Data Cleaning and Provenance Project - Phase II

Note: We received **96/100 pts for Phase I and choose to retain that score for Phase II*

Jon Laflamme
University of Illinois
at Urbana-Champaign
jml11@illinois.edu

Nodirbek Hojimatov
University of Illinois
at Urbana-Champaign
nh11@illinois.edu

***Abstract*—Data cleaning may never receive the same attention and recognition as an impactful data visualization, but it adds indispensable value to the analytics pipeline by improving the usability and ensuring the veracity of data. Similarly, provenance is what ensures reproducibility of the analytics pipeline and is the primary mechanism for facilitating transparency. Therefore, in this report we demonstrate the utility of data cleaning and provenance through a Paycheck Protection Program loan approval dataset from the state of Hawaii. In this report, we describe the dataset and discuss obvious data quality problems. We then propose a target use case where data cleaning is both necessary and sufficient, followed by two edge cases where data cleaning is necessary but not sufficient and sufficient but not necessary. Next, we outline our plan to make the data fit for use. Specifically, we provide details on the tools, steps and techniques we plan to deploy, and we discuss how we will measure data quality. Last, we provide insight into our provenance approach; namely, we discuss how we plan to document our cleaning steps and measure the types and amounts of changes we make to the dataset.**

Keywords—Paycheck Protection Program, PPP, Data Cleaning, Provenance

I. THE DATASET

We decided to use the Paycheck Protection Program (PPP) loan dataset from Kaggle [1]. This dataset includes a subset of loans from the state of Hawaii ranging in amounts up to $150,000. PPP was one component of the $2.2 trillion Coronavirus Aid, Relief and Economic Security Act (CARES) passed in March 2020. The program was initially funded with $250 billion to provide payroll support in the form of forgivable loans to small businesses that were impacted by the pandemic. We chose this dataset both for its public interest value and because it requires a non-trivial amount of data cleaning and accompanying provenance to extract that value.

There are 21,904 records in this dataset, which is encoded as a comma separated value file. Figure 1.1 provides details on each of the fields present in the dataset. Each of the fields are encoded as strings, but we present the logical data type that is inferred from the schema.

Column Name	Data Type	Description
LoanAmount	Number	Load amount given to borrower
City	String	City of business
State	String	State of business
Zip	Number	Zip code of business
NAICSCode	Number	North American Industry Classification System Code
BusinessType	String	Type of Business of borrower
RaceEthnicity	String	Race and ethnicity of business owner
Gender	String	Gender of business owner
Veteran	String	Veteran status of business owner
NonProfit	Boolean	Is business a nonprofit?
JobsReported	Number	Number if people employed by business
DateApproved	Date	Date the load is approved
Lender	String	Lender/Bank name
CD	AlphaNumeric	Certificate of Deposit

Figure 1.1

II. USE CASES

We propose three use cases for this dataset, labeled as U_0, U_1 and U_2. We define U_0 as the base case in which no data cleaning is required. We define U_1 as our target use case where data cleaning is both necessary and sufficient. Finally, we define U_2 as a use case where data cleaning is necessary but not sufficient – that is, no amount of data cleaning will make it suitable for use. Since data quality issues are addressed later in our report, we will simply introduce our use cases here as a collection of queries for each use case and later address why these queries belong in the use case category that we have assigned them.

For U_0 we will query the average and total loan amount by zip code. The SQL queries might look something like this:

Q_0: "SELECT Zip, AVG(LoanAmount) FROM PPP_Table

GROUP BY Zip"

Q_1: "SELECT Zip, SUM(LoanAmount) FROM PPP_Table

GROUP BY Zip"

For U_1, our target use case, we investigate the relationship of the following fields: Lender; DateAccepted; NAICSCode; and City. First we consider the average loan amounts accepted by the lender. The query will look something like this:

Q_0: "SELECT Lender, AVG(LoanAmount)

FROM PPP_Table

GROUP BY Lender”

Next, we consider the average loan amount by city, which will look something like this:

Q_1: “SELECT City, AVG(LoanAmount)

FROM PPP_Table

GROUP BY City”

Then, we will shift our focus to investigate the relationship of loan amounts and job types by city. These queries will require the use of external data sources, evinced by the view “City_Classification_PPP,” such that we can extract the job classification using the NAICSCode and also verify the existence of any referenced cities. Specifically, we try to determine how many loans were issued by job type for each city as well as the average amount of loans by job type for each city. The queries will looks something like this:

Q_2: “CREATE VIEW City_Classification_PPP

FROM

(SELECT

C.City,

N.JobType,

P.LoanAmount

FROM PPP_Table P

(INNER JOIN City_Table C

ON (P.City = C.City

AND

P.Zip = C.Zip)

) LEFT JOIN NAICS_Table N

ON P.NAICSCode = N.NAICS)”

Q_3: “SELECT

C.City,

N.JobType,

COUNT(P.LoanAmount)

FROM City_Classification_PPP

GROUP BY

C.City,

N.JobType”

Q_4: “SELECT

C.City,

N.JobType,

AVG(P.LoanAmount)

FROM City_Classification_PPP

GROUP BY

C.City,

N.JobType”

Finally, we shift our attention to the relationship of a lender to the date the loan was accepted and to the number and types of jobs created over a given period of time. Here, the reader can assume that we have already verified all lenders using an external data source. The SQL queries might look something like this:

Q_5: “CREATE VIEW Jobs

FROM

(SELECT Lender,

LoanAmount,

JobsReported,

MONTH(DateApproved)

AS Month,

(LoanAmount / JobsReported)

AS PerJobAmount

FROM PPP_Table)”

Q_6: “SELECT Lender, Month, AVG(PerJobAmount)

FROM Jobs

GROUP BY Lender, Month”

Q_7: “SELECT Lender, Month, AVG(JobsReported)

FROM Jobs

GROUP BY Lender, Month”

For Q_5-Q_7, we think these queries will provide a good picture of lender behavior at different points in the age of the program. Specifically, we think Q_7 will demonstrate whether lenders favored businesses with fewer or more employees at the earlier stages of the program. Similarly, Q_6 should show whether high paying or low paying jobs were favored earlier on. For the sake of completeness, we may also change the date queries to finer granularity, such as week or day to capture details during the initial surge of applications that occurred at the very beginning of the program.

Last, we consider U_2, which the reader will recall is our untenable use case, such that no amount of data cleaning can sufficiently remedy the data quality issues. Here, we hope to explore how the lending institution relates to ethnicity and

gender. We designate this query set as U_2 because the ethnicity and race fields are missing in more than 90 percent of the records, and no amount of data cleaning can remedy this deficiency. First we explore the total loan amounts issued by ethnicity. The SQL query might look something like this:

Q_0: SELECT Race, SUM(LoanAmount)

FROM PPP_Table

GROUP BY Race"

Next, we would drill deeper into the lender relationship to see whether gender or ethnicity show statistical significance in loan distributions. The SQL queries might look something like this:

Q_1: "SELECT Lender, RaceEthnicity, AVG(LoanAmount)

FROM PPP_Table

GROUP BY Lender, RaceEthnicity"

Q_2: "SELECT Lender, Gender, AVG(LoanAmount)

FROM PPP_Table

GROUP BY Lender, Gender"

While we would be very interested in knowing whether race or gender played a role in loan distribution patterns, the amount of missing records makes any conclusions drawn from such an inquiry problematic, since we cannot rule out the existence of selection bias of applicants. Further, the statistical significance of any findings would be far less meaningful with such a limited sample size.

III. DATA QUALITY PROBLEMS

As hinted at earlier in our report, the PPP loan dataset presents a number of serious data quality issues. We address these as data quality categories, each of which affects one or more fields in the dataset.

First, we consider completeness, where one or more records are either marked "Unanswered" or are missing for a given field. 20,215 of 21,904 records in the RaceEthnicity field are marked "Unanswered". Similarly, 18,454 of all records are marked "Unanswered" for the Gender field. For the Veteran field, 19,359 records are marked "Unanswered". For NonProfit, 21,226 records are missing, and JobsReported is missing 2,447 entries. NAICSCode is missing 105 records. BusinessType is only missing a single entry. The remaining fields (City, State, Zip, DateApproved, Lender, and CD) are all complete.

Next, we consider normalization, which we define as instances where data entries should be identical but because of typographical errors and formatting cannot be properly aggregated. The City field is the worst offender, where 88 clusters are present in the first pass of clustering on key collision using the fingerprint function in OpenRefine. For the Lender field, several different clustering techniques each yield a small number of collisions that will require examination and normalization.

Next, we consider integrity constraints (IC). There are five types of IC: null rule; unique column value; primary key; referential integrity; and complex integrity checking (ie. user-defined) [2]. While these rules apply more to relational database management systems (RDBMS) theory more than data cleaning, (since they are typically applied as triggers or checks for RDBMS insertions, updates and deletions), we can still retroactively enforce these rules in order to enforce data integrity in our dataset.

Therefore, we begin with the null rule, which disallows records where the value is missing for a given field. Already, we have identified columns that are missing a substantial number of values; these are the fields that will require consideration under the null rule. Next, we consider the unique column value constraint, which requires that there are no duplicate values in a given column. Since there are no uniquely identifying fields in our dataset (e.g. Employer Identification Number), we do not need to enforce this constraint.

The third constraint is the primary key constraint, which requires each record to be uniquely identifiable by one or more columns over the entire dataset. Since there are no unique column value constraints in our dataset, we will need to carefully select a collection of columns that will uniquely identify each record. We will need to proceed carefully in order to ensure there are no duplicate records while at the same time not eliminating unique records that coincidentally happen to share several attributes. In order to take the most conservative approach, we make the set of all fields the primary key, such that any duplicate records can be identified as IC violations and removed.

The fourth IC is referential integrity, also known as foreign key constraint, which ensures the existence of a referenced field. For example, if John Doe's John Deere tractor model number is referenced in a tractor owners table, we expect that tractor model number field to actually exist in a tractor models table. Therefore, we consider three instances in our dataset where referential integrity is a concern.

The first instance is the NAICSCode field, which references a federally designated industry category. Since we hope to provide our end users with the common name associated with this numerical identifier, we will need to introduce a supplementary NAICS dataset to extract this information using the NAICSCode given in the PPP dataset.

The second instance is the City field, which presents many problems due to key collisions related to spelling and formatting. We hope to remedy this problem by introducing yet another table, which contains the city, state and zip code for all municipalities in Hawaii. We can use a foreign key constraint

on City and Zip to ensure that each city referenced in the PPP dataset actually exists and is spelled correctly.

The third instance is the Lender field, which, like City and NAICSCode, we hope to validate through an external table that contains a more authoritative source of names of lending institutions in Hawaii. We demonstrate each of these three ICs in figure 1.2.

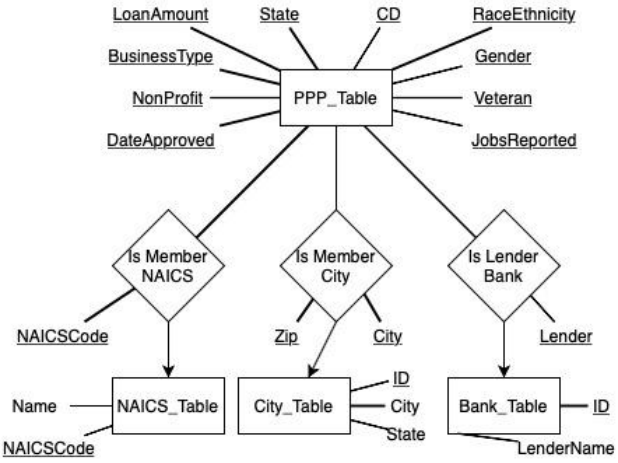

Figure 1.2

The fifth and final IC is complex integrity checking. The data quality issues we plan to check here concern the LoanAmounts and JobsReported fields. Here we hope to establish an upper bound flag for per-job loan amounts, such that DIVIDE(LoanAmount, JobsReported) > $25,000 would capture all loans in a higher income category (ie. user-defined as > $10,000/month), since the PPP loan maximum is 250% the average monthly employee payroll expenses for the previous year [3]. We have not yet decided what we may do with the results of this check, but likely we will create a view for these flagged loans. Similarly, we plan to flag the JobsReported with an upper and lower bound, such that records where JobsReported < 1 AND JobsReported > 250 will be identified and reviewed. This IC check is important because the eligibility of a small business to receive a PPP loan was determined in part by the number of employees, which for most business categories is capped at a maximum of 250 [4]. Likewise, JobsReported < 1 could yield unexpected behavior when performing calculations and aggregations over that field, such as when computing the PerJobAmount from DIVIDE(LoanAmount, JobsReported). Since Sole Proprietorships and Independent Contractors should legally have zero employees, we will need to determine some way to handle these records, such that the information is preserved without skewing aggregations and calculated fields. Likely, we will convert all cases of missing or zero-value entries to a default value of '1' in order to capture the spirit of our query, which is, "How many jobs does this loan support"?

Moving on from IC, the final data quality issue concerns data types. Note that we do not explicitly discuss each field where a data type conversion is required. The reason is that conversion from the raw string encoded CSV data to numeric, boolean or date datatypes is either automated by database software or else is trivial to perform. But we do point out that these conversions will be required for several of the fields in order to successfully run our queries.

IV. CLEANING PLAN

We expect to perform the bulk of our cleaning with OpenRefine, which is an open source software that provides many convenient features for exploring and cleaning datasets. It also provides built-in support for provenance, where each document transformation is recorded as a step in a recipe that can be downloaded. In fact, we have already relied heavily on OpenRefine's text facet filtering and clustering tools in order to evaluate data quality issues. Many of our syntactic data quality issues, such as key collisions due to formatting and spelling inconsistencies as well as data type conversions, we plan to remedy with OpenRefine.

Once our syntactic issues are resolved, we will shift our focus to ICs, where we will rely heavily on Datalog, which is an Answer Set Problem (ASP) declarative programming language that provides good support for ensuring that ICs are enforced. Since we have already covered each of the five categories of ICs in the previous section, we will not expand on our use of Datalog except to note that we plan to enforce all of our ICs with this tool.

Though we have not explicitly mentioned this until now, we expect to use the Regular Expression (RegEx) language to address typographical issues throughout our dataset. For example, many of the city entries have minor spelling and formatting variations that we will fix with RegEx. Again, this language feature is integrated with OpenRefine, which is where we will employ this approach to handle such syntactical issues.

V. PROVENANCE PLAN

Concerning documentation and provenance, we plan to use four tools to track changes made to our dataset throughout the cleaning process: OpenRefine Recipes; Jupyter Notebook for Datalog; ER Diagrams; and YesWorkflow. We will go through each in order.

First we consider OpenRefine Recipes. As mentioned in our data cleaning plan, OpenRefine automatically records each transformation made to the dataset as a series of executable steps that can be exported as a standalone file. Therefore will

use this feature to document any and all data cleaning that is performed in OpenRefine.

Next we consider Jupyter Notebook, which is a programming environment that provides an interactive shell with support for markdown languages and multiple programming languages. Aside from these features, the input and output of any Jupyter Notebook can be easily exported to a variety of formats, which makes it a convenient platform to document our use of Datalog.

Entity Relation (ER) diagrams are the third tool we plan to use for provenance. While OpenRefine and Jupyter Notebook capture individual transformations and IC assertions with precise detail, these platforms do not provide an overview of the dataset and how different components of the dataset relate to one another. Therefore, we will use ER diagrams to identify the entities and relations and ICs present in our dataset wherever additional clarity is required. A preliminary ER diagram is shown in figure 1.2. Others will be added in the next phase of our report.

We are investigating if we can also use YesWorkflow to provide additional provenance support in task II, but since we have not familiarized ourselves yet with that software, we cannot yet comment on how specifically that tool would be utilized.

==========

BEGINS PHASE II REPORT

==========

VI. EXTERNAL DATA SOURCES

While we originally expected to use three external sources, we found that the Lender field cleaning requirements were minimal enough that we could validate any suspicious fields on a case by case basis without the need to join the PPP dataset [16] with a separate banks and credit unions dataset. Instead, we used two external data sources to supplement and validate our primary PPP Loan Dataset. The first external dataset was sourced from a secondary publisher of the United States Postal Service's City-State-Zip code database. This dataset lists all 3-5 digit zip codes in the United states along with the associated city and state (or territorial) name [5, 16]. The second external dataset was obtained from the United States Census Bureau and contains the North American Industry Classification System (NAICS) codes along with a description field of the industry category associated with each NAICS code [6, 16]. We show a simple updated ER diagram below (Fig. 2.1), which details the revised structure of our data relations. However, the final database schema on which we performed our target queries is included in our supplementary materials [15].

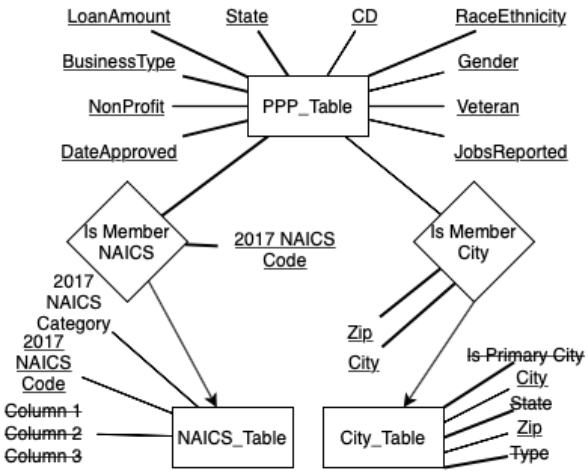

Fig 2.1

VII. CLEANING WITH PANDAS

The goal of the NAICS Code dataset was to extract a descriptive string for each NAICS code found in the PPP dataset. Since the United States Census Bureau data was available as an '.xlsx' file, we first loaded the data into Numbers, the Mac OS spreadsheet application and exported the file as a '.csv' file. This made it simpler to load the file into Python Pandas using the built-in 'pandas.read_csv()' function. The next step was formatting this file to make it suitable for joining with the PPP Loan dataset. Specifically, the following steps were applied in Pandas [8]:

1) Read both the PPP file and NAICS file into Pandas using pandas.read_csv()
2) Converted the datatype of the 'NAICSCode' field in the PPP DF to 'int32' from 'float64'; then converted this field to 'string' format. These steps were necessary because Pandas requires data types to match on joined fields.
3) Dropped three empty 'unnamed {i}' columns in the NAICS dataFrame. These fields were an artifact of the original file encoding and were not required for our target use case queries.
4) Dropped records with 'Nan' in the NAICS DF. This was necessary to prevent errors when performing data type conversions on whole columns, such as with step #5.
5) Converted the NAICS code in the NAICS DF to 'int32' from 'float64'; then converted to 'string'. Also converted the NAICS description field to 'string' from 'object' type in the NAICS DF. Same rational as step #2. Conversion of the latter description field to 'string' data type was to prevent encoding issues when exporting the resulting joined table to '.csv'.
6) Performed left join on the two dataFrames (DF), with the PPP DF on the left and the NAICS DF on the

right and joined on the matching NAICS fields. This step was for convenience to make our target queries simpler and faster.

7) Confirmed that the left join reported the same number of records in the resulting DF as were present in the PPP DF; confirmed in a subset of the resulting DF that all joined fields were present. These steps were to ensure that the join did not introduce any new records and that the join operation was completed as expected.
8) Exported the joined DF as a '.csv' file [7, 16]. This export allowed us to add an intermediate dataset to our repo as an additional provenance checkpoint. We chose this encoding format because it preserves data and schema and is readable by sql-lite.

Shown below, Fig. 2.2 and Fig. 2.3 quantify these operations:

PPP Dataset - Pandas		
Columns	**Operation**	**# Cells Affected**
NAICSCode	Dtype: double => int	21799 of 21799
NAICSCode	Dtype: int => string	21799 of 21799
Entire DF	Left Join PPP.NAICSCode = NAICS.NAICS	21904 of 2904

Fig. 2.2

NAICS Dataset - Pandas		
Columns	**Operation**	**# Cells Affected**
Unnamed 1, Unnamed 2, Unnamed 3	Dropped	1057 of 1057
*Row operation	Drop NaN	1 empty row
2017 NAICS Code	Dtype: double => int Dtype: int => string	1056 of 1057
2017 NAICS Category	Dtype: object => string	1056 of 1057

Fig. 2.3

VIII. CLEANING WITH OPEN-REFINE

We used OpenRefine for nearly all of our other data cleaning operations which allowed us to record each step of the process in a recipe, which we then exported as a serialized JSON file and persisted in our project repository. We used two separate recipes in OpenRefine – one for cleaning the City-State-Zip dataset [9] and the other for cleaning the dataset that resulted from joining the PPP dataset with the NAICS dataset and thereafter with the cleaned City-State-Zip dataset [10]. Note that we will refer to the dataset that resulted from joining the original PPP dataset and the NAICS dataset as PPP-Prime (PPP′), and we will refer to the dataset that resulted from joining PPP′ with the cleaned City-State-Zip dataset as PPP-Double-Prime (PPP″). Similarly, we will henceforth refer to the City-State-Zip dataset as (CSZ).

CSZ was not immediately suitable to join with PPP′ in its original form. First, the dataset was too large, since it contained data for all U.S. states and territories. Therefore, we used Python Pandas to filter out records that did not include the state of Hawaii [11]. This was mainly for convenience and efficiency since Github has strict limits on the size of repositories and because OpenRefine and Sql-lite run much more efficiently on smaller datasets. Since we did not require fields from any state or territory other than Hawaii, it was logical to prune the dataset before trying to clean it in OpenRefine [16]. For convenience and efficiency, we also dropped three columns that were not required for our target use case queries. These changes are detailed below in Fig. 2.4.

City-State-Zip Dataset (CSZ) - Pandas		
Columns	**Operation**	**# Cells Affected**
*Row operation	Make null rows where State != "HI"	50893 rows
*Row operation	Drop null rows	50893 rows
Is Primary City, Type, State	Drop columns	147 of 147

Fig. 2.4

The second issue with CSZ was that some zip codes mapped to more than one city. Since we were using this dataset to validate city names and spellings in PPP″, we needed to ensure that the zip code field was constrained to be a unique key for that dataset. We observed this data quality issue when we performed an exploratory left join in Python Pandas with CSZ from PPP′ on the zip code field. The resulting table showed that more than 600 new records were created as a result of the join, thereby demonstrating the need to enforce a unique key constraint on CSZ prior to joining with PPP′.

CSZ also showed some minor data quality issues, which we addressed using OpenRefine. The reason we addressed the remaining cleaning in OpenRefine for CSZ versus Pandas was to better support our Provenance modeling requirements and also to take advantage of the unique features OpenRefine offers, namely clustering. The following list summarizes some of the cleaning operations we applied to the dataset in OpenRefine:

1) Uppercased all city names. This was required to achieve field normalization.
2) Trimmed leading and trailing white spaces for city field. This was required to achieve field normalization.
3) Replaced multiple spaces between characters in City field with a single space. This was required to achieve field normalization.
4) Fixed abbreviated city names one by one. This was required to achieve field normalization.
5) Clustered city names and normalized by resolving collisions. This was required to achieve field normalization.
6) Exported the resulting file to .csv format. This was to prepare the data for consumption by our query engine, sql-lite.

The changes applied are shown below in Fig. 2.5

City-State-Zip Dataset (CSZ) - OpenRefine		
Columns	**Operation**	**# Cells Affected**
City	toUppercase()	0 of 147
City	trim()	0 of 147
City	replace(/\\+s/, ‘ ‘)	0 of 147
City	Cluster and Merge. Manual expansions for abbreviated city names	8 cells. See OpenRefine recipe for details
City and Zipcode	Create new “Temp” column based on concatenation of City and Zipcode columns	147 of 147
Temp	Blank down to blank out duplicate city and zip codes	5 of 147
*Row Operation	Remove duplicate rows	5 of 147
Temp	Remove temp column	142 of 142

Fig 2.5

Once these minor data quality issues were addressed in CSZ, we were ready to join this dataset with PPP′ using a simple SQL technique (ie. “LIMIT 1”) that constrained the number of joined records in PPP″ to not exceed the number of records present in PPP′. This satisfied the unique key constraint specified previously. Since we were using the CSZ city field mainly to validate spelling in PPP′ after joining as PPP″, the possibility that the “LIMIT 1” operation might return the wrong city for a given zip code presented at worst the need to manually reconcile those cases individually. The complete SQL command we used to create PPP″ from PPP′ and CSZ was as follows:

```
SELECT p.*, (
       SELECT city
       FROM zipcode as z
       WHERE z.zipcode = p.zip
       LIMIT 1
       ) AS CleanedCity
FROM ppp_naics as p;
```

Once we obtained PPP″ from Sql-lite, we saved the result as a ‘.csv’ file and loaded the newly created file into OpenRefine for cleaning. The cleaning steps we applied to PPP″ are summarized below as a list:

1) Removed unnecessary columns that were either duplicative or not required for our target use case queries. This was mostly for convenience but was also helpful to conserve storage requirements.
2) Removed leading and trailing whitespaces from all text fields. This was required to achieve field normalization.
3) Replaced multiple spaces between characters with a single space in all text fields. This was required to achieve field normalization.
4) Uppercased text fields for all text fields. This was required to achieve field normalization.
5) Converted JobsReported and LoanAmount from text type to numeric type. This was required to perform numeric operations in OpenRefine (step #6) and sql-lite.
6) Replaced entries with null and 0 values in the JobsReported field with a default value of 1*.
7) Converted DateApproved from text type to date type. This was required for step #8 and to perform date queries.
8) Created MonthApproved field from DateApproved field using ‘GREL’ function [14]. We derived the MonthApproved field because it was required for our target queries.
9) Resolved collisions in the Lender field. This was required to achieve field normalization.

*Note that the JobsReported field was set to a default value of 1 because the PPP loan program was constrained to supply

payroll funding for 1 or more jobs. Therefore, a value less than 1 is not possible. We recognize and draw attention to the fact that this action could produce an inaccuracy where a JobsReported entry should have shown a value greater than 1 but was unreported or missing from the original PPP dataset. The changes applied are shown below in Fig 2.6.

PPP-Double-Prime (PPP″) - OpenRefine		
Columns	**Operation**	**# Cells Affected**
Column, State, NAICSCode, BusinessType, RaceEthnicity, Gender, Veteran, NonProfit, CD	Drop columns	21904 of 21904
CleanedCity, JobTitle, Lender, City	toUppercase()	0 of 21904 21595 of 21904 21902 of 21904 0 of 21904
CleanedCity, JobTitle, Lender, City	trim()	0 of 21904
CleanedCity, JobTitle, Lender, City	replace(/\\+s/, ‘ ‘)	0 of 21904
JobsReported, LoanAmount	toNumber()	19457 of 21904 21904 of 21904
JobsReported	Grel: {0,null} => 1	5381 of 21904
DateApproved	toDate()	21904 of 21904
Lender	FirstBank => First Bank	1 entry
MonthApproved	New Column from Grel: Extract Month from DateApproved (eg.1 == JANUARY)	21904 of 21904

Fig 2.6

IX. TARGET USE CASE QUERIES

The queries we used for our target use case are available in our project repository as a ‘.sql’ file [12]. Similarly, the results from each of our queries are available in our repository as ‘.csv’ files [13].

X. YES WORKFLOW

We created four diagrams using YesWorkflow and have attached them in our supplementary materials. Among the diagrams are a linear [17] and parallel model [18] for the cleaning operations we performed on PPP" using OpenRefine. The diagrams also include a linear [19] and parallel model [20] for the OpenRefine operations performed on CSZ.

Once we completed cleaning PPP", we installed the OR2YW tool and executed the following commands on the terminal to generate inner and outer workflows. We chose OR2YW because it provided easy conversion of open-refine history to workflow diagrams. Here are the commands:

```
or2yw -i recipe_zipcode.json -o csz_parellel.pdf -ot pdf -t parallel

or2yw -i recipe_zipcode.json -o csz_linear.pdf -ot pdf

or2yw -i recipe_ppp.json -o ppp_parallel.pdf -ot pdf -t parallel

or2yw -i recipe_ppp.json -o ppp_linear.pdf -ot pdf
```

These commands generated teh PPP" and CSZ cleaning workflow diagrams in linear and parallel formats. Both formats have unique benefits to display the workflow so we decided to keep them both.

In the Appendix-XII-CSZ-Parallel.pdf file we described the cleaning workflow of the CSZ dataset. Input for this process was the raw CSZ dataset and output was cleaned CSZ dataset, ready to be joined with PPP', the result of which was PPP". First, we executed various transformation methods, shown in green rectangle boxes, on the city column (yellow boxes) for normalization. Then, we created a temporary column to identify the duplicate zip code and city combinations. Thereafter we removed the duplicate records.

In the Appendix-X-PPP-Parallel.pdf file we described the cleaning workflow of PPP". Input for this process was the raw PPP" dataset and output was cleaned PPP" dataset that was loaded by sqlite3 database. Key methods were column removal, month extraction from the “ApprovedDate” column, and cell transformation methods that were shown in green boxes.

XI. CONCLUSION

This project was a valuable introduction to a complete data cleaning workflow. We both appreciated the conceptual challenges of identifying used cases U_0 ,U_1 and U_2 for each of the candidate datasets, and, near the end of completing phase I, we even ended up changing our opinions about which queries should belong to U_1 and U_2, such that we had to shuffle our queries from one category to another and rewrite a large section of our report. We also appreciated the additional exposure to

and practice with OpenRefine and YesWorkflow. We found both of these applications to be powerful and helpful tools for completing our project. But the primary value we have both taken away from this project is an increased awareness of the conceptual considerations involved with data cleaning. While the tools and mechanics of data cleaning are certainly valuable, the consequence of properly identifying data quality issues and understanding how to guarantee fitness of use through process and provenance is the main lesson we learned through this project and what we expect to have the most impact in our future work.

XII. ACKNOWLEDGEMENTS

The responsibilities among team members were divided as follows:

Jon wrote both the phase I and phase II reports. He also researched and identified our two supplementary datasets to support the target dataset that was provided in course materials. Jon also performed all Python-based data cleaning and joining operations that were described in sections VI and VII. Additionally, Jon set up the project repository.

Nodirbek completed all OpenRefine and sql-lite operations described in sections VI, VII and VIII, which included cleaning, joins, and adapting our target use case queries from the Phase I report. Nodirbek also provided the YesWorkflow model, including both 'inner' and 'outer' diagrams.

We met twice weekly for most weeks after the final project details were released. During these meetings we discussed our independent findings from exploring each of the instructor-provided datasets in OpenRefine. We then selected the PPP dataset and sketched out some likely use cases to fulfill the U_0 , U_1 and U_2 phase I requirements. Throughout the remaining meetings, we discussed our progress and progressively discussed both conceptual and concrete data quality issues and decided on appropriate tools and resolutions to address these issues. We shared the responsibility of developing supplementary materials such as charts and graphs and preparing the project files for submission.

XIII. REFERENCES

[1] PPP Loan Level Data. Uploaded by Bruce Anders to Kaggle. September, 2020. https://www.kaggle.com/bruceanders1/ppp-loan-level-data

[2] Database Concepts (eBook), 10g Release 2 (10.2). Chapter 21: Data Integrity. Oracle, 2021. https://docs.oracle.com/cd/B19306_01/server.102/b14220/data_int.htm

[3] Second Draw PPP Loan. https://www.sba.gov/funding-programs/loans/covid-19-relief-options/paycheck-protection-program/second-draw-ppp-loan

[4] PPP Loans FAQs. June 8, 2021 https://home.treasury.gov/system/files/136/Paycheck-Protection-Program-Frequently-Asked-Questions.pdf

[5] United States Postal Service (USPS) Zip Codes: http://www.usboundary.com/Resources/USPS%20Zip%20Codes%20Download

[6] United States Census Bureau: North American Industry Classification System (NAICS). 6-digit 2017 Code File. https://www.census.gov/naics/2017NAICS/6-digit_2017_Codes.xlsx

[7] Github Project Repository. "ppp-naics-joined.txt" https://github.com/Jon-LaFlamme/data-cleaning-project/blob/main/data/ppp-naics-joined.txt

[8] See: "Appendix-II.ipynb".

[9] See: "Appendix-III-PPP-OpenRefineHistory.json"

[10] See: "Appendix-IV-CSZ-OpenRefineHistory.json"

[11] See: "Appendix-I.ipynb".

[12] See: "Appendix-V-Queries.sql"

[13] Github Project Repository. Results directory. https://github.com/Jon-LaFlamme/data-cleaning-project/tree/main/data/results

[14] See: "Appendix-VI-Grel.txt"

[15] See: "Appendix-VII-Schema.txt"

[16] See: "Appendix-VIII-DataLinks.txt"

[17] See: "Appendix-IX-PPP-Linear.pdf"

[18] See: "Appendix-X-PPP-Parallel.pdf"

[19] See: "Appendix-XI-CSZ-Linear.pdf"

[20] See: "Appendix-XII-CSZ-Parallel.pdf"