STATS 507

SQL/NoSQL

SQL(Sequel) - Structured Query Language

Databases

Information, organized so as to make retrieval fast and efficient

Examples: Census information, product inventory, library catalogue

relational databases

https://en.wikipedia.org/wiki/Relational_database

So-named because they capture relations between entities

In existence since the 1970s, and still the dominant model in use today

Outside the scope of this course: other models (e.g., object-oriented)

https://en.wikipedia.org/wiki/Database_model

Textbook: *Database System Concepts* by Silberschatz, Korth and Sudarshan.

Relational DBs: pros and cons

Pros:

Natural for the vast majority of applications

Numerous tools for managing and querying

Cons:

Not well-suited to some data (e.g., networks, unstructured text)

Fixed schema (i.e., hard to add columns)

Expensive to maintain when data gets large (e.g., many TBs of data)

Fundamental unit of relational DBs: the record

Each entity in a DB has a corresponding **record**

- Features of a record are stored in **fields**
- Records with same "types" of fields collected into **tables**
- Each record is a row, each field is a column

ID	Name	UG University	Specialization	Birth Year	Age at Death
101010	John Bardeen	University of Wisconsin	Electrical Engineering	1908	82
314159	Albert Einstein	ETH Zurich	Physics	1879	76
21451	Ronald Fisher	University of Cambridge	Statistics	1890	72

Table with six fields and three records.

https://en.wikipedia.org/wiki/John_Bardeen

Fields can contain different data types

ID	Name	UG University	Specialization	Birth Year	Age at Death
101010	John Bardeen	University of Wisconsin	Electrical Engineering	1908	82
314159	Albert Einstein	ETH Zurich	Physics	1879	76
21451	Ronald Fisher	University of Cambridge	Statistics	1890	72

Unsigned int, String, String, String, Unsigned int, Unsigned int

Of course, can also contain floats, signed ints, etc. Some DB software allows categorical types (e.g., letter grades).

By convention, each record has a **primary key**

ID	Name	UG University	Specialization	Birth Year	Age at Death
101010	John Bardeen	University of Wisconsin	Electrical Engineering	1908	82
314159	Albert Einstein	ETH Zurich	Physics	1879	76
21451	Ronald Fisher	University of Cambridge	Statistics	1890	72

Primary key used to uniquely identify the entity associated to a record, and facilitates joining information across tables.

ID	PhD Year	PhD University	Thesis Title
101010	1936	Princeton University	Quantum Theory of the Work Function
314159	1905	University of Zurich	A New Determination of Molecular Dimensions
21451			

Relational Database Management Systems (RDBMSs)

Program that facilitates interaction with database is called RDBMS

Public/Open-source options:

MySQL, PostgreSQL, **SQLite**

Proprietary:

IBM Db2, Oracle, SAP, SQL Server (Microsoft)

We'll use SQLite, because it comes built-in to Python. More later.

Note: R also has a SQLite package, which largely mirrors the Python one: https://db.rstudio.com/databases/sqlite/

ACID: Atomicity, Consistency, Isolation, Durability

Atomicity: to outside observer, every transaction (i.e., changing the database) should appear to have happened “instantaneously”.

Consistency: DB changes should leave the DB in a “valid state” (e.g., changes to one table that affect other tables are propagated before the next transaction)

Isolation: concurrent transactions don’t “step on each other’s toes”

Durability: changes to DB are permanent once they are committed

Normalization - a technique for optimal data storage with integrity constraints
1NF
A table is 1NF if every cell contains a single value, not a list of values. 1NF also prohibits repeating group of columns such as item1, item2.... itemN.
2NF
A table is 2NF, if it is 1NF and every non-key column is fully dependent on the primary key. If the primary key is made up of several columns, every non-key column shall depend on the entire set and not part of it.
3NF
A table is 3NF, if it is 2NF and the non-key columns are independent of each others and are only dependent on primary key and nothing else.
4NF AND BEYOND
You can still take it beyond the 3rd Normal form although that is beyond the scope of this course.

SQL (originally SEQUEL, from IBM)

Structured **Q**uery **L**anguage (**S**tructured **E**nglish **QUE**ry **L**anguage)

Language for interacting with relational databases

- Not the only way to do so, but by far most popular
- Slight variation from platform to platform ("dialects of SQL")

Good tutorials/textbooks:
https://www.w3schools.com/sql/sql_intro.asp
O'Reilly books: *Learning SQL* by Beaulieu
SQL Pocket Guide by Gennick
Severance, Chapter 14: http://www.pythonlearn.com/html-270/book015.html

Examples of database operations

ID	Name	GPA	Major	Birth Year
101010	John Bardeen	3.1	Electrical Engineering	1908
500100	Eugene Wigner	3.2	Physics	1902
314159	Albert Einstein	4.0	Physics	1879
214518	Ronald Fisher	3.25	Statistics	1890
662607	Max Planck	2.9	Physics	1858
271828	Leonard Euler	3.9	Mathematics	1707
999999	Jerzy Neyman	3.5	Statistics	1894
112358	Ky Fan	3.55	Mathematics	1914

- Find names of all physics majors
- Compute average GPA of students born in the 19th century
- Find all students with GPA > 3.0

SQL allows us to easily specify queries like these (and far more complex ones).

