Janel Chumley

Eid: jr43872

CS 327e

Final Lab Report

6th, November 2016

In order to understand the stages of learning related to fundamental relational database theory and hands-on applications, we will examine four consecutive and related laboratory exercises. The first set of lab exercises explores *Data Modeling and Schema Design*: the identification of datasets via web through third-party sources, the design of conceptual and logical entity-relationship models, and the creation of a physical database model using the SQL query language. The second set of exercises involves the hands-on application of *Data Loading* using the PyMySQL database interface. In particular, a python database interface will be used to perform *insert, delete*, and *alter* statements on the chosen schema.

The third set of lab exercises requires the development of a command-line user-interface using Python and MySQL, and the creation of SQL queries of varying degrees of complexity. The last set of exercises discussed in this paper is related to *Data Collection* using a RESTful API client. In particular, the Twitter RESTful API will be discussed.

It should be noted that the lab exercises discussed in this paper are designed for two participants, but are given through the experiences of one student. This is a result of interpersonal conflict. Communication issues related to differences in age and gender may have been contributing factors. However, the topic of this paper is about the learning processes involved in an undergraduate-level course on relational databases, not psychology. Hence,

interpersonal topics will not be discussed in further detail, but should be acknowledged as one of the challenges encountered in this course.

Data Modeling and Schema Design. In order to build a database, one is required to have data. Thus, the first step in this set of lab exercises requires locating one or more publically available datasets using online resources. Naturally, the Google search engine was used to locate viable sources.

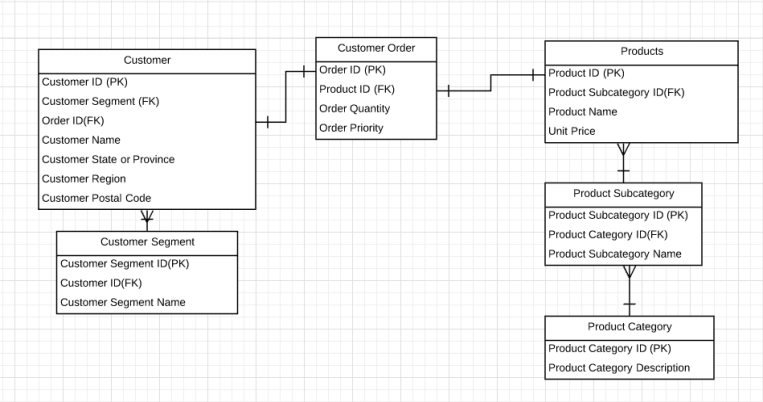
The first idea was to build a database schema using biological models. Thus, the Center for Disease Control (CDC) National Survey on Health and Nutrition was considered. However, after downloading and examining datasets from this source, it was soon realized that it was not ideal for this exercise. Identifying primary and foreign keys was a counter-intuitive process. Furthermore, the dataset was already decomposed into multiple tables. Several attempts at creating a conceptual diagram were made without success, so the dataset was discarded.

The second idea was to use geographic, spatial data to build a database on the Texas State Park system. It seemed like a viable option, mainly because of the many government sources that provide geographic data. However, it was soon realized that the geodatabase data type – shapefiles is a special datatype, thus was not suitable for the exercise. Thus, the datasets were not in one of the required file formats (xml, csv, JSON). Hence, the idea of building a geographic, spatial database was also discarded.

The third and final idea was to find a dataset suitable for E-commerce real-world web applications. This led to the discovery of the *Superstore Orders* dataset, an instructional tool used by data visualization software company Tableau. Since the dataset is designed for training purposes, it seemed to also be a viable dataset for this learning exercise.

After choosing an appropriate dataset, conceptual and logical blueprints are required to create a physical database model. This process involves normalization through multiple steps. The first objective is to get the data into tabular form. In order to accomplish this task, a data dictionary is created to identify entity classes and their associated attributes.

