NYPL Menu Dataset Analysis Report

*Team ID: Team69  
Name(Student ID):   
 Anthony Ravnic (aravni2),   
 Kumji Park(kupark2),   
 Mark Synowiec(ms217)  
Date: 7/4/25  
Course: CS513 Theory & Practice of Data Cleaning*

## **Description of DataSet**

### **Overview**

This report examines the structure and potential of a NYPL menu dataset, comprising four main components: Dish, Menu, MenuPage, and MenuItem. The objective is to understand how these entities relate to each other and propose meaningful data analysis scenarios (use cases) that leverage this data.

### **ER Diagram**

The dataset has been modeled into an Entity-Relationship (ER) Diagram to clarify how each table connects with the others:

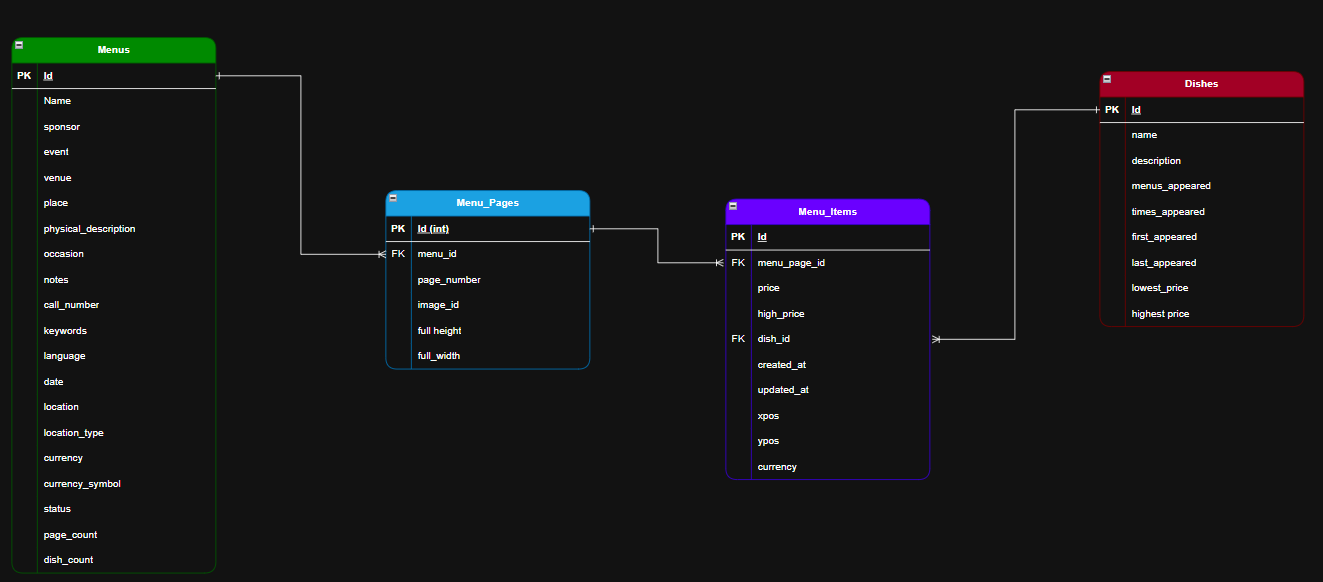
The ER diagram reflects a crows foot diagram showing the relationships of Menus, MenuPages, MenuItems, and Dishes. The Data Dictionary can be found in the “ColumnDescriptions.xlsx” file (see Appendix A) or in an archived page [Data\_Dictionary](https://web.archive.org/web/20221207031328/http://curatingmenus.org/data_dictionary/).

In the hierarchy of this NYPL menu model, the parent table would be “Menus” which gives info not only about the menu itself, but also the restaurant. The child table to this is the “MenuPages” table, which links via foreign key menu\_id to “Menu”.[Id]. “MenuPages” describes each page of its parent menu, including number, width, and height. The final child to this is “MenuItems” which lists the individual items on the menu, including when they’ve been created, updated, the price, the dish id, x position on menu, y position on menu, and currency used. Finally, we have Dishes, which will likely be an aggregated table of all the menu items table.

