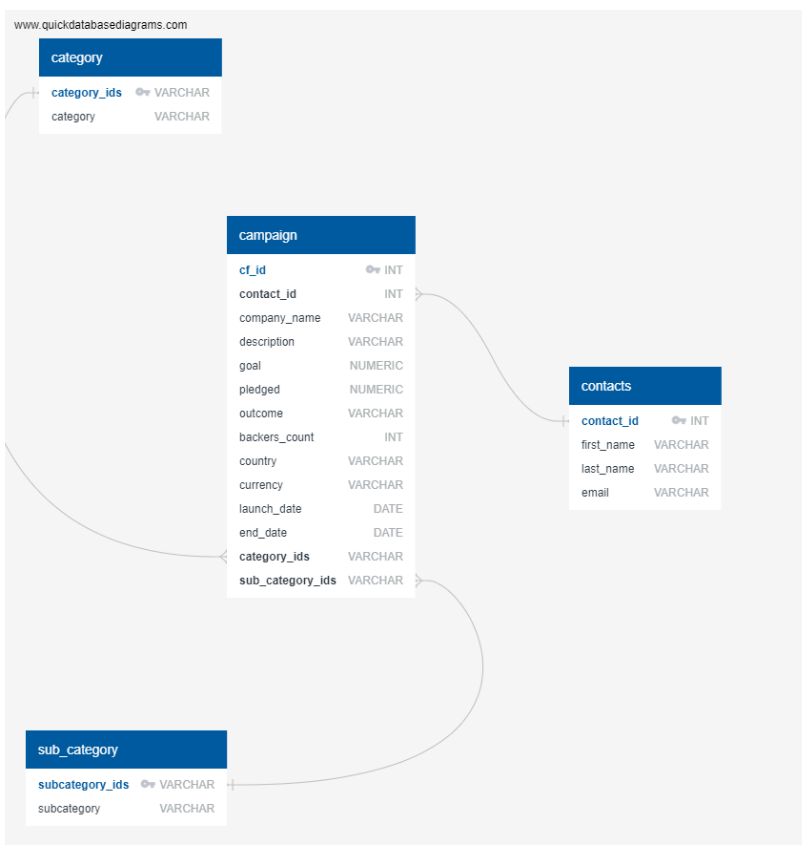
Project 2 Group 12: Write-up

In this project we converted some rough data into translatable csv files that then helped perform some basic data analysis. We also designed a database and used python code to load the csv files into pgAdmin. From there we performed some queries on our newly transformed data.

Step 0: ERD Diagram

An ERD diagram was engineered using quickdatabasediagrams.com. After understanding how we wished the data to be arranged, we used this website to architect our database hierarchy. The website also provided us with SQL code that was used in pgAdmin to troubleshoot our imports. The schema is as follows:



Step 1: Transformation – Jessie Wayne

The first step of this project was to transform two given Excel files into four usable .csv files for easy loading into a database for querying and creating visualizations.
Step one involved importing the crowdfunding Excel document into a pandas DataFrame for cleanup. Our next step was splitting the "category & subcategory" column into two separate columns using a string split and adding each piece to its respective column. Once these columns were established, we found the unique length of each to create new columns for unique IDs for each category and subcategory. These were then saved into two new DataFrames, which were then exported to .csv files for import.

```
# Get a brief summary of the crowdfunding_info DataFrame.
crowddf.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 15 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   cf_id                   1000 non-null   int64
 1   contact_id              1000 non-null   int64
 2   company_name            1000 non-null   object
 3   blurb                   1000 non-null   object
 4   goal                    1000 non-null   int64
 5   pledged                 1000 non-null   int64
 6   outcome                 1000 non-null   object
 7   backers_count           1000 non-null   int64
 8   country                 1000 non-null   object
 9   currency                1000 non-null   object
 10  launched_at             1000 non-null   int64
 11  deadline                1000 non-null   int64
 12  staff_pick              1000 non-null   bool
 13  spotlight               1000 non-null   bool
 14  category & sub-category 1000 non-null   object
dtypes: bool(2), int64(7), object(6)
memory usage: 103.6+ KB
```

```
category_df
```

	category_ids	category
0	cat1	food
1	cat2	music
2	cat3	technology
3	cat4	theater
4	cat5	film & video
5	cat6	publishing
6	cat7	games
7	cat8	photography
8	cat9	journalism

```
subcategory_df
```

	sub_category_ids	sub_category
0	subcat1	food trucks
1	subcat2	rock
2	subcat3	web
3	subcat4	plays
4	subcat5	documentary
5	subcat6	electric music
6	subcat7	drama
7	subcat8	indie rock
8	subcat9	wearables
9	subcat10	nonfiction
10	subcat11	animation
11	subcat12	video games
12	subcat13	shorts
13	subcat14	fiction
14	subcat15	photography books
15	subcat16	radio & podcasts
16	subcat17	metal
17	subcat18	jazz
18	subcat19	translations
19	subcat20	television
20	subcat21	mobile games
21	subcat22	world music
22	subcat23	science fiction
23	subcat24	audio

