Transformations for Data Analytics:

CPSC 4810

Khurram Shehzad

Computer Science & Information Systems
Langara College, Vancouver, BC.

Course Outline

- Introduction to Python for Data Wrangling
- Advanced Data Structures and File Handling
- Introduction to NumPy, Pandas, and Matplotlib
- Subsetting, Filtering, and Grouping Data
- Detecting Outliers and Handling Missing Values
- Concatenating, Merging, and Joining Data
- Getting Comfortable with Different Kinds of Data Sources
- Advanced List Comprehension and the zip Function
- Data Formatting, Identifying and Cleaning Outliers
- Advanced Web Scraping and Data Gathering
- **RDBMS and SQL**
- Application of Data Wrangling in Real Life

Lecture 14

RDBMS and SQL

Python for Data Analysis:

1st Edition, Wes McKinney / Misc Resources

Lecture Outcomes

1. Introduction
2. Refresher of RDBMS and SQL
3. SQL
4. Using an RDBMS (MySQL/PostgreSQL/SQLite)
5. Connecting to a Database in SQLite
6. DDL and DML Commands in SQLite
7. Reading Data from a Database in SQLite
8. Sorting Values in the Database

Lecture Outcomes...

9. Altering Table Structure and Updating New Columns
10. Grouping Values in Tables
11. Relation Mapping in Databases
12. Adding Rows to the comments Table
13. Joins
14. Deleting Rows From a Table
15. Updating Specific Values in a Table
16. RDBMS and DataFrames

Section 1 of 16

Introduction

Introduction

This lecture of our data journey is focused on RDBMS (Relational Database Management Systems) and SQL (Structured Query Language).

In one of the previous lectures, we stored and read data from a file. In this lecture, we will read structured data, design access to the data, and create query interfaces for databases.

We will also learn and play around with some basic and fundamental concepts of database and database management systems in this lecture.

Introduction...

Data has been stored in RDBMS format for years. The reasons behind it are as follows:

- RDBMS is one of the safest ways to store, manage, and retrieve data.
- They are backed by a solid mathematical foundation (relational algebra and calculus) and they expose an efficient and intuitive declarative language – SQL – for easy interaction.
- Almost every language has a rich set of libraries to interact with different RDBMS and the tricks and methods of using them are well tested and well understood.
- Scaling an RDBMS is a pretty well-understood task and there are a bunch of well trained, experienced professionals to do this job (DBA or database administrator).

Introduction...

As we can see in the following chart, the market of DBMS is big. This chart was produced based on market research that was done by **Gartner, Inc.** in **2016**:

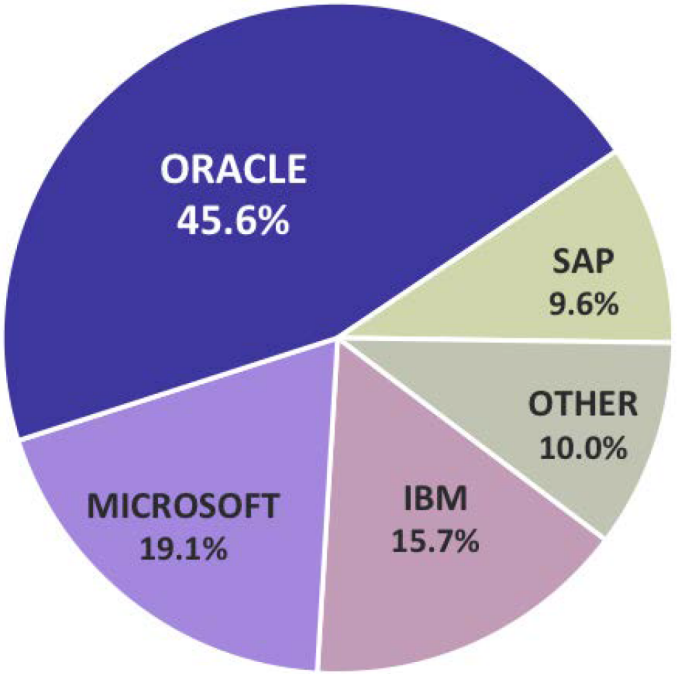

Figure 1: Commercial database market share in 2016

Section 2 of 16

Refresher of RDBMS and SQL

Refresher of RDBMS and SQL

An RDBMS is a piece of software that manages data (represented for the end user in a tabular form) on physical hard disks and is built using the Codd's relational model. Most of the databases that we encounter today are RDBMS.

In recent years, there has been a huge industry shift toward a newer kind of database management system, called **NoSQL** (**MongoDB**, **CouchDB**, **Riak**, and so on). These systems, although in some aspects they follow some of the rules of RDBMS, in most cases reject or modify them.

How is an RDBMS Structured?

The RDBMS structure consists of three main elements, namely the storage engine, query engine, and log management. Here is a diagram that shows the structure of an RDBMS:

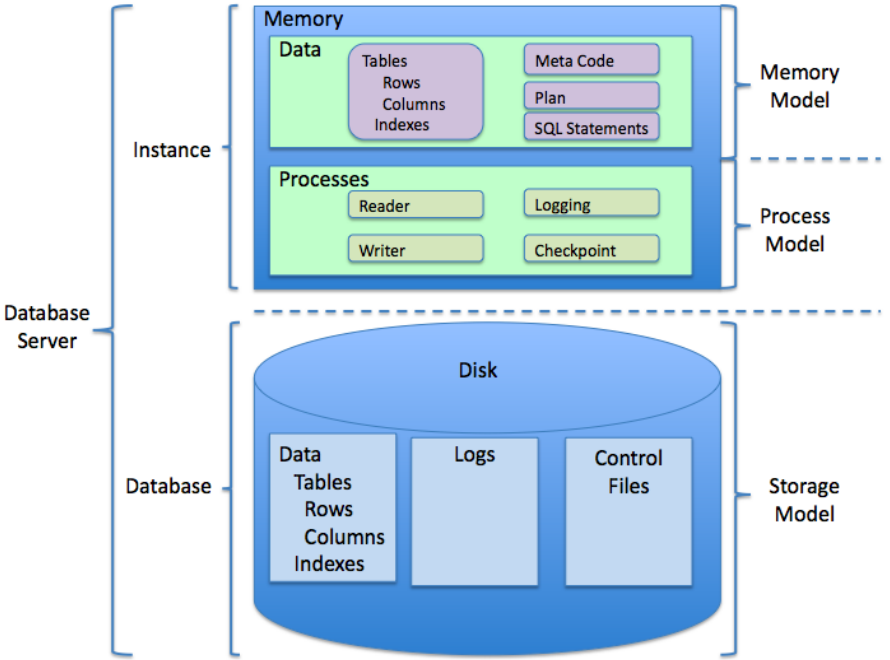

Figure 2: RDBMS Structure

How is an RDBMS Structured? ...

