Topic 11
Databases

CS 1MD3 • Introduction to Programming
Winter 2018

Dr. Douglas Stebila

McMaster University

Databases vs Programming Languages

- Like programming languages, databases allow us to process and store data. However:
  - data is stored in dedicated files and loaded as needed: data can be much larger than the available memory (retailer catalogue, bank accounts, personnel records); there is no need to explicitly open files and read data
  - data can be accessed "simultaneously" by "clients", either locally or remotely
  - data is atomically updated: data is either stored or not, but never becomes corrupt, even in case of failure (disk error, network disconnection)
  - data must be structured in specific ways, e.g. as tables in relational databases
  - complex queries can be written in a dedicated query language

Relational Databases

- Data items and their relationship is stored in tables.
- A **table** is a collection of **records** (a.k.a. rows, objects, entities) with **fields** (a.k.a. values, columns, attributes).

Example scenario: video rental store

- We want to create a database for running a video rental store

- We need to track:
  - All the movies we have available to rent
  - Our customers
  - Which customers have rented which movies

- We'll make three tables:
  - movies
  - customers
  - rentals

Example table: movies

movieid	title	genre	rating
101	Casablanca	drama romance	PG
102	Back to the Future	comedy adventure	PG
103	Monsters, Inc	animation comedy	G
104	Field of Dreams	fantasy drama	PG
5793	Life of Brian	comedy	R
7442	12 Angry Men	drama	PG

Database Schema

- A **database schema** specifies the names and types of the fields of each table
- Valid types:
  - INTEGER: up to 8 bytes integers
  - REAL: 8 byte floats
  - TEXT: Unicode strings
  - BOOLEAN: stored as 0 and 1
  - DATE: stored as numeric value

Example table: movies

movieid	title	genre	rating
101	Casablanca	drama romance	PG
102	Back to the Future	comedy adventure	PG
103	Monsters, Inc	animation comedy	G
104	Field of Dreams	fantasy drama	PG
5793	Life of Brian	comedy	R
7442	12 Angry Men	drama	PG

Our table's schema:

- `movieid INTEGER,`
- `title TEXT,`
- `genre TEXT,`
- `rating TEXT`

SQL: Structured Query Language

- Widely used database query language
- First proposed in 1974 for Codd's relational data model from 1970
- Can be used to interact with a database:
  - Fetch a set of records
  - Add data to a table
  - Modify data
  - Delete data
- The syntax is (mostly) independent of vendor

More advanced SQL

- SQL also supports **transactions** (begin, commit, rollback), authorization, stored functions, and more
- SQL is more general than the examples suggest, e.g. the result of an SQL query may be the input of another one

- Recently "NoSQL" databases for storing differently structured data and cloud storage are becoming popular

Database software

- There are many database programs available
  - Commercial: Oracle, Microsoft SQL Server
  - Open source: MySQL/MariaDB, PostgreSQL, SQLite, MongoDB
- Most use SQL for interacting with the database, although some have slight differences
- We will use SQLite in this course

SQLite

- SQLite is a widely used serverless relational database manager
  - **Serverless** means that database files are stored locally
    - Don't have to run any additional background service
    - Not as efficient for large databases or multiple users
    - But simpler and good enough for this course
- SQLite is preinstalled on macOS and Linux, as well as with Python, or can be downloaded at sqlite.org

Working with SQLite on the command line

```
$ sqlite3 videostore.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite>
...
sqlite> .quit
```

SQL command to create a table

movieid	title	genre	rating
101	Casablanca	drama romance	PG
102	Back to the Future	comedy adventure	PG
103	Monsters, Inc	animation comedy	G
104	Field of Dreams	fantasy drama	PG
5793	Life of Brian	comedy	R
7442	12 Angry Men	drama	PG

```
CREATE TABLE movies (movieid INTEGER,
title TEXT, genre TEXT, rating TEXT);
```

Field names are case sensitive

Keywords like CREATE TABLE and TEXT are case insensitive, but I like to use CAPS to make it clear what's a keyword and whats a field/value

All SQL commands terminated with ;

Special types of columns

Primary keys

- A primary key is a field that uniquely identifies a record.
- It is useful to have a primary key to cross reference tables (coming in a few slides).