Next, we returned to the original DataFrame for further clean-up which included removing and renaming columns, adjusting data types, and swapping out category and subcategory columns for their respective IDs. The "blurb," "launched at," and "deadline" columns were renamed to "description," "launch date," and "end date," respectively. The "goal" and "pledged" columns were changed to float data types, while the "launch date" and "end date" columns were changed to datetime. We then performed an inner join with the category and subcategory DataFrames to match the category and subcategory columns with their IDs. Finally, we dropped unnecessary columns before saving the cleaned DataFrame to a .csv file for import.

```
# Drop unwanted columns
campaign_merged_df.drop(['staff_pick', 'spotlight', 'category & sub-category', 'category', 'sub_category'], axis=1, inplace=True)
campaign_merged_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 14 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   cf_id             1000 non-null   int64
 1   contact_id        1000 non-null   int64
 2   company_name      1000 non-null   object
 3   description       1000 non-null   object
 4   goal              1000 non-null   float64
 5   pledged           1000 non-null   float64
 6   outcome           1000 non-null   object
 7   backers_count     1000 non-null   int64
 8   country           1000 non-null   object
 9   currency          1000 non-null   object
 10  launch_date       1000 non-null   datetime64[ns]
 11  end_date          1000 non-null   datetime64[ns]
 12  category_ids      1000 non-null   object
 13  sub_category_ids  1000 non-null   object
dtypes: datetime64[ns](2), float64(2), int64(3), object(7)
memory usage: 109.5+ KB
```

The final step was to create our contacts DataFrame from the second Excel sheet, where all information was listed in one column. We did this twice, using both JSON and regex methods.

Using JSON:

Our first step here was creating an empty list for the extracted values. Then we looped through all the rows, converting the JSON data into Python dictionaries and appending them to the list. Next, we converted the list of dictionaries into a DataFrame, checked the data types, split the "name" column into "first name" and "last name" columns, rearranged the columns for readability, and exported the DataFrame to a .csv file for import.

```
# Iterate through the contact_info_df and convert each row to a dictionary.
import json

dict_values = []
for i, row in contact_info_df.iterrows():
    data = row[0]
    converted_data = json.loads(data)
    # Iterate through each dictionary (row) and get the values for each row using list comprehension.
    row_values =[v for k, v in converted_data.items()]
    # Append the list of values for each row to a list.
    dict_values.append(row_values)

# Print out the list of values for each row.
print(dict_values)
```

Using regex:

Finally, we also used regex to create new columns by finding patterns in the single column of provided data. To extract the contact ID, we used the pattern “contact_id": (\d{4})”, which matches and extracts exactly four digits following "contact_id". Once extracted, we changed this data type to integer. Next, we extracted the name and email columns using the pattern “name": "([^"]*), which extracts text following "name:" or "email:" without capturing the double quotes. Finally, we dropped the original column, split the "name" column into "first name" and "last name" columns, and saved the cleaned DataFrame to a .csv file for import.

```
# Extract the four-digit contact ID number.
contact_info_df_copy['contact_id'] = contact_info_df_copy['contact_info'].str.extract(r'contact_id": (\d{4})')
contact_info_df_copy
```

```
# Check the datatypes.
contact_info_df_copy.info()
```

```
# Convert the "contact_id" column to an int64 data type.
contact_info_df_copy['contact_id']=contact_info_df_copy['contact_id'].astype(int)
contact_info_df_copy['contact_id'].dtype
```

```
# Extract the name of the contact and add it to a new column.
contact_info_df_copy['name'] = contact_info_df_copy['contact_info'].str.extract(r'"name": "([^"]*)"')
contact_info_df_copy
```

```
# Extract the email from the contacts and add the values to a new column.
contact_info_df_copy['email'] = contact_info_df_copy['contact_info'].str.extract(r'"email": "([^"]*)"')
contact_info_df_copy
```

