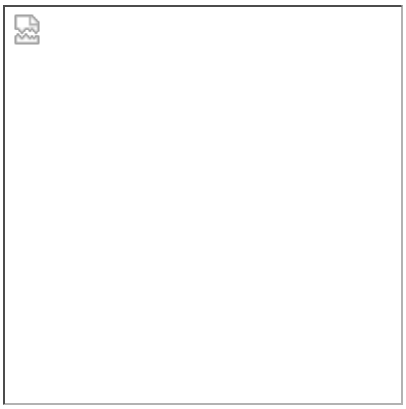
Matthew Peetz

Regis University

MSDS 621

Week 4 Lab: SQL vs NoSQL Databases



This week you will be building and querying both a SQL and NoSQL database. The dataset that we will be using is the MovieLens 1 million ratings dataset.

The MovieLens dataset is comprised of 3 files:

* Users
* Movies
* Reviews

One complete review consists of data from all three tables joined together. We will work through that part together. All 3 files and a dataset descriptive document are located in the assign_wk4 folder.

Assignment Requirements

At a high level, this assignment will follow our lecture material's progression:

- Load MovieLens tables into SQLite (good time to find that "multiple insert" I hinted about...)
- Create a query to retrieve reviews into a cursor
- Create a dataclass that represents a movie review
- Translate rows of the cursor into a list of MovieReview objects
- Translate the list of MovieReviews into a list of dictionaries
- Load the list of dictionaries into TinyDB (using `insert_multiple()`)

To help you out, I have written most of the code for you. You need to go through and replace all the `********` strings with the appropriate code. Also, there are questions imbedded in the code. So keep an eye out for those. You need to thoroughly answer all of these.

Oh! Feel free to add additional markdown text as you go to capture your thoughts/analysis.

Deliverables

Upload your Jupyter Notebook.

Note:: Make sure you have clearly indicated the answers to each question within your notebook.

Assignment Code

Part 1 - Storing in SQLite

In this part, you are expected to read MovieLens's README file to find information to proceed.

Hint:: Jupyter notebook and JupyterLabs can open it.

Questions::
1) What columns link the 3 files together to form a single review?
2) What seperator(s) are used in these files?

1) User ID is the unique key for the uses file. Movie ID is the unique key for the movies file. They are linked in the review file which has both of these foreign keys.

2) A double colon "::" is used to seperate the data in these files, follow by a line return

In [1]:
```
import dataset
```

In [4]:
```
# Fill in between the quotes for your own system
sql_db_path = "\\Users\matth\MSDS_621_Data_Wrangling\Week4"
```

In [5]:
```
# Fill in the connection string between the parentheses
db = dataset.connect('sqlite:///databases/movie_review.db')
```

In [6]:
```
# Are these files comma-separated? (Question #2 above)
separator = '::'
```

In [21]:
```
# Get column names from the README
# Replace *'s with column names
users_head = "UserID::Gender::Age::Occupation::Zip_code".split(separator)
movies_head = "MovieID::Title::Genres".split(separator)
ratings_head = "UserID::MovieID::Rating::Timestamp".split(separator)
```

Questions::

3) Before executing the next line, stop and think what should be output. Describe in detail what yo are expecting

3) I would expect to get the column header, without any "::" turned into a list

In [22]:
```
type(users_head)
```

Out[22]:
```
list
```

In [23]:
```
users_head
```

Out[23]:
```
['UserID', 'Gender', 'Age', 'Occupation', 'Zip_code']
```

Questions::

4) Does the actual output match your expectation? If not, explain why?

4) Yes, it does, I used the "type" command to make sure that I did have a list as well

Now it is time to create the database tables. As mentioned, there will be three of them. Interestingly, the `USERS` table and the `MOVIES` table both have unique ID fields alread - we will have to take that into account. The `RATINGS` table, on the other hand, does not have a unique ID column, so we don't have to worry about it.

The general, simple format to create a table is:

`table_variable = db.create_table("table_name")` . # This is what you use for ratings.

But, in the case where the data already has an ID, we have to tell DataSet about it. The general form is:

`table_variable = db.create_table("table_name", primary_id="ID_column_name", primary_type=db.types.integer)`

So, in the case of the `MOVIES` table, the `MovieID` column is the primary key.

