In [1]:

```
from collections import Counter
import mysql.connector
from mysql.connector import Error
import csv
import re
from neo4j import GraphDatabase
import pandas as pd
import numpy as np
from pandas.errors import ParserError
```

In [2]:

```
class Neo4jDB:

    def __init__(self, uri, user, pwd=''):
        self.__uri = uri
        self.__user = user
        self.__pwd = pwd
        self.__driver = None
        try:
            self.__driver = GraphDatabase.driver(self.__uri, auth=(self.__user, self.__pwd
        except Exception as e:
            print("Failed to create the driver:", e)

    def close(self):
        if self.__driver is not None:
            self.__driver.close()

    def query(self, query, parameters=None, db=None):
        assert self.__driver is not None, "Driver not initialized!"
        session = None
        response = None
        try:
            session = self.__driver.session(database=db) if db is not None else self.__dri
            response = list(session.run(query, parameters))
        except Exception as e:
            print("Query failed:", e)
        finally:
            if session is not None:
                session.close()
        return response

neoConn = Neo4jDB(uri="bolt://localhost:7687", user="")
```

In [3]:

```
host = 'localhost'
schema = 'sakila'
user = 'root'
password = 'sembiran2009'
```

In [4]:

```
try:
    connection = mysql.connector.connect(host=host, database=schema, user=user, password=p

    if connection.is_connected():
        db_Info = connection.get_server_info()
        print("Connected to MySQL Server version ", db_Info)
        cursor = connection.cursor()
        cursor.execute("select database();")
        record = cursor.fetchone()
        print("You're connected to database: ", record)

except Error as e:
    print("Error while connecting to MySQL", e)
```

```
Connected to MySQL Server version  8.0.29
You're connected to database:  ('sakila',)
```

In [5]:
```
def _clear_all(commit):
    if commit:
        query='''MATCH (n)
        DETACH DELETE n
        RETURN count(*) AS total'''
        return neoConn.query(query)
```

In [6]:
```
def get_dict(tables, keys):
    ref = {}
    for i in range(len(tables)):
        table = tables[i]
        if ref.get(table):
            ref[table] += ','+str(keys[i])
        else:
            ref[table] = str(keys[i])

    return ref
```

In [7]:
```
def populate_obj(name, df, excludes):
    header = list(df.columns)

    obj = '('+name.lower()+':'+name.capitalize()+'{'
    first = True
    for h in header:
        if h not in excludes:
            if first:
                first = False
            else:
                obj +=','
            obj += h+': row.'+h

    obj +='})'
    return obj
```

In [8]:
```
def _read_csv(table, s, e):
    file = 'db.localhost/{}/{}.csv'.format(schema, table)
    #print("file: {}".format(file))
    return pd.read_csv (file, sep = s, encoding = e)
```

In [9]:
```
def generate_single_graph(m, df_logs, commit):
    st = time.time()
    table = m.get('table')
    print("\n\n===> Processing records for Table '{}'".format(table))
    if not commit:
        print(m)

    records = _read_csv(table, ';', 'utf-8')

    query = '''UNWIND $rows AS row \nCREATE ''' + populate_obj(table.lower(), records, [])

    df_logs = _add_row(df_logs, table, ref_tables='', no_records=len(records), no_rels=0,

    if not commit:
        print(query)
    else:
```

```
        neoConn.query(query, parameters = {'rows': records.to_dict('records')})

    return df_logs
```