Required fields

- We can make fields be required by adding the **REQUIRED** keyword.
- If you try to insert a record but omit a required field, the command fails.

CREATE TABLE

movieid	title	genre	rating
101	Casablanca	drama romance	PG
102	Back to the Future	comedy adventure	PG
103	Monsters, Inc	animation comedy	G
104	Field of Dreams	fantasy drama	PG
5793	Life of Brian	comedy	R
7442	12 Angry Men	drama	PG

```
CREATE TABLE movies (movieid INTEGER
PRIMARY KEY, title TEXT REQUIRED, genre
TEXT, rating TEXT);
```

INSERT: Adding data to a table

- To insert data into a table, we insert rows one at a time
- Need to list the fields names and values
  - Any omitted fields get a default value of NULL

```
INSERT INTO movies (movieid, title, genre, rating)
VALUES (101, 'Casablanca', 'drama romance', 'PG');

INSERT INTO movies (movieid, title, genre, rating)
VALUES (102, 'Back to the Future', 'comedy adventure',
'PG');
```

SELECT: Getting data from a table

- Have to specify four things:
  1. Which fields to return
  2. Which table
  3. Which rows to select
  4. What order to sort

```
SELECT * FROM movies;
SELECT title, rating FROM movies;
SELECT * FROM movies WHERE genre = "comedy";
SELECT title FROM movies WHERE genre LIKE
"%comedy%";
SELECT * FROM movies ORDER BY title ASC;
```

SELECT: Getting data from a table

- Have to specify four things:
  1. **Which fields to return**
  2. Which table
  3. Which rows to select
  4. What order to sort

```
SELECT * FROM movies;


SELECT title, rating FROM movies;
```

SELECT: Getting data from a table

- Have to specify four things:
  1. Which fields to return
  2. **Which table**
  3. Which rows to select
  4. What order to sort

```
SELECT * FROM movies;
```

SELECT: Getting data from a table

- Have to specify four things:
  1. Which fields to return
  2. Which table
  3. **Which rows to select**
  4. What order to sort

```
SELECT * FROM movies; (Selects all rows)

SELECT * FROM movies WHERE rating = "PG";

SELECT * FROM movies WHERE genre LIKE "%drama%";
```

Use LIKE and % for wildcard matching on strings

SELECT: Getting data from a table

- Have to specify four things:
  1. Which fields to return
  2. Which table
  3. **Which rows to select**
  4. What order to sort

```
SELECT * FROM movies WHERE (rating = "PG" or
rating = "G") AND (genre LIKE "%comedy%");
```

SELECT: Getting data from a table

- Have to specify four things:
  1. Which fields to return
  2. Which table
  3. Which rows to select
  4. **What order to sort**

```
SELECT * FROM movies ORDER BY title;
SELECT * FROM movies ORDER BY title ASC;
SELECT * FROM movies ORDER BY title DESC;
```

SQLite3 in Python

- Python has a library module through which an SQLite3 database can be accessed
- Need to specify the filename of the database file stored on the computer
- Basic pattern:
  - Open the database to get a database handle
  - Do operations on the database using the database handle
  - Close the database handle

```
import sqlite3
db = sqlite3.connect('videostore.db')
...
db.close()
```

SQLite3 in Python

- Python will automatically convert some Python data types to SQL data types:

SQL data type	Python data type
NULL	None
INTEGER	int
REAL	float
TEXT	str
DATE	str

SELECT from Python

- All SQL statements are passed as string to the database.
- A **cursor** is needed to iterate over the results of an SQL query

```
cur = db.cursor()
cur.execute('SELECT * FROM movies')
print(cur.fetchone())
print(cur.fetchall())
```

SELECT from Python

- We can use a for statement to loop through the results of a SELECT query:

```
cur = db.cursor()
rows = cur.execute('SELECT * FROM movies')
for row in rows:
  print(row)
```