It is hard to tell the true relationship between dish and menu items at first glance but it would appear Dishes having aggregated values like menus\_appeard, times\_appeared, first\_appeared, last\_appeared, lowest and highest price, indicates that this is an aggregated/calculated table in which dish id is a primary key to the foreign key “dish\_id” in “MenuItems”. There is also a chance Dishes is a form of “silver table” that is created in a workflow after the base bronze tables (Menus, MenuPages, MenuItems) are ingested.

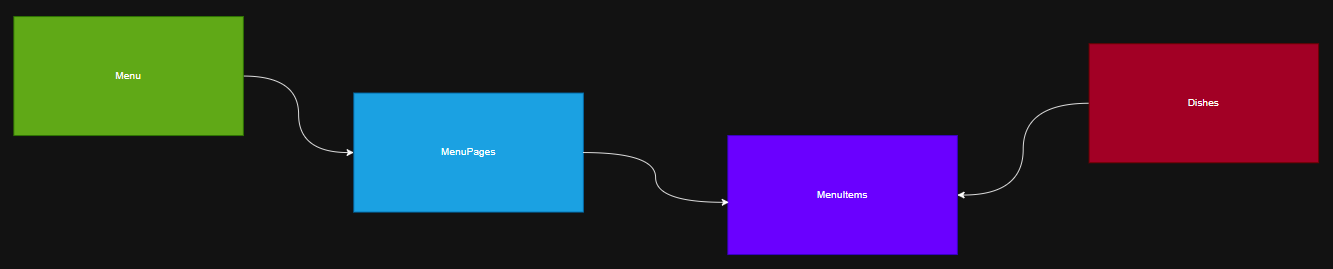
To better map the field descriptions found in this archived page [Data\_Dictionary](https://web.archive.org/web/20221207031328/http://curatingmenus.org/data_dictionary/), we re-formatted it into one document in the workbook “ColumnDescriptions.xlsx” (Appendix A). We also added more metadata fields to the tables in order to better map relationships. In smaller schemas, it doesn't matter as much, but when dealing with 100's-1000's of tables having the primary and foreign key relationships mapped out really helps. This data dictionary table can now be uploaded as a metadata table into the database. These added fields are as follows:

| Added Data Dictionary field | Description |
| --- | --- |
| Schema | Describes originating Schema (NYPL) |
| Table\_Name | Designates what table fields came from (color coded to match ERD table) |
| is\_primary\_key | y/n field to designate if primary key |
| is\_foreign\_key | y/n field to designate if field is foreign key |
| foreign\_key\_relationship | If table field is a foreign\_key, this denotes to which [table].[field] the foreign key holds a relationship to |



*“Crows foot” ERD diagram*

Whether Dishes is a separate aggregate or part of the key relationship in MenuItems. The general flow of ingestion would look something like this



### **Database Schema (Relational)**

CREATE TABLE Dishes(

id integer NOT NULL,

name VARCHAR(1387),

description VARCHAR(255), -- 0 length

menus\_appeared integer,

times\_appeared integer,

first\_appeared integer,

last\_appeared integer,

lowest\_price real,

highest\_price real,

PRIMARY KEY(id)

);

CREATE TABLE Menus(

id integer NOT NULL,

name VARCHAR(77),

sponsor VARCHAR(127),

event VARCHAR(194),

venue VARCHAR(47),

place VARCHAR(106),

physical\_description VARCHAR(118),

occasion VARCHAR(97),

notes VARCHAR(255), -- 0 length

call\_number VARCHAR(40),

keywords VARCHAR(255), -- 0 length

language VARCHAR(255), -- 0 length

date DATE,

location VARCHAR(127),

location\_type VARCHAR(255), -- 0 length

currency VARCHAR(26),

currency\_symbol VARCHAR(4),

status VARCHAR(12),

page\_count integer,

dish\_count integer,

PRIMARY KEY (id)

);

