**YouTube ETL Project**

1. Data Source
2. Data Understanding
3. Tools
4. ETL Process

**Data Source**

The datasource for the Youtube data came from a Kagel repository containing JSON and CSV files.

<https://www.kaggle.com/datasets/datasnaek/youtube-new>

**Data Understanding**

The data contained of multiple JSON formatted files whiche were derived from the categories of each country where as the CSV files derived from the videos created in each country.

The CSV files contained multiple variables including:

* video\_id – the url of the video
* trending\_date – the date the video was trending
* title – the name of the video
* channel\_title – the name of the channel the video was posted by
* category\_id – the id of the category (related to the JSON formatted files)
* publish\_time – the date and time the video was first uploaded
* tags – the tags assigned to the video
* views – the views recorded at the time
* likes – the likes recoreded at the time
* dislikes – the dislikes recoreded at the time
* comments\_count – the number of comments recorded at the time
* thumbnail\_link – the link to the thumbnail
* comments\_disabled – if any comments were disabled
* ratings\_disabled – if any ratings were disabled
* video\_error\_or\_removed – if the video has been removed
* description – the description given with the video

The category JSON formatted file were further investigated after pipelining the data into Python using Pandas library.

The data included related and unrelated category data which were:

* kind
* etag
* id
* snippet.channelId
* snippet.title
* snippet.assignable

**Tools**

* Excel
* Python (pandas, sqlalchemy, datetime) in Annaconda, Spyder IDE
* SQL Server

**ETL Process**

The data as mentioned previously were imported into Python following the ETL process.

Important libraries that were Pandas for data frame manipulation and CSV/JSON format reading and sqlalchemy, required to make a connection into SQL.



Another thing to do was to check to confirm the ODBC driver for SQL so that I could make a connection into SQL server:

A screen shot of a computer program

Description automatically generated

With research on the code, I have implemented the code into a function to easily call it as I will be running the ETL process in muliple instances.

All that was required as arguments were the tableName, which is the table name I wanted in SQL to be called, and the dataframe I wanted inside the table.

Additional arguments for the construction of the connection\_string included the SQL server name and the database name.

A computer screen shot of text

Description automatically generated

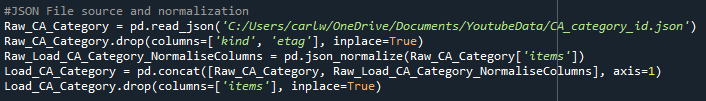
Issues: I had faced an issue where the code couldn’t run when having Trusted\_Connection=yes at the end. After the removal, the function could work as intended.

With a connection from the source to the destination, I could implement transformations to the data from the raw file, into staging tables, and from staging tables into dimension tables.

**Transforming – Load Tables**

The nature of the raw JSON file had 3 columns. The 3rd column however had a dictionary format inside of a dictionary format, like a nested dictionary.

For this, I had to do some normalization so that the columns could be separated into their own individual columns.



Here we can see that after inserting the data into a variable, I have dropped the first to columns as they meant nothing in terms of category. That would then leave with just the column that had to be normalized.

After that, the unwanted columns from the result of normalization could be dropped, which was the items.

Furthermore, creating the load tables were simple with the source and destination set up. However I believe that the loads should have the columns in their most optimal data types.

By doing this, I done a data profile on just the data types of the category data by running .info().

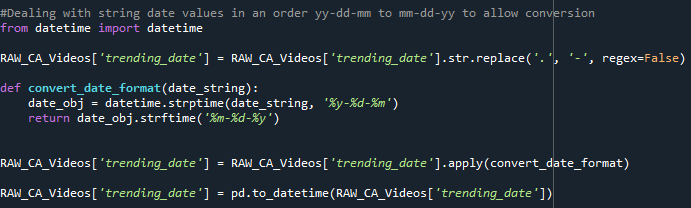
I found that one of the columns, “id” which will have to be related to “category\_id” in the videos file, were not in numeric format. I had to make sure the the column was reflected to the correct data type.



The rest of the data were fine, and could be piplined into the destination, creating a load table in SQL.



As for the Videos file, the data types were fine apart from one column, the trending\_date. This was in string format, and a few changes had to be made to change into date time format.



First datetime had to be imported to implement a function to convert the string format from y-d-m to m-d-y.

Followed by replacing forward slashes in the date column (/) to hyphins (-).

The function could then be called to have the string organised correctly, and then have the to\_datetime function convert the string into datetime.

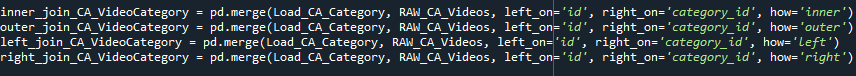


The same process were used for each county (DE, IN, US, GB & FR).

As for KR, RU and JP, there were some issues with these files in regard to the string values. For that I have excluded further analysis on those datasets.

**Transformation - Joins**

I took the opportunity to look over the different joins in Pandas which will come in handing during the Dim and Fact implementation of the data.



It is also reminded that both data types from the joining common columns have to be of the same data type.

**Transformation - Staging Tables**

During the staging table process, my concern was to take a step further with doing some mild data profiling, and this includes to see if there were any null values.

A screen shot of a computer screen

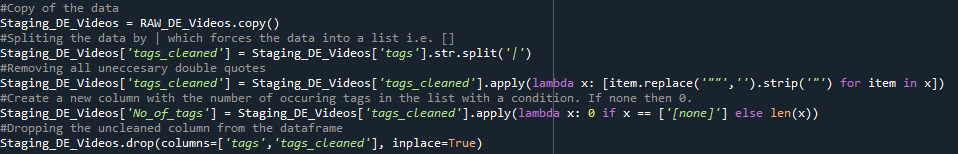
Description automatically generated

With no null values with the dataset, I could then proceed to make a direct copy of the category data, and include that into the staging table.



As for the videos file, there was one particular column, the tags, which I didn’t like the format of, and but I was interested in creating them into a list. Luckily by removing a character from the data, it would automatically store the data into a list for each row.

Eg. xyz | ‘”abc”’ | ‘”123’ -> [xyz, abc, 123]



1. Make a copy of the raw data into a Staging\_ variable.
2. Split the data by using str.split(‘|’) so that the data would split whenever | is used.
3. Use a lambda function to replace unecessary quotation marks i.e. (lambda x: [item.replace(‘“”’, ‘’).strip(‘”’) for item in x]
4. Count the number of occurences in a list, and if mentioned [‘none’], make it 0 in a new column.
5. Drop unwanted columns such as tags and tags\_cleaned.

The tags was not neccesary itself however if they were, one hot encoding could split the data into numerous columns, allthough not neccesary for the ETL set up.

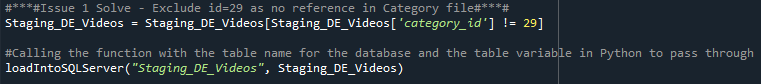
Issue: The data were tested in SQL server to join Video and Category, however there was one id that couldn’t make the relationship:

A white background with words

Description automatically generated

This only made more sense when I understood that the category\_id had to exists for results to reflect for the 29th id.

For that, all rows that had the category\_id = 29 were excluded from the staging table.



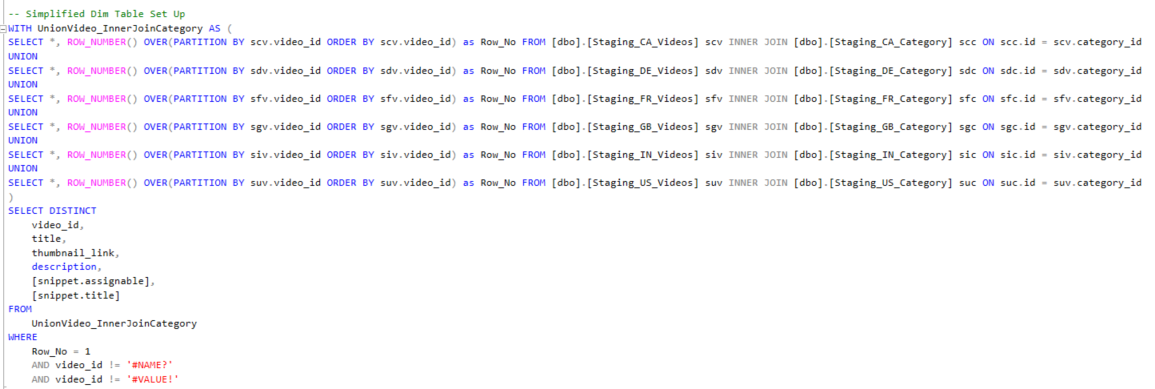
This was replicated for the rest of the different countries.

**Dimenstion Table Setup**

At first, I wanted to have all dimension tables set up individually from the staging table. Even though that seemed okay for the the scope of an ETL, I wanted to achieve an ETL that had the end result of a StarSchema model. This means any similarities I could put together such as video files having the same columns, categories having the same columns, could be unioned together vertically.

Below are a few query representations of how I wanted to implement the query.

These queries involve the use of CTE’s (Common Table Expressions), windows functions, and union to join the tables vertically.



A screenshot of a computer program

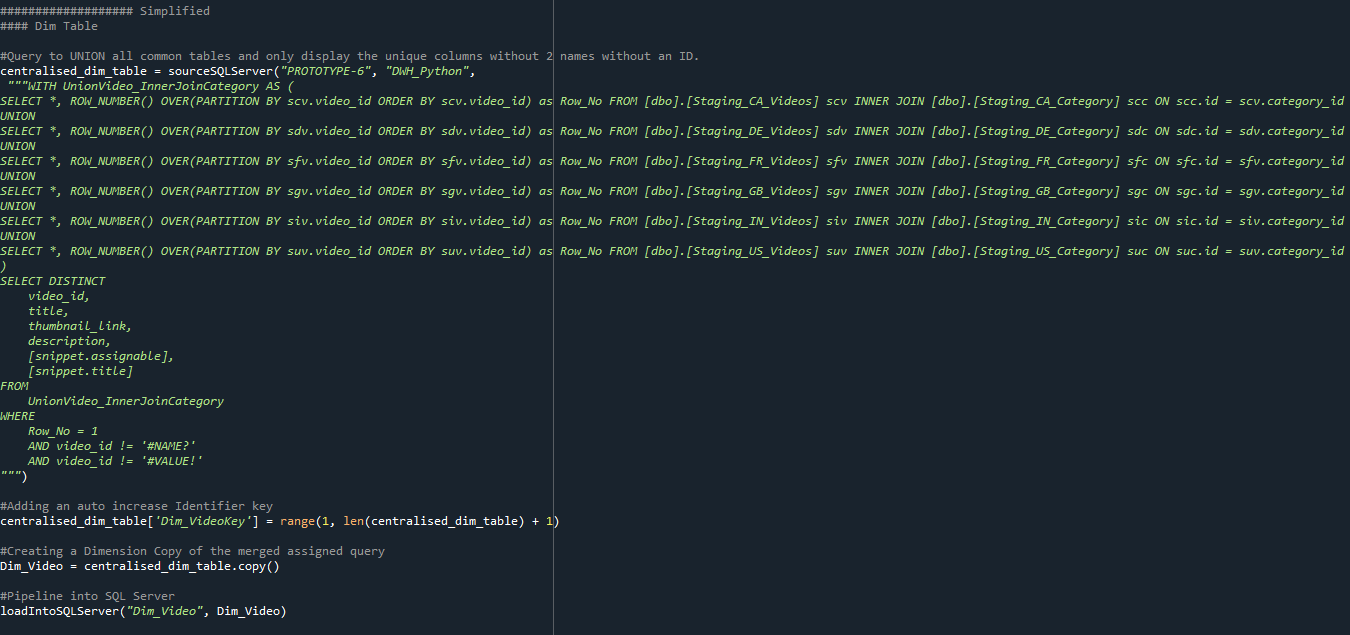
Description automatically generated

Back in Python, I had to set up a way of retrieving the data from SQL (as I only have set up as a destination) from a query.

A screen shot of a computer

Description automatically generated

**Creating a centralised Dim Table**



With the set up of the sourceSQLServer function, I could insert the query implemented to fetch the required data.

After the data has been inserted into a dataframe, I could ad a unique serrogate key, Dim\_VideoKey, with an autoincrement using the range function. i.e. range(1,len(centralised\_dim\_table) +1)

The dataframe could be stored in its proper variable name and then loaded into SQL with the original loadIntoSQLServer function.

**Creating a centralised Fact Table based on the centralised Dim Table**

A screen shot of a computer program

Description automatically generated

The same thing was done for the fact table to query the relevant facts from the videos data.

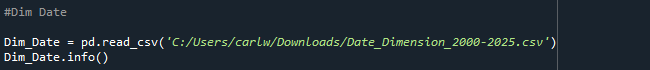
Once the data queried was stored into a dataframe, a merge function was used to merge the Dim\_Video table, using inner join, with the common columns (video\_id’s).

Once the data had been merged, an auto increment column was created for the Fact table, and the unessary columns not required were dropped, as the main purpose for the merge was to get the Dim\_VideoKey, to make the relationship in the star schema model.

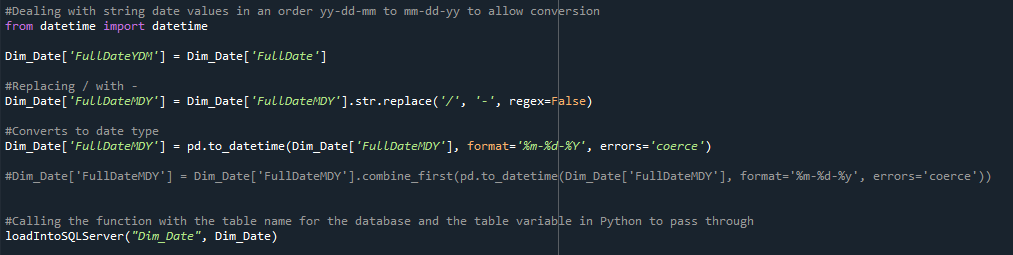
Furthermore, the data was copied into a variable that better suited the fact table, and then loaded into SQL server.

**Dim Date Table**

It is important to have a dimenstion date as the dimension date has all the hirachial values when it comes to reporting. For this I had to add a dim date by importing a ready file from CSV format.



A check on data types was the next step to ensure the connection with the fact table was suitable.



The transformation was changing a full date column into date data type from string so that the connection can be made. The data was then loaded into SQL server.

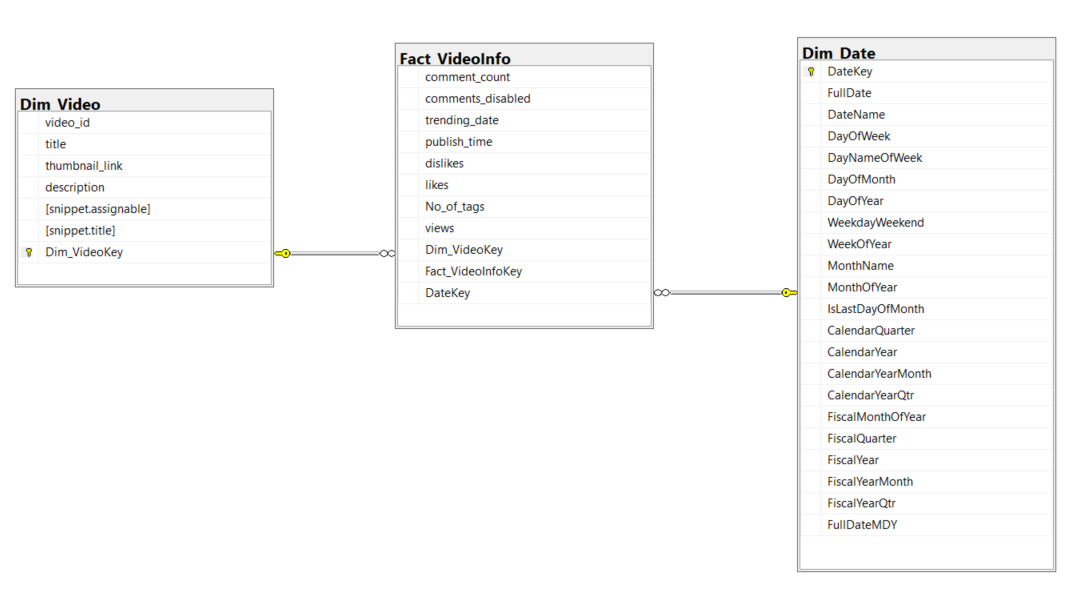
**Merging Dim Date with Fact\_VideoInfo**

Merging the two tables was the next solution so that the Dim\_DateKey could be available in the fact table. The common columns were the trending date and the FullDateMDY using an inner join.

The rest of the columns were dropped resulting in a copy of the data to a dataframe name of relvance and then loaded into SQL Server.

**SQL Server**

In SQL Server, the data could be connected creating a star schema, by making the primary keys of each dim table and the fact table, and then connecting them to the fact table’s foreign keys, which were implemented during the merges in Python.



**Limitations**

The limitations faced is that whenever data is loaded from the dimension tables, is that like SSIS, if the data is reference by a foreign key, then data will not be able to be loaded.

For this the relationship will have to be disconnected, or the fact table will have to be truncated before reloading data into SQL Server from Python.

Another limitation is that, when data is loaded from a fact table into SQL Server, the relationship breaks and the primary key gets removed. This means that the relationship will have to be reassigned and primary key be re-made.

**Further thoughts**

With the possibility of creating an ETL with Pandas and supporting libraries to SQL, I can look forward to taking a step further and looking into Pyspark which is more data engineering friendly than Pandas.