SELECT from Python

- If we add the line in red below, the rows we get back will be dictionary-like objects where we can pull out a field by name

```
db.row_factory = sqlite3.Row
cur = db.cursor()
rows = cur.execute('SELECT * FROM movies')
for row in rows:
  print(row["title"])
```

INSERT from Python

- We can do any SQL queries we want from Python – SELECT, INSERT, CREATE TABLE, etc.
- We can insert values from Python variables into an SQL query
- Use cur.commit() to ensure changes are saved

```
cur = db.cursor()
m = 17
t = 'Beauty and the Beast'
g = 'animation musical comedy'
r = 'G'
cur.execute('INSERT INTO movies VALUES (?, ?, ?, ?)', (m, t, g, r))
cur.commit()
```

These act as placeholders that are **bound** to the following values.

Relationships between tables

Suppose we want keep track which customer rented which video

- Add a table to keep track of each customer
- Add a table for rentals to keep track of a **relationship** between customers and movies

Creating the customer table

customerid	name	address
101	Dennis Cook	123 Broadwalk
102	Doug Nickle	456 Park Place
103	Randy Wolf	789 Pacific Avenue
104	Amy Yao	321 St James Place
105	Robert Mwanri	654 Marvin Gardens
106	David Coggin	987 Charles Place

```
CREATE TABLE customers (customerid
INTEGER PRIMARY KEY, name TEXT REQUIRED,
address TEXT);
```

Creating the customer table

customerid	name	address
101	Dennis Cook	123 Broadwalk
102	Doug Nickle	456 Park Place
103	Randy Wolf	789 Pacific Avenue

```
CREATE TABLE customers (customerid
INTEGER PRIMARY KEY, name TEXT REQUIRED,
address TEXT);
```

Creating the rentals table for customer-movie relationships

customerid	movieid	daterented	datedue
103	104	3-12-2017	3-13-2017
103	5022	3-28-2017	3-29-2017
105	107	3-28-2017	3-29-2017

We will use the ID fields from each table (the primary key) to cross reference uniquely

```
CREATE TABLE rentals (customerid INTEGER
REQUIRED, movieid INTEGER REQUIRED,
daterented DATE, datedue DATE);
```

Working with relationships

customerid	movieid	daterented	datedue
103	104	2018-2-12	2018-3-12
103	5022	2018-2-28	2018-3-28
105	107	2018-2-29	2018-3-29

- If we do a SELECT on the rentals table, we only get the customerid, not the customer name etc., and only the movieid, not the movie name etc.
- Naively we would have to do an extra SELECT to look up each customer one at a time
- But SQL provides a way to do this all at once

Working with relationships

rentals

customerid	movieid	daterented	datedue
103	104	2018-2-12	2018-3-12
103	5022	2018-2-28	2018-3-28

customers

customerid	name	address
101	Dennis Cook	123 Broadwalk
102	Doug Nickle	456 Park Place
103	Randy Wolf	789 Pacific Ave.

Returns all fields from both tables
but only in rows which match on customerid

```
SELECT * FROM rentals, customers WHERE
customers.customerid = rentals.customerid
```

customerid in the customers table

customerid in the rentals table

Working with relationships

```
SELECT * FROM rentals, customers WHERE
customers.customerid = rentals.customerid
```

- The result is as if we had a single table with the rows from each table joined together when they match on the customerid field
- This is also called a JOIN operation or an INNER JOIN

customerid	movieid	daterented	datedue	name	address
103	104	2018-2-12	2018-3-12	Randy Wolf	789 Pacific Ave.
103	5022	2018-2-28	2018-3-28	Randy Wolf	789 Pacific Ave.