CREATE TABLE MenuItems(

id integer NOT NULL,

menu\_page\_id integer,

price real,

high\_price real,

dish\_id integer,

created\_at DATE,

updated\_at DATE,

xpos real,

ypos real,

PRIMARY KEY (id),

FOREIGN KEY (dish\_id) REFERENCES Dishes(id)

);

CREATE TABLE MenuPages(

id integer NOT NULL,

menu\_id integer,

page\_number integer,

image\_id integer,

full\_height integer,

full\_width integer,

uuid VARCHAR(36),

PRIMARY KEY (id),

FOREIGN KEY (menu\_id) REFERENCES Menus(id)

);

ALTER TABLE MenuItems

ADD CONSTRAINT FK\_MenuPage\_id

FOREIGN KEY (menu\_page\_id) REFERENCES MenuPages(id);

## 

## **2. Use Cases**

| **Use Case** | **Description** |
| --- | --- |
| U0: Zero Cleaning | Count the total number of menus |
| U1: Historical Trends | Calculate average menu prices by region and time |
| U2: Never Enough | Analyze customer favorites, but the dataset lacks customer reviews. |

#### **Target (main) use case: U1** - Getting the average prices of menu items in the US throughout the 1800s and early 1900s.

#### A historian may want to take a look at the vast collection of menus and see how prices have changed in the US over time in the 1800s and early 1900s. He may also want to see this broken out by state or city. While the query itself seems somewhat easy, this would require a good amount of cleaning on several fronts

1. Key violations between menu items, menu pages, and menus
2. Incomplete/inconsistent location data, specifically US cities/states
3. Incomplete/inconsistent pricing data, along with outliers, can lead to bad averages/medians
4. Potential Gap years where data wasn’t available in a specific state or region
5. Date columns contain dates far into the future or past that would be considered bad data

#### 

#### **“Zero-Cleaning” use case: U0** - Counting the Total Number of Menus The user may want to understand the volume of menu data across the entire collection, regardless of dish-level inconsistencies or typos. This SQL query will return the result of how many distinct menus exist in the dataset over time. It does not require cleaning the dataset because the query only uses the ‘id’ column from the ‘Menus’ table, which is a stable primary key, and it does not depend on ‘dish\_id’, ‘dish\_name’, or any inconsistent fields.

#### SELECT COUNT(DISTINCT id) AS total\_menus FROM Menus

#### **“Never Enough” use case: U2**

#### - Analyzing Customer Favorites by Dish across Time

#### The user may want to analyze which dishes are most loved by customers across time. It requires computing an average customer rating per dish and ranking them, in order to recommend popular items or highlight bestsellers in the future. The existing dataset (Menus, MenuPages, MenuItems, Dishes) contains no information about customers, reviews, or ratings. Since the necessary data is missing, it will never be enough, even if we clean the data.

## **3. Data Quality Problems**

#### During the initial inspection of the dataset above, the following data quality issues were identified, which must be addressed to support Use Case 1 (U1)

**a) Inconsistent Dish Naming**   
Many dishes appear under slightly different names by casing, word order, or formatting matters. It requires normalization to prevent artificial fragmentation in counts and averages.  
ex) Cold roast beef: COLD ROAST BEEF, Cold roast beef, cold Roast Beef  
 Potatoes au gratine: Hased potatoes au gratin, mashed potatoes au gratin, Potatoes Au Gratin C. A. A.  
  
 **Why this matters for U1?**If the same dish *“Cold Roast Beef”* appears under slightly different names across menus, our price averages will treat them as separate items. This artificially fragments counts and price calculations, producing a misleading result.  
 - COLD ROAST BEEF (5 occurrences, avg $2.50)  
 - Cold roast beef (8 occurrences, avg $2.60)  
 - Roast Beef, Cold (3 occurrences, avg $2.45)  
To prevent this, we will use **OpenRefine** to cluster and merge these variants into one standardized name.

**b) Inconsistent Location Naming**City and state names are inconsistently formatted or abbreviated, e.g., New York City, NYC, N.Y.C., which should be grouped together.  
  