Step 2: Loading - Daniel Purrier

During the loading phase we ran into some minor issues that we were able to quickly overcome between us. A majority of the loading process was covered in the class example, and we only needed to plug in and tweak certain parts of the code to operate in Jupyter Notebook. Below is a snapshot of the start of the loading process.

```
[2]: SQL_USERNAME = "postgres"
     SQL_PASSWORD = "postgres" # change this
     SQL_IP = "localhost"
     PORT = 5432
     DATABASE = "project_2" # change this

[3]: connection_string = f"postgresql+psycopg2://{SQL_USERNAME}:{SQL_PASSWORD}@{SQL_IP}:{PORT}/{DATABASE}"
     engine = create_engine(connection_string)

[4]: # explore and understand the data

     # Create the inspector and connect it to the engine
     inspector = inspect(engine)

     # Collect the names of tables within the database
     tables = inspector.get_table_names()

     # Using the inspector to print the column names within each table and its types
     for table in tables:
         print(table)
         columns = inspector.get_columns(table)
         for column in columns:
             print(column["name"], column["type"])

         print()

     contacts
     contact_id INTEGER
     first_name VARCHAR
     last_name VARCHAR
     email VARCHAR
```

There is a bit more code than what is shown here. After inputting it we began to load the CSV's themselves. The Campaign csv is the first loaded on this notebook.

```
[5]: df_campaign = pd.read_csv("Resources/campaign.csv")
     df_campaign.head()
```

[5]:

	cf_id	contact_id	company_name	description	goal	pledged	outcome	backers_count	country	currency	launch_date	end_date	category_i
0	147	4661	Baldwin, Riley and Jackson	Pre-emptive tertiary standardization	100.0	0.0	failed	0	CA	CAD	1970-01-01 00:00:01.581573600	1970-01-01 00:00:01.614578400	ca
1	1621	3765	Odom Inc	Managed bottom-line architecture	1400.0	14560.0	successful	158	US	USD	1970-01-01 00:00:01.611554400	1970-01-01 00:00:01.621918800	ca
2	1812	4187	Melton, Robinson and Fritz	Function-based leadingedge pricing structure	108400.0	142523.0	successful	1425	AU	AUD	1970-01-01 00:00:01.608184800	1970-01-01 00:00:01.640844000	ca
3	2156	4941	Mcdonald, Gonzalez and Ross	Vision-oriented fresh-thinking conglomeration	4200.0	2477.0	failed	24	US	USD	1970-01-01 00:00:01.634792400	1970-01-01 00:00:01.642399200	ca
4	1365	2199	Larson-Little	Proactive foreground core	7600.0	5265.0	failed	53	US	USD	1970-01-01 00:00:01.608530400	1970-01-01 00:00:01.629694800	ca

We initially loaded all the csv's in Postgres and the campaign csv gave us the most trouble. To correct the issue we had to fix one of the column names for it to be recognized in Postgres. We also had to change the data types of the "goal" and "pledged" columns as they have decimals and are considered floats. Once these changes were made loading into Postgres was completed.

The next table loaded on our notebook was the category csv. We had no issues with loading this on either the Jupyter Notebook or Postgres.

[5]:
```
df_campaign = pd.read_csv("Resources/campaign.csv")
df_campaign.head()
```

[5]:

	cf_id	contact_id	company_name	description	goal	pledged	outcome	backers_count	country	currency	launch_date	end_date	category_i
0	147	4661	Baldwin, Riley and Jackson	Pre-emptive tertiary standardization	100.0	0.0	failed	0	CA	CAD	1970-01-01 00:00:01.581573600	1970-01-01 00:00:01.614578400	ca
1	1621	3765	Odom Inc	Managed bottom-line architecture	1400.0	14560.0	successful	158	US	USD	1970-01-01 00:00:01.611554400	1970-01-01 00:00:01.621918800	ca
2	1812	4187	Melton, Robinson and Fritz	Function-based leadingedge pricing structure	108400.0	142523.0	successful	1425	AU	AUD	1970-01-01 00:00:01.608184800	1970-01-01 00:00:01.640844000	ca
3	2156	4941	Mcdonald, Gonzalez and Ross	Vision-oriented fresh-thinking conglomeration	4200.0	2477.0	failed	24	US	USD	1970-01-01 00:00:01.634792400	1970-01-01 00:00:01.642399200	ca
4	1365	2199	Larson-Little	Proactive foreground core	7600.0	5265.0	failed	53	US	USD	1970-01-01 00:00:01.608530400	1970-01-01 00:00:01.629694800	ca

We did not have any issues with the subcategory csv either.