Working with relationships

rentals

customerid	movieid	daterented	datedue
103	104	2018-2-12	2018-3-12
103	5022	2018-2-28	2018-3-28

customers

customerid	name	address
101	Dennis Cook	123 Broadwalk
102	Doug Nickle	456 Park Place
103	Randy Wolf	789 Pacific Ave.

movies

movieid	title	genre	rating
101	Casablanca	drama romance	PG
104	Field of Dreams	fantasy drama	PG

```
SELECT * FROM rentals, customers, movies WHERE
(customers.customerid = rentals.customerid) AND
(movies.movieid = rentals.movieid)
```

Returns fields from all three tables which match across all three as described

UPDATE: Changing data

- We can update fields in one or more rows using an UPDATE command
  - Specify which fields should be changed
  - Specify which rows the change should be applied to using a WHERE condition

```
UPDATE movies
SET genre = 'horror', rating = 'R'
WHERE title = "Monsters, Inc";
```

UPDATE: Changing data

- Have to be careful
- If we omit the WHERE condition, the update applies to every row in the table
  - There is no undo with SQL databases!

```
UPDATE movies
SET genre = 'horror', rating = 'R';
```

Now everything is a horror movie! Scary!

DELETE: Removing data

- We can delete one or more rows using a DELETE command
  - Specify which rows the change should be applied to using a WHERE condition
- Have to be careful
  - If we omit the WHERE condition, the delete applies to every row in the table
    - There is no undo!

```
DELETE FROM movies WHERE movieid = 104;
```

DROP TABLE: Removing an entire table

- We can delete an entire table using the DROP TABLE command
  - Specify which table to delete by name
- Have to be careful
  - There is no undo!

```
DROP TABLE movies;
```

Using user input in a query

- What if we want to use user input inside a query?
  - E.g. search for all movies with a particular rating
- One approach:

```
rating = input("Enter a rating you'd like to
search for: ")
rows = cur.execute('SELECT * FROM movies WHERE
rating = "' + rating + '"')
for row in rows:
    print(list(row))
```

This ends up creating a query like:

```
SELECT * FROM movies WHERE rating = "PG"
```

Using user input in a query

- But what if the user enters a string with a quotation mark (") in it?
  - E.g. `Parental "Guidance"`

```
rating = input("Enter a rating you'd like to
search for: ")
rows = cur.execute('SELECT * FROM movies WHERE
rating = "' + rating + '"')
for row in rows:
    print(list(row))
```

This isn't a valid SQL query. The " before Guidance ends the string, so it thinks Guidance is an SQL keyword, and causes an error

This ends up creating a query like:

```
SELECT * FROM movies
WHERE rating = "Parental "Guidance""
```

Using user input in a query

- Even worse – what if the user enters something with " marks and then some valid SQL syntax
  - E.g. `PG" OR "X" = "X`

```
rating = input("Enter a rating you'd like to search for: ")
rows = cur.execute('SELECT * FROM movies WHERE rating = "' + rating + '"')
for row in rows:
    print(list(row))
```

The second condition "X" = "X" always evaluates to TRUE, so every row matches this query and we get back the whole table!

This ends up creating a query like:

```
SELECT * FROM movies
WHERE rating = "PG" OR "X" = "X"
```

SQL injection attacks

- When user input is mixed into a query string like this, applications become vulnerable to an SQL injection attack
  - This occurs when the attacker injects their own SQL commands as a string that accidentally gets executed by the database

- Can be used to see more data than intended
- Can be used to trick login checks and gain access without knowing the password
- Can be used to access data from other tables using JOIN commands
- Can even sometimes be used to modify or delete data!

SQL injection attacks

- What if the user enters the following?
  - `PG"; DELETE FROM movies WHERE "X" = "X`

```
rating = input("Enter a rating you'd like to search for: ")
rows = cur.execute('SELECT * FROM movies WHERE rating = "' + rating + '"')
f
```

Fortunately cur.execute will only execute one command and will complain if there are multiple commands

This ends up creating a query like:

SELECT * FROM movies WHERE rating = "PG"; DELETE FROM movies WHERE "X" = "X"

SQL injection attacks

- What if the user enters the following?
  - `5678`
  - `Fake"); DROP TABLE movies; --`