In [24]:
```
try:
    ratings_table = db.create_table("ratings")
except:
    print("that did not work")
```

In [25]:
```
try:
    users_table = db.create_table(
        "users", primary_id="User_ID", primary_type=db.types.integer
    )
except:
    print("that did not work")
```

In [26]:
```
try:
    movies_table = db.create_table(
        "movies", primary_id="Movie_ID", primary_type=db.types.integer
    )
except:
    print("that did not work")
```

You can, and probably should, put those `create_table()` function calls in `try / except` blocks.

Let's set up variables for the data file names:

In [27]:
```
users_file = "assign_wk4/users.dat"
movies_file = "assign_wk4/movies.dat"
ratings_file = "assign_wk4/ratings.dat"
```

Questions::
Having the ID column in the data caused one difference in our table creation (in comparison to how we did this in Week 2).
And yes, you might have to go back and review that!)

5) Do you notice any other differences, and if so, what are they?
6) If there is a difference, why is it different?
7) If there is a difference, how does it affect the data retrieved with a SELECT statement?

5) One other difference is the primary keys in the user and movies files, which allow the review file to by connected to them

6. The primary keys make sure that there is a unique value in each of the rows, ie. if there are multiple John Smiths they would each still have a key. It also makes the table joinable with the review files

7. The SELECT statement can now select date from multiple tables to JOIN together

OK. Here it is, the moment you've all been waiting for -- we can start stuffing data in the tables we created.

There is one more thing to show you about the insert. You might remember that this data set is called **"ml-1m"** which stands for *MovieLens - 1 million rows*. In the grand scheme of modern data storage, 1 million rows isn't a huge number, but it **is** enough to make even a fast laptop like mine choke a bit, so we are going to use a technique that many RDBMS systems call **Bulk Insert.**

Bulk insert is optimized for inserting large amounts of similarly-structured data. SQLite is relatively fast so let's do a quick comparison, using the user's table. After that, it will be **up to you to populate the other 2 tables.** We will also use that progress bar from the FTE, just for fun.

In [28]:
```
%%time
with open (users_file) as ufile:
    for line in ufile:
        u_dict = dict(zip(users_head, line.split("::")))
        users_table.insert(u_dict)
```

```
CPU times: total: 1.47 s
Wall time: 5.18 s
```

In [29]:
```
# Drop the table before trying to insert again
# You might remember how to do this from Week 2.

if (len(db.tables) > 0):
    for table in db.tables:
        db[table].drop()

# HINT: you need the table name, and the drop command...
```

In [30]:
```
%%time
users_list = []
with open(users_file) as ufile:
    for line in ufile:
        users_list.append(dict(zip(users_head, line.split("::"))))
users_table.insert_many(users_list)
```

```
CPU times: total: 0 ns
Wall time: 64.1 ms
```

Now **YOU** can decide how you want to do the other two tables, using `insert()` or `insert_many()`.

Since there are only 2 of them, I will let you do them one by one. *Don't get used to it!*

In [35]:
```
# code to insert or insert_many movies_file here
movies_list = []
with open(movies_file) as ufile:
    for line in ufile:
        movies_list.append(dict(zip(movies_head, line.split("::"))))
movies_table.insert_many(movies_list)
# I had an odd error in this file, the line below gets around it
# with open(movies_file, errors="ignore") as mfile:
```

In [36]:
```
# code to insert or insert_many ratings_file here
ratings_list = []
with open(ratings_file) as ufile:
    for line in ufile:
        ratings_list.append(dict(zip(ratings_head, line.split("::"))))
ratings_table.insert_many(ratings_list)
```

Success! At this point you should have a working relational database containing the MovieLens data!.

In [ ]:
```
# code to check that there is data in all your tables.
```

In [37]:
```
len(ratings_table)
```

Out[37]:
```
1000209
```

In [38]:
```
len(movies_table)
```

Out[38]:
```
3883
```

In [39]:
```
len(users_table)
```

Out[39]:
```
6040
```

SQL Joins

Records are divided into multiple tables due to the process of **data normalization**. We have to **join tables** in our `SELECT` queries to get one full movie rating.