Common database operations

SQL includes keywords for succinctly expressing all of these operations.

Extracting records: find all rows in a table - **select**

Filtering records: get only the records (rows) **where** the criterion matches

Sorting records: order by selected rows according to some field(s)

Adding/deleting records: insert new row(s) into a table or **delete** existing row(s)

Adding/deleting tables: create new or delete existing tables

Grouping records: group by rows according to some field

Retrieving records: SQL `SELECT` Statements

Basic form of a SQL `SELECT` statement:

```
SELECT [column names] FROM [table]
```

Example: we have table `customer` of customer IDs, names and companies

Retrieve all customer names: `SELECT name FROM customer`

Retrieve all company names: `SELECT company FROM customer`

References for naming convention: https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/ and http://leshazlewood.com/software-engineering/sql-style-guide/ for two people's (differing) opinions.

Table `student`

id	name	gpa	major	birth_year	pets	favorite_color
101010	John Bardeen	3.1	Electrical Engineering	1908	2	Blue
314159	Albert Einstein	4.0	Physics	1879	0	Green
999999	Jerzy Neyman	3.5	Statistics	1894	1	Red
112358	Ky Fan	3.55	Mathematics	1914	2	Green

```
SELECT id, name, birth_year FROM student
```

id	name	birth_year
101010	John Bardeen	1908
314159	Albert Einstein	1879
999999	Jerzy Neyman	1894
112358	Ky Fan	1914

Filtering records: SQL `WHERE` Statements

To further filter the records returned by a `SELECT` statement:

```
SELECT [column names] FROM [table] WHERE [filter]
```

Example: table `inventory` of product IDs, unit cost, and number in stock

Retrieve IDs for all products with unit cost at least $1:

```
SELECT id FROM inventory WHERE unit_cost >= 1
```

Note: Possible to do much more complicated filtering, e.g., regexes, set membership, etc. We'll discuss that more in a few slides.

Table `students`

id	name	gpa	major	birth_year	pets	favorite_color
101010	John Bardeen	3.1	Electrical Engineering	1908	2	Blue
314159	Albert Einstein	4.0	Physics	1879	0	Green
999999	Jerzy Neyman	3.5	Statistics	1894	1	Red
112358	Ky Fan	3.55	Mathematics	1914	2	Green

```
SELECT id, name FROM student WHERE birth_year >1900
```

id	name
101010	John Bardeen
112358	Ky Fan

`NULL` means Nothing!

Table `thesis`

id	phd_year	phd_university	thesis_title
101010	1936	Princeton University	Quantum Theory of the Work Function
314159	1905	University of Zurich	A New Determination of Molecular Dimensions
214511			
774477	1970	MIT	

```
SELECT id FROM thesis WHERE phd_year IS NULL
```

id
21451

`NULL` matches the *empty string*, i.e., matches the case where the field was left empty. Note that if the field contains, say, `' '`, then `NULL` will ***not*** match!

Ordering records: SQL `ORDER BY` Statements

To order the records returned by a `SELECT` statement:

```
SELECT [columns] FROM [table] ORDER BY [column] [ASC|DESC]
```

Example: table `inventory` of product IDs, unit cost, and number in stock

Retrieve IDs, # in stock, for all products, ordered by descending # in stock:

```
SELECT id, number_in_stock FROM inventory
ORDER BY number_in_stock DESC
```

Note: most implementations order ascending by default, but best to always specify, for your sanity and that of your colleagues!

Table `student`

id	name	gpa	major	birth_year	pets	favorite_color
101010	John Bardeen	3.1	Electrical Engineering	1908	2	Blue
314159	Albert Einstein	4.0	Physics	1879	0	Green
999999	Jerzy Neyman	3.5	Statistics	1894	1	Red
112358	Ky Fan	3.55	Mathematics	1914	2	Green

```
SELECT id, name, gpa FROM student ORDER BY gpa DESC
```

id	name	gpa
314159	Albert Einstein	4.0
112358	Ky Fan	3.55
999999	Jerzy Neyman	3.5
101010	John Bardeen	3.1

More filtering: `DISTINCT` Keyword

To remove repeats from a set of returned results:

```
SELECT DISTINCT [columns] FROM [table]
```

Example: table `student` of student IDs, names, and majors

Retrieve all the majors:

```
    SELECT DISTINCT major FROM student
```

Table `student`

id	name	gpa	major	birth_year	pets	favorite_color
101010	John Bardeen	3.1	Electrical Engineering	1908	2	Blue
314159	Albert Einstein	4.0	Physics	1879	0	Green
999999	Jerzy Neyman	3.5	Statistics	1894	1	Red
112358	Ky Fan	3.55	Mathematics	1914	2	Green

```
SELECT DISTINCT pets FROM student ORDER BY pets ASC
```

Test your understanding: what should this return?

Table `student`

id	name	gpa	major	birth_year	pets	favorite_color
101010	John Bardeen	3.1	Electrical Engineering	1908	2	Blue
314159	Albert Einstein	4.0	Physics	1879	0	Green
999999	Jerzy Neyman	3.5	Statistics	1894	1	Red
112358	Ky Fan	3.55	Mathematics	1914	2	Green

```
SELECT DISTINCT pets FROM student ORDER BY pets ASC
```

pets
0
1
2

More on `WHERE` Statements

`WHERE` keyword supports all the natural comparisons one would want to perform

(Numerical) Operation	Symbol/keyword
Equal	=
Not equal	<>, !=
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Within a range	BETWEEN … AND ...

Examples:

```
SELECT id from t_student WHERE …

    … gpa>=3.2

    … pets=1

    … gpa BETWEEN 2.9 AND 3.1

    … birth_year > 1900

    … pets <> 0
```

Caution: different implementations define `BETWEEN` differently (i.e., inclusive vs exclusive)! Be sure to double check!

More on `WHERE` Statements

`WHERE` keyword also allows (limited) regex support and set membership

```
SELECT id, major from student WHERE major IN (“Mathematics”,”Statistics”)

SELECT id, major from student WHERE major NOT IN (“Physics”)
```

Regex-like matching with `LIKE` keyword, wildcards `‘_’` and `‘%’`

```
SELECT first_name from simpsons_character WHERE first_name LIKE “M%”
```

Matches `‘Maggie’`, `‘Marge’` and `‘Moe’`

```
SELECT first_name from simpsons_character WHERE first_name LIKE “B_rt”
```

Matches `‘Bart’`, `‘Bert’`, `‘Bort’` ...

Aggregating results: `GROUP BY`

I have a DB of transactions at my internet business, and I want to know how much each customer has spent in total.

customer_id	customer	order_id	dollar_amount
101	Amy	0023	25
200	Bob	0101	10
315	Cathy	0222	50
200	Bob	0120	12
310	Bob	0429	100
315	Cathy	0111	33
101	Amy	0033	25
315	Cathy	0504	70

```
SELECT customer_id,SUM(dollar_amount)
FROM transaction GROUP BY customer_id
```

customer_id	dollar_amount
101	50
200	22
310	100
315	153

`GROUP BY field_x` combines the rows with the same value in the field `field_x`

More about GROUP BY

GROUP BY supports other operations in addition to SUM:

- COUNT, AVG, MIN, MAX
- Called **aggregate** functions

Can filter results *after* GROUP BY using the HAVING keyword

```
SELECT customer_id, SUM(dollar_amount) AS total_dollar FROM transaction
GROUP BY customer_id HAVING total_dollar > 50
```

customer_id	dollar_amount
101	50
200	22
310	100
315	40
315	100

customer_id	total_dollar
310	100
315	140

More about `GROUP BY`

`GROUP BY` supports other operations in addition to `SUM`:

- `COUNT, AVG, MIN, MAX`
- Called **aggregate** functions

Can filter results *after* `GROUP BY` using the `HAVING` keyword

```
SELECT customer_id, SUM(dollar_amount) AS total_dollar FROM transaction
GROUP BY customer_id HAVING total_dollar>50
```

Note: the difference between the `HAVING` keyword and the `WHERE` keyword is that `HAVING` operates *after* applying filters and `GROUP BY`.

The `AS` keyword just lets us give a nicer name to the aggregated field.

customer_id	dollar_amount
101	50
200	22
310	100
315	40
315	100

customer_id	total_dollar
310	100
315	140

Merging tables: `JOIN`

ID	Name	GPA	Major	Birth Year
101010	John Bardeen	3.1	Electrical Engineering	1908
314159	Albert Einstein	4.0	Physics	1879
999999	Jerzy Neyman	3.5	Statistics	1894
112358	Ky Fan	3.55	Mathematics	1914

P_ID	Pet	age
101010	snoopy	3
101010	scooby	2
999999	lizzy	1
112358	loki	5

Join tables based on primary key / foreign key relationship

ID	Name	GPA	Major	Birth Year	Pet	age
101010	John Bardeen	3.1	Electrical Engineering	1908	snoopy	3
101010	John Bardeen	3.1	Electrical Engineering	1908	scooby	2
999999	Jerzy Neyman	3.5	Statistics	1894	lizzy	1
112358	Ky Fan	3.55	Mathematics	1914	loki	5

Merging tables: `INNER JOIN`

student

id	name	gpa	major	birth_year
101010	John Bardeen	3.1	Electrical Engineering	1908
314159	Albert Einstein	4.0	Physics	1879
999999	Jerzy Neyman	3.5	Statistics	1894
112358	Ky Fan	3.55	Mathematics	1914

pets

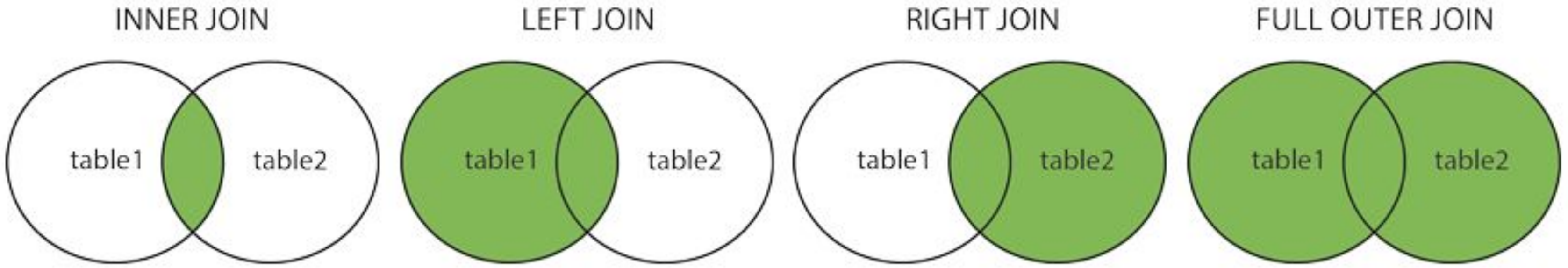
p_id	pet	age
101010	snoopy	3
101010	scooby	2
999999	lizzy	1
112358	loki	5

Join tables based on primary/foreign key

```
SELECT id, name,pet
FROM
student INNER JOIN pets
ON id = p_id
```

id	name	pet
101010	John Bardeen	snoopy
101010	John Bardeen	scooby
999999	Jerzy Neyman	lizzy
112358	Ky Fan	loki

Other ways of joining tables: `OUTER JOIN`



`(INNER) JOIN`: Returns records that have matching values in both tables

`LEFT (OUTER) JOIN`: Return all records from the left table, and the matched records from the right table

`RIGHT (OUTER) JOIN`: Return all records from the right table, and the matched records from the left table

`FULL (OUTER) JOIN`: Return all records when there is a match in either left or right table

Image credit: https://www.w3schools.com/sql/sql_join.asp

Creating/modifying/deleting rows

Insert a row into a table: `INSERT INTO`

```
INSERT INTO table_name [col1, col2, col3, …]
VALUES value1, value2, value3, …
```

Note: if adding values for all columns, you only need to specify the values.

Modify a row in a table: `UPDATE`

```
UPDATE table_name SET col1=value1,col2=value2,
WHERE condition
```

Delete rows from a table: `DELETE`

```
DELETE FROM table_name WHERE condition
```

Caution: if `WHERE` clause is left empty, you'll delete/modify the whole table!

Creating and deleting tables

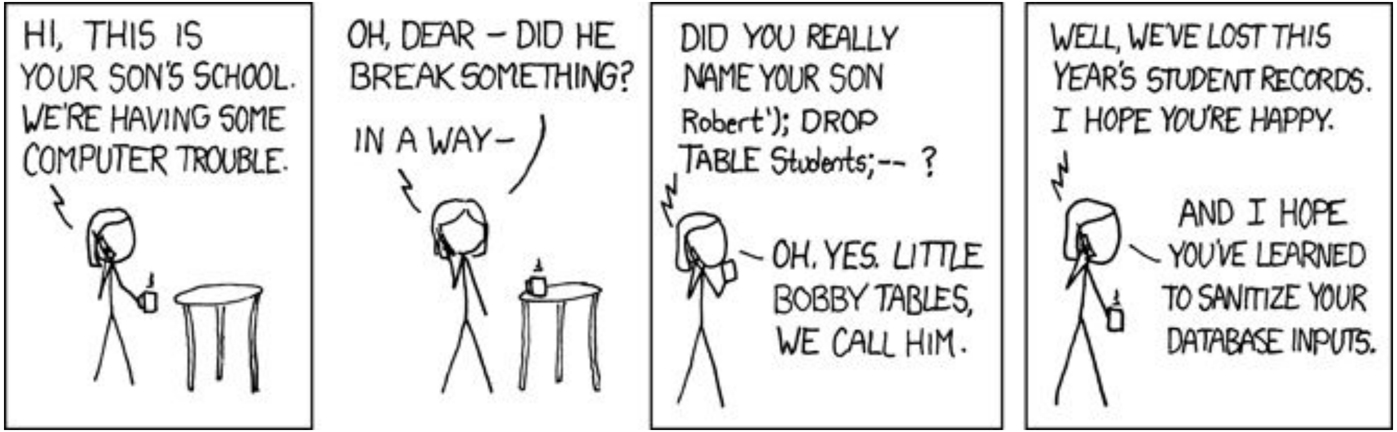
Create a new table: `CREATE TABLE`

```
CREATE TABLE table_name [col1 datatype, col2 datatype, …]
```

Delete a table: `DROP TABLE`

```
DROP TABLE table_name;
```

Be careful when dropping tables!



Python `sqlite3` package implements SQLlite

`Connection` object represents a database

- `Connection` object can be used to create a `Cursor` object
- `Cursor` facilitates interaction with database

`conn = sqlite3.connect('example.db')`

- establish connection to given DB file (creating it if necessary)
- return `Connection` object

`c = conn.cursor()`

- Creates and returns a `Cursor` object for interacting with DB

`c.execute( [SQL command] )`

- runs the given command; cursor now contains query results

Python `sqlite3` package

Important point: unlike many other RDBMSs, SQLite does not allow multiple connections to the same database at the same time.

So, if you're working in a distributed environment, you'll need something else
e.g., MySQL, Oracle, etc.

Python `sqlite3` in action

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor() # create a cursor object.
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
students = [(101010, 'John Bardeen', 'Electrical Engineering', 1908),
            (500100, 'Eugene Wigner', 'Physics', 1902),
            (314159, 'Albert Einstein', 'Physics', 1879),
            (214518, 'Ronald Fisher', 'Statistics', 1890),
            (662607, 'Max Planck', 'Physics', 1858),
            (271828, 'Leonard Euler', 'Mathematics', 1707),
            (999999, 'Jerzy Neyman', 'Statistics', 1894),
            (112358, 'Ky Fan', 'Mathematics', 1914)]
c.executemany('INSERT INTO t_student VALUES (?,?,?,?)', students)
conn.commit() # Write the changes back to example.db
for row in c.execute('''SELECT * from t_student'''):
    print(row)
```

```
(101010, 'John Bardeen', 'Electrical Engineering', 1908)
(500100, 'Eugene Wigner', 'Physics', 1902)
(314159, 'Albert Einstein', 'Physics', 1879)
(214518, 'Ronald Fisher', 'Statistics', 1890)
(662607, 'Max Planck', 'Physics', 1858)
(271828, 'Leonard Euler', 'Mathematics', 1707)
(999999, 'Jerzy Neyman', 'Statistics', 1894)
(112358, 'Ky Fan', 'Mathematics', 1914)
```

Python `sqlite3` in action

Create the database file and set up a `Cursor` object for interacting with it.

```
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor() # create a cursor object.
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
students = [(101010, 'John Bardeen', 'Electrical Engineering', 1908),
            (500100, 'Eugene Wigner', 'Physics', 1902),
            (314159, 'Albert Einstein', 'Physics', 1879),
            (214518, 'Ronald Fisher', 'Statistics', 1890),
            (662607, 'Max Planck', 'Physics', 1858),
            (271828, 'Leonard Euler', 'Mathematics', 1707),
            (999999, 'Jerzy Neyman', 'Statistics', 1894),
            (112358, 'Ky Fan', 'Mathematics', 1914)]
c.executemany('INSERT INTO t_student VALUES (?,?,?,?)', students)
conn.commit() # Write the changes back to example.db
for row in c.execute('''SELECT * from t_student'''):
    print(row)
```

```
(101010, 'John Bardeen', 'Electrical Engineering', 1908)
(500100, 'Eugene Wigner', 'Physics', 1902)
(314159, 'Albert Einstein', 'Physics', 1879)
(214518, 'Ronald Fisher', 'Statistics', 1890)
(662607, 'Max Planck', 'Physics', 1858)
(271828, 'Leonard Euler', 'Mathematics', 1707)
(999999, 'Jerzy Neyman', 'Statistics', 1894)
(112358, 'Ky Fan', 'Mathematics', 1914)
```

Python `sqlite3` in action

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor() # create a cursor object
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
students = [(101010, 'John Bardeen', 'Electrical Engineering', 1908),
            (500100, 'Eugene Wigner', 'Physics', 1902),
            (314159, 'Albert Einstein', 'Physics', 1879),
            (214518, 'Ronald Fisher', 'Statistics', 1890),
            (662607, 'Max Planck', 'Physics', 1858),
            (271828, 'Leonard Euler', 'Mathematics', 1707),
            (999999, 'Jerzy Neyman', 'Statistics', 1894),
            (112358, 'Ky Fan', 'Mathematics', 1914)]
c.executemany('INSERT INTO t_student VALUES (?,?,?,?)', students)
conn.commit() # Write the changes back to example.db
for row in c.execute('''SELECT * from t_student'''):
    print(row)
```

```
(101010, 'John Bardeen', 'Electrical Engineering', 1908)
(500100, 'Eugene Wigner', 'Physics', 1902)
(314159, 'Albert Einstein', 'Physics', 1879)
(214518, 'Ronald Fisher', 'Statistics', 1890)
(662607, 'Max Planck', 'Physics', 1858)
(271828, 'Leonard Euler', 'Mathematics', 1707)
(999999, 'Jerzy Neyman', 'Statistics', 1894)
(112358, 'Ky Fan', 'Mathematics', 1914)
```

Create the table. Note that we need not specify a data type for each column. SQLite is flexible about this.

Python `sqlite3` in action

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor() # create a cursor object.
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
students = [(101010, 'John Bardeen', 'Electrical Engineering', 1908),
            (500100, 'Eugene Wigner', 'Physics', 1902),
            (314159, 'Albert Einstein', 'Physics', 1879),
            (214518, 'Ronald Fisher', 'Statistics', 1890),
            (662607, 'Max Planck', 'Physics', 1858),
            (271828, 'Leonard Euler', 'Mathematics', 1707),
            (999999, 'Jerzy Neyman', 'Statistics', 1894),
            (112358, 'Ky Fan', 'Mathematics', 1914)]
c.executemany('INSERT INTO t_student VALUES (?,?,?,?)', students)
conn.commit() # Write the changes back to example.db
for row in c.execute('''SELECT * from t_student'''):
    print(row)
```

```
(101010, 'John Bardeen', 'Electrical Engineering', 1908)
(500100, 'Eugene Wigner', 'Physics', 1902)
(314159, 'Albert Einstein', 'Physics', 1879)
(214518, 'Ronald Fisher', 'Statistics', 1890)
(662607, 'Max Planck', 'Physics', 1858)
(271828, 'Leonard Euler', 'Mathematics', 1707)
(999999, 'Jerzy Neyman', 'Statistics', 1894)
(112358, 'Ky Fan', 'Mathematics', 1914)
```

Insert rows in the table.

Note: `sqlite3` has special syntax for parameter substitution in strings. Using the built-in Python string substitution is insecure-- vulnerable to SQL injection attack.

Python `sqlite3` in action

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor() # create a cursor object.
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
students = [(101010, 'John Bardeen', 'Electrical Engineering', 1908),
            (500100, 'Eugene Wigner', 'Physics', 1902),
            (314159, 'Albert Einstein', 'Physics', 1879),
            (214518, 'Ronald Fisher', 'Statistics', 1890),
            (662607, 'Max Planck', 'Physics', 1858),
            (271828, 'Leonard Euler', 'Mathematics', 1707),
            (999999, 'Jerzy Neyman', 'Statistics', 1894),
            (112358, 'Ky Fan', 'Mathematics', 1914)]
c.executemany('INSERT INTO t_student VALUES (?,?,?,?)', students)
conn.commit() # Write the changes back to example.db
for row in c.execute('''SELECT * from t_student'''):
    print(row)
```

```
(101010, 'John Bardeen', 'Electrical Engineering', 1908)
(500100, 'Eugene Wigner', 'Physics', 1902)
(314159, 'Albert Einstein', 'Physics', 1879)
(214518, 'Ronald Fisher', 'Statistics', 1890)
(662607, 'Max Planck', 'Physics', 1858)
(271828, 'Leonard Euler', 'Mathematics', 1707)
(999999, 'Jerzy Neyman', 'Statistics', 1894)
(112358, 'Ky Fan', 'Mathematics', 1914)
```

The `commit()` method tells `sqlite3` to write our updates to the database file. This makes our changes "permanent"

Python `sqlite3` in action

```python
import sqlite3
conn = sqlite3.connect('example.db')
c = conn.cursor() # create a cursor object.
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
students = [(101010, 'John Bardeen', 'Electrical Engineering', 1908),
            (500100, 'Eugene Wigner', 'Physics', 1902),
            (314159, 'Albert Einstein', 'Physics', 1879),
            (214518, 'Ronald Fisher', 'Statistics', 1890),
            (662607, 'Max Planck', 'Physics', 1858),
            (271828, 'Leonard Euler', 'Mathematics', 1707),
            (999999, 'Jerzy Neyman', 'Statistics', 1894),
            (112358, 'Ky Fan', 'Mathematics', 1914)]
c.executemany('INSERT INTO t_student VALUES (?,?,?,?)', students)
conn.commit() # Write the changes back to example.db
for row in c.execute('''SELECT * from t_student'''):
    print(row)
```

```
(101010, 'John Bardeen', 'Electrical Engineering', 1908)
(500100, 'Eugene Wigner', 'Physics', 1902)
(314159, 'Albert Einstein', 'Physics', 1879)
(214518, 'Ronald Fisher', 'Statistics', 1890)
(662607, 'Max Planck', 'Physics', 1858)
(271828, 'Leonard Euler', 'Mathematics', 1707)
(999999, 'Jerzy Neyman', 'Statistics', 1894)
(112358, 'Ky Fan', 'Mathematics', 1914)
```

Executing a query returns an iterator over query results.

Python `sqlite3` annotated

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

Establishes a connection to the database stored in `example.db`.

```python
c = conn.cursor()
```

`cursor` object is how we interact with the database. Think of it kind of like the cursor for your mouse. It points to, for example, a table, row or query results in the database.

```python
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
```

`cursor.execute` will run the specified SQL command on the database.

`executemany` runs a list of SQL commands.

```python
c.executemany('INSERT INTO t_student VALUES (?,?,?,?)', students)
conn.commit() # Write the changes back to example.db
```

`commit` writes changes back to the file. WIthout this, the next time you open `example.db`, the table `t_student` will be empty!

```python
conn.close()
```

Close the connection to the database. Think of this like Python file `close`.

Metainformation: `sqlite_master`

Special table that holds information about the “real” tables in the database

Two tables, named `t_student` and `t_thesis`

```
import os, sqlite3
os.remove('example.db') #remove old version of the database.
conn = sqlite3.connect('example.db')
c = conn.cursor()
c.execute('''CREATE TABLE t_student (id, name, field, birth_year)''')
c.execute('''CREATE TABLE t_thesis (thesis_id, phd_title phd_year)''')
for r in c.execute('''SELECT * FROM sqlite_master'''):
    print r
```

```
(u'table', u't_student', u't_student', 2, u'CREATE TABLE t_student (id, name, field, birth_year)')
(u'table', u't_thesis', u't_thesis', 3, u'CREATE TABLE t_thesis (thesis_id, phd_title phd_year)')
```

Retrieving column names in `sqlite3`

```
1  c.execute('''SELECT * from t_student''')
2  c.description
```

```
(('id', None, None, None, None, None, None),
 ('name', None, None, None, None, None, None),
 ('field', None, None, None, None, None, None),
 ('birth_year', None, None, None, None, None, None))
```

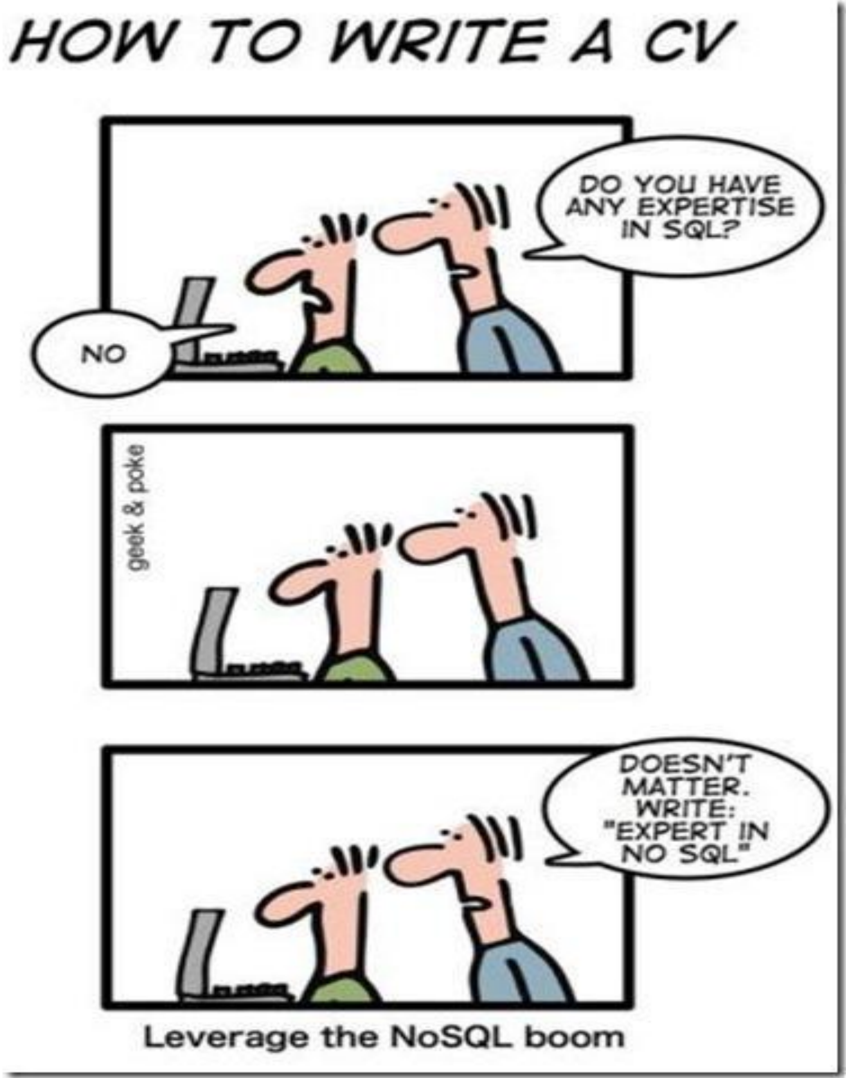
`description` attribute contains the column names; returned as a list of tuples for agreement with a different Python DB API.

```
1  [desc[0] for desc in c.description]
```

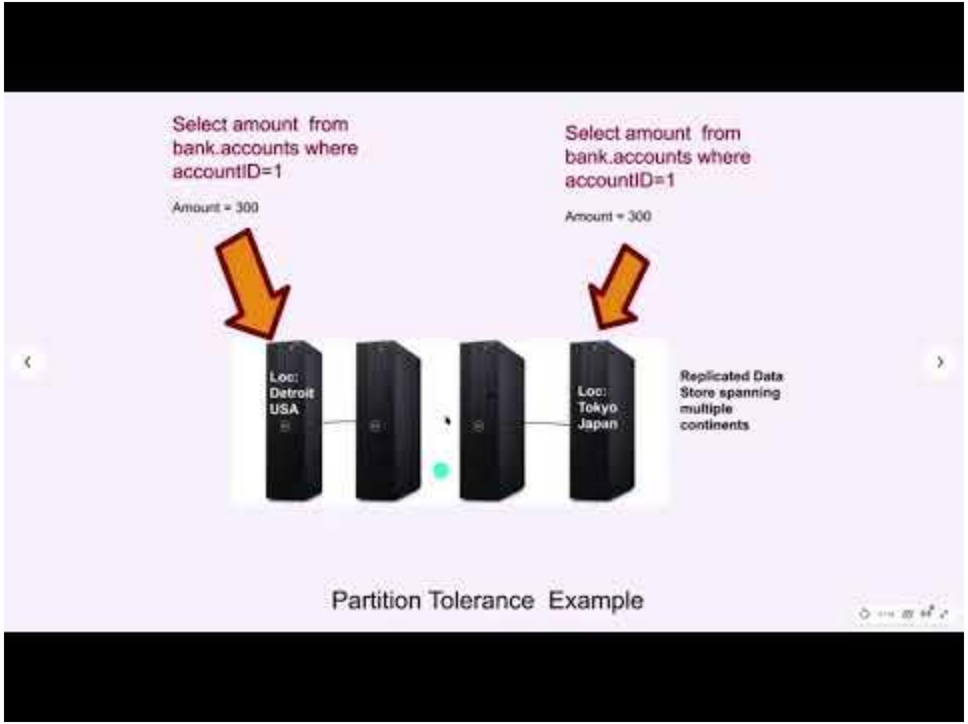
```
['id', 'name', 'field', 'birth_year']
```

Note: this is especially useful in tandem with the `mysql_master` table when exploring a new database, like in your homework!

NoSQL Databases



NoSQL Databases



CAP Theorem or Brewer's theorem

A distributed datastore can have only 2 of the 3 properties:

Consistency: In a system containing replicas, these must be all up-to-date before they can be accessed (access are under concurrency control with locks or multi-versions)

Availability: for a distributed system to be continuously available, every request received by a node in the system must result in a response. An available system must have replicas because failures on the server side may occur

Partition-tolerance: if replicas are on different nodes, the system must continue to operate even in the presence of a network partition (i.e. if I have a database running on 80 nodes across 2 racks and the connection between racks is lost, my database is now partitioned. If the system is partition-tolerant, then the database will still be able to perform read and write operations while partitioned)

Related: Brewer's Theorem https://en.wikipedia.org/wiki/CAP_theorem

What **data transaction properties** are required?

CAP Theorem

ACID

- Atomicity
- **Consistency**
- Isolation
- Durability

BASE

- Basically **Available**
- Soft state
- Eventual consistency

ACID vs BASE

BASE	ACID
• Basic Availability • Soft-state • Eventual consistency	• Atomic • Consistent • Isolated • Durable

NoSQL

Not Only SQL - NoSQL

Schema less

Horizontal scalable

Simple API

Implementation differences - few examples

- Key-Value Store
- Document Store
- Row, Column Store
- In Memory, Data-Structures

https://db-engines.com/en/ranking

Microsoft article:
https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-nosql-database/

What do we get with NoSQL?

- High scalability for simple operations (SO) on multiple nodes
  - By “simple operations”, we refer to key lookups, reads and writes of one record or a small number of records. With the advent of the Web 2.0 sites where millions of users may both read and write data, scalability for simple database operations has become more important
  - This is in contrast to complex queries or joins
  - SO are highly parallelizable

...cont

- Data partitioning (sharding) and replication on multiple nodes
  – Relaxed consistency therefore higher performance and availability
  – Flexibility on the data structure

  But with some sacrifice:
  – Transaction management less rigorous
  – Relaxed consistency
  – Queries that span multiple shards are very inefficient or impossible

What are we losing?

Many database innovations remain unique to the traditional stack

- Variety of indexing
- Query optimizations
- Storage optimizations

All of these are being rediscovered and re-invented in the newer NoSQL databases

Google Cloud Data Storage Solutions

	Cloud Datastore	Cloud Bigtable	Cloud Storage	Cloud SQL	Cloud Spanner	BigQuery
Type	NoSQL document	NoSQL wide column	Blobstore	Relational SQL for OLTP	Relational SQL for OLTP	Relational SQL for OLAP
Best for	Semi-structured application data, durable key-value data	"Flat" data, Heavy read/write, events, analytical data	Structured and unstructured binary or object data	Web frameworks, existing applications	Large-scale database applications (> ~2 TB)	Interactive querying, offline analytics
Use cases	Getting started, App Engine applications	AdTech, Financial and IoT data	Images, large media files, backups	User credentials, customer orders	Whenever high I/O, global consistency is needed	Data warehousing