-- is how you start a comment in SQL; equivalent to Python's #

```
movieid = input("Enter movie ID of new movie: ")
title   = input("Enter title of new movie: ")
cur.executescript('INSERT INTO movies (movieid, title)
VALUES (' + movieid + ', "' + title + '")')
```

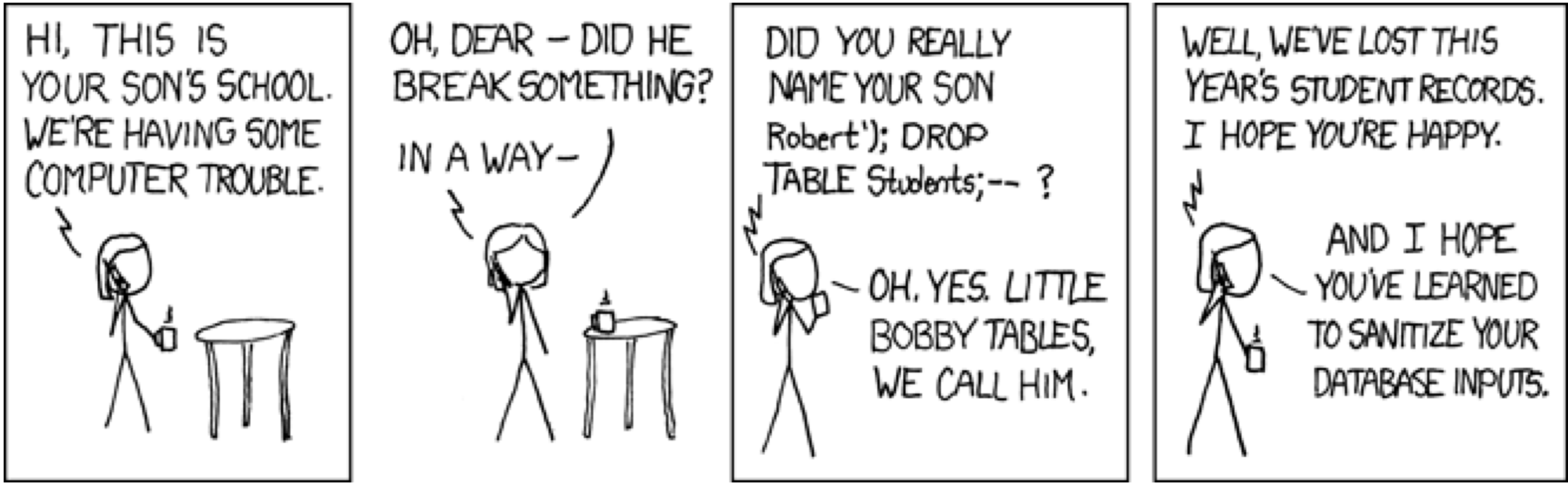
But there's a related function executescript which will execute multiple commands, and some programmers might use this intentionally or accidentally

This ends up creating a query like:

```
INSERT INTO movies (movieid, title) VALUES (5678,
"Fake"); DROP TABLE movies; --")
```

Text after ; ignored

Little Bobby Tables


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.

https://www.xkcd.com/327/

SQL injection attacks

- SQL injection attacks are the #1 security threat to web applications
  - According to Open Web Application Security Project
    - https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf
- Many large scale breaches due to SQL injection attacks
  - E.g. Yahoo had 450,000 login credentials stolen in 2012 using a JOIN-based SQL injection attack
    - http://www.zdnet.com/article/450000-user-passwords-leaked-in-yahoo-breach/
  - Many more attacks listed on Wikipedia
    - https://en.wikipedia.org/wiki/SQL_injection#Examples

Avoiding SQL injection attacks

- Sanitize user inputs by filtering out single and double quotation marks and other prohibited characters
  - Unreliable
  - Makes the O'Connor family unhappy
- Better approach: construct SQL queries by using special programming language features that safely insert user inputs

```
rows = cur.execute('SELECT * FROM movies
WHERE rating = ?', [rating])
```

This acts as a **placeholder** that is automatically **bound** to the subsequent value in a safe way.