In general, the **left join** or **left inner join** is the most common, although there are several types. The *left* part refers to the actual layout if you were putting the printed tables side by side on your desk. A left join/left inner join means you have a table with foreign keys on the left side and you are trying to match those keys to their primary keys on the right. Let's look at an example:

Movie

MovieID	Title	Genre		
1	Toy Story (1995)	Animation	Children's	Comedy
2	Jumanji (1995)	Adventure	Children's	Fantasy
3	Grumpier Old Men (1995)	Comedy	Romance	
4	Waiting to Exhale (1995)	Comedy	Drama	
5	Father of the Bride Part II (1995)	Comedy		

Users

UserID	Gender	Age	Occupation	ZipCode
1	F	1	10	48067
2	M	56	16	70072
3	M	25	15	55117
4	M	45	7	02460
5	M	25	20	55455

Ratings

UserID	MovieID	Rating	Timestamp
1	1193	5	978300760
1	661:	3	978302109
1	914:	3	978301968
1	3408	4	978300275
1	2355	5	978824291

It should be obvious in this small example that Ratings are linked to both Movie and Users through their ids. So, to get a complete rating record, we need the Movie record where the MovieIDs match and the user where the UserIDs match. In SQL that loobks like this:

SQL keywords are in caps.

```
SELECT m.title, m.genres, u.Gender, u.Age, u.Occupation, u.ZipCode,
r.Rating, r.Timestamp
```

```
    FROM movies m
    INNER JOIN ratings r ON m.MovieID = r.MovieID
    INNER JOIN users u ON r.UserID = u.UserID
    ORDER BY m.Title ASC;
```

Normally, when referencing columns from multiple tables, you have to prefix the column name with the table name, but in this case I used a shortcut -- in the FROM part, I gave each table a one-letter alias.

Also notice the last two lines. These will put all the matching movie titles together and then alphabetize the list.

Let's try it and see what comes out.

In [44]:
```
# Put the query in here. NOTE: If you break up the lines, you need
# a "continuation character" at the end of the line.

movie_query = "SELECT m.title, m.genres, u.Gender, u.Age, u.Occupation, u.Zip_code,
    FROM movies m \
    INNER JOIN ratings r ON m.MovieID = r.MovieID \
    INNER JOIN users u ON r.UserID = u.UserID \
    ORDER BY m.Title ASC;"
```

In [45]:
```
# Add the command to execute a query.
# Reference: https://dataset.readthedocs.io/en/latest/api.html#dataset.Database.que
query_result = db.query(movie_query)
```

In [47]:
```
print(query_result)
```

```
<dataset.util.ResultIter object at 0x0000019F679FB940>
```

In [48]:
```
# Convert that result into a list for ease of use.
movie_list = []
for movie in query_result:
    movie_list.append(movie)

# Print out first movie to see what is stored in the list
movie_list[0]
```

Out[48]:
```
OrderedDict([('Title', '$1,000,000 Duck (1971)'),
             ('Genres', "Children's|Comedy\n"),
             ('Gender', 'F'),
             ('Age', '35'),
             ('Occupation', '1'),
             ('Zip_code', '82601\n'),
             ('Rating', '3'),
             ('Timestamp', '975093319\n')])
```

Part 2 - Storing in TinyDB

Hopefully you remember that TinyDB inserts dictionaries as documents. This means that the data in the `movie_list` variable is in the correct form to insert.

In [49]:
```
from tinydb import TinyDB, Query, where

tiny_db = TinyDB("ml_nosql.json")
```

In [ ]:
```
tiny_db.insert_multiple(movie_list)
```

Success! At this point you should have a working NoSQL database containing the MovieLens data!.

Now we can actually start using this data.

SQL has some aggregation functions that can be interesting. For example, to find an average of a numeric column:

```
select avg(column) from table where condition;
```

Note: At this point, I'm not sure that the Dataset library gains us anything, since we are just passing straight SQL through it. You can continue to use Dataset or switch to the SQLite3 library. I'll stay with Dataset, since it is already loaded.