[11]:
```
df_subcategory = pd.read_csv("Resources/subcategory.csv")
df_subcategory.head()
```

[11]:

	sub_category_ids	sub_category
0	subcat1	food trucks
1	subcat2	rock
2	subcat3	web
3	subcat4	plays
4	subcat5	documentary

Lastly the contacts csv was loaded. We had to delete an extra row that had null data in it and also increase the memory for the amount of characters that were needed for the "emails" column before being able to import into Postgres. We then loaded it into Jupyter Notebook as follows:

```
13]: df_contacts = pd.read_csv("Resources/contacts.csv")
     df_contacts.head()
```

13]:

	contact_id	first_name	last_name	email
0	4661	Cecilia	Velasco	cecilia.velasco@rodrigues.fr
1	3765	Mariana	Ellis	mariana.ellis@rossi.org
2	4187	Sofie	Woods	sofie.woods@riviere.com
3	4941	Jeanette	Iannotti	jeanette.iannotti@yahoo.com
4	2199	Samuel	Sorgatz	samuel.sorgatz@gmail.com

Step 3: Analysis - James Lee

For our analysis we began by reading in our 4 csv files as data frames corresponding to each table in our database:

```
# Import Campaign CSV
df_campaign = pd.read_csv("Resources/campaign.csv")
# Import Category CSV
df_category = pd.read_csv("Resources/category.csv")
# Import Sub-Category CSV
df_subcategory = pd.read_csv("Resources/subcategory.csv")
# Import Contacts CSV
df_contacts = pd.read_csv("Resources/contacts.csv")
```

I then proceeded to convert those data frames into 4 tables that would be contained in a sqlite file:

```
import sqlite3

# Create a connection to the SQLite database
conn = sqlite3.connect('project_2.sqlite')

# Write the DataFrame to the SQLite database
df_campaign.to_sql('campaign', conn, if_exists='replace', index=False)
df_category.to_sql('category', conn, if_exists='replace', index=False)
df_subcategory.to_sql('sub_category', conn, if_exists='replace', index=False)
df_contacts.to_sql('contacts', conn, if_exists='replace', index=False)

# Commit and close the connection
conn.commit()
conn.close()

print("Database created successfully!")
```

After specifying my file path and creating my engine I wanted to verify that my file contained the necessary tables so I inspected the file using the code below:

```python
# INSPECT

# Create the inspector and connect it to the engine
inspector_gadget = inspect(engine)

# Collect the names of tables within the database
tables = inspector_gadget.get_table_names()

# print metadata for each table
for table in tables:
    print(table)
    print("-----------")

    # get columns
    columns = inspector_gadget.get_columns(table)
    for column in columns:
        print(column["name"], column["type"])

    print()
```

I was able to verify that all necessary tables existed in my file so I began my SQL queries. My first query was to explore the connection between the success and failure of companies based on their category:

category	company_count	outcome
film & video	11	canceled
film & video	59	failed
film & video	5	live
film & video	102	successful
food	4	canceled

I then expanded on this by performing a similar query to include sub-category:

category	sub_category	company_count	outcome
film & video	animation	1	canceled
film & video	animation	10	failed
film & video	animation	2	live
film & video	animation	21	successful
film & video	documentary	4	canceled

I then wanted to know how the average amount of backers differed between categories:

```
query = """SELECT c.category, AVG(backers_count) AS avg_backers_per_category
FROM campaign ca
JOIN category c ON ca.category_ids = c.category_ids
GROUP BY c.category;"""

cat_back = pd.read_sql(text(query), con=engine)
cat_back.set_index('category', inplace=True)
cat_back.head()
```

✓ 0.0s

category	avg_backers_per_category
film & video	684.691011
food	627.086957
games	784.625000
journalism	298.500000
music	737.154286

And then expanded the query in a similar way to my first two:

```
query = """SELECT sc.sub_category, AVG(backers_count) AS avg_backers_per_subcategory
FROM campaign ca
JOIN sub_category sc ON ca.sub_category_ids = sc.sub_category_ids
GROUP BY sc.sub_category;"""

sub_cat_back = pd.read_sql(text(query), con=engine)
sub_cat_back.set_index('sub_category', inplace=True)
sub_cat_back.head()
```

✓ 0.0s

sub_category	avg_backers_per_subcategory
animation	857.588235
audio	298.500000
documentary	714.950000
drama	438.243243
electric music	850.166667

Using these last two queries I created bar graphs out of their dataframes: