# Project A

**Question 1**

*Load data, preprocess it, apply transformations. Use Spark. This part is preliminary to the analysis part and the implementation part. Consider principals discussed in class: data formats, hierarchical data, (un)structured data, etc.*

**Answer 1**

**Loading and structuring CSV data**

At first, we loaded the CSV files as PySpark DataFrames.

We were receiving many cells as null types, as opposed to the Microsoft Excel reader.

This caused us to further examine the null cells as StringType prior to loading into the DataFrame.

We found the cause to the miss interpretation by Spark to the corrupt cells to be the char '"' quoted within an already quoted value. In addition to not interpreting the large multiline cells as multiline cells. Thankfully the Apache Spark documentation referred to this issue, (see: https://spark.apache.org/docs/latest/sql-data-sources-csv.html) and we found the following solution:

```
credits = spark.read.option("Header", True)\
.option("multiline", True)\
.option("escape", "\"")\
.csv(dataPath)
```

Output Prior to solution loading credits.csv:

```
credits.show(5)
```

```
+--------------------+--------------------+-----+
|                cast|                crew|   id|
+--------------------+--------------------+-----+
|[{'cast_id': 14, ...|"[{'credit_id': '...| null|
|[{'cast_id': 1, '...|[{'credit_id': '5...| 8844|
|[{'cast_id': 2, '...|[{'credit_id': '5...|15602|
|"[{'cast_id': 1, ...| 'credit_id': '52...| null|
|[{'cast_id': 1, '...|[{'credit_id': '5...|11862|
+--------------------+--------------------+-----+
```

Output using solution:

```
credits.show(5)
```

```
+--------------------+--------------------+-----+
|                cast|                crew|   id|
+--------------------+--------------------+-----+
|[{'cast_id': 14, ...|[{'credit_id': '5...|  862|
|[{'cast_id': 1, '...|[{'credit_id': '5...| 8844|
|[{'cast_id': 2, '...|[{'credit_id': '5...|15602|
|[{'cast_id': 1, '...|[{'credit_id': '5...|31357|
|[{'cast_id': 1, '...|[{'credit_id': '5...|11862|
+--------------------+--------------------+-----+
```

In cell B2 in credits.csv, causing the null output of first id in the DataFrame:

```
'Layout', 'name': 'DesirÃ©e Mourad', 'profile_path': Nor
Dresser', 'name': "Kelly O'Connell", 'profile_path': None},
resser', 'name': 'Sonoko Konishi', 'profile_path': None}, {
```

By solving this, we were able to load the CSV files as DataFrames containing StringType in each cell.

(See note on page 6, regarding further StringType cleaning and replacing done.)

At start we defined our data as structured, because we saw the comma delimiters and well-defined column titles. By further analysis we understood that our data is semi-structured because of columns containing data that could be structured as Arrays of JSONs but were interpreted as unusable Strings from the first loading of the DataFrame.

For example: if I would like to query the id of the first row in credits.csv, we would see the result in the cell (862), as opposed to the crew of the same row we would see: ("[{'credit_id': '52fe4284c3a36847f8024f49', 'department': 'Directing', 'gender': 2, 'id': 7879, 'job': 'Director', 'name': 'John Lasseter', 'profile_path': '/7EdqiNbr4FRjIhKHyPPdFf…"), requiring further cleaning and structuring of the data.

To clean the data and make it more accessible we transformed each column which contained cells of Strings that could be Arrays of JSON into a structure, as a table on its own.

This transformation led us to the point where we had loaded all the CSV files successfully and restructured them to access any data by column name.

Example: If we would like to know the IDs of all the cast in the first row of credits.csv, we would use the following code:

```
cast_row_1 = credits.collect()[0][0]

for elem in cast_row_1:
  print(elem['cast_id'])
```

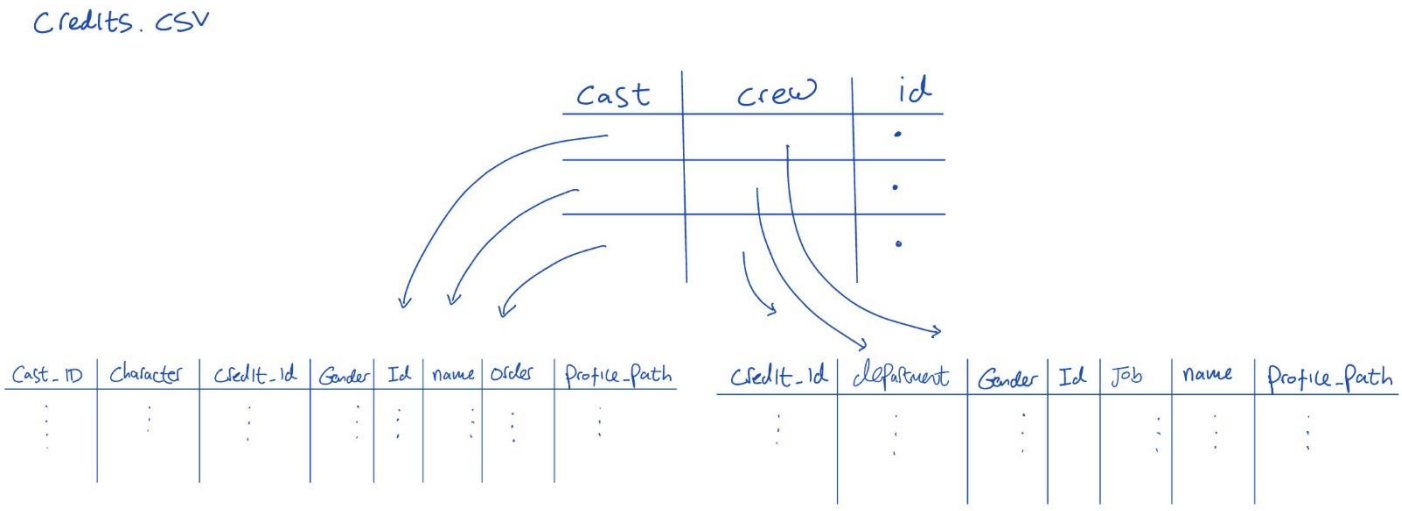
```
14
15
16
17
18
19
20
26
22
23
24
25
27
```

