional constraint (it must have a value).

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 letters*.

Commit your changes to the database:

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:

```
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
```

- 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 to our candidates table:

```
1 cursor.execute('''INSERT INTO candidates
2 (id, first_name, last_name, middle_initial, party)
3 VALUES (?, ?, ?, ?, ?)''', (32, 'Ron', 'Paul', '', 'R'))
4
5 cursor.execute('''INSERT INTO candidates
6 (id, first_name, last_name, middle_initial, party)
7 VALUES (?, ?, ?, ?, ?)''', (20, 'Barack', 'Obama', '', 'D'))
8
9 db.commit() # commit the changes to the database
```

SQLITE AND PYTHON

SQL queries:

- You can get all rows and columns from a table with the statement:

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

```
1 All rows and columns: got 3 rows
2 (16, 'Mike', 'Huckabee', '', 'R')
3 (20, 'Barack', 'Obama', '', 'D')
4 (32, 'Ron', 'Paul', '', 'R')
```

SQLITE AND PYTHON

SQL queries:

- Or select specific content using **WHERE**:

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

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

- Note the case-sensitive search. You need to use **LOWER()** or **UPPER()** to avoid case sensitivity, for example:

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

```
1 Looking for 'mike': got 1 rows
2 (16, 'Mike', 'Huckabee', '', 'R')
```

SQLITE AND PYTHON

SQL queries:

- If you only need data from a specific column, you can name the column directly:

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

```
1 Looking for first_name: got 3 rows
2 ('Mike',)
3 ('Barack',)
4 ('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.

```
1 cursor.execute('''CREATE TABLE contributors (  
2 id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,  
3 last_name TEXT,  
4 first_name TEXT,  
5 middle_name TEXT,  
6 street_1 TEXT,  
7 street_2 TEXT,  
8 city TEXT,  
9 state TEXT,  
10 zip TEXT,  
11 amount FLOAT(7,3),  
12 date DATETIME,  
13 candidate_id INTEGER NOT NULL,  
14 FOREIGN KEY(candidate_id) REFERENCES candidates(id))''')  
15 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:

```
1 cursor.execute('''CREATE TABLE contributors (  
2 id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,  
3 last_name TEXT,  
4 first_name TEXT,  
5 middle_name TEXT,  
6 street_1 TEXT,  
7 street_2 TEXT,  
8 city TEXT,  
9 state TEXT,  
10 zip TEXT,  
11 amount FLOAT(7,3),  
12 date DATETIME,  
13 candidate_id INTEGER NOT NULL,  
14 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 because of the AUTOINCREMENT attribute):

```
1 contributors = [  
2 ("Agee", "Steven", "", "549 Laurel Branch Road", "", "Floyd", "VA",  
3  int(24091), 500.0, '2007-06-30', 16),  
4 ("Buck", "Jay", "K.", "1855 Old Willow Rd Unit 322", "", "Northfield", "IL",  
5  int(600932918), 200.0, '2007-09-12', 20),  
6 ("Choe", "Hyeokchan", "", "207 Bridle Way", "", "Fort Lee", "NJ",  
7  int(70246302), -39.50, '2008-04-21', 32),  
8 ]  
9  
10 cursor.executemany(''INSERT INTO contributors  
11 (last_name, first_name, middle_name, street_1, street_2, city, state, zip, amount,  
12 date, candidate_id) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)'', contributors)  
13 db.commit()  
14  
15 cursor.execute('SELECT last_name FROM contributors WHERE amount <= 200')  
16 for row in cursor.fetchall():  
17     print(row) # 'Buck' and 'Choe' contribute 200 or less
```

SQLITE AND PYTHON

Adding contributors with an id mismatch into the candidates table:

- Adding contributors with non-existent candidate identifier:

```
1 cursor.execute('''INSERT INTO contributors
2 (last_name, first_name, middle_name, street_1, street_2, city, state, zip,
3 amount, date, candidate_id) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)''',
4 ("Buckler", "Steve", "", "24351 Armada Dr.", "", "Dana Point", "CA", int(926291),
5 50, '2007-07-30', 34))
```

```
1 Traceback (most recent call last):
2   File "/harvard/CS107/lecture21/presidential.py", line 121, in <module>
3     main()
4   File "/harvard/CS107/lecture21/presidential.py", line 112, in main
5     cursor.execute('''INSERT INTO contributors
6     sqlite3.IntegrityError: FOREIGN KEY constraint failed
```

- The foreign key *prevents* us from *entering invalid data* (there is no candidate_id==34). The exception above is raised because we have set

```
1 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 the sqlite3 database, *close* it with:

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

RECAP

- Introduction to databases
- Data models
- Structured query language (SQL)
- Python and SQLite

Further reading:

- E. F. Codd, "A relational model of data for large shared data banks", ACM, 1970 (<https://dl.acm.org/doi/10.1145/362384.362685>, pdf also in class repository)
- James R. Groff, Paul N. Weinberg and Andrew J. Oppel, "SQL The Complete Reference", McGraw-Hill, 3rd edition, 2010