We can modify our join from above to get an average rating from women for the movie "Die Hard" like this:

In [88]:
```
movie_query = "select m.title, u.Gender, avg(r.Rating)\
from movies m \
inner join ratings r on m.MovieID = r.MovieID \
inner join users u on r.UserID = u.UserID \
where u.Gender = 'F' and m.title = 'Die Hard (1988)';"
```

In [89]:
```
query_result = db.query(movie_query)
```

In [90]:
```
# A quick little list comprehension to extract the results
f_avg = [row for row in query_result]

f_avg
```

Out[90]:
```
[OrderedDict([('Title', 'Die Hard (1988)'),
              ('Gender', 'F'),
              ('avg(r.Rating)', 3.9185667752442996)])]
```

In [91]:
```
# So, to print it nicely:
print(f"Average female rating for {f_avg[0]['Title']} is {f_avg[0]['avg(r.Rating)']
```

```
Average female rating for Die Hard (1988) is 3.9185667752442996
```

That process is slightly more manual in TinyDB. Here, we can use TinyDB's `where()` command along with `matches()` to find movies with the right title, then use a logical and `&` to limit it to women. We can also take advantage of Python's built in `sum()` and `len()` commands to help us out.

It sounds more complicated than it is. Like this:

In [99]:
```
female_dh_set = tiny_db.search( (where('Title').matches('Die Hard')) & (where('Gend
```

That gives us a list of dictionaries, prove that is true to yourself, if you need to.

The rest is simple (Remember all numbers are stored as strings!):

In [100…
```
dh_avg_f = sum(int(r['Rating']) for r in female_dh_set) / len(female_dh_set)
```

In [101…
```
print(f'Average: {dh_avg_f}')
```

```
Average: 3.7107438016528924
```

Questions::

8) What is the age range of female reviewers of "Gone With The Wind?" (Hint: in SQL, you can use a column more than once.)(Hint2: There may be built in functions that help.)

9) Using the relational database you built, compare M and F average ratings for "Die Hard."

10) Do the same comparison, as in question 9, with the NoSQL database.

11) Do the averages match? If they don't, please provide a detailed explanation as to why not. You will need to write additional queries to back-up and defend your analysis.(Hint: They don't match!)

8. What is the age range of female reviewers of "Gone With The Wind"

In [116…
```
female_gwtw_set = tiny_db.search( (where('Title').matches('Gone with the Wind')) &

gwtw_min_age = min(int(r['Age']) for r in female_gwtw_set)
gwtw_max_age = max(int(r['Age']) for r in female_gwtw_set)

print(f'Minimum Age: {gwtw_min_age}')
print(f'Maximum Age: {gwtw_max_age}')
```

```
Minimum Age: 1
Maximum Age: 56
```

In [156…
```
movie_query = "select m.title, u.Gender, r.Rating, min(u.Age)\
from movies m \
inner join ratings r on m.MovieID = r.MovieID \
inner join users u on r.UserID = u.UserID \
where u.Gender = 'F' and m.title = 'Gone with the Wind (1939)';"
```

In [157…
```
query_result = db.query(movie_query)
```

In [158…
```
# A quick little list comprehension to extract the results
gwtw_min = [row for row in query_result]

gwtw_min
```

Out[158]:
```
[OrderedDict([('Title', 'Gone with the Wind (1939)'),
              ('Gender', 'F'),
              ('Rating', '5'),
              ('min(u.Age)', '1')])]
```

In [159…
```
movie_query = "select m.title, u.Gender, r.Rating, max(u.Age)\
from movies m \
inner join ratings r on m.MovieID = r.MovieID \
inner join users u on r.UserID = u.UserID \
where u.Gender = 'F' and m.title = 'Gone with the Wind (1939)';"
```

In [160…
```
query_result = db.query(movie_query)
```

In [161…
```
# A quick little list comprehension to extract the results
gwtw_max = [row for row in query_result]

gwtw_max
```

Out[161]:
```
[OrderedDict([('Title', 'Gone with the Wind (1939)'),
              ('Gender', 'F'),
              ('Rating', '5'),
              ('max(u.Age)', '56')])]
```

In [163…
```
# So, to print it nicely:
print("The Youngest Age for viewers of Gone With The wind is", {gwtw_min[0]['min(u.
print("The Youngest Age for viewers of Gone With The wind is", {gwtw_max[0]['max(u.
```

```
The Youngest Age for viewers of Gone With The wind is {'1'}
The Youngest Age for viewers of Gone With The wind is {'56'}
```

Unfortunately is would appear that someone set up an account and listed their age as 1, which means that that is the minimum age that someone watched Gone With the Wind at

9) Using the relational database you built, compare M and F average ratings for "Die Hard."

