

Prepare, Clean, and Analyze the National Health and Nutrition Examination Survey Data
James M. Gilson, Yang Yang, Junwei Xu, Lanxue Zhang, Miller Luo
School of Professional Studies
Columbia University

Data Preparation

Scenario

The National Health and Nutrition Examination Survey (NHANES) is a program created to evaluate the health and nutritional status of adults and children in the United States, which is a major program of the National Center for Health Statistics (NCHS).

We are hired by NCHS, and our job is to reorganize the survey results and design a database that is more accessible and powerful for medical research, analytics and customer querying.

On a daily basis, doctors need to diagnose based on their patients' data, such as history, profile, and medication usage. Researchers need to explore the relations between diseases and other variables such as examination, medication, and lab records. With more information technology entering the healthcare space, more data tools created, and greater regulations, organizations are looking for automated systems. NHANES is a great data resource, with which thousands of research findings are published.

However, the dataset is not normalized. Some tables such as medications contain hundreds of columns, and not all those columns are only related to the primary key. Non-normalized data is kind of messy, and harder to be used for analytics, so we are going to normalize those tables and dealing with missing values to make this dataset is more useful and accessible.

Our project output includes a redesigned database and a Metabase Dashboard for business analytics. Both of them will provide valuable benefits for our customers and the NCHS itself. Through the reorganized database and newly-designed metabase dashboard, it would be more convenient for doctors and researchers to access and make use of these survey results, and more research findings may come out. Especially with the development of the dashboard, the medical professionals can easily do analytics and generate insights. For the NCHS, it's also easier for the internal usage and provide a good frame for the future data management. As observed, NCHS did the survey each year, so the new data framework can provide guidance for the survey results management, and the dashboard is easy for NCHS to visualize the survey results.

Team Contract

Name	Responsibilities
James	Schema Design; Uploading Dataset into Group Server; Reviewing and Finalizing Jupyter Notebook; Demographics and Prescription Activity Dashboards

Yang Yang	Project Management ; Draft of Jupyter Notebook - Patient-related Tables and Examination-related Tables; Report Writing.
Junwei Xu	Draft of Jupyter Notebook - Medication-related Tables and Diet-related Tables; Analytics Procedures Design and SQL Query Writing; Report Writing.
Lanxue Zhang	Scenario Design and To C Applications Design; Draft of Jupyter Notebook - Labs-related Tables; Metabase Setting up; Report Writing.
Miller Luo	Finding Dataset; Slides Design; Analytics SQL Query Writing; Report Writing.

Sample of data




Our original data contains 6 tables: demographic, diet, examination, labs, medications, and questionnaire.

The original dataset can be found [here](#).

We chose this dataset based on the following reasons:

1. It's a large-scale dataset. Main tables such as examination, labs and diet contain hundreds of columns, which means we have enough data for normalization, database design and future analytics.
2. This is a dataset that contains both examination results and survey results, giving access to both objective information and subjective information about patients.
3. This dataset contains a tremendous amount of information about different diseases, offering the potential to explore many different medical questions. For the purposes of the analytic components of this project, we will focus mainly on questions associated with diabetes.
4. The way the information is currently organized is difficult to work with. A restructuring of this information has the potential to offer significant benefits.

Here is a sample of the medications table.

medications.csv (1.24 MB)							12 of 13 columns		Views		
# SEQN	# RXDUSE	A RXDDRUG	A RXDDRGID	# RXDDAYS	A RXDRSC1						
		<div>[null] 30%</div> <div>LISINOPRIL 2%</div> <div>Other (698) 67%</div>	<div>[null] 31%</div> <div>d00732 2%</div> <div>Other (695) 67%</div>		<div>[null]</div> <div>I10</div> <div>Other (472)</div>						
73557	1	99999									
73557	1	INSULIN	d00262	1460	E11						
73558	1	GABAPENTIN	d03182	243	G25.81						
73558	1	INSULIN GLARGINE	d04538	365	E11						
73558	1	OLMESARTAN	d04801	14	E11.2						

Besides the 6 original tables, we also extracted the data dictionaries from NHANES through the following links:

Demographics	Diet	Examinations
Labs	Medications	

Data Cleaning and Loading

Normalization Plan

Our [Lucidchart](#).

For our normalization plan, we chose not to use the questionnaire portion of the dataset to keep the overall scope of this project at a reasonable level. Similarly, for the Demographics table, since there are many columns, each of which has the potential to correspond to a separate lookup table, we only selected several dimensions that we anticipate being especially relevant for analysis. We picked marital status, ethnicity, country, and income. These attributes may be more closely related to our examinees' health. In the original demographics data, we are missing references for the values. For instance, in the marital status, a value of 1 could refer to married, a value of 2 could refer to single, etc. It took us some time to finally find it on the original website. Hence we chose to add lookup tables, when people use our dataset, they could easily know what each value means, instead of looking for it on a separate website. For our lookup

tables, we have the id to be a primary key and they correspond to a value. All the ids reference to the ids in the patients table.