The following are the main concepts of any RDBMS structure:

- **Storage engine**: This is the part of the RDBMS that is responsible for storing the data in an efficient way and also to give it back when asked for, in an efficient way. As an end user of the RDBMS system, we will never need to interact with this layer directly.
- **Query engine**: This is the part of the RDBMS that allows us to create data objects (tables, views, and so on), manipulate them (create and delete columns, create/delete/update rows, and so on), and query them (read rows) using a simple yet powerful language.
- **Log management**: This part of the RDBMS is responsible for creating and maintaining the logs. If you are wondering why the log is such an important thing, then you should look into how replication and partitions are handled in a modern RDBMS (such as PostgreSQL) using something called Write Ahead Log (or WAL for short).

We will focus on the query engine in this lecture.

Section 3 of 16

SQL

SQL

Introduction

Structured **Q**uery **L**anguage or **SQL** (pronounced sequel), as it is commonly known, is a domain-specific language that was originally designed based on E.F. Codd's relational model and is widely used in today's databases to define, insert, manipulate, and retrieve data from them.

It can be further sub-divided into four smaller sub-languages, namely **DDL** (Data Definition Language), **DML** (Data Manipulation Language), **DQL** (Data Query Language), and **DCL** (Data Control Language).

SQL...

Advantages

There are several advantages of using SQL, with some of them being as follows:

- It is based on a solid mathematical framework and thus it is easy to understand.
- It is a declarative language, which means that we actually never tell it how to do its job. We almost always tell it what to do. This frees us from a big burden of writing custom code for data management. We can be more focused on the actual query problem we are trying to solve instead of bothering about how to create and maintain a data store.
- It gives you a fast and readable way to deal with data.
- SQL gives you out-of-the-box ways to get multiple pieces of data with a single query.

SQL...

Description of Sub-languages

Let's learn about DDL, DML, and DQL in a bit more detail. The DCL part is more for database administrators, so we won't cover it.

- **DDL**: This is how we define our data structure in SQL. As RDBMS is mainly designed and built with structured data in mind, we have to tell an RDBMS engine beforehand what our data is going to look like. We can update this definition at a later point in time, but an initial one is a must. This is where we will write statements such as **CREATE TABLE** or **DROP TABLE** or **ALTER TABLE**.

SQL...

Description of Sub-languages...

- **DML**: DML is the part of SQL that let us insert, delete, or update a certain data point (a row) in a previously defined data object (a table). This is the part of SQL which contains statements such as **INSERT INTO**, **DELETE FROM**, or **UPDATE**.
- **DQL**: With DQL, we enable ourselves to query the data stored in an RDBMS, which was defined by DDL and inserted using DML. It gives us enormous power and flexibility to not only query data out of a single object (table), but also to extract relevant data from all the related objects using queries. The frequently used query that's used to retrieve data is the **SELECT** command. We will also see and use the concepts of the primary key, foreign key, index, joins, and so on.

SQL...

Description of Sub-languages...

Once you define and insert data in a database, it can be represented as follows:

First Name	Last Name	Address	City	Age
Mickey	Mouse	123 Fantasy way	Anaheim	73
Bat	Man	321 Cavern Ave	Gotham	54
Wonder	Woman	987 Truth way	Paradise	39
Donald	Duck	555 Quack Street	Mallard	65
Bugs	Bunny	567 Carrot Street	Rascal	58
Wiley	Coyote	999 Acme Way	Canyon	61
Cat	Woman	234 Perfect Street	Hairball	32
Tweet	Bird	543 Ample Ave	Idola	28

Figure 3: Table displaying sample data

SQL...

Description of Sub-languages...

Another thing to remember about tables in an RDBMS is relations. Generally, in a table, we have one or more columns that will have unique values for each row in the table. We call them **primary keys** for the table.

We should be aware that we will encounter unique values across the rows, which are not primary keys. The main difference between them and primary keys is the fact that a primary key cannot be null.

By using the primary key of one table and mentioning it as a foreign key in another table, we can establish relations between two tables. A certain table can be related to any finite number of tables. The relations can be 1:1, which means that each row of the second table is uniquely related to one row of the first table, or 1 :N, N:1, or N: M.

SQL...

Description of Sub-languages...

An example of relations is as follows:

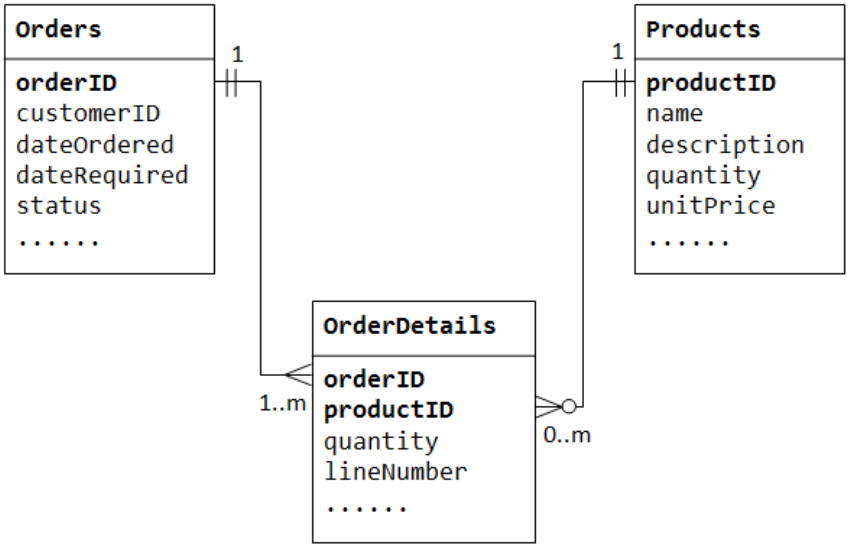

Figure 4: Diagram showing relations

With this brief refresher, we are now ready to go ahead and write some SQL to store and retrieve data.

Section 4 of 16

Using an RDBMS (MySQL/PostgreSQL/SQLite)

Using an RDBMS (MySQL/PostgreSQL/SQLite)

- In this section, we will focus on how to write some basic SQL commands, as well as how to connect to a database from Python and use it effectively within Python.
- The database we will choose here is SQLite. Although, there are other databases such as **Oracle**, **MySQL**, **Postgresql**, and **DB2**, the main tricks that you are going to learn here will not change based on what database you are using.

Using an RDBMS (MySQL/PostgreSQL/SQLite)...

- What will change for different databases however is that you would need to install different third-party Python libraries (such as **Psycopg2** for **Postgresql**, and so on).
- The reason they all behave the same way (apart for some small details) is the fact that they all adhere to **PEP249** (commonly known as Python DB API 2). This is a good standardization and saves us a lot of headaches while porting from one RDBMS to another.

Note: Most of the industry standard projects which are written in Python and use some kind of RDBMS as the data store, most often rely on an ORM or Object Relational Mapper. An ORM is a high-level library in Python which makes many tasks, while dealing with RDBMS, easier. It also exposes a more Pythonic API than writing raw SQL inside Python code.

Section 5 of 16

Connecting to a Database in SQLite

Connecting to a Database in SQLite

- In this section, we will look into the first step toward using an RDBMS in Python code.
- All we are going to do is connect to a database and then close the connection.
- We will also learn about the best way to do this.
- So let's get started.

Connecting to a Database in SQLite...