In [192…
```
movie_query = "select m.title, u.Gender, avg(r.Rating)\
from movies m \
inner join ratings r on m.MovieID = r.MovieID \
inner join users u on r.UserID = u.UserID \
where u.Gender = 'M' and m.title = 'Die Hard (1988)';"
```

In [193…
```
query_result = db.query(movie_query)
```

In [194…
```
# A quick little list comprehension to extract the results
m_avg = [row for row in query_result]

m_avg
```

Out[194]:
```
[OrderedDict([('Title', 'Die Hard (1988)'),
              ('Gender', 'M'),
              ('avg(r.Rating)', 4.1677704194260485)])]
```

The female average rating for Die Hard was 3.92, the mail average was 4.17.

10) Do the same comparison, as in question 9, with the NoSQL database.

In [201…
```
male_dh_set = tiny_db.search( (where('Title').matches('Die Hard')) & (where('Gender

dh_avg_m = sum(int(r['Rating']) for r in male_dh_set) / len(male_dh_set)

print(f'Average for Females: {dh_avg_f}')

print(f'Average for Males: {dh_avg_m}')
```

```
Average for Females: 3.7107438016528924
Average for Males: 3.833167825223436
```

The female average using the SQL database is 3.92, while the male average is 4.17. If the same comparison is done using the NoSQL database the female average is 3.71 and the male average is 3.83. Why is this?

In [202…
```
# Looking at the length of the male_dh_set NoSQL
print(len(male_dh_set))
```

```
3021
```

In [203…
```
# Looking at the length of the male info in the SQL set
movie_query = "select m.title, u.Gender, COUNT(r.Rating)\
from movies m \
inner join ratings r on m.MovieID = r.MovieID \
inner join users u on r.UserID = u.UserID \
where u.Gender = 'M' and m.title = 'Die Hard (1988)';"
```

In [204…
```
query_result = db.query(movie_query)
```

In [205…
```
# A quick little list comprehension to extract the results
m_count = [row for row in query_result]

m_count
```

Out[205]:
```
[OrderedDict([('Title', 'Die Hard (1988)'),
              ('Gender', 'M'),
              ('COUNT(r.Rating)', 1359)])]
```

The count for the NoSQL set is 3,021. Meaning that there are 3,021 reviews of Die Hard that meet the criteria. The SQL set is only 1,359, or about half the size. This means that the two commands are somehow pulling a different set of data out of the data base. Why is this?

In [216…
```
male_dh_set = tiny_db.search( (where('Title').matches('Die')) & (where('Gender').ma

dh_avg_m = sum(int(r['Rating']) for r in male_dh_set) / len(male_dh_set)

print(f'Average for Females: {dh_avg_f}')

print(f'Average for Males: {dh_avg_m}')
```

```
Average for Females: 3.7107438016528924
Average for Males: 3.832230311052283
```

In [ ]:
```
print(male_dh_set)
```

NOTE: I cleared the output of the above command as it created a 143 page notebook. It has a listed where you could see that all the die hard movies were being pulled, and not just the original christmas classic.

When you rerun the search you can use the word "Die" and see what the problem is. The term it not specific and it is pulling all the movies that have "Die" in the title. This includes Die Hard, Die Hard with a Vengance, Dir Harder, etc. All great movies but not the one that we were searching for.

Conclusion

In previous cources at Regis I have used the cmd terminal to create SQL databases, I have also played around (a little) with MSSql. Both of them require a great deal of patience as you punch commands into the terminal and wait for a response. Using TinyDB, python, and a jupyter notebook made storing and accessing data in a database much more stream lined and easier.

I also learned that you have to be really careful with your query commands. I would have never though twice about either of the numbers reported for the Die Hard ratings if I had been using just one of the programs. It was a good demonstration on how to utlize query commands, analyze the data that is returned, and double check it.

In [ ]: