CS 513 Group Project

Part 1

Team ID: 37

Team Name: PADS

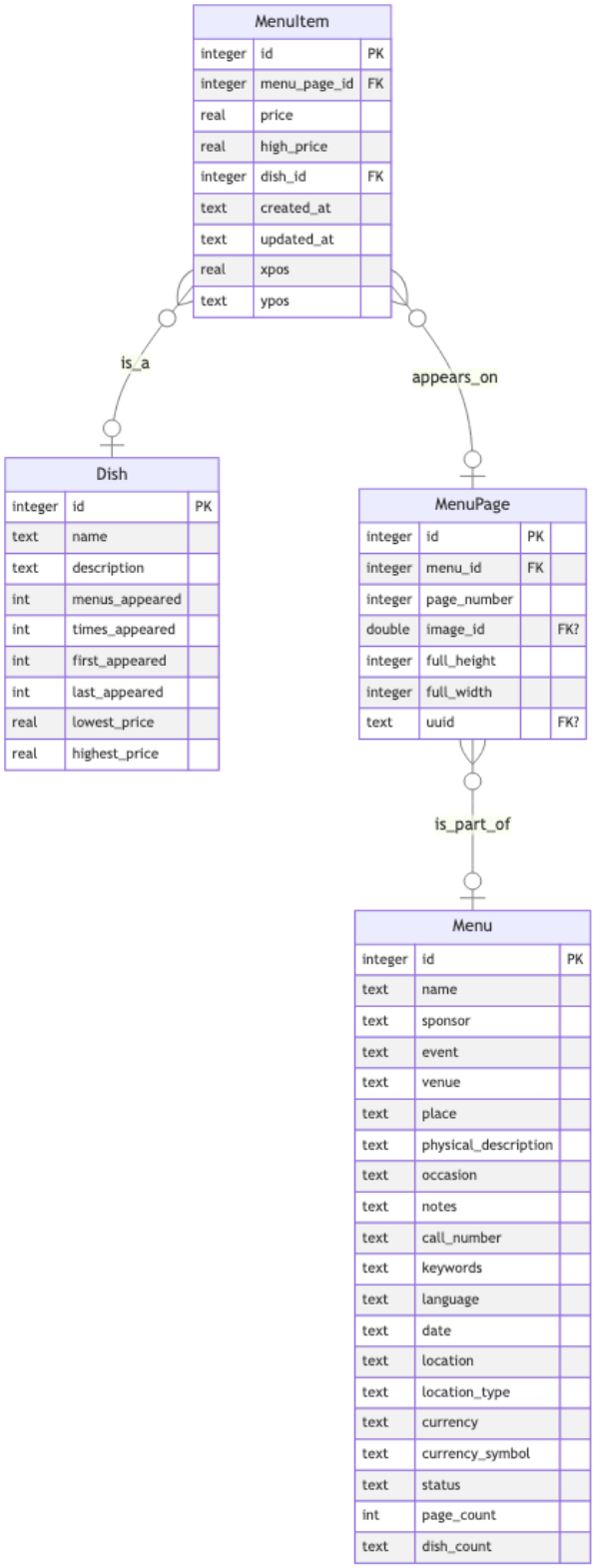
Aaron Schlesinger and David Pankros

{aschle2, pankros2}@illinois.edu

Description of Dataset (25 points)

Note: Throughout this and all subsequent sections, we'll use database-style terminology to refer to the data. Generally speaking, "tables" or "collections" will refer to individual CSV files (e.g. `Menu.csv`, `MenuItem.csv`, etc...), "rows" will refer to individual rows within a given CSV file, and "columns" will refer to an individual column within a given CSV file.

Overall, the dataset appears to be data captured from digitized menus which was subsequently structured into the menu, pages, menu items in pages and then the dishes on the pages. Each will be described in more detail, below.


MenuItem
integer id PK
integer menu_page_id FK
real price
real high_price
integer dish_id FK
text created_at
text updated_at
real xpos
text ypos
is_a
appears_on
Dish
integer id PK
text name
text description
int menus_appeared
int times_appeared
int first_appeared
int last_appeared
real lowest_price
real highest_price
MenuPage
integer id PK
integer menu_id FK
integer page_number
double image_id FK?
integer full_height
integer full_width
text uuid FK?
is_part_of
Menu
integer id PK
text name
text sponsor
text event
text venue
text place
text physical_description
text occasion
text notes
text call_number
text keywords
text language
text date
text location
text location_type
text currency
text currency_symbol
text status
int page_count
text dish_count

ER Diagram

`Dish` The `Dish` table contains information about food dishes, the number of menus in which they appear, a high and low price for each `Dish`, and a first and last appearance year. Thus, "Bread" could appear as a row in `Dish` and it may be referenced by several `Menus` as `MenuItems`.