The medications table has medication_id as its primary key, and functions both as a unique list of all medications, and a lookup table for medication_id. The patient_medications table indicates what medications have been prescribed to each patient. We used the combination of patient_id, medication_id, and prescription_number as our primary key, with patient_id identifying the patient, medication_id identifying the medication the patient was prescribed, and prescription_number identifying how many times this medication has been prescribed to the same person (in the original table, there are a number of instances where the same patient is prescribed the same drug more than once with different reasons for the prescription). Then we have days_taken, which tells us how many days our participants has been taking the medication. The prescription_reasons table captures the reasons that a patient was prescribed a given medication. We have a combination of patient_id, medication_id, prescription_number, and reason_number as primary key (there are up to three prescription reasons associated with each prescription). The reason_for_prescription attributes explains the prescription reason for each combination.

We also have three tables for examinations information. We have an examination_categories table, in which each examination_category_id corresponds to one examination_category name (a number of examination_types may be associated with the same category, i.e. systolic and dystolic are two separate examination types that both correspond to a blood pressure examination category). For the examination_types table, we have examination_id to be the primary key, examination_name that corresponds to examination_id and examination_category_id referencing the examination_categories table. For the examination table, we have patient_id as our primary key, which references to our patient table, examination_id and value. For labs information, we have an identical design. The diet information, is stored in a similar way. The only difference is that we don't have diet_categories since after cleaning data, there is only one category left, there is no need for a separate table for type.

SQL Tables Creation

A github repository of our ETL codes can be found [here](#)

For all our SQL codes, we implemented SQL constraints.

First we created the lookup tables for patients information. They all consist of an id and a value.

```
1. CREATE TABLE MaritalStatus_lookup (
2.     maritalStatus_id      int,
3.     maritalStatus_value   varchar(50) NOT NULL,
4.     PRIMARY KEY(maritalStatus_id)
5. );
6. CREATE TABLE Ethnicity_lookup(
7.     ethnicity_id          int,
8.     ethnicity_value       varchar(50) NOT NULL,
9.     PRIMARY KEY(ethnicity_id)
10.);
11. CREATE TABLE Country_lookup(
12.     country_id           int,
13.     country_value        varchar(50) NOT NULL,
14.     PRIMARY KEY(country_id)
15.);
16. CREATE TABLE Income_lookup(
17.     income_id            int,
18.     income_value         varchar(50) NOT NULL,
19.     PRIMARY KEY(income_id)
20.);
```

Then we created the patients table. Patient_id is the primary key and all other ids correlate to the id from the lookup tables.

```
1. CREATE TABLE Patients (
2.     patient_id            int,
3.     maritalStatus_id      int NOT NULL,
4.     ethnicity_id          int NOT NULL,
5.     country_id            int NOT NULL,
6.     income_id             int NOT NULL,
7.     PRIMARY KEY (patient_id),
8.     FOREIGN KEY (maritalStatus_id) REFERENCES MaritalStatus_lookup
9.     (maritalStatus_id)
10.     ON UPDATE CASCADE ON DELETE CASCADE,
11.     FOREIGN KEY (ethnicity_id) REFERENCES Ethnicity_lookup (ethnicity_id)
12.     ON UPDATE CASCADE ON DELETE CASCADE,
13.     FOREIGN KEY (country_id) REFERENCES Country_lookup (country_id)
14.     ON UPDATE CASCADE ON DELETE CASCADE,
15.     FOREIGN KEY (income_id) REFERENCES Income_lookup (income_id)
16.     ON UPDATE CASCADE ON DELETE CASCADE
17.);
```

Then we create the medications table which contains medication_id, and medication_name, with medication_id being the primary key. Then the Prescription_Reasons table which contains patient_id, medication_id, prescription_number, reason_number and reason_for_prescription with first three being the primary key. Patient_id relates to the id in the patients table, while the medication_id refers to the medication table. The Patients_Medication table includes patients_id, medication_id, prescription_number, and days_taken, with the first three being the primary key. Patient_id relates to the id in the patients table, while the medication_id refers to the medication table.

```
1. CREATE TABLE Medications (  
2.     medication_id      varchar(10),  
3.     medication_name    varchar(100) NOT NULL,  
4.     PRIMARY KEY (medication_id)  
5. );  
6.  
7. CREATE TABLE Prescription_Reasons (  
8.     patient_id         int,  
9.     medication_id      varchar(10),  
10.    prescription_number int,  
11.    reason_number       int,  
12.    reason_for_prescription varchar(150) NOT NULL,  
13.    PRIMARY KEY (patient_id, medication_id, prescription_number,  
14.                reason_number),  
15.    FOREIGN KEY (patient_id) REFERENCES Patients (patient_id)  
16.        ON UPDATE CASCADE ON DELETE CASCADE,  
17.    FOREIGN KEY (medication_id) REFERENCES Medications (medication_id)  
18.        ON UPDATE CASCADE ON DELETE CASCADE  
19. );  
20. CREATE TABLE Patient_Medications (  
21.     patient_id         int,  
22.     medication_id      varchar(10),  
23.     prescription_number int,  
24.     days_taken         int NOT NULL,  
25.     PRIMARY KEY (patient_id, medication_id, prescription_number),  
26.     FOREIGN KEY (patient_id) REFERENCES Patients (patient_id)  
27.         ON UPDATE CASCADE ON DELETE CASCADE,  
28.     FOREIGN KEY (medication_id) REFERENCES Medications (medication_id)  
29.         ON UPDATE CASCADE ON DELETE CASCADE  
30. );
```