**Why this matters for U1?**Since the historian wants to see price trends by state or city, these variations can split records into separate groups, distorting the average for that location. For example, A single year’s menus from New York and NYC would appear as two separate cities, each with half the data, leading to misleading averages.  
We will use **OpenRefine** to cluster and standardize location names. Also, we will handle missing or ambiguous location info.

#### **c) Missing Prices in Menu Items (Empty Cells)**

A significant number of ‘MenuItem’ entries have missing or zero prices  
  
**Why this matters for U1?**  
Missing or zero prices make it impossible to compute reliable averages.

For example, if a 1910 Chicago menu lists 15 items but only 5 have valid prices, the city’s average will be skewed low.

We plan to identify and handle these cases with **OpenRefine/Python**, either importing or excluding them with clear rules.

**d) Outliers (Extreme Price Values)**  
A few menu items may contain unrealistically high or low prices due to OCR errors or manual data entry mistakes, such as misplaced decimal points.

**Why this matters for U1:**U1 depends on calculating accurate averages over time. Outliers can significantly distort trends, especially when the average is calculated for a small city or a single year. For instance, if a soup menu item was mistakenly entered as $100 instead of $1.00 for a 1910 menu, the average price for that year would be unrealistically high.   
  
We will use **OpenRefine** to identify extreme price outliers. Suspicious records will be reviewed and either corrected if possible or removed from the final calculations.

**e) Implausible or Bad Dates**Some date fields contain unrealistic or incorrect years, such as 2200 or 1700, due to scanning or manual entry errors.

**Why this matters for U1:** Since U1 analyzes price trends by year/decade, menus with implausible dates will end up in the wrong time bins, creating false spikes or gaps in the trend lines. Both outliers and bad formats could cause misinterpretations of the data and lead to incorrect conclusions  
 We will set a valid date range filter (e.g., 1800–1950) and use **Python/OpenRefine** to flag and review dates outside this range. Invalid dates will be corrected if possible or set to NULL to avoid skewing the time series.

**f) Broken Foreign Key Relationship**  
There are cases of missing keys referenced across tables. For example, some MenuItems rows reference ‘dish\_id’ values that do not exist in ‘Dishes’ (e.g., dish\_id 220797…). This breaks the link needed to group and clean dish names correctly. Likewise, there are missing links between ‘MenuItems’ → ‘MenuPages’ and ‘MenuPages’ → ‘Menus’.

#### menuitem.csv referenced dish\_id 220797 on line 588748 does not exist.

#### menuitem.csv referenced dish\_id 329183 on line 797057 does not exist.

#### menuitem.csv referenced dish\_id 395403 on line 1001145 does not exist.

#### menupage.csv referenced menu\_id 12460 on line 1 does not exist.

#### menupage.csv referenced menu\_id 12460 on line 2 does not exist.

#### menupage.csv referenced menu\_id 12460 on line 3 does not exist.

#### **Why this matters for U1:**

Broken foreign key connections between tables can cause valid menu item prices to disappear from our queries altogether. For instance, if a MenuItem references menu\_id = 12460 that doesn’t exist in menus, any price linked to it is lost in the join. We also lose the connection to the menu table, in which we can’t assign a date to the menuitem/dishes price.

#### We’ll use **SQL** to detect and fix or remove these orphaned keys.

#### **g) Missing Currency Information (Empty Cells)** Many ‘Menu’ entries lack currency and symbol data, making price interpretation ambiguous, which may cause unreliability when comparing prices.

#### **Why this matters for U1:**

#### Missing currency/price values may cause us to filter out menu items (when filtering on dollars) and potentially skew the average prices for some U.S. cities/states.

#### Given we don’t have the data available, it is best to filter these out and only use the data available to us to determine the average price

## 

## **4. Initial Plan for Phase II**

### **S1: Description of Dataset (D) and Use Case (U1)**

#### Dataset (D): The NYPL Menu dataset consists of four tables: Dishes, Menus, MenuPages, and MenuItems. It includes historical menus from the 1800s to the early 1900s with menu item names, prices, venues, and related metadata.

#### Target Use Case (U1): *Analyze the average prices of menu items in the US across different cities and decades.* To support this use case, the dataset must have reliable price data, consistent menu item names, valid currency information, and complete foreign key references between related tables.