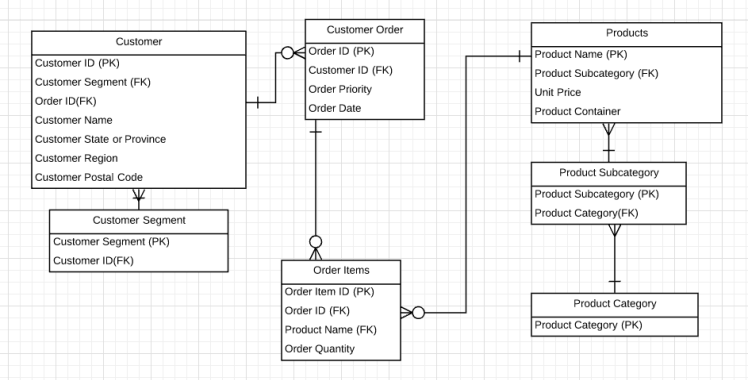
The next step in the process of normalization is the creation of conceptual and logical diagrams using LucidChart. To create the conceptual and logical schemas, it was necessary to determine primary and foreign key relationships using the data dictionary. Furthermore, subtypes and supertypes needed to be determined between the chosen entity classes. The following is the first attempt at designing a conceptual model, created during lab 1:



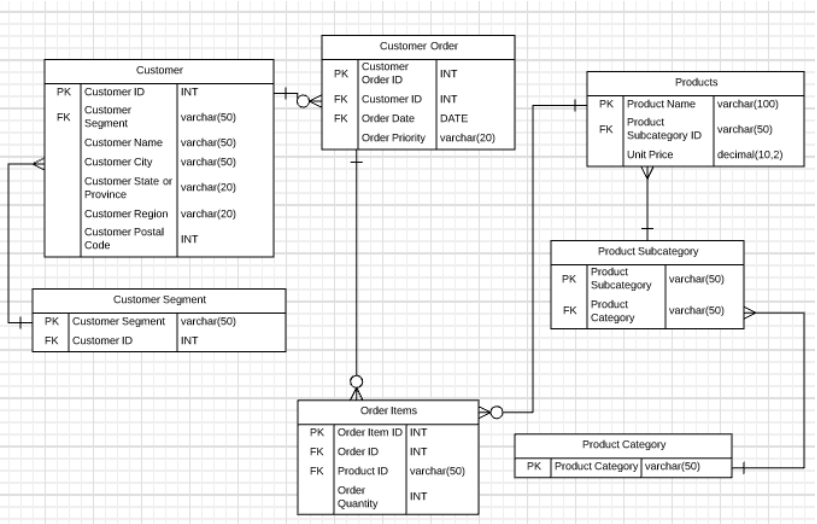
While the conceptual model provided above reflects an understanding of hierarchies and levels between entity-classes, it is erroneous with respect to cardinality. In particular, the Customer to Customer Order relationship should be one-to-many, not one-to-one. The Customer Order to Products table should be one-to-many as well. In other words, this schema required a junction table.

Without a junction table, the schema would be vulnerable to data redundancy, causing problems in the implementation of the physical model. Hence, it was necessary to create a new entity-class to serve as a junction table. It should be noted that this realization didn't occur until the second lab assignment, when issues arose with the insertion, deletion, and altering of data in the physical

model. The following is the conceptual schema that was created during the second laboratory exercises, which includes a junction table :



Notice that the Order Items table serves as a junction table between Customer Orders and Products. Furthermore, the relationship between Customer and Customer Orders was changed from one-to-one to one-to-many. After determining that the conceptual schema was in first normal form, a logical model could be derived. The following is the logical model with respect to the revisions performed during the second lab assignment:

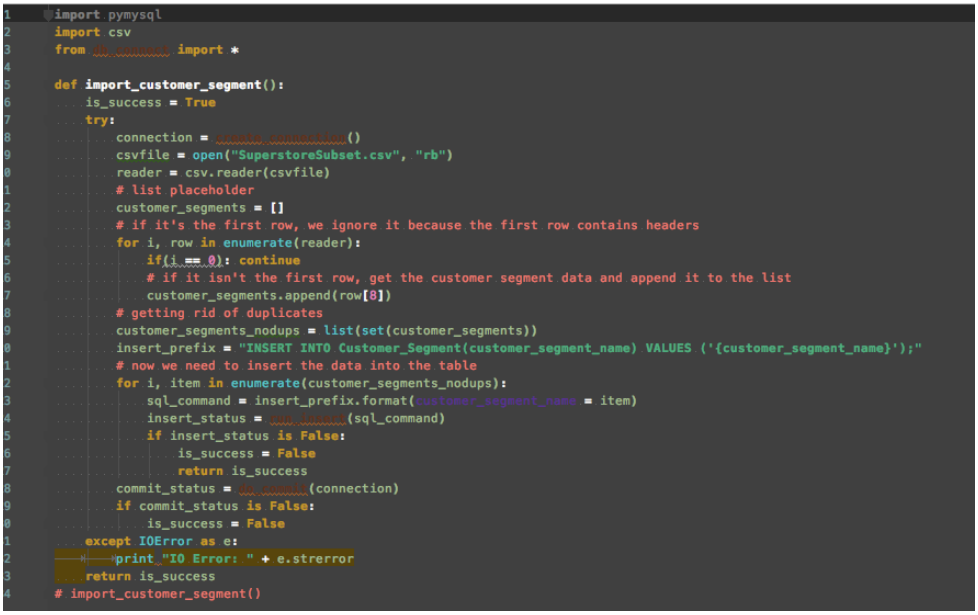


As can be seen in the diagram above, the translation between the conceptual and logical models was a lateral move. The only additional step was to include an additional column defining the

data types of each of the attributes when deriving the logical model. After deriving the logical model, it was suitable to build a physical database model. The process required knowledge of basic SQL commands as well as an understanding of primary and foreign key relationships.

After designing the physical model, it was time to import the data into the database. In order to execute this process, the Python MySQL database interface was used to populate the tables. First, it was necessary to install and set-up the PyMySQL library through the Bash terminal window. This step also required the creation of a username and password in order to connect to the MySQL database through the Python script. After ensuring that the installation and set-up was properly executed, the data could then be added to the database. The PyMySQL library was used to connect to the MySQL database and to perform SQL commands.

The next step was importing the data. The screenshot shows one of the seven import scripts that were written during this exercise:



```
import pymysql
import csv
from db_connect import *

def import_customer_segment():
    is_success = True
    try:
        connection = create_connection()
        csvfile = open("SuperstoreSubset.csv", "rb")
        reader = csv.reader(csvfile)
        # list placeholder
        customer_segments = []
        # if it's the first row, we ignore it because the first row contains headers
        for i, row in enumerate(reader):
            if(i == 0): continue
            # if it isn't the first row, get the customer segment data and append it to the list
            customer_segments.append(row[8])
        # getting rid of duplicates
        customer_segments_nodups = list(set(customer_segments))
        insert_prefix = "INSERT INTO Customer_Segment(customer_segment_name) VALUES ('{customer_segment_name}');"
        # now we need to insert the data into the table
        for i, item in enumerate(customer_segments_nodups):
            sql_command = insert_prefix.format(customer_segment_name = item)
            insert_status = run_insert(sql_command)
            if insert_status is False:
                is_success = False
                return is_success
        commit_status = do_commit(connection)
        if commit_status is False:
            is_success = False
    except IOError as e:
        print "IO Error: " + e.strerror
    return is_success
# import_customer_segment()
```

Overall, the writing and debugging of the import scripts was a time-intensive process. For one, after each script was ran, it was necessary to log into the MySQL database via the Bash terminal

window to verify that the insertions were executed properly. The process was tedious, but also provided extensive hands-on application, which contributed to the learning process.

The next step was checking the run-time of each of the seven import scripts using the Sublime text-editor. Following the assignment specification, text files were created for each of the respective import scripts noting the run-time, number of insertions, and any error messages. It soon became apparent that duplicate entries were resulting in longer run-times for two of the tables. In particular, the import scripts for the Customer and Customer Order tables resulted in several duplicate entry messages.

The subsequent step involved writing rollback scripts in Python. Similar to the import scripts described on page 5, a script was required for each of the seven tables in the database. These scripts used the *delete* and *alter* SQL commands, which were called through the db_connect.py script. This step wasn't as involved as the scripts significantly shorter, as can be seen in the following example:

```
import pymysql
from [redacted] import *
def rollback():
    try:
        delete_stmt = "DELETE FROM Customer"
        [redacted](delete_stmt)
    except IOError as e:
        print "IO Error: " + e.strerror
# def destroy_connection(conn):
#     conn.close()
rollback()
```

The next step involved the integration of the import and rollback scripts into a main script. The import and rollback scripts were to be called in correct sequential order, allowing the program to verify that the data was imported successfully. Since the ordering of the import and rollback scripts was not initially executed correctly, further revisions were made in the third laboratory exercise. After closer inspection of the example provided in the Datasnippets repo, it became evident that the rollback scripts were to be imported in the inverse order of the import scripts.

The following screenshot is taken from the final version of the populate_database.py (main) script:

```
import pymysql
from rollback_order_products import *
from rollback_customer_orders import *
from rollback_products import *
from rollback_product_subcategory import *
from rollback_product_category import *
from rollback_customer import *
from rollback_customer_segment import *



from import_customer_segment import *
from import_customer import *
from import_product_category import *
from import_product_subcategory import *
from import_products import *
from import_customer_orders import *
from import_order_products import *
```

Query Interface. The next stage of the database development process involves the creation of a query command-line use interface written in Python. To implement this design, it was necessary to write a SQL script with a collection of queries of varying complexity. The SQL script was used as a blueprint in the development of Python command-line interface. An example is provided below:

```
--create view for products
--This view is created with an inner join, left outer join, and an order by clause
CREATE VIEW product_vw
AS SELECT p.product_id, p.product_name, p.unit_price, pc.product_category_name, psc.product_subcategory_name
FROM Products p INNER JOIN Product_Subcategory psc
ON p.product_subcategory_id = psc.product_subcategory_id
LEFT OUTER JOIN Product_Category pc
ON pc.product_category_id = psc.product_category_id
ORDER BY product_id;



--This view shows product category names with average unit price
CREATE VIEW product_cat_ave_price_vw
AS SELECT pc.product_category_name, FORMAT(avg(unit_price), 2) ave_unit_price
FROM Products p INNER JOIN Product_Subcategory psc
ON p.product_subcategory_id = psc.product_subcategory_id
LEFT OUTER JOIN Product_Category pc
ON pc.product_category_id = psc.product_category_id
GROUP BY product_category_name
ORDER BY ave_unit_price DESC;

--Selecting by specific category
SELECT * FROM product_cat_ave_price_vw WHERE product_category_name = ["Technology"];
SELECT * FROM product_cat_ave_price_vw WHERE product_category_name = ["Office Supplies"];
SELECT * FROM product_cat_ave_price_vw WHERE product_category_name = ["Furniture"];
```

The next step was writing a Python script using the queries written in the first step. The program provides users with various options to explore customer, product, and order data. Sub-menus with queries related to customer region, customer segment, order region, order year, and product category are also included. Furthermore, the command-line interface forms a hierarchy with 7 main-menu options and 31 sub-menu options, mapping to 38 SQL queries in total.

After verifying the functionality of the query interface discussed in the previous paragraph, it was necessary to perform SQL injections as a security measure. To perform this task, both the run_stmt() and run_prepared_stmt() functions were called in the try-except blocks, depending on whether the related query included user-input parameters. In particular, if further user-input wasn't required, then select statements were ran through the run_stmt() function. If the sub-menu option required further user-input, then the run_prepared_stmt() function was used.

Data Collection. The fourth set of lab exercises involved data collection through a RESTful API client. Students were allowed to choose any publically available RESTful API that returns JSON, but were encouraged to use Twitter. This is related to the textbook material related to this particular course, which includes a detailed and comprehensive example of collecting data on Twitter.

The tasks of collecting data related to the Superstore Orders dataset via Twitter was initially problematic. For one, the dataset is an educational tool consisting of both real and fake data. Since this particular learning exercise employed a real-world application, it was necessary to identify the real attributes from the Superstore Orders database.

Assuming the customer and order data was not real, the best option was to use the products data. However, the products data from this particular dataset is related to office supplies, a topic that on the surface would receive few if any mentions on Twitter. After doing a

few manually-tested Twitter searches with lackluster results, the Amazon and Walmart RESTful APIs were considered.

However, after reviewing Amazon's documentation, it became apparent that their RESTful API return XML and SOAP, not JSON. So, Amazon wasn't a viable option. Walmart's RESTful API returns JSON, but no examples of its implementation in Python were given on the company's website. Due to time limitations, Twitter was left as the best and only option for this lab exercise. Further manual exploration of the products data from the existing database was required, in order to identify which products would potentially be mentioned on Twitter. The two product types that seemed suitable for this exercise were *calculators* and *keyboards*.

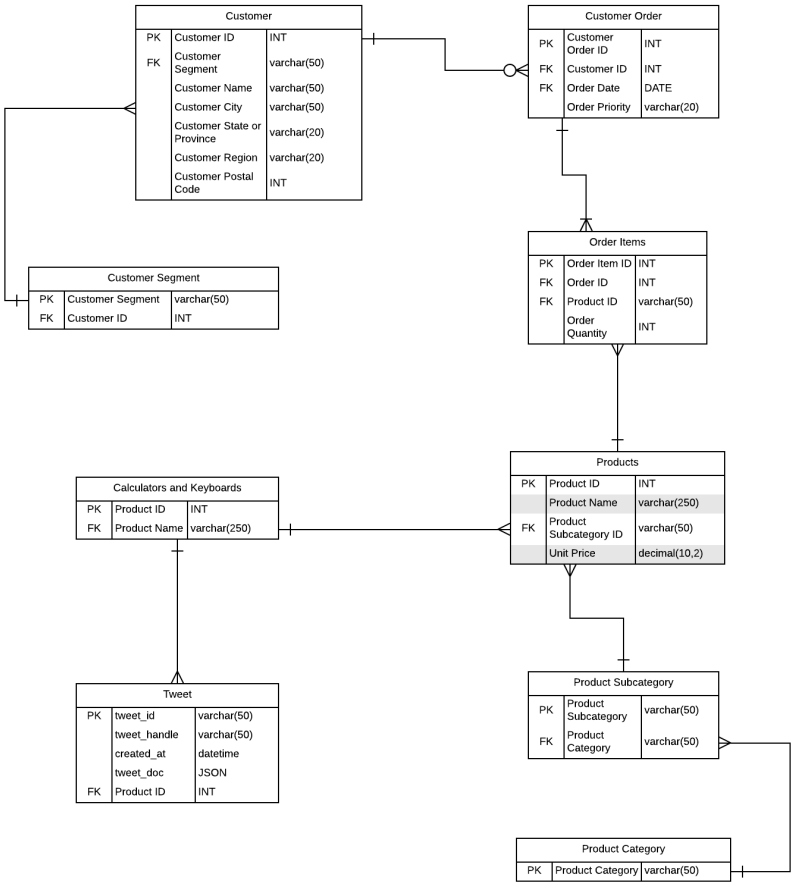
The first implementation of the extended logical and physical models involved the creation of a table with hardcoded attributes. For example, "Logitech" and "keyboard" would be inserted into *brand* and *item_type* columns in the new table. A new hardcoded table was created and populated in the extend_database.sql script.

The next step was to create a Tweet table containing data collected from SQL queries using concatenations of the data from the *brand* and *item*_type columns of the Calculator_Keyboards table. The primary key of the Tweet table was generated from the JSON document with a foreign key constraint on the product_id primary key of the Calculators_Keyboards table. The example provided in the Datasnippets repo was used to achieve this result.

The Tweet table would be populated using a Python RESTful API client via the Tweepy module. The twitter_client.py example provided in the Datasnippets repo was used as a template in creating the api_module.py. However, a *rate limit* error message along with 1000 Twitter

search results was returned upon running this module. It was apparent that the search queries were too broad.

This led to the redesign to the Calculators_Keyboards table. Instead of using a concatenation of attributes, distinct product names would be derived from the existing Products table. For example, “Microsoft Internet Keyboard” or “TI 36X Solar Scientific Calculator” could be used as search queries on Twitter. So, a new table consisting only of products with “calculator” or “keyboard” in the name would be created. The revised logical ER model is provided below:



After revising the extended logical model, the physical model was also modified to reflect any changes made. First, a table with calculator and keyboards would be created using an inner join between the existing Products and Product Subcategory tables. Furthermore, the LIKE SQL command would be used in conjunction with wildcard characters to filter out the calculator and keyboard products.

The new table, Calculators_Keyboards, would also have a foreign key constraint using the product_id primary key from the Products table. The Tweet table from the first extended logical model would be used again. After running the api_module.py script again and receiving only 79 Twitter search results, it was acceptable to move on.

The subsequent step involves running SQL queries on the JSON data derived from the Tweet table. These queries would be used to extend the user command-line interface that was created during the data loading exercises discussed on pages 7 and 8 of this document. A top-menu option called *Twitter data related to searches on calculators and keyboards* was added. This new main-menu option is mapped to sub-menu items allowing the user to retrieve data on tweets and retweets relevant to the products in the Calculators_Keyboards table. Finally, after the command-line interface was extended to include the data collected through the Twitter RESTful API client, the **mysqldump** utility was used to back-up the database.

Conclusion. Learning the fundamentals of relational databases in a formal academic setting is a process involving an exhaustive exploration of theoretical concepts as well as extensive hands-on experience through an amalgamation of software applications. It requires the gathering of information through multiple sources including lecture material, textbook readings, and website tutorials.

Additionally, the acquisition of multiple software development tools is necessary in the database software development process. For example, the Github graphical-user interface, LucidChart web-based diagramming application, Python object-orient programming language, MySQL Workbench, and Twitter RESTful API technology were used in the various lab exercises that were explored in this paper. Overall, the knowledge gained throughout this course enables students to transition between academic and real-world applications in relational databases.