STAT679
Computing for Data Science and Statistics

Lecture 13: Databases with SQL
Last lecture: HTML, XML and JSON

Each provided a different (though similar) way of storing data

Key motivation of JSON (and, sort of, HTML and XML): self-description

But we saw that JSON could get quite unwieldy quite quickly…
Example of a more complicated JSON object

```
complex_json_string=""{
    "id": "0001",
    "type": "donut",
    "name": "Cake",
    "ppu": 0.55,
    "batters":
    {
        "batter":
        [
            { "id": "1001", "type": "Regular" },
            { "id": "1002", "type": "Chocolate" },
            { "id": "1003", "type": "Blueberry" },
            { "id": "1004", "type": "Devil's Food" }
        ],
        "topping":
        [
            { "id": "5001", "type": "None" },
            { "id": "5002", "type": "Glazed" },
            { "id": "5005", "type": "Sugar" },
            { "id": "5007", "type": "Powdered Sugar" },
            { "id": "5006", "type": "Chocolate with Sprinkles" },
            { "id": "5003", "type": "Chocolate" },
            { "id": "5004", "type": "Maple" }
        ]
    }
}"
```

What if I have hundreds of different kinds of cakes or donuts? The nestedness of JSON objects makes them a little complicated. Generally, JSON is good for delivering (small amounts of) data, but for storing and manipulating large, complicated collections of data, there are better tools, namely databases.

**Note:** there are also security and software engineering reasons to prefer databases over JSON for storing data, but that's beyond the scope of our course.
Why use a database?

Database (DB) software hides the problem of actually handling data
  As we’ll see in a few slides, this is a complicated thing to do!
  Indexing, journaling, archiving handled automatically

Allow fast, concurrent (i.e., multiple users) access to data
  ACID transactions (more on this in a few slides)

Access over the web
  DBs can be run, e.g., on a remote server

Again, JSON/XML/HTML/etc good for delivering data, DBs good for storing
Databases

Information, organized so as to make retrieval fast and efficient

Examples: Census information, product inventory, library catalogue

This course: 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)


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

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UG University</th>
<th>Field</th>
<th>Birth Year</th>
<th>Age at Death</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>University of Wisconsin</td>
<td>Electrical Engineering</td>
<td>1908</td>
<td>82</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>ETH Zurich</td>
<td>Physics</td>
<td>1879</td>
<td>76</td>
</tr>
<tr>
<td>21451</td>
<td>Ronald Fisher</td>
<td>University of Cambridge</td>
<td>Statistics</td>
<td>1890</td>
<td>72</td>
</tr>
</tbody>
</table>

Table with six fields and three records.

Fields can contain different data types

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UG University</th>
<th>Field</th>
<th>Birth Year</th>
<th>Age at Death</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>University of Wisconsin</td>
<td>Electrical Engineering</td>
<td>1908</td>
<td>82</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>ETH Zurich</td>
<td>Physics</td>
<td>1879</td>
<td>76</td>
</tr>
<tr>
<td>21451</td>
<td>Ronald Fisher</td>
<td>University of Cambridge</td>
<td>Statistics</td>
<td>1890</td>
<td>72</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>UG University</th>
<th>Field</th>
<th>Birth Year</th>
<th>Age at Death</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>University of Wisconsin</td>
<td>Electrical Engineering</td>
<td>1908</td>
<td>82</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>ETH Zurich</td>
<td>Physics</td>
<td>1879</td>
<td>76</td>
</tr>
<tr>
<td>21451</td>
<td>Ronald Fisher</td>
<td>University of Cambridge</td>
<td>Statistics</td>
<td>1890</td>
<td>72</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>ID</th>
<th>PhD Year</th>
<th>PhD University</th>
<th>Thesis Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>1936</td>
<td>Princeton University</td>
<td>Quantum Theory of the Work Function</td>
</tr>
<tr>
<td>314159</td>
<td>1905</td>
<td>University of Zurich</td>
<td>A New Determination of Molecular Dimensions</td>
</tr>
<tr>
<td>21451</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

**Note:** some systems achieve faster performance, at cost of one or more of above

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/](https://db.rstudio.com/databases/sqlite/)
SQL (originally SEQUEL, from IBM)

Structured Query Language (Structured English QUEry Language)

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
Examples of database operations

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

- 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

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

**Filtering records**: retain only the records (rows) that match some criterion

**Sorting records**: reorder selected rows according to some field(s)

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

**Adding/deleting tables**: create new or delete existing tables

**Grouping records**: gather rows according to some field

**Merging tables**: combine information from multiple tables into one table
Common database operations

Extracting records: find all rows in a table

Filtering records: retain only the records (rows) that match some criterion

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

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

Adding/deleting tables: create new or delete existing tables

Grouping records: gather rows according to some field

Merging tables: combine information from multiple tables into one table

SQL includes keywords for succinctly expressing all of these operations.
Retrieving records: SQL SELECT Statements

Basic form of a SQL SELECT statement:

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

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

**Retrieve all customer names:** `SELECT name FROM t_customers`

**Retrieve all company names:** `SELECT company FROM t_customers`

**Note:** by convention (and good practice), one often names tables to be prefixed with “TB_” or “t_”. In our illustrative examples, I won’t always do this for the sake of space and brevity, but I highly recommend it in practice. See [https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/](https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/) and [http://leshazlewood.com/software-engineering/sql-style-guide/](http://leshazlewood.com/software-engineering/sql-style-guide/) for two people’s (differing) opinions.
### Table t_students

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

**SELECT id, name, birth_year FROM t_students**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>birth_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>1908</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>1879</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>1894</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>1914</td>
</tr>
</tbody>
</table>
Filtering records: SQL \textbf{WHERE} Statements

To further filter the records returned by a \texttt{SELECT} statement:

\texttt{SELECT [column names] FROM [table] WHERE [filter]}

\textbf{Example}: table \texttt{t\_inventory} of product IDs, unit cost, and number in stock

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

\begin{verbatim}
SELECT id FROM t_inventory WHERE unit_cost>=1
\end{verbatim}

\textbf{Note}: Possible to do much more complicated filtering, e.g., regexes, set membership, etc. We'll discuss that more in a few slides.
<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>gpa</th>
<th>major</th>
<th>birth_year</th>
<th>pets</th>
<th>favorite_color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>3.1</td>
<td>Electrical Engineering</td>
<td>1908</td>
<td>2</td>
<td>Blue</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>4.0</td>
<td>Physics</td>
<td>1879</td>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>3.5</td>
<td>Statistics</td>
<td>1894</td>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>3.55</td>
<td>Mathematics</td>
<td>1914</td>
<td>2</td>
<td>Green</td>
</tr>
</tbody>
</table>

SELECT id, name FROM t_students WHERE birth_year > 1900

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
</tr>
</tbody>
</table>
**NULL** means Nothing!

### Table `t_thesis`

<table>
<thead>
<tr>
<th>id</th>
<th>phd_year</th>
<th>phd_university</th>
<th>thesis_title</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>1936</td>
<td>Princeton University</td>
<td>Quantum Theory of the Work Function</td>
</tr>
<tr>
<td>314159</td>
<td>1905</td>
<td>University of Zurich</td>
<td>A New Determination of Molecular Dimensions</td>
</tr>
<tr>
<td>21451</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>774477</td>
<td>1970</td>
<td>MIT</td>
<td></td>
</tr>
</tbody>
</table>

**SELECT** `id` **FROM** `t_thesis` **WHERE** `phd_year` **IS** **NULL**

**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 `t_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 t_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 t_students

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

```sql
SELECT id, name, gpa FROM t_students ORDER BY gpa DESC
```
More filtering: **DISTINCT** **Keyword**

To remove repeats from a set of returned results:

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

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

**Retrieve all the majors:**

```
SELECT DISTINCT major FROM t_student
```
### Table t_students

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

```sql
SELECT DISTINCT pets FROM t_students ORDER BY pets ASC
```

**Test your understanding:** what should this return?
### Table t_students

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

```sql
SELECT DISTINCT pets FROM t_students ORDER BY pets ASC
```
More on **WHERE** Statements

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

<table>
<thead>
<tr>
<th>(Numberical) Operation</th>
<th>Symbol/keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>=</td>
</tr>
<tr>
<td>Not equal</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>&gt;=</td>
</tr>
<tr>
<td>Within a range</td>
<td>BETWEEN … AND …</td>
</tr>
</tbody>
</table>

**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

```sql
SELECT id, major from t_student WHERE major IN ("Mathematics","Statistics")
SELECT id, major from t_student WHERE major NOT IN ("Physics")
```

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

```sql
SELECT first_name from t_simpsons_characters WHERE first_name LIKE "M%"
```

Matches ‘Maggie’, ‘Marge’ and ‘Moe’

```sql
SELECT first_name from t_simpsons_characters WHERE first_name LIKE "B_rt"
```

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

<table>
<thead>
<tr>
<th>customer_id</th>
<th>customer</th>
<th>order_id</th>
<th>dollar_amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Amy</td>
<td>0023</td>
<td>25</td>
</tr>
<tr>
<td>200</td>
<td>Bob</td>
<td>0101</td>
<td>10</td>
</tr>
<tr>
<td>315</td>
<td>Cathy</td>
<td>0222</td>
<td>50</td>
</tr>
<tr>
<td>200</td>
<td>Bob</td>
<td>0120</td>
<td>12</td>
</tr>
<tr>
<td>310</td>
<td>Bob</td>
<td>0429</td>
<td>100</td>
</tr>
<tr>
<td>315</td>
<td>Cathy</td>
<td>0111</td>
<td>33</td>
</tr>
<tr>
<td>101</td>
<td>Amy</td>
<td>0033</td>
<td>25</td>
</tr>
<tr>
<td>315</td>
<td>Cathy</td>
<td>0504</td>
<td>70</td>
</tr>
</tbody>
</table>

SELECT customer_id, SUM(dollar_amount) FROM t_transactions GROUP BY customer_id

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 t_transactions
GROUP BY customer_id HAVING total_dollar>50
```

<table>
<thead>
<tr>
<th>customer_id</th>
<th>dollar_amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>50</td>
</tr>
<tr>
<td>200</td>
<td>22</td>
</tr>
<tr>
<td>310</td>
<td>100</td>
</tr>
<tr>
<td>315</td>
<td>153</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>customer_id</th>
<th>total_dollar</th>
</tr>
</thead>
<tbody>
<tr>
<td>310</td>
<td>100</td>
</tr>
<tr>
<td>315</td>
<td>153</td>
</tr>
</tbody>
</table>
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 t_transactions
GROUP BY customer_id HAVING total_dollar > 50
```

<table>
<thead>
<tr>
<th>customer_id</th>
<th>dollar_amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>50</td>
</tr>
<tr>
<td>200</td>
<td>22</td>
</tr>
<tr>
<td>310</td>
<td>100</td>
</tr>
<tr>
<td>315</td>
<td>153</td>
</tr>
</tbody>
</table>

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.
Merging tables: **JOIN**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>GPA</th>
<th>Major</th>
<th>Birth Year</th>
<th>#Pets</th>
<th>Favorite Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>3.1</td>
<td>Electrical Engineering</td>
<td>1908</td>
<td>2</td>
<td>Blue</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>4.0</td>
<td>Physics</td>
<td>1879</td>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>3.5</td>
<td>Statistics</td>
<td>1894</td>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>3.55</td>
<td>Mathematics</td>
<td>1914</td>
<td>2</td>
<td>Green</td>
</tr>
</tbody>
</table>

Join tables based on primary key

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>GPA</th>
<th>Major</th>
<th>Birth Year</th>
<th>#Pets</th>
<th>Favorite Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>3.1</td>
<td>Electrical Engineering</td>
<td>1908</td>
<td>2</td>
<td>Blue</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>4.0</td>
<td>Physics</td>
<td>1879</td>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>3.5</td>
<td>Statistics</td>
<td>1894</td>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>3.55</td>
<td>Mathematics</td>
<td>1914</td>
<td>2</td>
<td>Green</td>
</tr>
</tbody>
</table>
### Merging tables: JOIN

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>GPA</th>
<th>Major</th>
<th>Birth Year</th>
<th>#Pets</th>
<th>Favorite Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>3.1</td>
<td>Electrical Engineering</td>
<td>1908</td>
<td>2</td>
<td>Blue</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>4.0</td>
<td>Physics</td>
<td>1879</td>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>3.5</td>
<td>Statistics</td>
<td>1894</td>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>3.55</td>
<td>Mathematics</td>
<td>1914</td>
<td>2</td>
<td>Green</td>
</tr>
</tbody>
</table>

Join tables based on primary key
Merging tables: INNER JOIN

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>gpa</th>
<th>major</th>
<th>birth_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>3.1</td>
<td>Electrical Engineering</td>
<td>1908</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>4.0</td>
<td>Physics</td>
<td>1879</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>3.5</td>
<td>Statistics</td>
<td>1894</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>3.55</td>
<td>Mathematics</td>
<td>1914</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>pets</th>
<th>favorite_color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>2</td>
<td>Blue</td>
</tr>
<tr>
<td>314159</td>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>999999</td>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>112358</td>
<td>2</td>
<td>Green</td>
</tr>
</tbody>
</table>

Join tables based on primary key

```
SELECT id, name, favorite_color
FROM t_student INNER JOIN t_personal
ON t_student.id=t_personal.id
```
### Merging tables: INNER JOIN

#### t_student

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>gpa</th>
<th>major</th>
<th>birth_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>3.1</td>
<td>Electrical Engineering</td>
<td>1908</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>4.0</td>
<td>Physics</td>
<td>1879</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>3.5</td>
<td>Statistics</td>
<td>1894</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>3.55</td>
<td>Mathematics</td>
<td>1914</td>
</tr>
</tbody>
</table>

#### t_personal

<table>
<thead>
<tr>
<th>id</th>
<th>pets</th>
<th>favorite_color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>2</td>
<td>Blue</td>
</tr>
<tr>
<td>314159</td>
<td>0</td>
<td>Green</td>
</tr>
<tr>
<td>999999</td>
<td>1</td>
<td>Red</td>
</tr>
<tr>
<td>112358</td>
<td>2</td>
<td>Green</td>
</tr>
</tbody>
</table>

#### Join tables based on primary key

```
SELECT id, name, favorite_color
FROM t_student INNER JOIN t_personal
ON t_student.id = t_personal.id
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>favorite_color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101010</td>
<td>John Bardeen</td>
<td>Blue</td>
</tr>
<tr>
<td>314159</td>
<td>Albert Einstein</td>
<td>Green</td>
</tr>
<tr>
<td>999999</td>
<td>Jerzy Neyman</td>
<td>Red</td>
</tr>
<tr>
<td>112358</td>
<td>Ky Fan</td>
<td>Green</td>
</tr>
</tbody>
</table>
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](https://www.w3schools.com/sql/sql_join.asp)
Creating/modifying/deleting rows

Insert a row into a table: 
```
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
    table_name SET col1=value1, col2=value2,
WHERE condition
```

Delete rows from a table: 
```
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 SQLite

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

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

```python
import sqlite3

conn = sqlite3.connect('example.db')
c = conn.cursor()  # create a cursor object.

# Create table
conn.execute('''CREATE TABLE t_student 
    (id int, name text, field text, birth_year int);''')

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.

# Insert rows in the table.
c.executemany('INSERT INTO t_student VALUES (?, ?, ?, ?, ?)', students)

for row in c.execute('SELECT * from t_student '):
    print(row)
```

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)]
conn.execute(INSERT INTO t_student VALUES (?, ?, ?, ?))
for row in c.execute('SELECT * FROM t_student'):
    print(row)
```

The `commit()` method tells sqlite3 to write our updates to the database file. This makes our changes “permanent”
```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)
corn.commit()  # Write the changes back to example.db
for row in c.execute('SELECT * from t_student '):
    print(row)
```

Executing a query returns an iterator over query results.
Establishes a connection to the database stored in `example.db`.

```python
import sqlite3

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

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 = conn.cursor()
```

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

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

executemany runs a list of SQL commands.

```python
c.executemany('INSERT INTO t_student VALUES (?, ?, ?, ?)', students)
```

c.commit # Write the changes back to example.db

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

```python
conn.close()
```
Metainformation: sqlite_master

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

Two tables, named t_student and t_thesis

```python
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

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

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

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