`Menu` Conceptually, `Menu` represents one physical menu that is provided to a guest at a restaurant or an event. The `Menu` table is relatively more complex and contains a collection of the following columns:

- sponsors,
- events,
- venue types,
- venue name and general location,
- a description of the physical menu,
- the occasion for the meal,
- notes about the physical menu,
- a "call number,"
- keywords and language (both blank),
- the date of the menu,
- a location type (unused),
- the currency on the menu (sparely used),
- a currency symbol (used when currency is designated),
- a status field (presumably for an internal workflow),
- a menu page count, and
- the number of dishes on the `Menu` row

`MenuPage` The `MenuPage` table is a collection of information about individual physical pages for a `Menu`. It contains a reference to a `menu_id` and a `page_number`.

The `image_id` column presumably references an image in another table, but there is no provided table that it references.

The `full_height` and `full_width` fields seem to indicate an image size in pixels, though some appear incorrect or are altogether missing (such as row id 329).

There is a `uuid` field that stores many distinct values but lacks a consistent format (some are upper case, while most are lower) and there is no indication what it references. It could be a primary key to some other system, but the actual use of this column is a mystery.

`MenuItem` The `MenuItem` table is a collection of what must be assumed to be individual blocks of text that map to `Dish`es that appear on a `MenuPage`. `MenuItem`s include the following columns:

- A `price` column indicating the price of the `Dish` on the `Menu`.
- A `high_price` column which is presumably the highest price among a collection of `MenuItem`s, but the domain of such a set is unclear.
- A `dish_id` column referencing a row in the `Dish` table (multiple `MenuItem` rows may reference the same `Dish` row for comparison purposes).
- A `created_at` column indicating when a `MenuItem` row was created, and an `updated_at` column indicating when it was updated (or, more accurately, when the database row was created and updated).
- `xpos` and `ypos` column, presumably representing the location relative to the overall width and height of the `Dish` on the `MenuPage`. All `xpos` values are in the domain `[0,1)` and `ypos` values are in the domain `[0,1]`. This domain was verified using SQL query `SELECT min(xpos), max(xpos), min(ypos), max(ypos) FROM MenuItem;`).

Use Cases (30 points)

Use case U1 - Target (Main) use case You'd likely be able to use standard data cleaning techniques to produce a dataset suitable for at least the following applications:

- Non-production-critical online applications like reference systems that allow experienced, trained users to search for and view `Menu`s and `Dish`es. Such a system could be used by researchers, historians, or other professionals who need to efficiently access the data in a structured format.
- Data mining applications like analytics systems concerning the New York City restaurant industry. Data mining applications and reference systems like those mentioned above might share some common functionality.
- Unsupervised machine learning applications, which would likely be most valuable for pattern recognition and possible generative AI use cases. These machine learning applications might share some common functionality with data mining applications.
- There may also be semantically consistent subsets of the data that may be able to be cleaned and used for some applications beyond those mentioned above. For example, a subset of `Menu` data after a certain date or belonging to a certain geographical region may be consistent, and therefore usable, whereas other data that is inconsistent would require highly error-prone manual cleaning. We expect to be able to determine if such subsets exist as we proceed with data cleaning.

Use case U0 - “Zero data cleaning” Since these raw data contain at least several inconsistencies we consider to be severe, we believe they are unsuitable for use in most user-facing applications. Further, we don't believe that data mining applications over the raw data would produce reliable results.

We believe that only some unsupervised learning applications would be suitable for the raw, un-cleaned data. The accuracy of results generated from these applications should not be directly relied upon for critical applications, however. For example, an unsupervised learning application that can identify `MenuItem` trends, clusters, or relative pricing over time could be appropriate, but an application that directly used those results to forecast revenue for an active restaurant business would likely not be appropriate.

Use case U2 - "Never Enough" We believe there are many applications for which even data cleaning would not be sufficient, including the following:

- Any application that involves knowing true prices. Our pricing information is semantically incomplete so we can *guess* but never *know* whether prices are in comparable units.
- Any application that requires knowing the exact location of a `Dish` on a `MenuPage`. The `xpos` and `ypos` columns, discussed elsewhere in this document, are not well-defined and would require additional information to be useful.
- Any application that relies on strong referential integrity, like lookup or reference systems that rely on foreign keys. Our data has missing foreign keys, foreign keys referencing non-existent rows, and even foreign keys that reference non-existent tables.

Generally speaking, any application for which data accuracy and consistency is critical, especially applications used to ensure safety, drive revenue, or perform other critical tasks, should not be built on top of these data. We anticipate many, but not all, of these applications will primarily be user-facing.

Data Quality Problems (30 points)

Generally speaking, the schema for these data is largely denormalized, which reduces or eliminates the opportunity to establish referential integrity in many cases. In cases where a table appears to have a column *intended to be* a foreign key, not all values in that column are valid (i.e. do not reference a valid row in the target table, or appear to reference a table that was not provided).

These reductions or limitations can in turn can reduce the overall quality of the data.

We believe it's important to note, however, that many front-end applications could benefit from this denormalized arrangement, particularly for performance reasons performance (i.e. a denormalized, read-only view of the data).

We detail specific data quality problems, including examples, for each table below.

`Dish` The schema and supplementary information for this table is as follows:

Table	Column	Type	Null Count	Domain
Dish	id	int	0	
Dish	name	text	0	
Dish	description	text	423397	
Dish	menus_appeared	int	0	[0,7800]
Dish	times_appeared	int	0	[-100,8500]
Dish	first_appeared	int	0	[0,3000]
Dish	last_appeared	int	0	[0,3000]
Dish	lowest_price	float	29100	[0,1100]
Dish	highest_price	float	29100	[0,3100]
Menu	id	int	0	

This table has several data quality problems, including the following:

- The `description` column is wholly unused -- its values are all missing.
- The `menus_appeared` column contains values from 0 to 7800. We assume this column indicates the number of `Menu` rows in which the applicable `Dish` appears, and many rows (including rows with IDs `88446`, `132992`, `164029`, and more) have `0` in this column, indicating there are 0 menus in which the `Dish` appeared. The fact that rows in this collection indicate there are dishes that don't appear in any `Menu` strongly suggests that those rows are incorrect. This inconsistency calls into question the accuracy of all the data stored in this column.
- The `times_appeared` column contains negative numbers, which likely have no meaning in this context, calling into question the accuracy of data in this column. Examples of rows with negative numbers include IDs `445236`, `457504`, `510718`, and `223497`.
- The `first_appeared` and `last_appeared` columns seem to represent the year at which the `Dish` first and last appeared on a `Menu`. Several `Dish`es contain `0` in one or both of these columns, and several others contain values into the future. Examples of both cases are shown below. The former case indicates a `Dish` first or last appeared on a `Menu` over 2000 years ago, and the latter case indicates a `Dish` that will appear on a `Menu` in the future. Both cases are obviously incorrect.
  - Several `Dish`es contain `0` in one or both of these columns, including IDs `340`, `2005`, and `2056`; and

  - Further, several `Dishes` contain values in the future, including IDs `415521`, `415523`, `415525`, and `415526`.

- The `lowest_price` and `highest_price` columns presumably contain currency values, but some values are missing, and no units are referenced by the table. Values in these columns have several issues, calling into question the usability of both of these columns:
  - Many rows are missing values for one or both of these columns, including `Dishes` with IDs `34`, `39`, and `60`;
  - Other rows have `0` for one or both of these columns, including `Dishes` with IDs `933`, `934`, and `936`; and
  - No currency is listed for any values in either of these columns, calling into question both the usability of these values as-is, and how this column would resolve values expressed in different currencies.

Menu The schema and supplementary information for this table is as follows:

Table	Column	Type	Null Count	Domain
Menu	id	int	0	
Menu	name	text	14348	
Menu	sponsor	text	1561	
Menu	event	text	9391	
Menu	venue	text	9414	
Menu	place	text	9422	
Menu	physical_description	text	2777	
Menu	occasion	text	13742	
Menu	notes	text	6932	
Menu	call_number	text	1562	
Menu	keywords	text	17545	
Menu	language	text	17545	
Menu	date	date	586	
Menu	location	text	0	
Menu	location_type	text	17545	

Table	Column	Type	Null Count	Domain
Menu	currency	text	11089	
Menu	currency_symbol	text	11089	
Menu	status	text	0	complete,under review
Menu	page_count	int	0	[1,75]
Menu	dish_count	int	0	[0,4100]

This table contains several significant data quality problems that call into question the integrity and accuracy of these data. These problems include:

- The `name`, `occasion` and `location_type` columns are frequently blank or null, including in row IDs `12463`, `12465`, and `12468`. Further, some menus appear to be aggregates of multiple menus, such as in row IDs `31054` and `31230`. These aggregate representations include descriptions such as `"62 menus bound into 1 volume"`. Such semantic inconsistencies will be difficult to remove from the data by automated methods.
- The `physical_description` and `occasion` columns are really separate collections of "tags" that describe the `Menu`, and these data would thus be better stored as a separate table. Formats of these columns vary: some are blank, some are non-empty but not delimited, some are non-empty but delimited, and some are non-empty and encapsulated in hard-brackets (and seemingly limited to 8 characters). Some examples of these inconsistencies are listed below:
  - Row ID `12465`: `occasion` is empty;
  - Row ID `12836`: `occasion` is non-empty and non-delimited (`COMPLIMENTARY/TESTIMONIAL`);
  - Row ID `31575`: `physical_description` is non-empty and delimited (`30x22.5cm folded; 30x45cm open`); and
  - Row ID `13926`: `occasion` is non-empty and encapsulated in brackets (`[COMPLIMENTARY TO COMMODORE C. WE. BLISS]`).
- 5 `Menu` rows indicate a different number of pages than are actually stored in the referenced `MenuPage` table. These rows have IDs `26577`, `26739`, `32086`, `32663`, and `33648`.
  - Assuming all of the CSV files are loaded into SQLite, the following query can be used to fetch all 5 rows adhering to these criteria:

```
select M.id, count(MP.id), M.page_count from Menu M left join
↪  main.MenuPage MP on M.id = MP.menu_id group by M.id having
↪  count(MP.id) <> M.page_count;
```

- 217 Menu rows differ in the number of Dishes stored versus the number of Dishes referenced in the dish_count column. Rows with these inconsistencies include IDs 12474, 12498, 12611, and more. In each of these instances, only the first page number is stored.
  - Assuming all of the CSV files are loaded into SQLite, the following query can be used to fetch all 217 rows adhering to these criteria:

```
select M.id, MP.page_number, M.page_count, count(D.id) as 'Dishes
↪  in DB', M.dish_count from Menu M left join main.MenuPage MP on
↪  M.id = MP.menu_id left join main.MenuItem MI on MP.id =
↪  MI.menu_page_id left join main.Dish D on MI.dish_id = D.id
↪  group by M.id having count(D.id) <> M.dish_count order by M.id
↪  asc, MP.page_number asc;
```

MenuPage The schema and supplementary information for this table is as follows:

Table	Column	Type	Null Count	Domain
MenuPage	id	int	0	
MenuPage	menu_id	int	0	
MenuPage	page_number	int	1202	[1,75]
MenuPage	image_id	int	0	
MenuPage	full_height	int	329	[0,13000]
MenuPage	full_width	int	329	[500,9200]
MenuPage	uuid	text	0	

This table contains the following data quality issues:

- The page_number column contains blank values. It is unknown if a blank should be considered the first page, such as in a one-page menu or if it is simply erroneous. Row IDs with empty page_number values include 44543, 44544, and 44545.
- Some Menus seemingly contain hundreds of dishes, because Menus are not used in a semantically consistent way. That is, some Menus are individual menus and others are collections. For those that are collections, page_number may reference the page_number within the collection of menus, even though each physical menu within the collection is only one page, for example. For example, there are 176 MenuItem rows that belong to the MenuPage with ID 168.

- The `image_id` column appears to be a foreign key to an unprovided table of images or references to images, but many errors occur when treating it as such because while predominantly integers are present, 23 outlier values such as `psnypl_rbk_936` (in row ID `45598`), which are nonsensical, also appear in that column.
- The `full_height` and `full_width` columns likely represent the width in pixels of the referenced image. One would expect the pages of a `Menu` to have the same proportions, but some rows with identical `menu_ids` and different `page_numbers` have transposed `full_height` and `full_width` values indicating that one (or more) page(s) is/are, possibly, rotated. Row IDs `119` and `120` hold an example of this transposition.
- There is no indication of the use for the `uuid` column and the column name provides no clues. Their format generally indicates they are valid UUIDs, and a quick "spot check" indicates many of them are parseable as such, but we haven't yet tried to parse all values.
- 5789 rows in the `MenuPage` table are missing their referenced `Menus`. Row IDs with missing `Menus` include `119`, `120`, `952`, `1022`, and more. Assuming all CSVs are loaded into SQLite, the following query can be used to return all rows adhering to these criteria:

```
select * from MenuPage MP
left outer join main.Menu M on M.id = MP.menu_id
where M.id is NULL;
```

`MenuItem` The schema and supplementary information for this table is as follows:

Table	Column	Type	Null Count	Domain
MenuItem	id	int	0	
MenuItem	menu_page_id	int	0	
MenuItem	price	float	445916	[0,190000]
MenuItem	high_price	float	1240821	[0,7900]
MenuItem	dish_id	int	241	
MenuItem	created_at	datetime	0	
MenuItem	updated_at	datetime	0	
MenuItem	xpos	float	0	[0,1)
MenuItem	ypos	float	0	[0,1]

This table has the following quality issues:

- The `price` column appears to have a similar purpose as other collections, but like in those other collections, it does not indicate a currency and it's values vary wildly from 0 to 190,000, and include empty values.
  - Because the `price` column does not indicate units, you could presumably join `MenuItem` to `MenuPage` and then to `Menu` to (sometimes) discern the price *and* the currency of the price.
  - Row IDs with empty `price` columns include the following: `15`, `31`, `48`, and more.
  - Row IDs with the value `0` in the `price` column include the following: `13009`, `24903`, and `31144`.
  - Row IDs with values above `100,000` in the `price` column, along with their values include the following:
    * `485274`: `110000.0`
    * `485279`: `180000.0`
    * `485280`: `180000.0`
    * `485281`: `160000.0`
- The `high_price` column is a bigger mystery in this table. In many rows, its value is completely missing, and when it's present, we can currently only guess its purpose.
  - `1240821` rows in the `MenuItem` have missing `high_price` values, while the remaining `91905` rows do not. Non-blank `high_price` values are in the range `[0, 7800.0]`.
- The `dish_id` column contains 241 blank values. Row IDs in which `dish_id` are blank include the following: `19171`, `22322`, `31566`, and more.
- The `created_at` and `updated_at` columns seem to be fairly-standard database creation/update timestamps and have no observable issues.
- The `xpos` and `ypos` columns have semantic issues making it impossible to understand their use without more information. These issues include the following:
  - These columns appear to represent coordinates, but we can only guess to what;
  - There is no indication whether the origin they represent is a center point, a corner, or something else; and
  - Finally, we also are unclear why there is no width and height to define a bounding box.
- 35 `MenuItems` are missing their referenced `MenuPage`, via the `menu_page_id` column.
- 244 `MenuItems` are missing their referenced `Dish`, via the `dish_id` column.

Supporting use case U1

We assume the "main use case" from U1 is non-production-critical online systems (rather than data mining/analytics or unsupervised learning applications), such as reference systems.

If we were to build a reference system over the raw data, the many inconsistencies and other issues with the data would produce results upon which we could not rely for any reasonable application, even if a single result in question were accurate. This is particularly true for semantic issues where no straight-forward data manipulation will provide reliable data, other than manual human review.

We must clean the data to reduce the number and severity of these inconsistencies enough to be able to rely upon at least the majority of the results fetched from the reference system.

Initial Plan for Phase-II (15 points)

As we document our steps for Phase II, we will include responsibilities for each part, but please understand that this team has completed significant projects in the past. All its members have a proven track record of successfully dividing up tasks in a fair and amicable fashion, without an extensive prior project plan.

Further, both team members are comfortable with and have been successful working with agile methodologies. All task assignments will be subject to change based on what we discover as we proceed through each step, our development of the Python client (discussed below in steps 3 and 5), and our own personal and professional workloads.

Step 1: Review (and update if necessary) your use case description and dataset description We expect to discover more features of (and issues with) this dataset, and thus additional use cases, as we proceed with data cleaning (DC). Thus, we'll review and update our use cases and dataset descriptions iteratively and as necessary. This work will initially be done by Aaron.

For example, as we proceed with our DC work, we expect we may gain more context as to what, exactly, the various `xpos` and `ypos` values represent in the `MenuItem` table. If this happens, we will update our dataset description accordingly.

Step 2: Profile D to identify DQ problems: How do you plan to do it? What tools are you going to use? We believe we currently know enough about the DQ problems in this dataset to proceed with DC, but we'll refer back to our DQ research process as we proceed. To start the DC process, we plan to use OpenRefine to isolate and fix the known "low hanging fruit" like the heterogenous tag delimiter syntax in the applicable tables (discussed above). Tasks like these can be done relatively easily using OpenRefine's basic filtering features.

After we tackle low-hanging fruit, we plan to use a combination of SQL (primarily SQLite), OpenRefine's more complex features like facets and clustering, and Pandas to do more complex data cleaning operations. Such operations might include normalizing the various `*_price` columns discussed

above, attempting to "fix" `xpos` and `ypos` coordinate values (if possible), or filtering foreign key constraints.

As we proceed with our profiling process, we plan to feed our discoveries back into step 1 to update our dataset description.

As this work has significant overlap with step 1, it will likely be a combined effort between Aaron and Dave.

Step 3: Perform DC "proper": How are you going to do it? What tools do you plan to use? Who does what? This step has some overlap with step 2 since our profiling process will likely involve some DC work. In other words, as we identify issues with our data, we'll likely be able to more easily fix them.

In more detail, we expect to primarily use OpenRefine to do DC. We already have a good idea of the domain of values stored in each column. From there we can assign a suitable type to each. For values that are text, for example, we can further designate a canonical capitalization format, a canonical delimiter, and convert the values to those. Once values are standardized, they can be separated and clustered into groups of like tags.

We have already tried to script all the data cleaning using OpenRefine's API and the Python `refine-client`. This idea was discounted when it was observed that the last update to that library was over 10 years ago, and it barely supports python 3. In fact, trying to run it using python 3 results in syntax errors from the library. Thus, any further attempts to script data cleaning with the official python library have been abandoned.

In light of that, Dave has undertaken to create a new client library as a proof of concept and good progress has already been made. Dave will likely continue this work, but creating a repeatable cleaning workflow allows either member of the team to contribute and change the cleaning process. This functionality would thus allow for better collaboration and we anticipate that, like in step 2, this work will be a combined effort between Aaron and Dave.

Step 4: Data quality checking: is D' really "cleaner" than D? Develop test examples / demos We aim to determine two things about a new dataset `D'`:

1. Is it a dataset that still represents D accurately?
2. Is it really cleaner than D?

While we plan to use OpenRefine to clean the data, we plan to use the features of a SQL database (primarily SQLite) to test for these two features.

To determine (1), we plan to attempt to normalize the data and enforce a SQL schema. We can use views to represent a version of the data matching the original dataset. If we can do so with minimal or no errors in the import process, we expect features of the database like referential integrity and strict data typing will ensure the dataset still represents D accurately.

We expect these features to also contribute to ensuring (2), but we also aim to develop a suite of database queries -- primarily those that join across more than 2 tables -- and acceptance criteria for the various results of each query.

Example test cases:

- For tables from the original dataset that purport to include the number of pages, dishes, etc (the "Original Number"), is the Original Number accurate?
  - If so, can a query be constructed for the cleaned dataset that returns the same value as the Original Number?
- Are all foreign key violations resolved? (i.e. For all cases, are there primary keys in the foreign table corresponding to the foreign key?)

As we begin to make meaningful progress on cleaning, we plan to discuss amongst ourselves who will take on the work to develop these test cases.

Step 5: Document and quantify change (e.g. columns and cells changed, IC violations detected: before vs after, etc.) If we continue with the Python implementation, we will have executable documentation of provenance through that workflow and the associated OpenRefine cleaning operations we perform, and both the SQL import process and our acceptance-test suite of queries will give us a detailed, standard outline of IC violations etc...

If we do not continue with the Python implementation, we plan to still export the standard OpenRefine-provided provenance data and the full suite of SQL queries we ran to perform all steps.

Regardless of direction, we expect the work to complete this step will be largely done for us as we complete DC. Any remaining work will be split between Dave and Aaron.