(There was an issue with None types in the JSONs we came across while trying to structure these cells, we chose to solve the issue by using a UDF, which converts the None appearance in the String type to 'None', when loaded to JSON – the new None is read as a String and not None type.)

```
# --- Phase 1: All cells are Strings ---

fix_cols = f.udf(lambda x: x.replace('None', '\'None\''))
```

Hierarchical schema of credits.csv post cleaning and structuring:



**Data cleaning based on 'queries.csv' data**

Considering the output of the query data, we concluded that we should clean the corresponding data to match the user's requirements.

Example: While the genres data has been collected from movies.csv, from column 'genres', there it appears as [{'id': 37, 'name': 'Western'}, {'id': 9648, 'name': 'Mystery'}].

'genres' column output in queries.csv is an Array of genre names, row 1: ['Western', 'Mystery'].

We decided to reorganize the data to answer the query requirements by adding **accessibility** (from JSONs to Array of the data needed), **relevance** (If the 'id' of genres is never required in the user query, is it relevant?) and **memory utilization** (with many cells, containing redundant information, we can save a lot of space to improve the database performance).

Therefore, we transformed the following columns by the output in the queries.csv file:

Movies.csv

```
# --- Phase 3: Extract feilds based on the query data ---

# Extracts the value of 'name' in each json in the cell of column
extract_name = f.udf(lambda x: ",".join([elem['name'] for elem in x]) if x is not None else "-", StringType())


#Extracts year from the date format
extract_year = f.udf(lambda x: int(x[-4:]) if '-' not in x else int(x[:4]), IntegerType())

movies = movies\
      .withColumn("genres", extract_name('genres'))\
      .withColumn("production_companies", extract_name('production_companies'))\
      .withColumn("production_countries", extract_name('production_countries'))\
      .withColumn("spoken_languages", extract_name('spoken_languages'))\
      .withColumn("release_date", extract_year('release_date'))
```

Credits.csv

```
# --- Phase 3: Extract feilds based on the query data ---
extract_name = f.udf(lambda x: ",".join([elem['name'] for elem in x]) if x is not None else "-", StringType())

# Extract the director's 'name' only
extract_director_name = f.udf(lambda x: ",".join([elem['name'] for elem in x if elem['job'] == 'Director'])
        if x is not None else "-", StringType())
```

```
credits = credits\
      .withColumn("cast", extract_name('cast'))\
      .withColumn("crew", extract_director_name('crew'))
```

**Final structuring**

At last, once we interpreted the data, loaded, and cleaned it to our requirements – we needed to convert all the columns containing strings that could be structured as Arrays of Strings. We couldn't load the data at first as an Array of String because CSV cannot contain such a data structure.

The process was needed on the queries.csv, credits.csv and movies.csv.

An example of queries pre and post final structuring:

```
queries.printSchema()

root
 |-- user_id: integer (nullable = true)
 |-- genres: string (nullable = true)
 |-- lang: string (nullable = true)
 |-- actors: string (nullable = true)
 |-- director: string (nullable = true)
 |-- cities: string (nullable = true)
 |-- country: string (nullable = true)
 |-- from_realese_date: string (nullable = true)
 |-- production_company: string (nullable = true)
```

```
queries.printSchema()

root
 |-- user_id: string (nullable = true)
 |-- genres: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- lang: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- actors: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- director: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- cities: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- country: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- from_realese_date: array (nullable = true)
 |    |-- element: integer (containsNull = true)
 |-- production_company: array (nullable = true)
 |    |-- element: string (containsNull = true)
```

**We were oriented towards cleaning, structuring, and reformatting all the data so that in the following sections we would be able to work smoothly without converting Strings to Integers and fully utilizing the Data Frames abilities once all the data is in the correct form.**

(Example: in the above printSchema() image, the from_release_date column could have stayed as a String and we might have not felt the difference. We decided to take the assignments as serious as possible and pay attention to every detail)

***Note regarding further cleaning:***

*We needed to decide how to work with null types and empty '[]' (we will cast to arrays but at this stage they are still strings.). We replaced the NONE which was misinterpreted by the parsing to the StringType 'None'. We replaced '[]' with '['-']' and replaced empty release dates with the year 2030 (year 2030 replacement contained approx. 80 tuples; didn't change the stats we were researching. We chose this because the release date query is defined as the earliest date, we had an empty cell, chose to give it a date that clearly isn't true[hasn't happened yet…] but won't be affected by the queries), this was needed to smoothly research all the insights in the next question using only spark with as many transformations as possible. (We decided to note these minor changes so that If you come across these in the data, you'll know where it came from)*

**Summary:**

Tickets and Users data was a simple loading process, needing only to declare the data types.

Credits and Movies data was a little more of a challenge, we divided each into 4 phases.

- **Phase 1**: All cells were loaded as StringTypes and cleaned by column.
- **Phase 2**: All semi structured columns were converted into structured ArrayType columns.
- **Phase 3**: We extracted required fields based on the query data. For instance: in credits, the only field needed in crew column is the director's name.
- **Phase 4**: Converted all Strings which represented an Array of Strings to an Array of Strings.

Queries data needed only the first and second phases mentioned above.