Import the **sqlite3** library of Python by using the following command:

In [1]:
```
import sqlite3
```

Use the **connect** function to connect to a database. In case the database with the name passed to **connect** doesn't already exist, it would be created. If you already have some experience with databases, you will notice that we are not using any **server address**, **user name**, **password**, or other credentials to connect to a database. This is because these fields are not mandatory in **sqlite3**, unlike in **Postgresql** or **MySQL**. The main database engine of SQLite is embedded:

In [2]:
```
conn = sqlite3.connect("lesson.db")
```

Here we stored the connection to our database **lesson.db** as the object **conn**, which we will use later by invoking several methods on top of it.

Connecting to a Database in SQLite...

Finally, once we are finished interacting with our database, we can close the connection by invoking the **close** method on top of our **conn** object as follows:

In [3]:
```
conn.close()
```

This **conn** object is the main connection object, and we will need that to get a second type of object (**cursor**) in the future once we want to interact with the database. We need to be careful about closing any open connection to our database.

Connecting to a Database in SQLite...

There is also an alternative for connecting to and interacting with our database, which is by using the same **with** statement from Python, just like we did for files, as follows:

In [4]:
```
with sqlite3.connect("lesson.db") as conn:
    pass
```

Here, I used the **pass** statement which is simply ignored by the Python interpreter and can be seen as a null statement. It basically means "do nothing" as I don't want to do anything at the moment using the database connection.

So, we have just learned how to connect to a database using Python.

Section 6 of 16

DDL and DML Commands in SQLite

DDL and DML Commands in SQLite

We will now look at how we can create a table in our **lesson** database using **sqlite3**, and we will also insert some data in it.

As the name suggests, DDL (Data Definition Language) is the way to communicate to the database engine in advance to define what the data will look like. The database engine creates a table object based on the definition provided and prepares it.

To create a table in SQL, we use the **CREATE TABLE** SQL clause. This will need the *table name* and the table definition. The *table name* is a unique identifier for the database engine to find and use the table for all future transactions. It can be anything (any alphanumeric string), as long as it is unique.

DDL and DML Commands in SQLite...

We add the *table definition* in the form of (column_name_1 data_type, column_name_2 data type, …). For our purpose, we will use the **text** and **integer** datatypes, but usually a standard database engine supports many more datatypes, such as float, double, date time, Boolean, and so on.

We will also need to specify a *primary key*. A primary key is a unique, non-null identifier that's used to uniquely identify a row in a table. In our case, we will use email as a primary key. A primary key can be an integer or text.

The last thing you need to know is that unless you call a **commit** on the series of operations you just performed (together, we formally call them a **transaction**), nothing would be actually performed and reflected in the database. This property is called **atomicity**. In fact, for a database to be industry standard (to be useable in real life), it needs to follow the ACID (Atomicity, Consistency, Isolation, Durability) properties.

DDL and DML Commands in SQLite...

So let's go ahead and use SQLite's **connect** function to connect to the **lesson.db** database we created earlier:

In [5]:
```
conn = sqlite3.connect("lesson.db")
```

We can now go ahead and create a **cursor** object by calling **conn.cursor()**.

In [6]:
```
cursor = conn.cursor()
```

This **cursor** object, by means of its **execute** method, will be used to make edits to our database just like we use a normal cursor to edit an MS Word document, for example.

DDL and DML Commands in SQLite...

So let's use **cursor.execute** to create a table named **user** with the columns *email*, *first_name*, *last_name*, *address*, and *age*:

In [7]:
```
cursor.execute("CREATE TABLE IF NOT EXISTS user (email text, first_n
ame text, last_name text, address text, age integer, PRIMARY KEY (em
ail))")
```

Out[7]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

The first bit **CREATE TABLE IF NOT EXISTS** in the above statement creates the table only if it does not exist already. This is to avoid getting an error had we used **CREATE TABLE** only (which throws an error the next time we run the same code). The last bit **PRIMARY KEY (email)** in the above statement tells SQL that the *email* column should be used as the primary key.

We are now ready to to go ahead and insert some data (rows) into our table. So let's do that on the next slide.

DDL and DML Commands in SQLite...

So let's use **cursor.execute** once again to insert some data into our table **user**:

In [8]:
```
cursor.execute("INSERT INTO user VALUES ('bob@example.com', 'Bob',
 'Codd', '123 Fantasy lane, Fantasu City', 31)")
cursor.execute("INSERT INTO user VALUES ('tom@web.com', 'Tom', 'Fak
e', '123 Fantasy lane, Fantasu City', 39)")
```

Out[8]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

Once the above code is executed, you would notice a temporary file **lesson.db-journal** created in the same folder from which your Notebook is running. That's because all the changes we have made so far are temporary, they only exist in the computer's memory.

DDL and DML Commands in SQLite...

Alternatively, you can also write all your data to a list first and then write it to the database's table all at once. But before we do that, let's delete all the rows we just wrote to the table **user**:

In [9]:
```
cursor.execute("DELETE FROM user")
```

Out[9]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

Using **DELETE FROM user** without the **WHERE** clause simply deletes all rows from the table. Now let's go ahead and store our data rows as tuples into a **data** list and also define the statement (**stmt**) to be executed:

In [10]:
```
data = [('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantas
u City', 31),
        ('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu Ci
ty', 39)]
stmt = "INSERT INTO user VALUES(?, ?, ?, ?, ?)"
```

DDL and DML Commands in SQLite...

Now that we have created our **data** list and written the SQL statement **stmt** on the previous slide, we can go ahead and use the method **executemany** on top of our **cursor** object as follows:

In [11]:
```
cursor.executemany(stmt, data)
```

Out[11]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

This accomplishes exactly the same as the code on slide 35.

DDL and DML Commands in SQLite...

Finally, we can use **conn.commit()** to commit the changes (i.e. the table and its rows) to our database. When you do so, the changes will become permanent and **lesson.db-journal** would be deleted.

In [12]:
```
conn.commit()
```

Section 7 of 16

Reading Data from a Database in SQLite

Reading Data from a Database in SQLite

In the preceding section, we created a table and stored data in it. Now, we will learn how to read the data that's stored in this database.

The **SELECT** clause is immensely powerful, and it is really important for a data practitioner to master **SELECT** and everything related to it (such as conditions, joins, group-by, and so on).

Reading Data from a Database in SQLite...

So let's go ahead and access the data we just wrote to our database:

In [13]:
```
rows = cursor.execute('SELECT * FROM user')
for row in rows:
    print(row)
```

```
('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu Ci
ty', 31)
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39)
```

The * after **SELECT** tells the engine to select all of the columns from the table. It is a useful shorthand. We have not mentioned any condition for the selection (such as above a certain age, first name starting with a certain sequence of letters, and so on). We are practically telling the database engine to select all the rows and all the columns from the table.

Reading Data from a Database in SQLite...

Alternatively, I can also access the data by storing the query in **rows** (just like before) and then invoking the **fetchall** method on top of it:

In [14]:
```
rows = cursor.execute('SELECT * FROM user')
rows.fetchall()
```

Out[14]:
```
[('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu C
ity', 31),
 ('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu Cit
y', 39)]
```

Reading Data from a Database in SQLite...

SELECT * is time-consuming and less effective if we have a huge table. In such situations (which is almost always the case), we can use the **LIMIT** clause to limit the number of rows we want (similar to **df.head()**):

In [15]:
```
rows = cursor.execute('SELECT * FROM user LIMIT 1')
for row in rows:
    print(row)
```

```
('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu Ci
ty', 31)
```

Section 8 of 16

Sorting Values in the Database

Sorting Values in the Database

We can use the **ORDER BY** clause to sort the rows of the table **user** in ascending order with respect to a column (here we sort by *age*):

In [16]:
```
rows = cursor.execute('SELECT * FROM user ORDER BY age')
for row in rows:
    print(row)
```

```
('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu Ci
ty', 31)
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39)
```

Sorting Values in the Database...

Similarly, using **DESC** in conjunction with **ORDER BY** will sort the rows in descending order with respect to *age*:

In [17]:
```
rows = cursor.execute('SELECT * FROM user ORDER BY age DESC')
for row in rows:
    print(row)
```

```
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39)
('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu Ci
ty', 31)
```

Section 9 of 16

Altering Table Structure and Updating New Columns

Altering Table Structure and Updating New Columns

Now, we are going to add a column to our table **user** using **ALTER** and later use **UPDATE** to set all the values for the newly added column to a certain value.

So let's go ahead and alter our table structure by adding a new column:

In [18]:
```
cursor.execute("ALTER TABLE user ADD COLUMN gender text")
```

Out[18]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

The **ALTER TABLE user** bit in the above statement tells SQL to alter the structure of the table **user** whereas the **ADD COLUMN gender text** bit tells how the structure is going to be altered, i.e. by adding a new column named **gender** with the data type **text**.

Altering Table Structure and Updating New Columns...

We can now **UPDATE** our table **user** by using **SET** to populate the entire column **gender** with the value 'M' regardless of row. The **UPDATE** command is basically used to edit/update any row after it has been inserted:

In [19]:
```
cursor.execute("UPDATE user SET gender='M'")
```

Out[19]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

The **UPDATE user** bit in the above statement tells SQL to update the table **user** whereas the **SET gender='M'** bit tells how the table is going to be updated, i.e. by setting the value for the column **gender** to 'M' for all the rows. Finally we can commit the changes:

In [20]:
```
conn.commit()
```

Altering Table Structure and Updating New Columns...

Now let's go ahead and see what the table **user** looks like:

In [21]:
```
rows = cursor.execute('SELECT * FROM user')
for row in rows:
    print(row)
```

```
('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu Ci
ty', 31, 'M')
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39, 'M')
```

We can clearly see that the value for the newly added column **gender** has been set to 'M' for all the rows (where 'M' stands for male). In practise, you should be careful with this as using **UPDATE** without selective clauses such as **WHERE** affects the entire table.

Section 10 of 16

Grouping Values in Tables

Grouping Values in Tables

Now let's learn about a concept that we have already learned about in pandas. This is the **GROUP BY** clause. The **GROUP BY** clause is a technique that's used to group the rows based on the value of one or more columns and then apply some function on the resulting groups. The following diagram explains how the **GROUP BY** clause works.

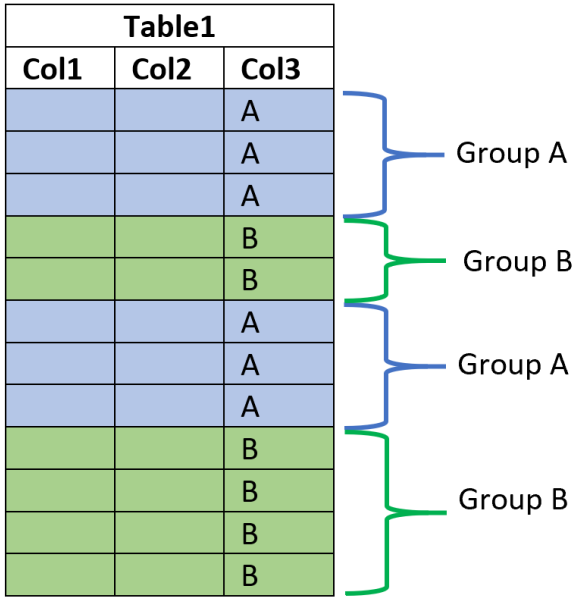

Figure 5: Illustration of the GROUP BY clause on a table

Grouping Values in Tables...

Before we go ahead to see how the **GROUP BY** clause works, let's add a female user named 'Shelly' to the table:

In [22]:
```
cursor.execute("INSERT INTO user VALUES ('shelly@www.com', 'Shelly',
'Milar', '123, Ocean View Lane', 39, 'F')")
conn.commit()
```

Grouping Values in Tables...

Now let's go ahead and see what the table **user** looks like:

In [23]:
```
rows = cursor.execute('SELECT * FROM user')
for row in rows:
    print(row)
```

```
('bob@example.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu Ci
ty', 31, 'M')
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39, 'M')
('shelly@www.com', 'Shelly', 'Milar', '123, Ocean View Lane', 3
9, 'F')
```

Grouping Values in Tables...

We will now retrieve the count of all the rows from our table **user** based on the gender of the users:

In [24]:
```
rows = cursor.execute("SELECT COUNT(*), gender FROM user GROUP BY ge
nder")
for row in rows:
    print(row)
```

```
(1, 'F')
(2, 'M')
```

The first bit **SELECT COUNT(*), gender FROM user** in the above statement tells SQL to retrieve the **count of all the rows** and the **gender** column from the **user** table. Whereas the second bit **GROUP BY gender** tells SQL that the counts need to be calculated separately for each group in the **gender** column, i.e. separately for the group 'M' and separately for the group 'F'. So this returns the counts for each group separately.

Section 11 of 16

Relational Mapping in Databases

Relational Mapping in Databases

- So far, we have been working with a single table and altering it, as well as reading back the data.
- However, the real power of an RDBMS comes from the handling of relationships among different objects (tables).
- In this section, we are going to create a new table called **comments** and link it with the **user** table in a 1:N relationship. This means that one user can have multiple comments.
- The way we are going to do this is by adding the **user** table's primary key as a foreign key in the **comments** table. This will create a 1:N relationship.

Relational Mapping in Databases...

When we link two tables, we need to specify to the database engine what should be done if the parent row is deleted, which has many children in the other table. As we can see in the following diagram, we are asking what happens at the place of the question marks when we delete 'email1', which is one of the values for the primary key **email** in the **user** table. The following figure illustrates this concept:

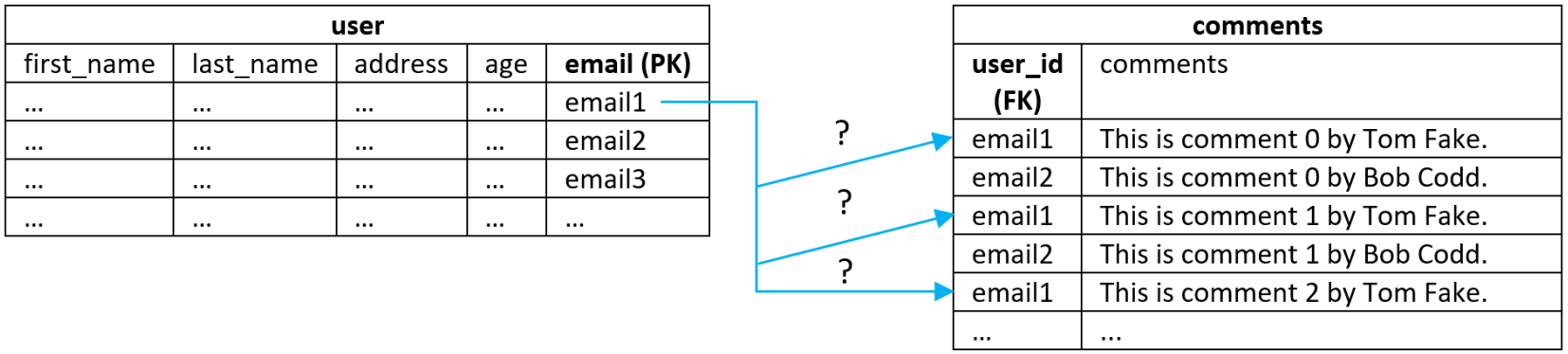

Figure 6: Illustration of relations between **user** and **comments** tables

Relational Mapping in Databases...

- In a non-RDBMS situation, this situation can quickly become difficult and messy to manage and maintain.
- However, with an RDBMS, all we have to tell the database engine, in very precise ways, is what to do when a situation like this occurs. The database engine will do the rest for us.
- We use **ON DELETE** to tell the engine what we do with all the rows of a linked table (the one with the foreign key) when the parent row for one or more of its rows gets deleted in the linking table (the one with the primary key).

Relational Mapping in Databases...

So let's see how we can create the table **comments** which has a foreign key in it. But before we can do that, we need to tell **sqlite3** that the table we are about to create will have a single foreign key in it (actually there can be multiple foreign keys but we don't need that at the moment):

In [25]:
```
cursor.execute("PRAGMA foreign_keys = 1")
```

Out[25]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

You may be wondering what the strange looking **PRAGMA foreign_keys = 1** above means. It is there just because SQLite does not use the normal foreign key features by default. It is this line that enables that feature. Therefore this step is specific to SQLite only and we won't need it for any other DBMS.

Relational Mapping in Databases...

In [26]:
```
sql = """
    CREATE TABLE comments (
        user_id text,
        comments text,
        FOREIGN KEY (user_id) REFERENCES user (email)
        ON DELETE CASCADE ON UPDATE CASCADE
    )
"""
```

In the above SQL statement, we are creating our new table **comments** just like we created the table **user** before. The only change is the line **FOREIGN KEY (user_id) ... CASCADE**. The **FOREIGN KEY** modifier specifies that the column **user_id** will be used as a foreign key. Further, the **REFERENCES** keyword tells SQL that this column is related to the **user** table's primary key (**email**). The **ON DELETE CASCADE / ON UPDATE CASCADE** inform the database engine that we want to delete / update all the children rows in the **comments** table when the parent row gets deleted / updated in the **user** table. We could have also used **NO ACTION** for either of these.

Relational Mapping in Databases...

Finally, we can execute the SQL query on the previous slide and commit the changes to our database which would create the table **comments** along with its associated fields:

In [27]:
```
cursor.execute(sql)
conn.commit()
```

Section 12 of 16

Adding Rows to the comments Table

Adding Rows to the comments Table

We have created a table called **comments**. Now let's dynamically generate an insert query so that we can insert 10 generic comments for each user, as follows:

In [28]:
```
sql = "INSERT INTO comments VALUES ('{}', '{}')"
rows = cursor.execute('SELECT * FROM user ORDER BY age')
for row in rows:
    email = row[0]
    print("Going to create rows for {}".format(email))
    name = row[1] + " " + row[2]
    for i in range(10):
        comment = "This is comment {} by {}".format(i, name)
        conn.cursor().execute(sql.format(email, comment))
conn.commit()
```

```
Going to create rows for bob@example.com
Going to create rows for tom@web.com
Going to create rows for shelly@www.com
```

Adding Rows to the comments Table...

Now let's see what a portion of the table **comments** looks like. Due to shortage of space on the slide, I would print only a few rows from the table:

In [29]:
```
rows = cursor.execute('SELECT * FROM comments')
i = 0
for row in rows:
    if (i % 5 == 0):
        print(row)
    i += 1
```

```
('bob@example.com', 'This is comment 0 by Bob Codd')
('bob@example.com', 'This is comment 5 by Bob Codd')
('tom@web.com', 'This is comment 0 by Tom Fake')
('tom@web.com', 'This is comment 5 by Tom Fake')
('shelly@www.com', 'This is comment 0 by Shelly Milar')
('shelly@www.com', 'This is comment 5 by Shelly Milar')
```

Section 13 of 16

Joins

Joins

In this section, we will learn how to exploit the relationship we just built. This means that if we have the primary key from one table, we can retrieve all the data needed from that table and also all the linked rows from the child table. To achieve this, we will use something called a join.

A join is basically a way to retrieve linked rows from two tables using any kind of primary key - foreign key relation that they have. There are many types of join, such as **INNER**, **LEFT OUTER**, **RIGHT OUTER**, **FULL OUTER**, and **CROSS**. They are used in different situations. However, most of the time, in simple 1:N relations, we end up using an **INNER** join.

Joins...

In one of the earliest few lectures, we learned about sets, and we can actually view an **INNER JOIN** as an intersection of two sets. The following diagram illustrates the concept:

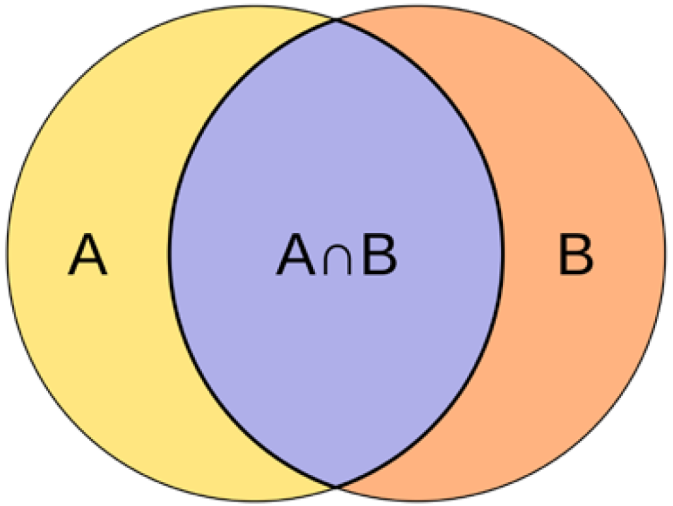

Figure 7: Intersection Join

Here, **A** represents one table and **B** represents another.

Joins...

The meaning of having common members is to have a relationship between them. It takes all of the rows of **A** and compares them with all of the rows of **B** to find the matching rows that satisfy the join predicate.

This can quickly become a complex and time-consuming operation. Joins can be very expensive operations. Usually, we use some kind of **WHERE** clause, after we specify the join, to shorten the scope of rows that are fetched from table **A** or **B** to perform the matching.

Joins...

In our case, our first table, **user**, has three entries, with the primary key being the **email**. We can make use of this in our query to get comments just from **Bob** (here I use **LIMIT 2** to reduce output due to space constraints):

In [30]:
```
sql = """
    SELECT * FROM comments
    JOIN user ON comments.user_id = user.email
    WHERE user.email='bob@example.com'
    LIMIT 2
"""
rows = cursor.execute(sql)
for row in rows:
    print(row)
```

```
('bob@example.com', 'This is comment 0 by Bob Codd', 'bob@exampl
e.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu City', 31,
'M')
('bob@example.com', 'This is comment 1 by Bob Codd', 'bob@exampl
e.com', 'Bob', 'Codd', '123 Fantasy lane, Fantasu City', 31,
'M')
```

Joins...

We have seen that we can use a **JOIN** to fetch the related rows from two tables. However, if we look at the results on the previous slide, we will see that it returned all the columns, thus combining both tables. This is not very concise. What if we only want to see the emails and the related comments, and not all the data? We can do that as follows:

In [31]:
```
sql = """
    SELECT comments.* FROM user
    JOIN comments ON comments.user_id = user.email
    WHERE user.email='bob@example.com'
    LIMIT 4
"""
rows = cursor.execute(sql)
for row in rows:
    print(row)
```

```
('bob@example.com', 'This is comment 0 by Bob Codd')
('bob@example.com', 'This is comment 1 by Bob Codd')
('bob@example.com', 'This is comment 2 by Bob Codd')
('bob@example.com', 'This is comment 3 by Bob Codd')
```

Joins...

We can also select a subset of the columns from the resulting join:

In [32]:
```
sql = """
    SELECT first_name, last_name, comments FROM user
    JOIN comments ON comments.user_id = user.email
    WHERE user.email='bob@example.com'
"""
rows = cursor.execute(sql)
for row in rows:
    print(row)
```

```
('Bob', 'Codd', 'This is comment 0 by Bob Codd')
('Bob', 'Codd', 'This is comment 1 by Bob Codd')
('Bob', 'Codd', 'This is comment 2 by Bob Codd')
('Bob', 'Codd', 'This is comment 3 by Bob Codd')
('Bob', 'Codd', 'This is comment 4 by Bob Codd')
('Bob', 'Codd', 'This is comment 5 by Bob Codd')
('Bob', 'Codd', 'This is comment 6 by Bob Codd')
('Bob', 'Codd', 'This is comment 7 by Bob Codd')
('Bob', 'Codd', 'This is comment 8 by Bob Codd')
('Bob', 'Codd', 'This is comment 9 by Bob Codd')
```

Joins...

You might be wondering how in the SQL query on the previous slide, apart from the fields *first_name* and *last_name*, you could also **SELECT** the field *comments* **FROM** the **user** table. That's because we are not selecting the field *comments* from the **user** table, but rather from the join of **user** on **comments** based on the condition that the *user_id* field in the **comments** table is equal to the *email* field in the **user** table. The join of the two tables **user** and **comments** creates a temporary table in the memory (aka a view). Therefore, **user JOIN comments** in the query on the previous slide should have been more appropriately written on the same line as follows. However, starting with **JOIN** on the new line is a convention in the SQL community and as such should be respected.

In [33]:
```
sql = """
    SELECT first_name, last_name, comments FROM
    user JOIN comments ON comments.user_id = user.email
    WHERE user.email='bob@example.com'
"""
```

Joins...

In fact, it's better to qualify field names as follows to avoid ambiguity:

In [34]:
```
sql = """
    SELECT user.first_name, user.last_name, comments.comments FROM
    user JOIN comments ON comments.user_id = user.email
    WHERE user.email='bob@example.com'
"""
rows = cursor.execute(sql)
for row in rows:
    print(row)
```

```
('Bob', 'Codd', 'This is comment 0 by Bob Codd')
('Bob', 'Codd', 'This is comment 1 by Bob Codd')
('Bob', 'Codd', 'This is comment 2 by Bob Codd')
('Bob', 'Codd', 'This is comment 3 by Bob Codd')
('Bob', 'Codd', 'This is comment 4 by Bob Codd')
('Bob', 'Codd', 'This is comment 5 by Bob Codd')
('Bob', 'Codd', 'This is comment 6 by Bob Codd')
('Bob', 'Codd', 'This is comment 7 by Bob Codd')
('Bob', 'Codd', 'This is comment 8 by Bob Codd')
('Bob', 'Codd', 'This is comment 9 by Bob Codd')
```

Section 14 of 16

Deleting Rows From a Table

Deleting Rows From a Table

- We are now going to delete a row from the **user** table and observe the effects it will have on the **comments** table.
- Be very careful when running this command as it can have a destructive effect on the data.
- Remember that the **DELETE** command has to almost always be run accompanied by a **WHERE** clause so that we delete just a part of the data and not everything.

Deleting Rows From a Table...

So let's go ahead and delete the user Bob from the **user** table:

In [35]:
```
cursor.execute("DELETE FROM user WHERE email='bob@example.com'")
conn.commit()
```

Let's see what the **user** table looks like afer the user Bob has been deleted:

In [36]:
```
rows = cursor.execute("SELECT * FROM user")
for row in rows:
    print(row)
```

```
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39, 'M')
('shelly@www.com', 'Shelly', 'Milar', '123, Ocean View Lane', 3
9, 'F')
```

Deleting Rows From a Table...

We saw on the previous slide that the user 'Bob' has been deleted.

Now, moving on to the **comments** table, we have to remember that we had mentioned **ON DELETE CASCADE** while creating the table. The database engine knows that if a row is deleted from the parent table (**user**), all the related rows from the child tables (**comments**) will have to be deleted.

So let's see on the next slide what the **comments** table looks like now that the user 'Bob' has been deleted from the **user** table.

Deleting Rows From a Table...

In [37]:
```
rows = cursor.execute("SELECT * FROM comments LIMIT 18")
for row in rows:
    print(row)
```

```
('tom@web.com', 'This is comment 0 by Tom Fake')
('tom@web.com', 'This is comment 1 by Tom Fake')
('tom@web.com', 'This is comment 2 by Tom Fake')
('tom@web.com', 'This is comment 3 by Tom Fake')
('tom@web.com', 'This is comment 4 by Tom Fake')
('tom@web.com', 'This is comment 5 by Tom Fake')
('tom@web.com', 'This is comment 6 by Tom Fake')
('tom@web.com', 'This is comment 7 by Tom Fake')
('tom@web.com', 'This is comment 8 by Tom Fake')
('tom@web.com', 'This is comment 9 by Tom Fake')
('shelly@www.com', 'This is comment 0 by Shelly Milar')
('shelly@www.com', 'This is comment 1 by Shelly Milar')
('shelly@www.com', 'This is comment 2 by Shelly Milar')
('shelly@www.com', 'This is comment 3 by Shelly Milar')
('shelly@www.com', 'This is comment 4 by Shelly Milar')
('shelly@www.com', 'This is comment 5 by Shelly Milar')
('shelly@www.com', 'This is comment 6 by Shelly Milar')
('shelly@www.com', 'This is comment 7 by Shelly Milar')
```

Section 15 of 16

Updating Specific Values in a Table

Updating Specific Values in a Table

We will now see how we can update rows in a table. We have already looked at this previously but, as I mentioned, at a table level only. Without **WHERE**, updating is often a bad idea.