In [10]:

```
def generate_joined_graph(m, s, df_logs, commit):
    table = m.get('table')
    print("\n\n===> Processing records for Table '{}'".format(table))
    if not commit:
        print(m)

    records = _read_csv(table, ';', 'utf-8')

    pri_key = m.get('pri_key').split(',')
    for_keys = m.get('for_keys').split(',')
    # print(for_keys)
    ref_tables = m.get('ref_tables').split(',')
    # print(ref_tables)
    ref_keys = m.get('ref_keys').split(',')
    # print(ref_keys)
    ref = get_dict(ref_tables, ref_keys)
    # print(ref)

    #if sorted(pri_key) == sorted(for_keys):
    #    query = '''UNWIND $rows AS row\nCREATE ''' + populate_obj(table.lower(), records,
    #else:
    #    query = '''UNWIND $rows AS row\nCREATE ''' + populate_obj(table.lower(), records,
    #include ids
    query = '''UNWIND $rows AS row\nCREATE ''' + populate_obj(table.lower(), records, [])

    start = 0
    end = 0
    no_edges = 0
    for ref_table, ids in ref.items():

        ref_ids = ids.split(',')
        # print(ref_table)
        # print(ref_ids)

        end = start+len(ref_ids)
        fk = for_keys[start:end]

        #skip, if it is a cyclic table
        if s and ref_table in s:
            print("********** Table '{}' is a cyclic referenced by Table '{}'..skipped it
            start = end
            continue

        print("Processing reference Table '{}' ...".format(ref_table))
        query += '''\nWITH distinct row, ''' + table.lower()

        no_edges += 1

        prev_fk = None
        duplicate = False
        uwind = {}
        for i in range(len(fk)):
            if (prev_fk == ref_ids[i]):
                uwind[i] = '''\nUNWIND row.''' + fk[i] + ''' AS _''' + fk[i]
            else:
                query += '''\nUNWIND row.''' + fk[i] + ''' AS _''' + fk[i]
            prev_fk = ref_ids[i]

        if len(uwind) > 0:
            for i in range(len(fk)):
```

```
                if i>0:
                    query += '''\nWITH distinct row, ''' + table.lower()
                    query += uwind.get(i)

                query += '''\nMATCH (''' + ref_table.lower() + ''':'''+ ref_table.capitali
                query += ref_ids[i] + ''': _'''+fk[i]
                query += '''})\nMERGE ('''+table.lower()+''')-[:'''+ table.upper() + '''_'
            else:
                query += '''\nMATCH (''' + ref_table.lower() + ''':'''+ ref_table.capitalize()
                first = True
                for i in range(len(fk)):
                    if not first:
                        query += ''','''
                    else:
                        first = False
                    query += ref_ids[i] + ''': _'''+fk[i]
                query += '''})\nMERGE ('''+table.lower()+''')-[:'''+ table.upper() + '''_''' +

            start = end

        query += '''\nRETURN count(*) AS total'''

        df_logs = _add_row(df_logs, table, ref_tables=ref_tables, no_records=len(records), no_

        if not commit:
            print(query)
        else:
            neoConn.query(query, parameters = {'rows': records.to_dict('records')})

        return df_logs
```

In [11]:

```
def generate_edges(m, r, df_logs, commit):
    table = m.get('table')
    print("\n\n===> Creating additional Edges between Table '{}' and Table(s) '{}' ".forma
    if not commit:
        print(m)
        print(r)

    records = _read_csv(table, ';', 'utf-8')

    pri_key = m.get('pri_key').split(',')
    for_keys = m.get('for_keys').split(',')
    # print(for_keys)
    ref_tables = m.get('ref_tables').split(',')
    # print(ref_tables)
    ref_keys = m.get('ref_keys').split(',')
    # print(ref_keys)
    ref = get_dict(ref_tables, ref_keys)
    # print(ref)

    query = '''UNWIND $rows AS row'''
    for prim in pri_key:
        query += '''\nUNWIND row.''' + prim + ''' AS _''' + prim

    query += '''\nMATCH (''' + table.lower() + ''':'''+ table.capitalize() +''' {'''
    first = True
    for i in range(len(pri_key)):
        if not first:
            query += ''','''
        else:
            first = False
        query += pri_key[i] + ''': _'''+pri_key[i]
    query += '''})'''
```

```
    start = 0
    end = 0
    no_edges = 0
    for ref_table, ids in ref.items():

        ref_ids = ids.split(',')
        #print(ref_table)
        #print(ref_ids)

        end = start+len(ref_ids)
        fk = for_keys[start:end]

        if ref_table not in r:
            print('********** Skipping table {} **********'.format(ref_table))
            start = end
            continue

        print("Creating an Edge from Table '{}' --> Table '{}' ...".format(table, ref_tabl
        query += '''\nWITH distinct row, ''' + table.lower()

        no_edges += 1

        prev_fk = None
        duplicate = False
        uwind = {}
        for i in range(len(fk)):
            if (prev_fk == ref_ids[i]):
                uwind[i] = '''\nUNWIND row.''' + fk[i] + ''' AS _''' + fk[i]
            else:
                query += '''\nUNWIND row.''' + fk[i] + ''' AS _''' + fk[i]
            prev_fk = ref_ids[i]

        if len(uwind) > 0:
            for i in range(len(fk)):
                if i>0:
                    query += '''\nWITH distinct row, ''' + table.lower()
                    query += uwind.get(i)

                query += '''\nMATCH (''' + ref_table.lower() + ''':'''+ ref_table.capitali
                query += ref_ids[i] + ''': _'''+fk[i]
                query += '''})\nMERGE ('''+table.lower()+''')-[:'''+ table.upper() + '''_'
        else:
            query += '''\nMATCH (''' + ref_table.lower() + ''':'''+ ref_table.capitalize()
            first = True
            for i in range(len(fk)):
                if not first:
                    query += ''','''
                else:
                    first = False
                query += ref_ids[i] + ''': _'''+fk[i]
            query += '''})\nMERGE ('''+table.lower()+''')-[:'''+ table.upper() + '''_''' +

        start = end

    query += '''\nRETURN count(*) AS total'''

    df_logs = _add_row(df_logs, table, ref_tables=ref_tables, no_records=len(records), no_

    if not commit:
        print(query)
    else:
        neoConn.query(query, parameters = {'rows': records.to_dict('records')})

    return df_logs
```

