CS513 Final Report

BFS DATA JANITORS

Rick Bischoff (rdb4)

Wenyi Shang (wenyis3)

Stanley Lim (sflim2)

Introduction

Imagine you're a modern 2020s hipster foodie, sans-covid-19, time-traveler. You suddenly find yourself transported a century prior to 1920s New York. You feel hopeless without your modern Bon Appetit magazines and food vloggers telling you where to eat.

Suddenly you remember you downloaded the NYPL Labs historic menu data for an awesome class you took at UIUC. You're in luck! You set out to create an app that can serve your needs, but first, the data…

Initial Assessment of the Data

The data is provided by the New York Public Library, which painstakingly collected 45,000+ menus from the 1840s to present day. These datasets were curated by crowdsourcing the transcriptions, which has resulted in a massive volume of data albeit at the cost of cleanliness!

We are using CS513's reference copy of the dataset as of June 2020. The data is also available at http://menus.nypl.org/data but as it is a living project, specific values may vary.

The raw data comes as four relations: Menu, MenuPage, MenuItem, and Dish:

1. Menu.csv: This is the messiest of all 4 data sets. It contains detailed information about the provider of the menu such as name of provider, event for the menu, number of pages of the menu, and number of dishes. The available variables are: id, name, sponsor,

event, venue, place, physical_description, occasion, notes, call_number, keywords, language, date, location, location_type, currency, currency_symbol, status, page_count, and dis_count. There are a total of 17,545 observations.

   a. There are many empty cells in the variable "name".
   b. Variable "name" appears to be a duplicate of variable sponsor when the former contains non-empty cells. As such, we can potentially remove this variable from the data.
   c. Variable "location" and variable "sponsor" carry the same information, and so we can possibly drop one of the two.
   d. Some variables contain capitalized letters (e.g. sponsor) whereas others contain lower case letters (e.g. location). We can standardize to make downstream processing easier.
   e. Some variables contain no values. This includes "keywords", "language", and "location_type". Therefore, we can drop these columns.
   f. All variables are in text format. Because some variables are numeric, they could be transformed into numeric format.

2. Dish.csv: This data contains detailed dish information capturing number of times a dish appears on menus, first appearance, last appearance, lowest price, and highest price. The available variables are: id, name, description, menus_appeared, tims_appeared, first_appeared, last_appeared, lowest_price, and highest_price. There are a total of 423,397 observations. Id can be matched with dish_id in MenuItem.csv. There is also a "name" field, which at first glance appears to be mostly unique.
   a. The variable "description" can be dropped as it contains no values.
   b. Conversion of text to numeric for: "id", "menus_appeared", "times_appeared", "first_appeared", "last_appeared", "lowest_price", "higest_price".

The other two data sets are relatively clean.

3. MenuPage.csv: This data contains id and menu id with the corresponding image information such as image id, dimensions (height and width) and a unique identifier. The available variables are: id, menu_id, page_number, image_id, full_height, full_width, and uuid. There are a total of 66,937 observations.

   a. Data appears to be cleaned and possibly only requires conversion of text to numeric for: “id”, “menu_id”, “page_number”, “image_id”, “full_height”, and “full_width”.

4. MenuItem.csv: This data contains mainly prices, including high price of menu items. Time stamps are provided to indicate relevance of price information. That is, whether the information are up to date. The available variables are: id, menu_page_id, price, high_price, dish_id, created_at, updated_at, xpos, and ypos. There is a total of 1,332,726 observations. Menu_page_id can be matched with menu_id in MenuPage.csv.
   a. Data appears to be cleaned and possibly only requires conversion of text to numeric for: “id”, “menu_id”, “menu_page_id”, “price”, “high_price”, “dish_id”, “xpos”, and “ypos”.

Assessment of Data for Hypothetical Use-Case

Our hypothetical use-case is to build an app that will let us find the meals we want around New York City in the 1920s. We want to search for specific restaurants, our favorite ingredients, our favorite meals, find the cheapest food (when the depression hits!), and finally find some recommendations for similar restaurants to the ones we love.

Presently there are significant challenges to using the data as-is in this context:

1) Menu location is very messy. It is a combination of street addresses, cities, countries, and planes, trains and cruise ships. We will need to spend effort on cleaning the location up if we hope it to be useful for a New York centric application.
2) Dish description is completely blank, so we will have to rely on parsing information out of the “name” field. Our initial assessment was that “name” looked relatively clean, but it turns out that it is not and significant effort went towards cleaning that field and making it useful for this application.
3) MenuPage will not be needed for this use-case.
4) MenuItem does get some cleaning via the transitive property. We change “Dish” which is connected to MenuItem via “dish_id” and so there were some changes there.

There are many possible uses for this data. You could expand the scope of our hypothetical app by making it worldwide in scope, for instance. But this would pose many challenges, as there are menus present from trains, planes, and cruise ships. This might get frustrating to our end-users if they are suggested a meal they can't actually get. There would also be some real challenges in dealing with so many currencies across so wide of a time-frame.

Other possible use cases:

- Search for places that offer menus within a price range
- Menus available at a specific hotel
- Menus for occasions (e.g. Christmas dinner)
- Based on the menus offered by restaurants/hotels and for different occasions, homemakers can use them as reference for preparing meals at home
- New business owners can refer to the available menus at different locations to understand local preferences on food (presumably food institutions serve food that meets local preferences) and prepare their own menus

The data as-is is not presently in a form that would suggest a use-case other than as a historic museum of menus. For example, because there are no consumer reviews nor rating provided by consumers who have dined at specific places, we are not able to rank the recommendations (e.g., food or restaurants). Moreover, other than the menus of food, it may be difficult for people to choose restaurants/hotels to dine at that provide them specific ambience, service quality, seating and car park availability, and timing of operation.

Data Cleaning

OpenRefine

1. Menu.csv
    a. Drop variable name.
    b. Drop variable location.
    c. Drop the following variables as cell values are blank: keywords, language, location_type.

d. Trim leading and trailing white spaces and collapse consecutive white spaces for all columns.

Trim leading and trailing white spaces	
Variable	No. of changes
Id	0
Sponsor	15
Event	3
Venue	0
Place	8
Physical_description	384
Occasion	0
Notes	125
Call_number	9
Language	0
Date	0
Location_type	0
Currency	2
Currency_symbol	4
Status	0
Page_count	0
Dish_count	0

Collapse consecutive white spaces	
Variable	No. of changes
Id	0
Sponsor	127
Event	6
Venue	0
Place	45

Physical_description	38
Occasion	3
Notes	195
Call_number	0
Language	0
Date	0
Location_type	0
Currency	0
Currency_symbol	0
Status	0
Page_count	0
Dish_count	0

e. Check for incorrect data types and convert text to numeric format for: id, call_number, page_count, and dish_count.
f. Remove special characters for variable occasion:
    i. Using GREL value.replace(";",""): 2332 changes.
    ii. Using GREL value.replace("[",""): 25 changes.
    iii. Using GREL value.replace("]”,""): 25 changes.
    iv. Using GREL value.replace("(",""): 127 changes.
    v. Using GREL value.replace(")”,""): 136 changes.
    vi. Using GREL value.replace("?”,""): 62 changes.

g. Convert upper to low cases for sponsor, event, venue, place. Occasion, notes,
h. Clustering for the following variables to group similar texts together, for example, dinner, [dinner], (dinner) and convert foreign language to English (e.g. mittagesen to lunch). During this process, we also removed special characters such as (), [].
    - sponsor
    - event
    - venue
    - place

2. MenuPage.csv
    a. Transform the following variables to numeric format: ID, menu_id, page_number, image_id, full_height, full_width.
    b. Trim leading and trailing white spaces for all.

3. MenuItem.csv:
    a. Transform the following variables to numeric format: id, menu_page_id, price, high_price, dish_id, xpos, ypos.
    b. Trim leading and trailing white spaces for menu_page_id and id.

4. Dish:
    a. Drop variable description as it contains no values.
    b. Transform the following variables to numeric format: ID, menus_appeared, times_appeared, first_appeared, last_appeared, lowest_price, higest_price.
    c. Trim leading and trailing white spaces and collapse consecutive white spaces for variable name.

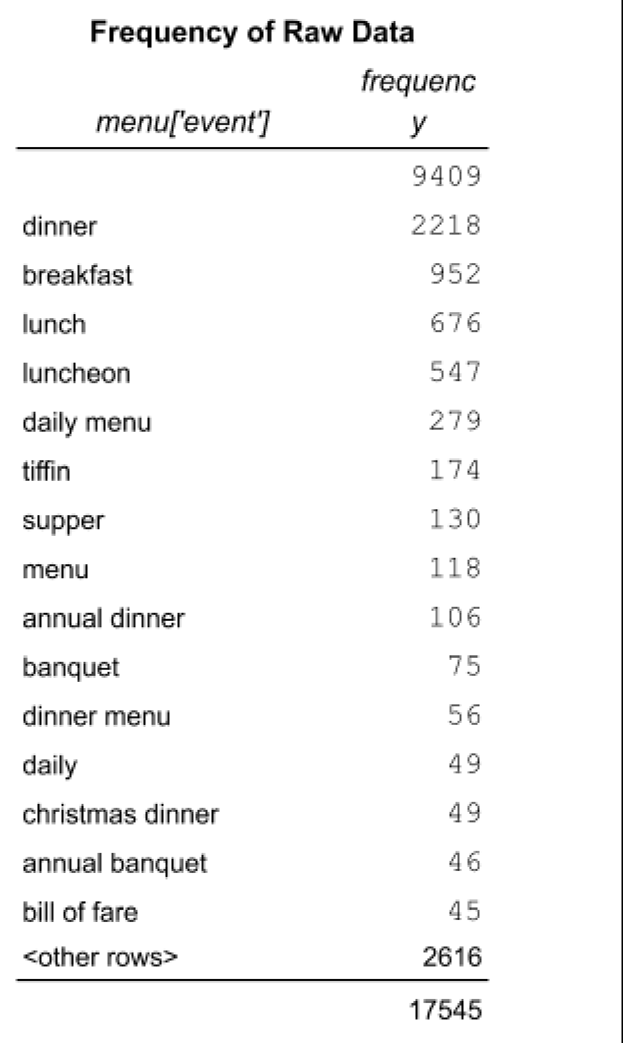
The operations history for each of the above data sets and the processed data sets are available as .json and .csv files, respectively, in the supplemental materials. Note that while the operations history feature in Open Refine captures the history of actions taken on the data, it does not capture the number of changes made.

Python

We also utilized Python Pandas library to facilitate some of the cleaning. This was preferable to OpenRefine in some ways as it allowed us finer control of some of the clustering steps, which we'll explain later.

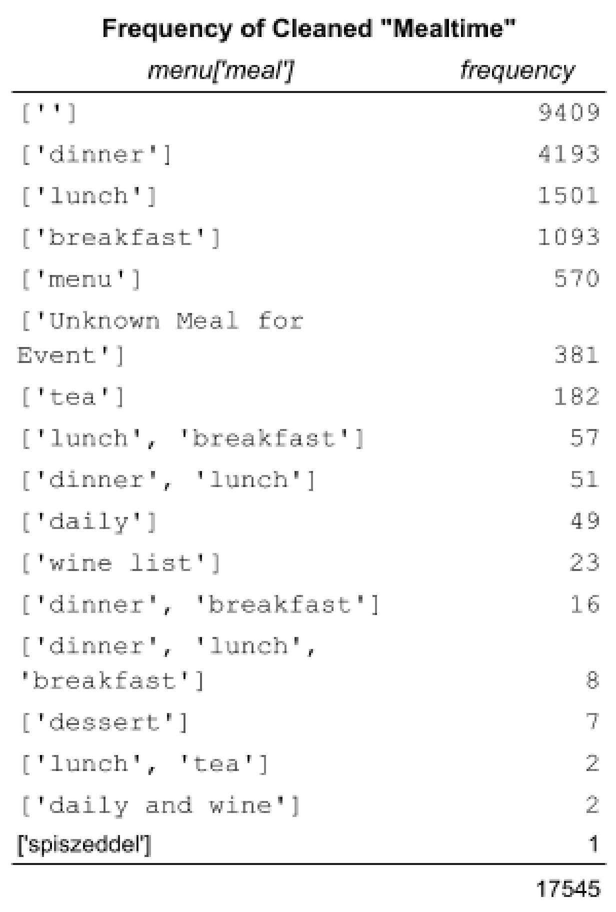
Menu - (03_python/f04_cleanMenu.py)

1. The field "event" contained a rich mix of "time-of-day" information (such as dinner, lunch, breakfast), as well as details about holiday celebrations, reunions, as well as typical restaurant menus. We created a series of regular expressions to determine if there was a meal-time indicated for the menu, and if so, to cluster that into a single label. For instance "dinner", "diner", "supper", "abendessen", "evening menu", etc. would all get mapped to the singular "dinner". We also manually grouped many events with an unclear meal time to "Unknown Meal for Event".



Frequency of Raw Data

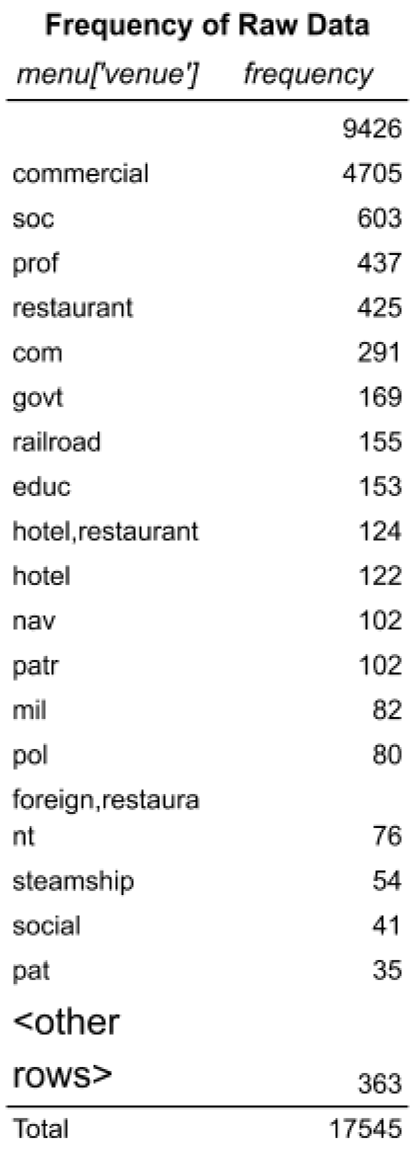
menu['event']	frequency
	9409
dinner	2218
breakfast	952
lunch	676
luncheon	547
daily menu	279
tiffin	174
supper	130
menu	118
annual dinner	106
banquet	75
dinner menu	56
daily	49
christmas dinner	49
annual banquet	46
bill of fare	45
<other rows>	2616
	17545



Frequency of Cleaned "Mealtime"

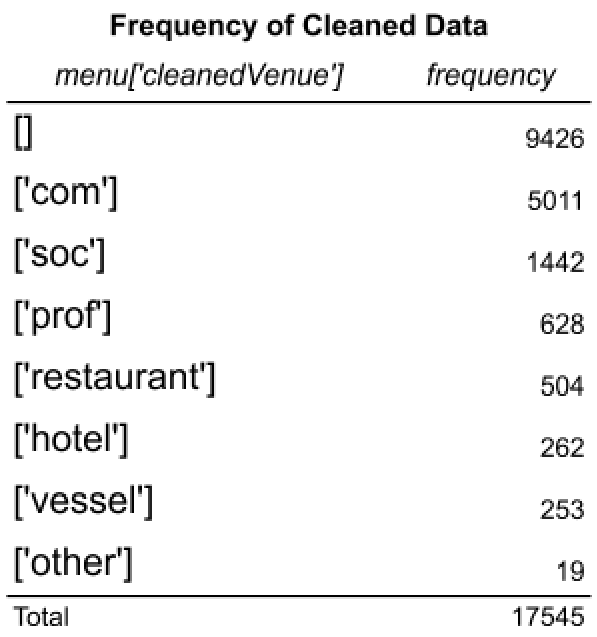
menu['meal']	frequency
['']	9409
['dinner']	4193
['lunch']	1501
['breakfast']	1093
['menu']	570
['Unknown Meal for Event']	381
['tea']	182
['lunch', 'breakfast']	57
['dinner', 'lunch']	51
['daily']	49
['wine list']	23
['dinner', 'breakfast']	16
['dinner', 'lunch', 'breakfast']	8
['dessert']	7
['lunch', 'tea']	2
['daily and wine']	2
['spiszeddel']	1
	17545

2. The field “venue” contained a detailed but undocumented ontology of different venues. Most of the detail was unneeded for our application, so we clustered these up into only 6 major categories:
    a. Vessel - Was this a restaurant used on a ship, plane, etc.
    b. Restaurant - Was this an “ordinary” restaurant
    c. Professional - Was this a gathering of like minded professionals
    d. Hotel - Was this a hotel specific menu
    e. Government - Was this a government entity
    f. Commercial
    g. Other



Frequency of Raw Data

menu['venue']	*frequency*
	9426
commercial	4705
soc	603
prof	437
restaurant	425
com	291
govt	169
railroad	155
educ	153
hotel,restaurant	124
hotel	122
nav	102
patr	102
mil	82
pol	80
foreign,restaurant	76
steamship	54
social	41
pat	35
<other rows>	363
Total	17545



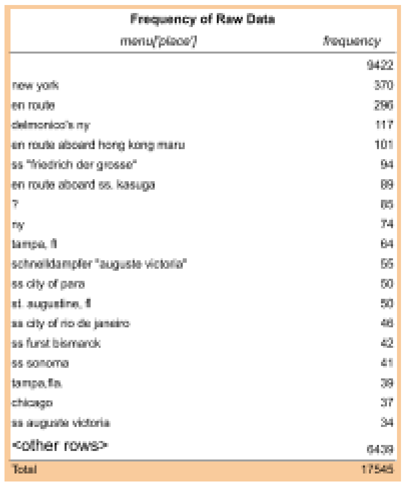
Frequency of Cleaned Data

menu['cleanedVenue']	*frequency*
[]	9426
['com']	5011
['soc']	1442
['prof']	628
['restaurant']	504
['hotel']	262
['vessel']	253
['other']	19
Total	17545

3. The field “place” contained quite the mix of information. In some cases, there was actually a physical address. In other cases, a city was listed. In others, a country. And

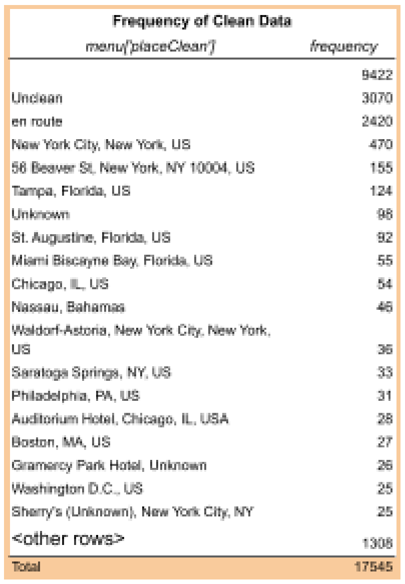
in quite a few, “en route” about a vessel. We undertook an effort to cluster these together and apply some common formatting manually:

a. If “place” was about a vessel, place was set as “en route” and the vessel name into a separate field. If we knew there was a vessel, but did not know its name “Unknown” was used. If there is no possibility a vessel was present, we used “NA”. And finally, if we knew what kind of vessel, but not it’s name, we used “Unknown (Train)”, if it were a train.
b. For everything else, there was some consistent formatting applied based on the amount of information available:
    i. Country
    ii. State/Province, Country
    iii. City, State/Province, Country
    iv. St. Address, City, STate/Province, Country
    v. Incomplete Information was shown with a “?” at the end representing incomplete information.
c. This information was split out into two new fields: placeClean and vesselName.
d. Note: The manual step was quite labor intensive, and we only did the most common entries--leaving 3070 rows uncleaned.



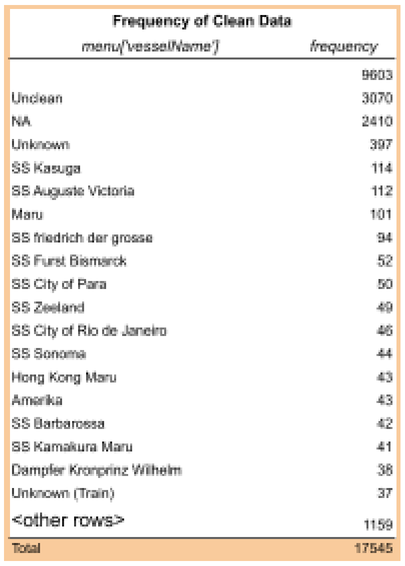
Frequency of Raw Data

menu['place']	*frequency*
	9422
new york	370
en route	296
delmonico's ny	117
en route aboard hong kong maru	101
ss "friedrich der grosse"	94
en route aboard ss. kasuga	89
?	85
ny	74
tampa, fl	64
schnelldampfer "auguste victoria"	55
ss city of para	50
st. augustine, fl	50
ss city of rio de janeiro	46
ss furst bismarck	42
ss sonoma	41
tampa,fla.	39
chicago	37
ss auguste victoria	34
<other rows>	6439
Total	17545



Frequency of Clean Data

menu['placeClean']	*frequency*
	9422
Unclean	3070
en route	2420
New York City, New York, US	470
58 Beaver St, New York, NY 10004, US	155
Tampa, Florida, US	124
Unknown	98
St. Augustine, Florida, US	92
Miami Biscayne Bay, Florida, US	55
Chicago, IL, US	54
Nassau, Bahamas	46
Waldorf-Astoria, New York City, New York, US	36
Saratoga Springs, NY, US	33
Philadelphia, PA, US	31
Auditorium Hotel, Chicago, IL, USA	28
Boston, MA, US	27
Gramercy Park Hotel, Unknown	26
Washington D.C., US	25
Sherry's (Unknown), New York City, NY	25
<other rows>	1308
Total	17545



Frequency of Clean Data

menu['vesselName']	*frequency*
	9603
Unclean	3070
NA	2410
Unknown	397
SS Kasuga	114
SS Auguste Victoria	112
Maru	101
SS friedrich der grosse	94
SS Furst Bismarck	52
SS City of Para	50
SS Zeeland	49
SS City of Rio de Janeiro	46
SS Sonoma	44
Hong Kong Maru	43
Amerika	43
SS Barbarossa	42
SS Kamakura Maru	41
Dampfer Kronprinz Wilhelm	38
Unknown (Train)	37
<other rows>	1159
Total	17545

Dish - (03_python/f04_cleanDish.py)

1. Preliminary frequency on dish “name” sorted in order of frequency, shows there were only a few duplicates:

Amount of Occurences by dish['name']

Occurences	*Unique Dishes*	*Total Count*
1	411634	411634
2	5441	10882
3	255	765
4	24	96
20	1	20
Total	417355	423397

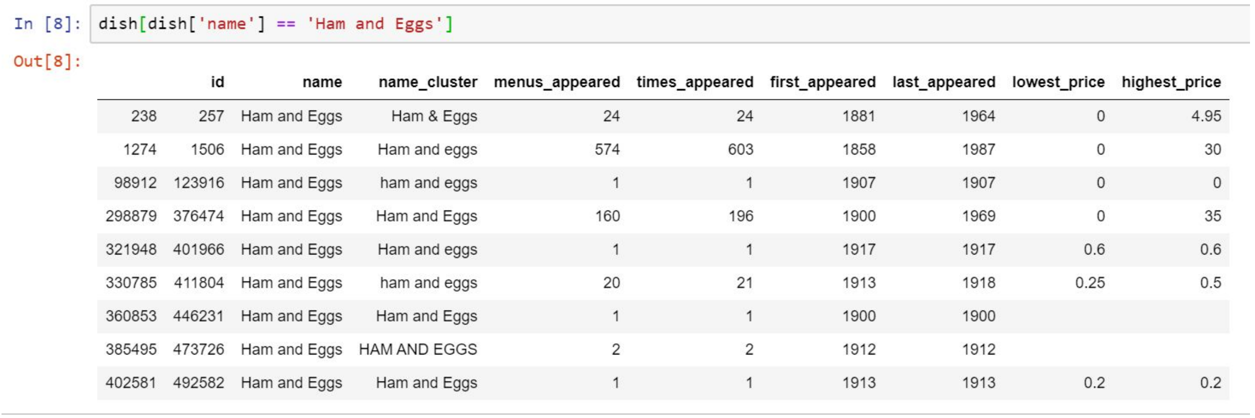
2. Based on this, we thought ““name” was relatively clean, but was soon discovered here to be quite messy! First we apply a common “Title” case function to the field, replace “&” with “and”, and also convert small words like “a la”, “of”, etc into lowercase.
    a. 230,444 entries were changed by this operation.
3. Rerunning the frequency on dish[‘name’] now reveals there is a significant duplication issue:

Amount of Occurences by dish['name_tmp']

Occurences	*Unique Dishes*	*Total Count*
1	364707	364707
2	17244	34488
3	4012	12036
4	1516	6064
5	581	2905
6	259	1554
7	113	791
8	53	424
9	27	243
10	7	70
11	4	44
12	2	24
13	1	13
14	1	14
20	1	20
Total	388528	423397

4. We then set out about deduplicating Dish. It was assumed that “dish_name” was to be a primary key of this relation, so we set out to make sure that integrity constraint still held.
    a. We first grouped together common dish names.
    b. Next, we recalculated the aggregate values “menus_appeared”, “times appeared”, “first_appeared”, “last_appeared”, “lowest_price”, “highest_price” by applying the right aggregation across common names. For instance “menu_appeared” was the sum of all “menu_appeared” rows with the same dish name.
    c. Then we generated a new entry for “Dish” that contained a new unique identifier “id”, and contained the unique “name” and the recalculated values listed in step (b).
5. Next, we deleted the duplicate rows.
6. Because MenuItem had referential integrity to “dish”, it was also necessary to update the references there to maintain that integrity.

To make things more concrete, here is an example of what was deduplicated:

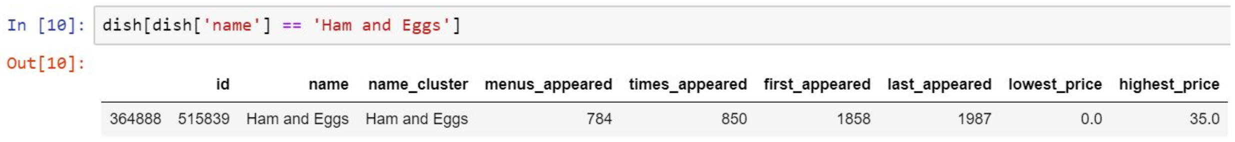


```
In [8]: dish[dish['name'] == 'Ham and Eggs']
```

Out[8]:

	id	name	name_cluster	menus_appeared	times_appeared	first_appeared	last_appeared	lowest_price	highest_price
238	257	Ham and Eggs	Ham & Eggs	24	24	1881	1964	0	4.95
1274	1506	Ham and Eggs	Ham and eggs	574	603	1858	1987	0	30
98912	123916	Ham and Eggs	ham and eggs	1	1	1907	1907	0	0
298879	376474	Ham and Eggs	Ham and Eggs	160	196	1900	1969	0	35
321948	401966	Ham and Eggs	Ham and eggs	1	1	1917	1917	0.6	0.6
330785	411804	Ham and Eggs	ham and eggs	20	21	1913	1918	0.25	0.5
360853	446231	Ham and Eggs	Ham and Eggs	1	1	1900	1900		
385495	473726	Ham and Eggs	HAM AND EGGS	2	2	1912	1912		
402581	492582	Ham and Eggs	Ham and Eggs	1	1	1913	1913	0.2	0.2

Here it is clear that all of these rows are in fact referring to the same dish “Ham and Eggs”. So we collapse all of these entries into a new row of “dish” and assign a new “id” to it:

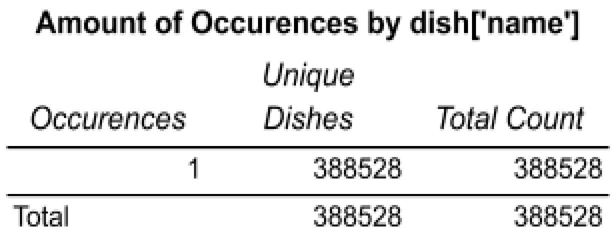


```
In [10]: dish[dish['name'] == 'Ham and Eggs']
```

Out[10]:

	id	name	name_cluster	menus_appeared	times_appeared	first_appeared	last_appeared	lowest_price	highest_price
364888	515839	Ham and Eggs	Ham and Eggs	784	850	1858	1987	0.0	35.0

And here is evidence that dish_name is now completely unique:



Amount of Occurences by dish['name']

Occurences	*Unique Dishes*	*Total Count*
1	388528	388528
Total	388528	388528

Understanding the record chain here is a little confusing--how did we go from 423397 to 388528--so let’s break it down. Most of the rows in “dish” (364,707) were unique by the clean name. However the remainder 58,690 were duplicates--their name collided with another row. These 58,690 rows in “dish” only made up 23,821 unique dishes. Therefore the duplicate 58,690 rows were deleted from “dish” and 23,821 new rows were added to represent the new unique dish_name. 423,397 (original entries in dish) - 58,690 (duplicate entries) + 23,821 (summarized duplicates) = 388,528.

7. Rows in “MenuItem” pointing to any of the deleted rows in “Dish” then had to be corrected by pointing to the newly added rows.
    a. menuItem has 1,332,726 rows
    b. 743,227 of these rows pointed to one of the 58,690 records and needed to be changed to point to the new dish ID.
    c. 589,499 of these rows pointed to one of the other dishes that were not touched.
8. Finally, “name” was tokenized (e.g. split into word tokens that were separated by white-space) and exported to an external spreadsheet. This was then manually labelled, and is documented in the next section.

Manual Labelling Step

The name of “dish” was so dirty, it was deemed unlikely that a regular expression would be able to extract reliable consistent information. Automatic clustering was considered, but because there were 400k+ unique entries, a simpler approach seemed warranted.

First, one of our team members studied the dish names and noticed some common themes:

1. Menu items frequently contained preparation adjectives, like “Broiled”, “Braised”, “Roasted”.
2. Legitimate Menu items always have an ingredient--e.g. “Chicken” or “eggs” or “beef tongue”
3. Menus often contain flowery adjectives that are meaningless--e.g. “House” or “Fresh”
4. Often there were descriptions of the portion size or quantity--e.g. 1 doz.
5. Further ontological structure was discovered--Region names (English, French, Japanese, etc.) were frequently used, as were Brand names (e.g. Anhauser-Busch).
6. Often the names were foreign and easily translated to English (e.g. “*salade”* fo salad)

If this were a real commercial project, we’d partner with some culinary experts and define a proper grammar of dish names. As it were, we are simply grad students and we pretended to be experts and manually labelled the tokens that appeared in the top 60% of data.

Dish - (03_python/f05_cleanDish.py)

1. First we imported the cleaned token list as described above, as well as the dish dataset from the previous 03_python/f04_cleanDish.py step. Dish still had 388,528 rows.
2. A simpler translation and whitespace removal process took place:
   a. Foreign words that showed up in our manually labelled step were translated per the label.
   b. Extraneous whitespace in the middle of the name was removed.
   c. For efficiency, these steps were carried out simultaneously. 33,244 rows were changed.
3. Next we looped over the Dish dataset and created new fields corresponding to the ontology above:
   a. ingredients - space separated list of all ingredients present in the dish
   b. preparation - space separated list of all preparation verbs present in the dish
   c. Adjectives -- space separated list of flowery adjectives
   d. Region -- space separated list of all present region names in the dish
   e. Quantity -- space separated list of all quantity information present
   f. Brand -- space separated list of all brands in the dish.
   g. Incomplete tokens -- all tokens that were not labelled. This is a treasure trove of further research and is where you'd want to get started to further refine.

It should be noted again we only cleaned 60% of the dish tokens, which we felt was enough to demonstrate the intent and satisfy project requirements. See table on following page for resulting frequencies of extracted information.

Extracted information From Dish

Ingredients	*Frequency*
	162203
Chicken	6218
Wine	3144
Potatoes	2976
Beef	2726
Butter	2378
Ham	2165
Lamb	2164
Cheese	2127
Fries	2074
Veal	2048
Lobster	2026
Ice	1864
Egg	1704
Champagne	1690
Potato	1686
Eggs	1593
Tomato	1411
Fruit	1386
<other values>	184945
Total	388528

Region	*Frequency*
	350463
French	8439
Chateau	3212
English	1680
American	1213
Virginia	1000
Boston	965
California	934
Spanish	909
Italian	843
Parisienne	824
Bordelaise	804
Maryland	777
Madeira	698
Bordeaux	687
York	675
German	667
Rhine	649
Newburg	648
<other values>	12441
Total	388528

Adjectives	*Frequency*
	312796
White	3649
Extra	2749
New	2462
Spring	2361
Red	2162
Sweet	1984
House	1877
Old	1768
Prime	1460
Vanilla	1337
Breast	1317
Supreme	1308
Plain	1289
Choice	1267
Assorted	1246
Les	1225
Club	1204
Chops	1196
<other values>	43871
Total	388528

Preparation	*Frequency*
	209676
Cream	7540
Salad	7434
Sauce	6546
Broiled	4352
Fried	4255
Fresh	3277
Soup	2909
Roast	2692
Filet	2575
Dry	2378
Cold	2173
Consomme	2077
Baked	2011
Imported	1769
Saute	1593
Jelly	1568
Boiled	1553
Glazed	1491
<other values>	120659
Total	388528

Quantity	*Frequency*
	351124
Half	3046
Cup	2030
2	1695
Two	1489
Small	1392
1/2	1345
Grand	1088
1	1063
Glass	1050
per	805
Long	794
For	784
Jumbo	772
Large	770
Bowl	729
Little	719
Petits	675
Double	654
<other values>	16504
Total	388528

Relational Schema

After cleaning the dataset in the syntactic level and solving the violations, we further cleaned the dataset in the schema/integrity level, where we developed the relational schema of the dataset, identified the logical integrity constraints (ICs), used SQL queries to profile the dataset and check the data that violates the ICs, and finally solved these schema/integrity violations.

Schemas

Using the syntactically cleaned dataset generated in the last step, we first analyzed the database schema by interpreting the four tables in the original dataset into four separate schemas, and identified a primary key for each of them. Next, since the four tables are correlated, we identified three attributes that act as a primary key for one table but as a foreign key of the other, by which we created a logic structure of the dataset: the “Menu” table is the “parent table” for the “MenuPage” table, and both “MenuPage” table and “Dish” table are “parent tables” for the “MenuItem table”. The database schema is shown in figure 1:

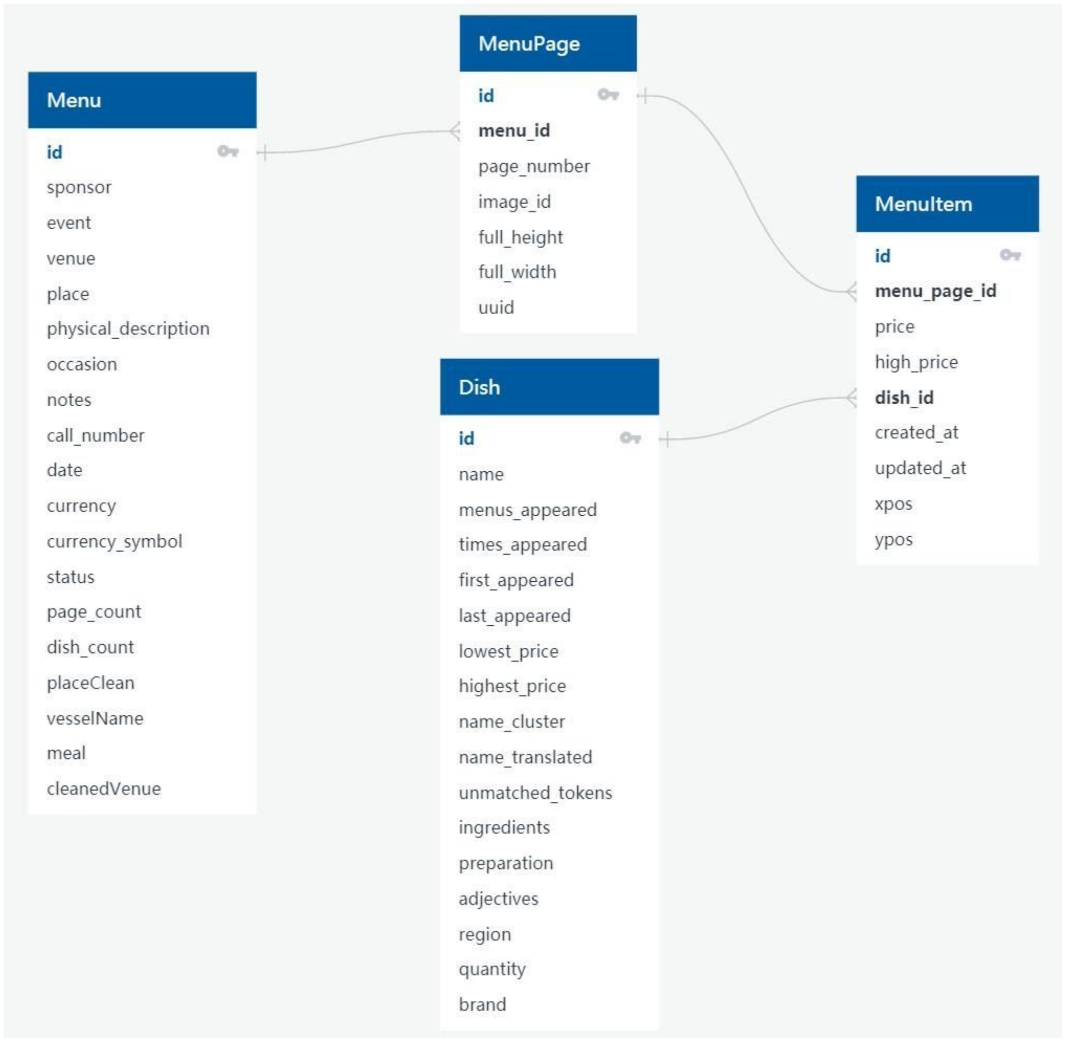


Figure 1: Database Schema

This database schema clearly demonstrates the overall structure of the dataset; however, it is not abstract enough for the specific logic relations, which are necessary for identifying the logical integrity constraints. Therefore, based on the database schema, we further developed the entity-relationship schema (E-R schema) of the dataset, which is shown in figure 2:

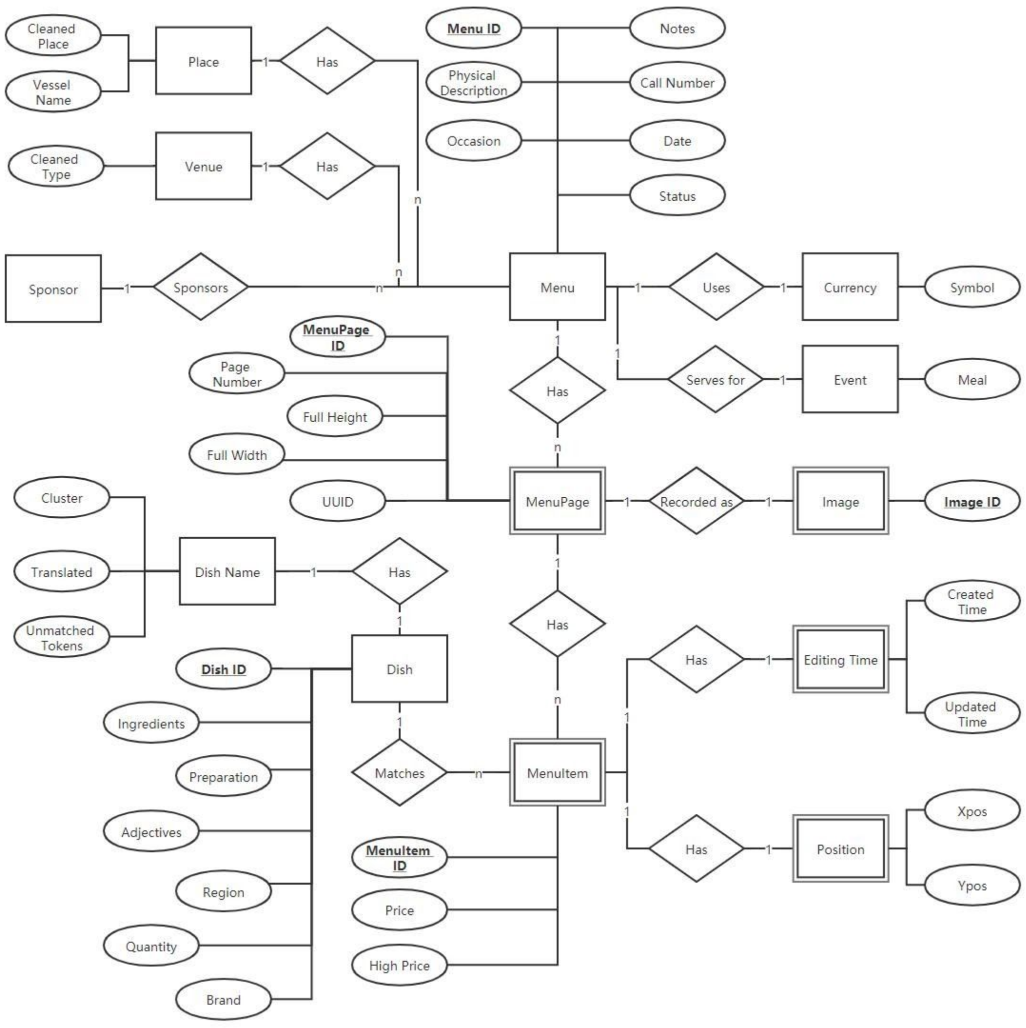
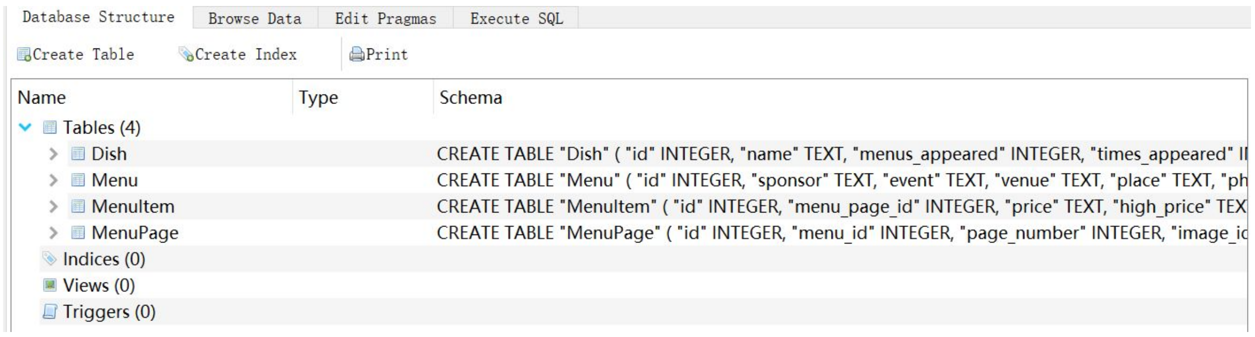


Figure 2: Entity-Relationship Schema

Here, the logic structure of the dataset in the attribute level is demonstrated. Comparing figure 2 to figure 1, we can see that, attributes in figure 1 including “menus_appeared”, “times_appeared”, “first_appeared”, “last_appeared”, “lowest_price”, “highest_price” in the “Dish” table and “page_count” and “dish_count” in the “Menu” table are not included in figure 2, since they are logically derived from other attributes in other tables, and are not independent attributes. These relationships consist of 8 logical integrity constraints across tables. Besides, there are also some logical integrity constraints within individual tables.

Logical Integrity Constraints (ICs)

In order to check the logical integrity constraints, we first loaded the data into a SQLite database based on the schema described above. We loaded the data using one of the most widely-used GUI for SQLite database, DB Browser for SQLite (DB4S)[1], in which we created a database, and imported four tables from csv. The database structure shown in DB4S is shown in figure 3:



Database Structure | Browse Data | Edit Pragmas | Execute SQL

Create Table | Create Index | Print

Name	Type	Schema
Tables (4)		
Dish		CREATE TABLE "Dish" ("id" INTEGER, "name" TEXT, "menus_appeared" INTEGER, "times_appeared" I
Menu		CREATE TABLE "Menu" ("id" INTEGER, "sponsor" TEXT, "event" TEXT, "venue" TEXT, "place" TEXT, "ph
MenuItem		CREATE TABLE "MenuItem" ("id" INTEGER, "menu_page_id" INTEGER, "price" TEXT, "high_price" TEX
MenuPage		CREATE TABLE "MenuPage" ("id" INTEGER, "menu_id" INTEGER, "page_number" INTEGER, "image_ic
Indices (0)		
Views (0)		
Triggers (0)		

Figure 3: Screenshot of the Database Schema in DB Browser for SQLite

Afterwards, we began to identify logical integrity constraints in this dataset, based on which we wrote SQL queries to identify the violations of these constraints, and ran the SQL queries in the "executing SQL" tab.

The integrity constraints and the corresponding SQL queries used to check them are listed as follows. First, we identified 8 logical integrity constraints that deal with the logical relationships across the tables:

1. The number of "menus_appeared" of each dish should be the same as the number of menus that contain this dish.

   SQL query:

   SELECT Dish.id, Dish.menus_appeared, COUNT (DISTINCT Menu.id) AS

   true_menus_count

   FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish

   WHERE Menu.id = MenuPage.menu_id AND MenuItem.menu_page_id = MenuPage.id

   AND Dish.id = MenuItem.dish_id

   GROUP BY Dish.id

   HAVING Dish.menus_appeared!=true_menus_count[2]

2. The number of "times_appeared" of each dish should be the same as the number of menu items that contain this dish.

   SQL query:

[1] https://sqlitebrowser.org.

[2] In queries 1, 2, 7 and 8, we used "DISTINCT" function to get the distinct number of menu, menu item, menu page, and dish.

SELECT Dish.id, Dish.times_appeared, COUNT (DISTINCT MenuItem.id) AS true_menuitem_count
FROM MenuItem JOIN Dish
WHERE Dish.id = MenuItem.dish_id
GROUP BY Dish.id
HAVING Dish.times_appeared!=true_menuitem_count

3. The "first_appeared" year of each dish should be the same as the year in the earliest "date" of the menus that contain this dish.
   <u>SQL query</u>:
   SELECT Dish.id, Dish.first_appeared, MIN(SUBSTR (Menu.date,1,4)) AS early_date
   FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish
   WHERE Menu.id = MenuPage.menu_id AND MenuItem.menu_page_id = MenuPage.id AND Dish.id = MenuItem.dish_id
   GROUP BY Dish.id
   HAVING Dish.first_appeared!=early_date[3]
4. The "last_appeared" year of each dish should be the same as the year in the latest "date" of the menus that contain this dish.
   <u>SQL query</u>:
   SELECT Dish.id, Dish.last_appeared, MAX(SUBSTR (Menu.date,1,4)) AS late_date
   FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish
   WHERE Menu.id = MenuPage.menu_id AND MenuItem.menu_page_id = MenuPage.id AND Dish.id = MenuItem.dish_id
   GROUP BY Dish.id
   HAVING Dish.last_appeared!=late_date
5. The "highest_price" of each dish should be the same as the maximum value of the "price" and "high price" of the menus that contain this dish.
   <u>SQL query</u>:
   SELECT Dish.id, Dish.highest_price,
   MAX(CAST((COALESCE(MenuItem.price,0)+COALESCE(MenuItem.high_price,0)+ABS (COALESCE(MenuItem.price,0)-COALESCE(MenuItem.high_price,0)))/2 AS REAL)) AS real_highest_price
   FROM MenuItem JOIN Dish

[3] In queries 3 and 4, we used "SUBSTR" function to get the "year" information in the "date".

WHERE Dish.id = MenuItem.dish_id

GROUP BY Dish.id

HAVING Dish.highest_price!=real_highest_price[4]

6. The "lowest_price" of each dish should be the same as the minimum value of the "price" of the menus that contain this dish.

   <u>SQL query</u>:

   SELECT Dish.id, Dish.lowest_price, MIN(CAST(MenuItem.price AS REAL)) AS real_lowest_price

   FROM MenuItem JOIN Dish

   WHERE Dish.id = MenuItem.dish_id

   GROUP BY Dish.id

   HAVING Dish.lowest_price!=real_lowest_price[5]

7. The "page_count" of each menu should be the same as the number of menu pages that are in this menu.

   <u>SQL query</u>:

   SELECT Menu.id, Menu.page_count, COUNT (DISTINCT MenuPage.id) AS true_page_count

   FROM Menu JOIN MenuPage

   WHERE Menu.id = MenuPage.menu_id

   GROUP BY Menu.id

   HAVING Menu.page_count!=true_page_count

8. The "dish_count" of each menu should be the same as the number of dishes that are contained in this menu.

   <u>SQL query</u>:

   SELECT Menu.id, Menu.dish_count, COUNT (DISTINCT Dish.id) AS true_dish_count

   FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish

   WHERE Menu.id = MenuPage.menu_id AND MenuItem.menu_page_id = MenuPage.id AND Dish.id = MenuItem.dish_id

   GROUP BY Menu.id

[4] In query 5, we used the "COALESCE" function to deal with the null values (the null values can never be the maximum price, and thus can be substituted with 0), which is necessary when selecting the maximum value from multiple columns. Besides, we used "CAST AS REAL" to convert the values to float numbers.

[5] In query 6, we also used "CAST AS REAL" to convert the values to float numbers, but "COALESCE" was no longer used, as we only need to deal with a single column here, since the "lowest price" should never be in the "high_price" column logically.

HAVING Menu.dish_count!=true_dish_count

In addition to the above 8 logical integrity constraints that deal with the logical relationships across the tables, we also identified 6 logical integrity constraints that deal with the logical relationships within individual tables:

1. In each menu, records with the same "currency" should also have the same "currency_symbol".

   <u>*SQL query*</u>:

   SELECT m1.id, m1.currency, m1.currency_symbol, m2.id, m2.currency, m2.currency_symbol

   FROM Menu m1, Menu m2

   WHERE m1.id<m2.id AND m1.currency=m2.currency AND m1.currency_symbol!=m2.currency_symbol

2. Menu pages that are in the same menu should have different "page number".

   <u>*SQL query*</u>:

   SELECT m1.id, m1.currency, m1.currency_symbol, m2.id, m2.currency, m2.currency_symbol

   FROM Menu m1, Menu m2

   WHERE m1.id<m2.id AND m1.currency=m2.currency AND m1.currency_symbol!=m2.currency_symbol

3. The "high_price" of each menu item should be higher than or equal to "price" of it.

   <u>*SQL query*</u>:

   SELECT id, price, high_price

   FROM MenuItem

   WHERE price-0>high_price-0

4. The "updated_at" time of each menu item should be no earlier than "created_at" time of it.

   <u>*SQL query*</u>:

   SELECT id, created_at, updated_at

   FROM MenuItem

   WHERE created_at>updated_at

5. The "xpos" position of each menu item should be in the range of [0, 1].

   <u>*SQL query*</u>:

   SELECT id, xpos

FROM MenuItem

WHERE xpos<0 OR xpos>1

6. The “ypos” position of each menu item should be in the range of [0, 1].

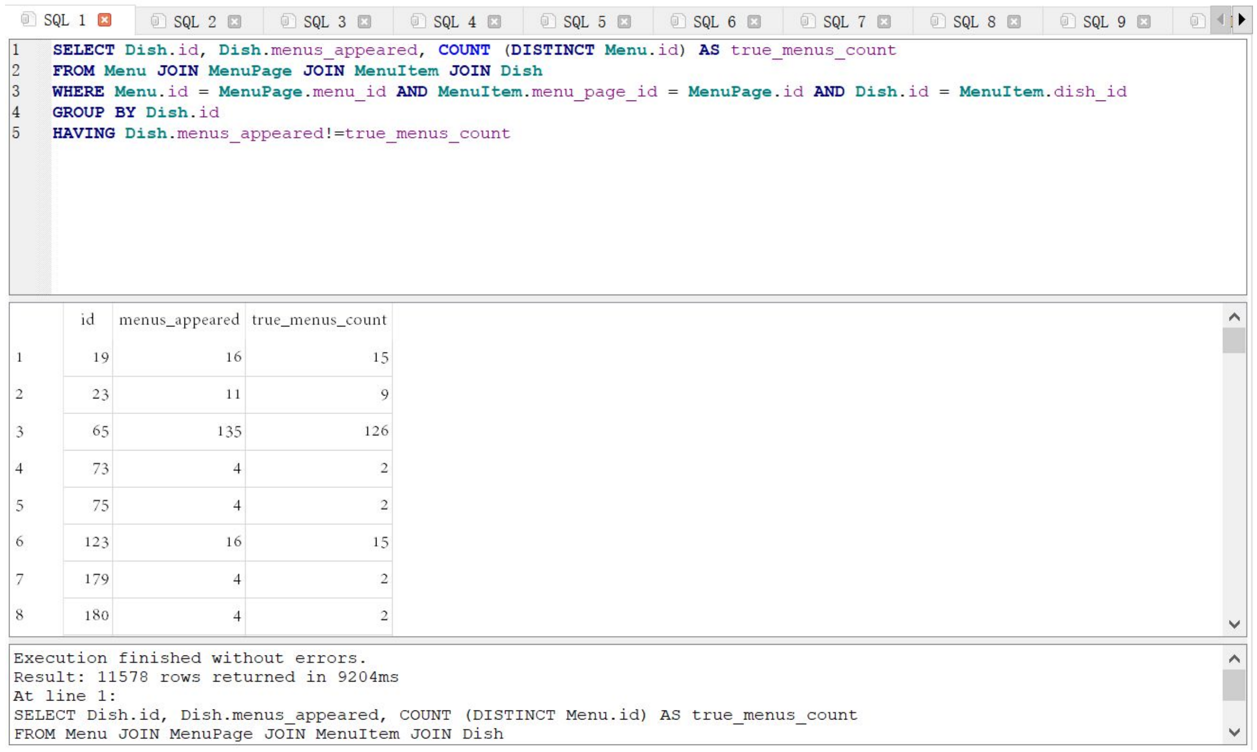
   <u>*SQL query*</u>:

   SELECT id, ypos

   FROM MenuItem

   WHERE ypos<0 OR ypos>1

An example query and the results of executing it in DB Browser for SQLite are shown in figure 4:



```
SELECT Dish.id, Dish.menus_appeared, COUNT (DISTINCT Menu.id) AS true_menus_count
FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish
WHERE Menu.id = MenuPage.menu_id AND MenuItem.menu_page_id = MenuPage.id AND Dish.id = MenuItem.dish_id
GROUP BY Dish.id
HAVING Dish.menus_appeared!=true_menus_count
```

	id	menus_appeared	true_menus_count
1	19	16	15
2	23	11	9
3	65	135	126
4	73	4	2
5	75	4	2
6	123	16	15
7	179	4	2
8	180	4	2

```
Execution finished without errors.
Result: 11578 rows returned in 9204ms
At line 1:
SELECT Dish.id, Dish.menus_appeared, COUNT (DISTINCT Menu.id) AS true_menus_count
FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish
```

Figure 4: Screenshot of the Results of an Example Query Before Solving the Violations

Violations and Solutions

The number of violations found in each of the above 8 logical integrity constraints that deal with the logical relationships across the tables and the 6 logical integrity constraints that deal with the logical relationships within individual tables are shown in table 1:

Integrity Constraint Type	Integrity Constraint ID	Number of violations	Number of total records
Across the tables	1-1	11578	423397

	1-2	135	423397
	1-3	2482	423397
	1-4	2381	423397
	1-5	109077	423397
	1-6	115492	423397
	1-7	0	17545
	1-8	5513	17545
Within individual tables	2-1	0	17545
	2-2	129	66937
	2-3	1274	582193
	2-4	2886	582193
	2-5	0	582193
	2-6	0	582193

Table 1: Number of Violations Found in Each Logical Integrity Constraints

Among the 14 logical integrity constraints, 4 of them do not contain any violations. Besides, the violations of the integrity constraint 2 within individual tables, menu pages that are in the same menu should have different “page number”, cannot be solved, since we do not have the information of what the “real” page number of these violations should be. For example, if two menu pages are both labelled as “page 15”, we do not know which should be changed, and what should it be changed to.

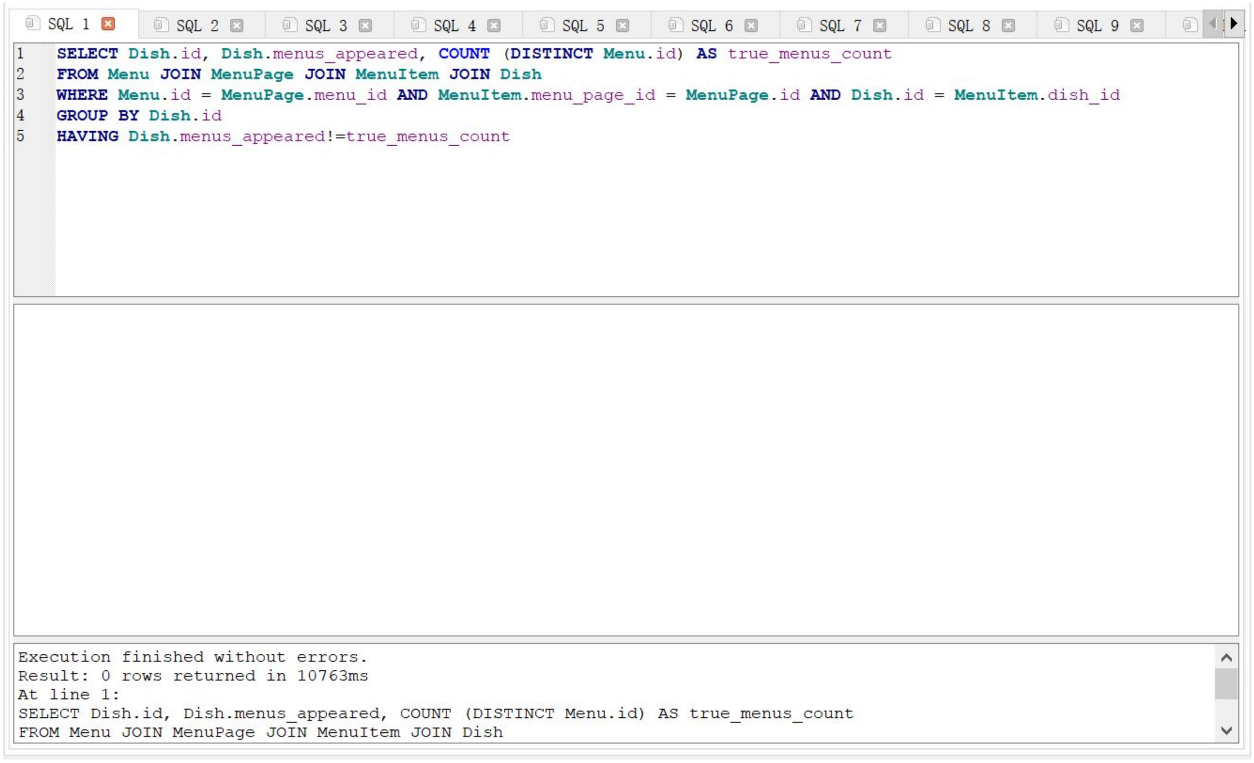
For the other 9 logical integrity constraints, we collected the violation data, and solved these problems. The ideas of the solutions are listed as follows:

1. Change the number of “menus_appeared” of each dish in violations of constraint 1-1 to be the same as the number of menus that contain this dish.
2. Change the number of “times_appeared” of each dish in violations of constraint 1-2 to be the same as the number of menu items that contain this dish.
3. Change the “first_appeared” year of each dish in violations of constraint 1-3 to be the same as the year in the earliest “date” of the menus that contain this dish.
4. Change the “last_appeared” year of each dish in violations of constraint 1-4 to be the same as the year in the latest “date” of the menus that contain this dish.
5. Change the “highest_price” of each dish in violations of constraint 1-5 to be the same as the maximum value of the “price” and “high price” of the menus that contain this dish.

6. Change the "lowest_price" of each dish in violations of constraint 1-6 to be the same as the minimum value of the "price" of the menus that contain this dish.
7. Change the "page_count" of each menu in violations of constraint 1-8 to be the same as the number of menu pages that are in this menu.
8. Delete the "high_price" of each menu item in violations of constraint 2-3.
9. Change the "updated_at" time of each menu item in violations of constraint 2-4 to the same as the "created_at" time of it.

After we have solved these violations according to the above 9 solutions, we re-ran the SQL queries to check the logical integrity constraints of the dataset. This time, all the problems are solved and no violation is detected (except for the violations of constraint 2-2, which, as noted above, is not solvable).

An example query and the results of executing it in DB Browser for SQLite after solving the violations are shown in figure 5.



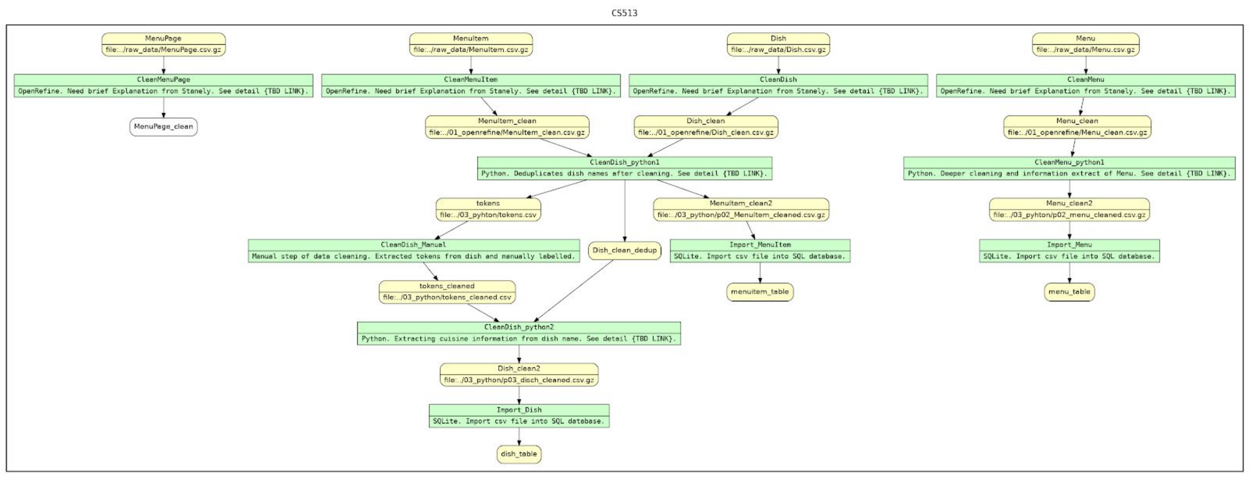
```sql
SELECT Dish.id, Dish.menus_appeared, COUNT (DISTINCT Menu.id) AS true_menus_count
FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish
WHERE Menu.id = MenuPage.menu_id AND MenuItem.menu_page_id = MenuPage.id AND Dish.id = MenuItem.dish_id
GROUP BY Dish.id
HAVING Dish.menus_appeared!=true_menus_count
```

```
Execution finished without errors.
Result: 0 rows returned in 10763ms
At line 1:
SELECT Dish.id, Dish.menus_appeared, COUNT (DISTINCT Menu.id) AS true_menus_count
FROM Menu JOIN MenuPage JOIN MenuItem JOIN Dish
```

Figure 5: Screenshot of the Results of an Example Query After Solving the Violations

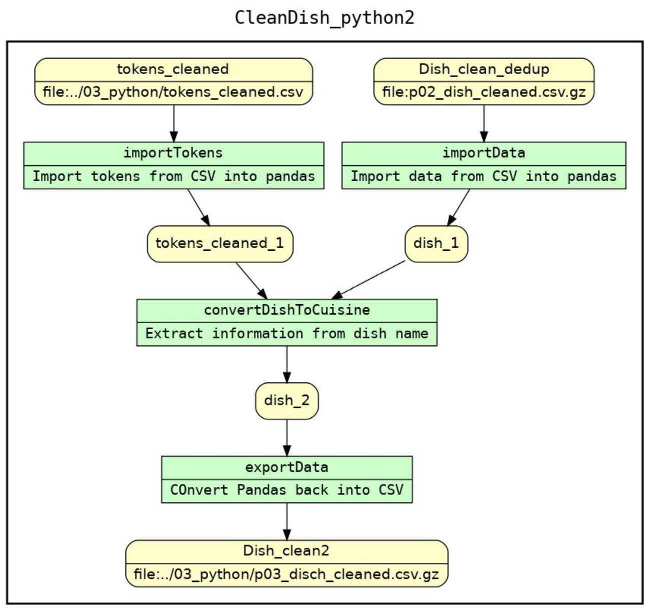
Workflow Model

The overall workflow is depicted here:


CS513
MenuPage
file:../raw_data/MenuPage.csv.gz
CleanMenuPage
OpenRefine. Need brief Explanation from Stanely. See detail {TBD LINK}.
MenuPage_clean
MenuItem
file:../raw_data/MenuItem.csv.gz
CleanMenuItem
OpenRefine. Need brief Explanation from Stanely. See detail {TBD LINK}.
MenuItem_clean
file:../01_openrefine/MenuItem_clean.csv.gz
Dish
file:../raw_data/Dish.csv.gz
CleanDish
OpenRefine. Need brief Explanation from Stanely. See detail {TBD LINK}.
Dish_clean
file:../01_openrefine/Dish_clean.csv.gz
CleanDish_python1
Python. Deduplicates dish names after cleaning. See detail {TBD LINK}.
tokens
file:../03_pyhton/tokens.csv
CleanDish_Manual
Manual step of data cleaning. Extracted tokens from dish and manually labelled.
Dish_clean_dedup
tokens_cleaned
file:../03_python/tokens_cleaned.csv
CleanDish_python2
Python. Extracting cuisine information from dish name. See detail {TBD LINK}.
Dish_clean2
file:../03_python/p03_disch_cleaned.csv.gz
Import_Dish
SQLite. Import csv file into SQL database.
dish_table
MenuItem_clean2
file:../03_python/p02_MenuItem_cleaned.csv.gz
Import_MenuItem
SQLite. Import csv file into SQL database.
menuitem_table
Menu
file:../raw_data/Menu.csv.gz
CleanMenu
OpenRefine. Need brief Explanation from Stanely. See detail {TBD LINK}.
Menu_clean
file:../01_openrefine/Menu_clean.csv.gz
CleanMenu_python1
Python. Deeper cleaning and information extract of Menu. See detail {TBD LINK}.
Menu_clean2
file:../03_pyhton/p02_menu_cleaned.csv.gz
Import_Menu
SQLite. Import csv file into SQL database.
menu_table

It is also available in the supplemental materials, 05_yesworkflow/master_workflow.png. This workflow depicts the highest-level building blocks of our workflow. The top most inputs are the raw data. Each of the steps is depicted identifying the high level tool used, and a general description of the tasks. First all four datasets were cleaned in OpenRefine. Next, there was some complex processing done in Python, in conjunction with manual labelling efforts. Finally, there was SQLite processing and integrity checks.

There is detail on the OpenRefine and Python steps available in the supplement materials as well. These are all contained in the folder 05_yesworkflows. Here is an example of the last python step, which also shows the ingestion of the manually created label corrections:


CleanDish_python2
tokens_cleaned
file:../03_python/tokens_cleaned.csv
Dish_clean_dedup
file:p02_dish_cleaned.csv.gz
importTokens
Import tokens from CSV into pandas
importData
Import data from CSV into pandas
tokens_cleaned_1
dish_1
convertDishToCuisine
Extract information from dish name
dish_2
exportData
COnvert Pandas back into CSV
Dish_clean2
file:../03_python/p03_disch_cleaned.csv.gz

Provenance

Based on the overall workflow and OpenRefine workflows obtained in the last step, we developed provenance queries to show the dependencies of different data and steps.

First, we generated the "port" information from each of our yesworkflow models, which is self-sufficient for provenance analysis since it provides both information of the edges and the type of the nodes in the workflow graph. In this step, a text file "overall.txt" of the port information of the overall workflow, as well as four text files "openrefine_*.txt" of the port information of the four OpenRefine workflows on each of the four tables.

Next, we wrote code to load the port information and parse it into dataframes. Since all edges in the workflow are directed, these four-column dataframes contain information about the start and end point of each edge, as well as the type (whether it is a "data" or a "step") of each of these start and end points. Afterwards, we parsed these dataframes into strings, which take the format of "edge(x, y)" and "data/type(x)", and are cleaned to remove the signals that are not acceptable to clingo. Then these statements were considered as "facts" and were passed to the Clingo function.

In order to delve into these facts and analyze the dependencies of different data and steps, we developed 9 provenance rules:

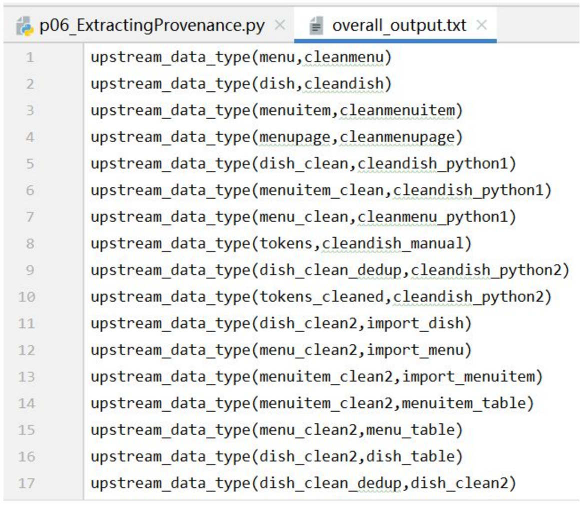
1. X is upstream of Y when X is the start point and Y is the end point in an edge, or X is the start point and Z is the end point in an edge, and Z is upstream of Y.
   Datalog query:
   upstream(X,Y) :- edge(X,Y).
   upstream(X,Y) :- edge(X,Z), upstream(Z,Y).
2. X is downstream of Y when X is the end point and Y is the start point in an edge, or Z is the end point and Y is the start point in an edge, and X is downstream of Z.
   Datalog query:
   downstream(X,Y) :- edge(Y,X).
   downstream(X,Y) :- edge(Y,Z), downstream(X,Z).
3. X is parallel to Y when Z is the start point and X is the end point in an edge, and Z is the start point and Y is the end point in an edge, and X is smaller than Y (to avoid duplicates).
   Datalog query:

parallel(X,Y) :- edge(Z,X), edge(Z,Y), X<Y.

4. X is data-type upstream of Y when X is upstream of Y and the type of X is data.
   Datalog query:
   upstream_data_type(X,Y) :- upstream(X,Y), data(X).
5. X is step-type upstream of Y when X is upstream of Y and the type of X is step.
   Datalog query:
   upstream_step_type(X,Y) :- upstream(X,Y), step(X).
6. X is data-type downstream of Y when X is downstream of Y and the type of X is data.
   Datalog query:
   downstream_data_type(X,Y) :- downstream(X,Y), data(X).
7. X is step-type downstream of Y when X is downstream of Y and the type of X is step.
   Datalog query:
   downstream_step_type(X,Y) :- downstream(X,Y), step(X).
8. X is parallel data of Y when X is parallel to Y and the type of both X and Y are data.
   Datalog query:
   parallel_data_type(X,Y) :- parallel_data_type(X,Y) :- parallel(X,Y), data(X), data(Y).
9. X is parallel step of Y when X is parallel to Y and the type of both X and Y are steps.
   Datalog query:
   Parallel_step _type(X,Y) :- parallel_step_type(X,Y) :- parallel(X,Y), step(X), step(Y).[6]

By adding facts and these rules into the Clingo server, we are able to obtain all dependencies of different data and steps in both the overall workflow and the OpenRefine workflows. We finally cleaned the output txt files and removed the facts and the results of the first three rules (since they are used to generate the last six rules), and re-arrange the output into different lines. The first 17 rows in the output file of the overall workflow is shown in figure 7.

[6] Note that the parallel data/step relationship requires the types of both X and Y are data/steps, but the data/step-type upstream/downstream relationship only requires the type of X is data/step (Y can be either data or step). This is because a data cannot be a parallel data of a step and vice versa, while a data can be an upstream/downstream of a step, and a step can be an upstream/downstream of a data.



```
upstream_data_type(menu,cleanmenu)
upstream_data_type(dish,cleandish)
upstream_data_type(menuitem,cleanmenuitem)
upstream_data_type(menupage,cleanmenupage)
upstream_data_type(dish_clean,cleandish_python1)
upstream_data_type(menuitem_clean,cleandish_python1)
upstream_data_type(menu_clean,cleanmenu_python1)
upstream_data_type(tokens,cleandish_manual)
upstream_data_type(dish_clean_dedup,cleandish_python2)
upstream_data_type(tokens_cleaned,cleandish_python2)
upstream_data_type(dish_clean2,import_dish)
upstream_data_type(menu_clean2,import_menu)
upstream_data_type(menuitem_clean2,import_menuitem)
upstream_data_type(menuitem_clean2,menuitem_table)
upstream_data_type(menu_clean2,menu_table)
upstream_data_type(dish_clean2,dish_table)
upstream_data_type(dish_clean_dedup,dish_clean2)
```

Figure 7: Screenshot of the Results of the Sample Query in the Overall Workflow

It is very easy to manipulate the output and only keep the required relationships. For example, if we want to investigate the parallel data, we can eliminate all the other outputs and keep the "parallel_data_type" results only.

Alternatively, we can directly query the results we want, which is especially useful when we want to investigate the dependencies of a specific data or step. We provided some sample queries to show the dependencies in related to the data "tokens" in the overall workflow:

Datalog query:

upstream_data_type_tokens(X) :- upstream_data_type(X, tokens).

upstream_step_type_tokens(X) :- upstream_step_type(X, tokens).

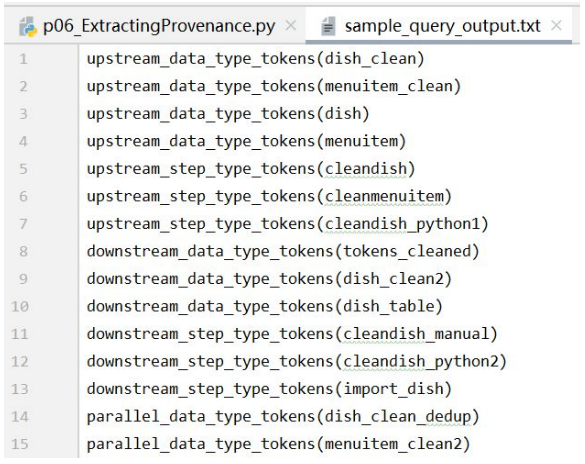
downstream_data_type_tokens(X) :- downstream_data_type(X, tokens).

downstream_step_type_tokens(X) :- downstream_step_type(X, tokens).

parallel_data_type_tokens(X) :- parallel_data_type(X, tokens).

parallel_data_type_tokens(X) :- parallel_data_type(tokens, X).

The results of these queries are shown in figure 7: this data depends on 4 data, 3 steps, and there are 3 data and 3 steps depend on it. Besides, it has 2 parallel data derived from the same step.


p06_ExtractingProvenance.py × sample_query_output.txt ×

```
upstream_data_type_tokens(dish_clean)
upstream_data_type_tokens(menuitem_clean)
upstream_data_type_tokens(dish)
upstream_data_type_tokens(menuitem)
upstream_step_type_tokens(cleandish)
upstream_step_type_tokens(cleanmenuitem)
upstream_step_type_tokens(cleandish_python1)
downstream_data_type_tokens(tokens_cleaned)
downstream_data_type_tokens(dish_clean2)
downstream_data_type_tokens(dish_table)
downstream_step_type_tokens(cleandish_manual)
downstream_step_type_tokens(cleandish_python2)
downstream_step_type_tokens(import_dish)
parallel_data_type_tokens(dish_clean_dedup)
parallel_data_type_tokens(menuitem_clean2)
```

Figure 7: Screenshot of the Results of the Sample Query in the Overall Workflow

Conclusion

We set out to clean the dataset for the use case of building an application around searching for favorite foods and restaurants in New York City. Were we successful?

- We proposed and partially implemented a strategy to clean the restaurant location in order to make searching both locally and globally feasible. In addition our strategy also allows for expanded usefulness for those folks frequently traveling on steamships and trains.
- We proposed and partially implemented a strategy to decompose the "dish name" into a structure that will stand the test of time. We developed an ontology of how foods should be parsed and this will give our application developers easy access to merge with modern culinary databases, or to search for ingredients (like cheese!) and preparation methods.

Things that still could be done:

- Due to constraints on time and diminishing returns, we did not 100% clean important fields like "dish name" or "menu place". This would be a first next step for a true commercial application.
- Meeting with culinary experts to help us further refine the grammar of dish names.
- integrating with likely with other culinary databases for caloric and nutritional information.

In hindsight, we would have approached the project a little differently:

1) Many of our individual data steps had to be manually woven together, either via export-import steps or by shell scripts. This is very error prone and it would hae been better to capture the workflow in a single tool to broker all of this together.
2) We waited until the end to incorporate "Yesworkflow" and "provenance" queries. Both of these would have been useful to have in a data-pipeline for real time debugging and support.
3) We didn't leverage machine learning or NLP tools to help automatically parse fields like "dish name." While we are comfortable that our approach is repeatable and useful, there would have been potentially less labor involved had we utilized modern AI techniques.

Despite this, we all-in-all did a thorough job of cleaning most of the data, and provided a good framework for future application developers to leverage this dataset for purpose.

Supplemental Materials

This section outlines important artifacts available in the “dropbox” link. The “zip” archive subsection contains the information called out per the rubric specifically.

github https://github.com/rdbisch/BLS_Data_Janitors
dropbox https://www.dropbox.com/sh/w36btljjgjk8v5e/AACShbDvzauP_4-jcNinrY_Ba?dl=0

ZIP Archive

- Operation History -- the rubric asked for a single JSON file, but this was not possible for us as we cleaned all 4 datasets across multiple tools. The operation history is therefore offered as distinct files

- Queries -- again, we had many tables to check and it would be potentially confusing to put these all in a single file.
    - 04_sql/Queries.txt
    - 04_sql/s04_ResolveViolations.ipnyb

- Workflow Model -- We provide a manually created overall workflow model and gv files. Note the generated graphics are also available in the larger archive.

Outline of Github/DropBox - The entire archive of our project is quite large so we’ve made an index of all important files. The files that were copied into the supplemental zip archive are highlighted in orange for easy reference.

- Raw Data -- This was the class reference files for the NYPL dataset. We include a copy here for reference.
    - raw_data/Dish.csv.gz
    - raw_data/Menu.csv.gz
    - raw_data/MenuItem.csv.gz
    - raw_data/MenuPage.csv.gz
- OpenRefine Data Cleaning Outputs
    - OpenRefine JSON workflows

        - 01_openrefine/Dish.json
        - 01_openrefine/Menu.json
        - 01_openrefine/MenuItem.json
        - 01_openrefine/MenuPage.json
    - Intermediate Data Products
        - 01_openrefine/Dish_clean.csv.gz
        - 01_openrefine/Menu_clean.csv.gz
        - 01_openrefine/MenuItem_clean.csv.gz
        - 01_openrefine/MenuPage_clean.csv.gz
- Python Data Cleaning Outputs
    - Python scripts for cleaning
        - 03_python/f04_cleanDish.py
        - 03_python/f04_cleanMenu.py
        - 03_python/f05_cleanDish.py
        - 03_python/helper.py
    - Intermediate Data Outputs
        - 03_python/tokens.csv
        - 03_python/tokens_cleaned.csv
        - 03_python/p02_dish_cleaned.csv.gz
        - 03_python/02_menu_cleaned.csv.gz
        - 03_python/02_MenuItem_cleaned.csv.gz
- SQL Data Cleaning and Validation Outputs
    - Scripts
        - 04_Sql/Queries.txt
        - 04_sql/s04_ResolveViolations.ipnyb
    - Intermediate Output Data
        - 04_sql/violated_data/SQL*csv
    - Final Output Data
        - 04_sql/s04_dish_cleaned.csv.gz
        - 04_sql/menu_cleaned.csv.gz
        - 04_sql/menuitem_cleaned.csv.gz

- YesWorkflow Artifacts
    - Overall Master Workflow
        - 05_yesworkflow/Overall_workflow.py(manually generated)
    - OpenRefine and Python YesWorkflows - Other than one file, all of the sub-workflows are generated directly from the OpenRefine JSON or the python source files. There is a script that processes all of these files and creates the corresponding “yw”, “gz” and “png” outputs.
        - 05_yesworkflow/f04_cleanMenu_yw.py (simplified version of 03_python/f04_cleanMenu.py to get around yw bugs)
        - Scripts
            - createGraphs.sh - Compiles YW files from each of the sources.
        - Derived Files - Created by “createGraphs.sh”
            - Overall_Workflow.yw Overall_Workflow.gv Overall_Workflow.png
            - f04_cleanDish.yw f04_cleanDish.gv f04_cleanDish.png
            - f04_cleanMenu.yw f04_cleanMenu.gv f04_cleanMenu.png
            - f05_cleanDish.yw f05_cleanDish.gv f05_cleanDish.png
            - or_dish.yw or_dish.gv or_dish.png
            - or_menu.yw or_menu.gv or_menu.png
            - or_menuitem.yw or_menuitem.gv or_menuitem.png
            - or_menupage.yw or_menupage.gv or_menupage.png
- Provenance Queries
    - Facts
        - 06_provenance/overall.txt
        - 06_provenance/openrefine_cleandish.txt
        - 06_provenance/openrefine_cleanmenu.txt
        - 06_provenance/openrefine_cleanmenupage.txt
        - 06_provenance/openrefine_cleanmenuitem.txt
    - Output
        - 06_provenance/overall_output.txt
        - 06_provenance/cleandish_output.txt
        - 06_provenance/cleanmenu_output.txt
        - 06_provenance/cleanmenuitem_output.txt

        - 06_provenance/cleanmenupage_output.txt
    - Python/CLingo scripts
        - 06_provenance/p06_ExtractingProvenance.py
        - 06_provenance/provenance rules and queries.txt

Project Work

While this was a group project and we all worked together, the rubric requested documentation for primary authorship of who did what. The high-level assignments were as follows:

01_openrefine/	Stanley Lim
Assessment of Dataset for use case	Stanley Lim
03_python/	Rick Bischoff
04_sql/	Wenyi Shang
05_yesworkflow/	Rick Bischoff
06_provenance/	Wenyi Shang