So let's go ahead and combine **UPDATE** with **WHERE** to selectively update the *first_name* of the user 'Tom' with 'Chris' and then see what the updated table looks like:

In [38]:
```
cursor.execute("UPDATE user SET first_name='Chris' WHERE email='tom@
web.com'")
conn.commit()
rows = cursor.execute("SELECT * FROM user")
for row in rows:
    print(row)
```

```
('tom@web.com', 'Chris', 'Fake', '123 Fantasy lane, Fantasu Cit
y', 39, 'M')
('shelly@www.com', 'Shelly', 'Milar', '123, Ocean View Lane', 3
9, 'F')
```

Now let's see on the next slide what the **comments** table looks like now that the *first_name* of the user 'Tom' has been updated to 'Chris' in the **user**
table.

Updating Specific Values in a Table...

In [39]:
```
rows = cursor.execute("SELECT * FROM comments LIMIT 18")
for row in rows:
    print(row)
```

```
('tom@web.com', 'This is comment 0 by Tom Fake')
('tom@web.com', 'This is comment 1 by Tom Fake')
('tom@web.com', 'This is comment 2 by Tom Fake')
('tom@web.com', 'This is comment 3 by Tom Fake')
('tom@web.com', 'This is comment 4 by Tom Fake')
('tom@web.com', 'This is comment 5 by Tom Fake')
('tom@web.com', 'This is comment 6 by Tom Fake')
('tom@web.com', 'This is comment 7 by Tom Fake')
('tom@web.com', 'This is comment 8 by Tom Fake')
('tom@web.com', 'This is comment 9 by Tom Fake')
('shelly@www.com', 'This is comment 0 by Shelly Milar')
('shelly@www.com', 'This is comment 1 by Shelly Milar')
('shelly@www.com', 'This is comment 2 by Shelly Milar')
('shelly@www.com', 'This is comment 3 by Shelly Milar')
('shelly@www.com', 'This is comment 4 by Shelly Milar')
('shelly@www.com', 'This is comment 5 by Shelly Milar')
('shelly@www.com', 'This is comment 6 by Shelly Milar')
('shelly@www.com', 'This is comment 7 by Shelly Milar')
```

Updating Specific Values in a Table...

We just saw on the previous slide that updating the *first_name* from 'Tom' to 'Chris' in the **user** table had no effect on the **comments** table because there was no field in the **comments** table linked to the *first_name* field in the **user** table.

Updating Specific Values in a Table...

Before proceeding further, let's update the *first_name* from 'Chris' back to 'Tom', so that the rest of the lesson remains unaffected:

In [40]:
```
cursor.execute("UPDATE user SET first_name='Tom' WHERE email='tom@we
b.com'")
conn.commit()
rows = cursor.execute("SELECT * FROM user")
for row in rows:
    print(row)
```

```
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39, 'M')
('shelly@www.com', 'Shelly', 'Milar', '123, Ocean View Lane', 3
9, 'F')
```

Updating Specific Values in a Table...

Now let's go ahead with another update, this time around updating the *email* field in the **user** table for the user 'Tom':

In [41]:
```
cursor.execute("UPDATE user SET email='tom@outlook.com' WHERE email
='tom@web.com'")
conn.commit()
rows = cursor.execute("SELECT * FROM user")
for row in rows:
    print(row)
```

```
('tom@outlook.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu Ci
ty', 39, 'M')
('shelly@www.com', 'Shelly', 'Milar', '123, Ocean View Lane', 3
9, 'F')
```

As before, let's see on the next slide what the **comments** table looks like, now that the *email* of the user 'Tom' has been updated to 'tom@outlook.com' in the **user** table.

Updating Specific Values in a Table...

In [42]:
```
rows = cursor.execute("SELECT * FROM comments LIMIT 18")
for row in rows:
    print(row)
```

```
('tom@outlook.com', 'This is comment 0 by Tom Fake')
('tom@outlook.com', 'This is comment 1 by Tom Fake')
('tom@outlook.com', 'This is comment 2 by Tom Fake')
('tom@outlook.com', 'This is comment 3 by Tom Fake')
('tom@outlook.com', 'This is comment 4 by Tom Fake')
('tom@outlook.com', 'This is comment 5 by Tom Fake')
('tom@outlook.com', 'This is comment 6 by Tom Fake')
('tom@outlook.com', 'This is comment 7 by Tom Fake')
('tom@outlook.com', 'This is comment 8 by Tom Fake')
('tom@outlook.com', 'This is comment 9 by Tom Fake')
('shelly@www.com', 'This is comment 0 by Shelly Milar')
('shelly@www.com', 'This is comment 1 by Shelly Milar')
('shelly@www.com', 'This is comment 2 by Shelly Milar')
('shelly@www.com', 'This is comment 3 by Shelly Milar')
('shelly@www.com', 'This is comment 4 by Shelly Milar')
('shelly@www.com', 'This is comment 5 by Shelly Milar')
('shelly@www.com', 'This is comment 6 by Shelly Milar')
('shelly@www.com', 'This is comment 7 by Shelly Milar')
```

Updating Specific Values in a Table...

As we saw on the previous slide, updating the *email* for the user 'Tom' from 'tom@web.com' to 'tom@outlook.com' in the **user** table also changed the value of *user_id* in all children rows of 'Tom' in the **comments** table. That's because the field *user_id* in the **comments** table was linked to the field *email* in the **user** table and we had used **ON UPDATE CASCADE** when creating the table **comments** on slide 61.

Updating Specific Values in a Table...

Once again, before proceeding further, let's update the *email* for 'Tom' from 'tom@outlook.com' back to 'tom@web.com', so that the rest of the lesson remains unaffected:

In [43]:
```
cursor.execute("UPDATE user SET email='tom@web.com' WHERE email='tom
@outlook.com'")
conn.commit()
rows = cursor.execute("SELECT * FROM user")
for row in rows:
    print(row)
```

```
('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu City',
39, 'M')
('shelly@www.com', 'Shelly', 'Milar', '123, Ocean View Lane', 3
9, 'F')
```

Section 16 of 16

RDBMS and DataFrames

RDBMS and DataFrames

- We have looked into many fundamental aspects of storing and querying data from a database so far.
- However, as a data wrangling expert, we need our data to be packed and presented as a DataFrame so that we can perform quick and convenient operations on them.
- So let's see how we can do that on the next couple of slides.

RDBMS and DataFrames

As a first step, let's import the **pandas** library:

In [44]:
```
import pandas as pd
```

Now let's create a list of names for the columns of our Dataframe from the fields *email*, *first_name*, *last_name*, *age*, *gender*, and *comments* in our database. Also, let's create an empty **data** list for the data rows:

In [45]:
```
columns = ["Email", "First Name", "Last Name", "Age", "Gender", "Com
ments"]
data = []
```

RDBMS and DataFrames...

We can now write the SQL query to **SELECT** the columns we are interested in from the temporary table resulting from the **JOIN** of the two tables in our database based **ON** the **comments** table's *user_id* field:

In [46]:
```
sql = """
    SELECT user.email, user.first_name, user.last_name, user.age, us
er.gender, comments.comments FROM comments
    JOIN user ON comments.user_id = user.email
"""
```

Now let's use the **execute** method on top of our **cursor** to execute the above SQL command and store it as **rows**:

In [47]:
```
rows = cursor.execute(sql)
rows
```

Out[47]:
```
<sqlite3.Cursor at 0x120ba35eb20>
```

RDBMS and DataFrames...

Let's append each row from our query result **rows** to our **data** list:

In [48]:
```
for row in rows:
    data.append(row)
```

In [49]:
```
data[:2]
```

Out[49]:
```
[('tom@web.com', 'Tom', 'Fake', 39, 'M', 'This is comment 0 by T
om Fake'),
 ('tom@web.com', 'Tom', 'Fake', 39, 'M', 'This is comment 1 by T
om Fake')]
```

Finally, let's create a Dataframe with the **data** rows above and the **columns** header we created earlier on slide 92:

In [50]:
```
df = pd.DataFrame(data, columns=columns)
```

RDBMS and DataFrames...

Let's have a look at the resulting Dataframe:

In [51]:
```
df
```

Out[51]:

	Email	First Name	Last Name	Age	Gender	Comments
0	tom@web.com	Tom	Fake	39	M	This is comment 0 by Tom Fake
1	tom@web.com	Tom	Fake	39	M	This is comment 1 by Tom Fake
2	tom@web.com	Tom	Fake	39	M	This is comment 2 by Tom Fake
3	tom@web.com	Tom	Fake	39	M	This is comment 3 by Tom Fake
4	tom@web.com	Tom	Fake	39	M	This is comment 4 by Tom Fake
5	tom@web.com	Tom	Fake	39	M	This is comment 5 by Tom Fake
6	tom@web.com	Tom	Fake	39	M	This is comment 6 by Tom Fake
7	tom@web.com	Tom	Fake	39	M	This is comment 7 by Tom Fake
8	tom@web.com	Tom	Fake	39	M	This is comment 8 by Tom Fake
9	tom@web.com	Tom	Fake	39	M	This is comment 9 by Tom Fake
10	shelly@www.com	Shelly	Milar	39	F	This is comment 0 by Shelly Milar
11	shelly@www.com	Shelly	Milar	39	F	This is comment 1 by Shelly Milar
12	shelly@www.com	Shelly	Milar	39	F	This is comment 2 by Shelly Milar
13	shelly@www.com	Shelly	Milar	39	F	This is comment 3 by Shelly Milar
14	shelly@www.com	Shelly	Milar	39	F	This is comment 4 by Shelly Milar
15	shelly@www.com	Shelly	Milar	39	F	This is comment 5 by Shelly Milar
16	shelly@www.com	Shelly	Milar	39	F	This is comment 6 by Shelly Milar
17	shelly@www.com	Shelly	Milar	39	F	This is comment 7 by Shelly Milar
18	shelly@www.com	Shelly	Milar	39	F	This is comment 8 by Shelly Milar
19	shelly@www.com	Shelly	Milar	39	F	This is comment 9 by Shelly Milar

RDBMS and DataFrames...

There's also a quick alternative to create a Dataframe from an SQLite database. But before we look at that, we need to know about the **description** attribute of our **cursor** object when we use it to select data from one or more tables:

In [52]:
```
cursor = cursor.execute('select * from user')
cursor.description
```

Out[52]:
```
(('email', None, None, None, None, None, None),
 ('first_name', None, None, None, None, None, None),
 ('last_name', None, None, None, None, None, None),
 ('address', None, None, None, None, None, None),
 ('age', None, None, None, None, None, None),
 ('gender', None, None, None, None, None, None))
```

RDBMS and DataFrames...

In order to fetch the rows from my **cursor**, I invoke the **fetchall** method on top of it:

In [53]:
```
rows = cursor.fetchall()
rows
```

Out[53]:
```
[('tom@web.com', 'Tom', 'Fake', '123 Fantasy lane, Fantasu Cit
y', 39, 'M'),
 ('shelly@www.com', 'Shelly', 'Milar', '123, Ocean View Lane', 3
9, 'F')]
```

And now I can use the following code snippet to create a Dataframe from the **user** table:

In [54]:
```
pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
```

Out[54]:

	email	first_name	last_name	address	age	gender
0	tom@web.com	Tom	Fake	123 Fantasy lane, Fantasu City	39	M
1	shelly@www.com	Shelly	Milar	123, Ocean View Lane	39	F

RDBMS and DataFrames...

Similarly, I can use the **cursor.description** code to create my Dataframe from the join of the tables **user** and **comments** just by replacing **select * from user** with the **sql** query I created earlier:

In [55]:
```
cursor = cursor.execute(sql)
cursor.description
```

Out[55]:
```
(('email', None, None, None, None, None, None),
 ('first_name', None, None, None, None, None, None),
 ('last_name', None, None, None, None, None, None),
 ('age', None, None, None, None, None, None),
 ('gender', None, None, None, None, None, None),
 ('comments', None, None, None, None, None, None))
```

In [56]:
```
rows = cursor.fetchall()
```

RDBMS and DataFrames...

Finally, I can create the Dataframe:

In [57]:
```
pd.DataFrame(rows, columns=[x[0] for x in cursor.description])
```

Out[57]:

	email	first_name	last_name	age	gender	comments
0	tom@web.com	Tom	Fake	39	M	This is comment 0 by Tom Fake
1	tom@web.com	Tom	Fake	39	M	This is comment 1 by Tom Fake
2	tom@web.com	Tom	Fake	39	M	This is comment 2 by Tom Fake
3	tom@web.com	Tom	Fake	39	M	This is comment 3 by Tom Fake
4	tom@web.com	Tom	Fake	39	M	This is comment 4 by Tom Fake
5	tom@web.com	Tom	Fake	39	M	This is comment 5 by Tom Fake
6	tom@web.com	Tom	Fake	39	M	This is comment 6 by Tom Fake
7	tom@web.com	Tom	Fake	39	M	This is comment 7 by Tom Fake
8	tom@web.com	Tom	Fake	39	M	This is comment 8 by Tom Fake
9	tom@web.com	Tom	Fake	39	M	This is comment 9 by Tom Fake
10	shelly@www.com	Shelly	Milar	39	F	This is comment 0 by Shelly Milar
11	shelly@www.com	Shelly	Milar	39	F	This is comment 1 by Shelly Milar
12	shelly@www.com	Shelly	Milar	39	F	This is comment 2 by Shelly Milar
13	shelly@www.com	Shelly	Milar	39	F	This is comment 3 by Shelly Milar
14	shelly@www.com	Shelly	Milar	39	F	This is comment 4 by Shelly Milar
15	shelly@www.com	Shelly	Milar	39	F	This is comment 5 by Shelly Milar
16	shelly@www.com	Shelly	Milar	39	F	This is comment 6 by Shelly Milar
17	shelly@www.com	Shelly	Milar	39	F	This is comment 7 by Shelly Milar
18	shelly@www.com	Shelly	Milar	39	F	This is comment 8 by Shelly Milar
19	shelly@www.com	Shelly	Milar	39	F	This is comment 9 by Shelly Milar