In [12]:
```
def get_ddl(c1, table):
    q2 = "SHOW CREATE TABLE %s;" % table
    c1.execute(q2)
    result = c1.fetchone()
    return list(result.values())[1]
```

In [13]:
```
def get_metadata(c1, table):
    ddl = get_ddl(c1, table)

    prog = re.compile('CREATE TABLE.*?`(.*).*?`\s\(', re.IGNORECASE)
    table = prog.findall(ddl)[0]

    prog = re.compile('PRIMARY KEY\s\S(.*)(?=\))', re.IGNORECASE)
    pri_key = prog.findall(ddl)[0].replace('`', '')

    prog = re.compile('FOREIGN KEY.*?\(`(.*).*?`\)\sREFERENCES', re.IGNORECASE)
    for_keys = prog.findall(ddl)
    for_keys = ",".join(for_keys)

    prog = re.compile('REFERENCES.*?`(.*).*?`\s', re.IGNORECASE)
    ref_tables = prog.findall(ddl)
    ref_tables = ",".join(ref_tables)

    prog = re.compile('REFERENCES.*?\(`(.*).*?`\)', re.IGNORECASE)
    ref_keys = prog.findall(ddl)
    ref_keys = ",".join(ref_keys)

    return {'table':table, 'pri_key':pri_key, 'for_keys': for_keys, 'ref_tables':ref_table
```

In [14]:
```
def _compute_out(data):
    return len(data.split(','))

def _compute_in(key, unsorted_data):
    counter = 0
    for k, v in unsorted_data.items():
        if k != key and key in v:
            dependents = v.split(',')
            for d in dependents:
                if d.strip() == key:
                    counter += 1
    return counter

def _score_data(unsorted_data):
    scores = {}
    for key, data in unsorted_data.items():
        out_elms = 0
        in_elms = 0
        if data.strip() == '':
            scores[key] = 0
        else:
            #compute: score = no.outgoung / no.incoming
            out_elms = _compute_out(data)
            in_elms = _compute_in(key, unsorted_data)
            scores[key] = np.inf if in_elms == 0 else (out_elms/in_elms)
        #print ("Key [{}], Out[{}], In[{}], Score[{}]".format(key, out_elms, in_elms, sco

    scores = dict(sorted(scores.items(), key=lambda item: item[1]))
    #print(scores)
    return scores

def _remove_best(unsorted_data, key, sorted_data, relations):
    for k, v in unsorted_data.items():
        if key in v:
```

```
            deps = v.split(',')
            dependents = deps.copy()
            for d in deps:
                if d.strip() == key:
                    dependents.remove(d)

                    #add the relation between this entry's key and the 'key', if record ex
                    rels = relations.get(k)
                    if rels:
                        relations[k] = rels + ',' + key
                    else:
                        relations[k] = key

            #dependents = list(filter(lambda d: d.strip() != key, dependents))
            unsorted_data[k] = ','.join(dependents)

    unsorted_data.pop(key, None)

def _remove_cyclic(unsorted_data, key, sorted_data, relations, cyclics):
    #1) add the 'key' as one of the cyclic dependencies
    cyclics[key] = unsorted_data.get(key)
    #2) remove from those depend on the 'key', record their relations to it, and finally r
    _remove_best(unsorted_data, key, sorted_data, relations)
    #3) finally, add the 'key' as one of the sorted keys
    sorted_data.append(key)

def _sort(unsorted_data, sorted_data, relations, cyclics):
    scores = _score_data(unsorted_data)

    best_key = list(scores.keys())[0]
    best_val = list(scores.values())[0]
    if best_val == 0:
        #remove best[0] from unsorted_data
        _remove_best(unsorted_data, best_key, sorted_data, relations)
        sorted_data.append(best_key)
    else:
        print('CYCLIC detected!')
        _remove_cyclic(unsorted_data, best_key, sorted_data, relations, cyclics)

    return unsorted_data
```

In [15]:

```
df_logs = pd.DataFrame(columns=['table', 'ref_tables', 'no_records', 'no_rels', 'no_nodes'

def _add_row(df_logs, table, **kwargs):
    idx = df_logs.index[df_logs['table']==table]
    if len(idx):
        for key, value in kwargs.items():
            if isinstance(value, list):
                value = ','.join(value)
            if key not in (['no_records', 'ref_tables']) and df_logs.iloc[idx][key].any():
                if isinstance(value, int):
                    df_logs.at[idx, key] = df_logs.iloc[idx][key] + value
                else:
                    df_logs.at[idx, key] = df_logs.iloc[idx][key] + ',' + value
            else:
                df_logs.at[idx, key] = value
    else:
        df_logs = df_logs.append({'table':table}, ignore_index=True)

    return df_logs
```

In [16]:

```
def _populate_graph(metas, sorted_data, relations, cyclics, df_logs, commit):
    for data in sorted_data:
```

```
        meta = metas[data]
        table = meta['table']
        df_logs = _add_row(df_logs, table)
        rels = relations.get(table)
        skip = cyclics.get(table)
        if rels:
            #joined
            df_logs = generate_joined_graph(meta, skip, df_logs, commit)
        else:
            #single
            df_logs = generate_single_graph(meta, df_logs, commit)

    #create relation for cyclic references
    for table, refs in cyclics.items():
        meta = metas[table]
        df_logs = generate_edges(meta, refs, df_logs, commit)

    return df_logs
```

In [17]:

```
def _print(tables, metas, sorted_data, relations, cyclics):

    df_print1 = pd.DataFrame(columns=['table', 'ref_tables'])
    sep = '---------------'
    for table in tables:
        m = metas[table]
        ref_tables = m.get('ref_tables')
        rels = ref_tables.split(',')
        if ref_tables == '':
            rels = None
        df_print1 = df_print1.append({'table':table, 'ref_tables':rels}, ignore_index=True
    df_print1 = df_print1.append({'table':sep, 'ref_tables':sep}, ignore_index=True)

    df_print2 = pd.DataFrame(columns=['table', 'ref_tables'])

    for data in sorted_data:
        meta = metas[data]
        table = meta['table']
        rs = relations.get(table)
        rels = None
        if rs:
            rels = rs.split(',')
        cs = cyclics.get(table)
        skip = None
        if cs:
            skip = cs.split(',')
        if skip and rels:
            rels = [i for i in rels if i not in skip]
        df_print2 = df_print2.append({'table':table, 'ref_tables':rels}, ignore_index=True
    df_print2 = df_print2.append({'table':sep, 'ref_tables':sep}, ignore_index=True)

    df_print3 = pd.DataFrame(columns=['table', 'ref_tables'])
    for table, refs in cyclics.items():
        df_print3 = df_print3.append({'table':table, 'ref_tables':refs}, ignore_index=True
    df_print3 = df_print3.append({'table':sep, 'ref_tables':sep}, ignore_index=True)

    df_print = df_print1.append(df_print2).append(df_print3)
    display(df_print)
```

In [18]:

```
def _generate_graph(df_logs, commit):
    q1 = ("SHOW TABLES FROM " + schema)
    c1 = connection.cursor(dictionary=True, buffered=True)
    c1.execute(q1)
```

```
    table_list = c1.fetchall()
    tables = []
    metas = {}
    unsorted_data = {}
    for entry in table_list:
        _, table = entry.popitem()
        tables.append(table)
        meta = get_metadata(c1, table)
        metas[table]=meta
        unsorted_data[meta.get('table')] = meta.get('ref_tables')

    sorted_data = []
    relations = {}
    cyclics = {}
    half_sort = dict(sorted(unsorted_data.items(), key=lambda item: item[1]))
    while True:
        half_sort = _sort(half_sort, sorted_data, relations, cyclics)
        if (len(half_sort) == 0):
            break

    _print(tables, metas, sorted_data, relations, cyclics)

    return _populate_graph(metas, sorted_data, relations, cyclics, df_logs, commit)
```

In [19]:

```
import time
commit = True

start_time = time.time()

_clear_all(commit)

elapsed_time = time.time() - start_time
print('Execution time:', time.strftime("%H:%M:%S", time.gmtime(elapsed_time)))

start_time = time.time()

_sorted = []
df_logs = _generate_graph(df_logs, commit)

elapsed_time = time.time() - start_time
print('Execution time:', time.strftime("%H:%M:%S", time.gmtime(elapsed_time)))
df_logs
```

```
Execution time: 00:00:14
CYCLIC detected!
```

	table	ref_tables
0	actor	None
1	address	[city]
2	category	None
3	city	[country]
4	country	None
5	customer	[address, store]
6	film	[language, language]
7	film_actor	[actor, film]
8	film_category	[category, film]
9	film_text	None

	table	ref_tables
10	inventory	[film, store]
11	language	None
12	payment	[customer, rental, staff]
13	rental	[customer, inventory, staff]
14	staff	[address, store]
15	store	[address, staff]
16	---	---
0	actor	None
1	category	None
2	country	None
3	film_text	None
4	language	None
5	city	[country]
6	address	[city]
7	film	[language, language]
8	film_actor	[actor, film]
9	film_category	[category, film]
10	store	[address]
11	customer	[address, store]
12	staff	[address, store]
13	inventory	[film, store]
14	rental	[customer, staff, inventory]
15	payment	[customer, staff, rental]
16	---	---
0	store	staff
1	---	---

```
===> Processing records for Table 'actor'

===> Processing records for Table 'category'

===> Processing records for Table 'country'

===> Processing records for Table 'film_text'

===> Processing records for Table 'language'

===> Processing records for Table 'city'
Processing reference Table 'country' ...
```

```
===> Processing records for Table 'address'
Processing reference Table 'city' ...


===> Processing records for Table 'film'
Processing reference Table 'language' ...


===> Processing records for Table 'film_actor'
Processing reference Table 'actor' ...
Processing reference Table 'film' ...


===> Processing records for Table 'film_category'
Processing reference Table 'category' ...
Processing reference Table 'film' ...


===> Processing records for Table 'store'
Processing reference Table 'address' ...
********** Table 'staff' is a cyclic referenced by Table 'store'..skipped it *********


===> Processing records for Table 'customer'
Processing reference Table 'address' ...
Processing reference Table 'store' ...


===> Processing records for Table 'staff'
Processing reference Table 'address' ...
Processing reference Table 'store' ...


===> Processing records for Table 'inventory'
Processing reference Table 'film' ...
Processing reference Table 'store' ...


===> Processing records for Table 'rental'
Processing reference Table 'customer' ...
Processing reference Table 'inventory' ...
Processing reference Table 'staff' ...


===> Processing records for Table 'payment'
Processing reference Table 'customer' ...
Processing reference Table 'rental' ...
Processing reference Table 'staff' ...


===> Creating additional Edges between Table 'store' and Table(s) 'staff'
********** Skipping table address **********
Creating an Edge from Table 'store' --> Table 'staff' ...
Execution time: 00:10:09
```

Out[19]:

	table	ref_tables	no_records	no_rels	no_nodes	no_edges
0	actor		200	0	1	NaN
1	category		16	0	1	NaN
2	country		109	0	1	NaN
3	film_text		1000	0	1	NaN

	table	ref_tables	no_records	no_rels	no_nodes	no_edges
4	language		6	0	1	NaN
5	city	country	600	1	1	1
6	address	city	603	1	1	1
7	film	language,language	1000	2	1	1
8	film_actor	actor,film	5462	2	1	2
9	film_category	category,film	1000	2	1	2
10	store	address,staff	2	4	1	2
11	customer	address,store	599	2	1	2
12	staff	address,store	2	2	1	2
13	inventory	film,store	4581	2	1	2
14	rental	customer,inventory,staff	16044	3	1	3
15	payment	customer,rental,staff	16049	3	1	3