### **S2: Profiling of D to Identify Data Quality Problems (P)**

### During Phase I, we used SQLite browser, OpenRefine, and custom profiling tools to get an initial understanding of the dataset.

#### Examples of profiling results:

#### Found that the ‘location’ column in the Menus table has a maximum length of 127 characters, and has variability in the format which could cause potential inconsistencies.

#### Used a regex-based tool to replace numbers with NUM and all texts with TEXT.

#### For example, the ‘location’ column in Menus.csv showed:

#### TEXT: 14470 rows

#### TEXT'TEXT: 1062 rows

* + TEXT&TEXT: 296

#### TEXTNUM: 50 rows

#### TEXT\TEXTé\"": 47 rows

#### This gives us a higher-level view of the data to see how data might be cleaned, or if the regular expressions used to group characters should be modified to better group values. Using **OpenRefine** will provide a more sophisticated method to see how data can be grouped and cleaned. During Phase II, we will use the same tools and process, but in greater depth to help plan our cleaning process.

**S3: Performing the Data Cleaning Process**

To address the various data quality problems identified during profiling, our team plans to use a combination of complementary tools and methods. We will primarily use **Python**, **OpenRefine** to group similar city/state names into clusters, parse and standardize them, ensuring that variations like casing or minor spelling differences do not fragment our counts and averages. **Python** scripts and **OpenRefine** will also help us detect and clean irregular text patterns more systematically. Also, **SQL, Datalog** will be used to specify IC violations detected and missing foreign key references. And **YesWorkflow** to document and visualize the entire cleaning process.

**S4: Data quality checking: is D’ really “cleaner” than D?**

* Running **SQL/Datalog** queries to show that FK violations have been resolved and improvements in the data.
* Comparing before/after profiles of NULLs, value lengths, and pattern matches.
* Using sample queries to show average prices grouped by decade and region to demonstrate U1 has a meaningful outcome using the cleaned dataset.

**S5: Document and quantify change (e.g., columns and cells changed, IC violations detected: before vs after, etc.)**

* Maintain an **OpenRefine cleaning recipe** and export it for reproducibility.
* Log the number of rows affected by automated tools, SQL, and/or Datalog(e.g., how many location variants were grouped).
* Record IC violation counts before and after.
* Use **YesWorkflow** to diagram the overall cleaning workflow.

**Phase II Schedule**

For Phase I, our team has worked well with weekly tagups on a Slack Huddle and a Slack channel for daily discussion on tasks and progress. We’ve set up a GitLab repo to share data, analysis, and tools as we perform tasks. As work is performed, we have discussed and critiqued results and provided regular support to one another to overcome technical hurdles and bounce ideas within the team. This has worked well for us and we plan to continue this cadence for Phase II.

Our team will submit Phase I after our final tag-up to review the material on or before July 6. This will give us 4 weeks to complete Phase II. We expected progress to follow this schedule, with some overlap between steps:

| Step | Dates | Description |
| --- | --- | --- |
| S1 | July 6 - July 11 | Use case and dataset description |
| S2 | July 6 – July 16 | Profiling for data quality issues |
| S3 | July 13 - July 27 | Data cleaning with tools |
| S4 | July 20 - July 30 | Checking that D′ is improved |
| S5 | July 10 - Aug 2 | Documenting all changes and workflow from the start of Phase II |

**Team Roles for Phase II**

To ensure that our data cleaning process fully supports the main use case (U1), each team member will focus on specific tools and tasks:  
 - **Mark** will lead the use of **OpenRefine** for clustering and standardizing date and location names to clean data.  
 - **Kumji** will maintain our **YesWorkflow** diagrams to document the overall cleaning pipeline and dependencies.  
 - **Anthony** will focus on **SQL scripts and formal proofs** to verify that FK violations have been resolved, and improvements in the data also confirm that the changes we will make for cleaning dataset are addressed intentionally. Optionally, he will also explore using **Python scripts** for additional address parsing (breaking out cities and states) if necessary.

## Appendix A - ColumnDescriptions.xlsx