Then we created the Examination_Categories table with the id being the primary key and name correspond to it. The Examination_Types table uses examination_id as the primary key, which also refers to the examination table. Other info included are examination_name and

examination_category. Examinations table uses patient_id and examination_id as its primary key, each correspond to Patients and Examination_Type table.

```

1. CREATE TABLE Examination_Categories (
2.     examination_category_id    int,
3.     examination_category_name  varchar(150) NOT NULL,
4.     PRIMARY KEY (examination_category_id)
5. );
6. CREATE TABLE Examination_Types (
7.     examination_id             varchar(10),
8.     examination_name           varchar(300) NOT NULL,
9.     examination_category_id    int NOT NULL,
10.    PRIMARY KEY (examination_id),
11.    FOREIGN KEY (examination_category_id) REFERENCES Examination_Categories
    (examination_category_id)
12.    ON UPDATE CASCADE ON DELETE CASCADE
13.);
14. CREATE TABLE Examinations (
15.     patient_id                int,
16.     examination_id            varchar(10) NOT NULL,
17.     value                     varchar(20) NOT NULL,
18.     PRIMARY KEY (patient_id, examination_id),
19.     FOREIGN KEY (patient_id) REFERENCES Patients (patient_id)
20.     ON UPDATE CASCADE ON DELETE CASCADE,
21.     FOREIGN KEY (examination_id) REFERENCES Examination_Types
    (examination_id)
22.     ON UPDATE CASCADE ON DELETE CASCADE
23.);

```

The layout for lab tables are identical to the layout of the examination table.

```

1. CREATE TABLE Lab_Categories (
2.     lab_category_id           int,
3.     lab_category_name         varchar(150) NOT NULL,
4.     PRIMARY KEY (lab_category_id)
5. );
6.
7. CREATE TABLE Lab_Types (
8.     lab_id                    varchar(10),
9.     lab_name                   varchar(300) NOT NULL,
10.    lab_category_id           int NOT NULL,
11.    PRIMARY KEY (lab_id),
12.    FOREIGN KEY (lab_category_id) REFERENCES Lab_Categories
    (lab_category_id)
13.    ON UPDATE CASCADE ON DELETE CASCADE
14.);
15.

```



```

16. CREATE TABLE Labs (
17.     patient_id      int,
18.     lab_id           varchar(10) NOT NULL,
19.     value            int NOT NULL,
20.     PRIMARY KEY (patient_id, lab_id),
21.     FOREIGN KEY (patient_id) REFERENCES Patients (patient_id)
22.         ON UPDATE CASCADE ON DELETE CASCADE,
23.     FOREIGN KEY (lab_id) REFERENCES Lab_Types (lab_id)
24.         ON UPDATE CASCADE ON DELETE CASCADE
25.);

```

The layout of diet tables are also very similar. The only difference is that after cleaning data, there is only one category left, so we got rid of the category table.

```

1. CREATE TABLE Diet_Types (
2.     diet_id          varchar(10),
3.     diet_name         varchar(300) NOT NULL,
4.     PRIMARY KEY (diet_id)
5. );
6.
7. CREATE TABLE Diet (
8.     patient_id        int,
9.     diet_id           varchar(10) NOT NULL,
10.    value              int NOT NULL,
11.    PRIMARY KEY (patient_id, diet_id),
12.    FOREIGN KEY (patient_id) REFERENCES Patients (patient_id)
13.        ON UPDATE CASCADE ON DELETE CASCADE,
14.    FOREIGN KEY (diet_id) REFERENCES Diet_Types (diet_id)
15.        ON UPDATE CASCADE ON DELETE CASCADE
16.);

```

ETL Process

We started our ETL process with loading in csv files for all relevant data, and data dictionaries.

We load in information from demographics file, medication file, diet and diet type file, examination and examination_type file, labs and lab type file.

Transform Patients Tables

We did a primary check on whether all the patient ids are in the demographics table, and the answer appears to be yes. But we still join all the table using the patient_id just in case and called it patients_t1. Then we merged the combined table with our demographics table.

```

1. patients_t1 = pd.DataFrame(set(pd.concat(objs = [demographics.SEQN, diet.SEQN,
2. exams.SEQN, labs.SEQN])), columns = ['SEQN'])
3. patients_t2 = pd.merge(left = patients_t1, right = demographics)

```

Then we selected the columns we need and rename them so that our table is more intelligible. We also replace the missing values with '0' to ensure NOT NULL constraints are not violated.

```
1. patients = patients_t2[['SEQN', 'DMDMARTL', 'RIDRETH3', 'DMDHRBR4',
    'INDHHIN2']]
2. patients.rename(columns= {"SEQN": "patient_id",
3.                             "DMDMARTL": "maritalstatus_id",
4.                             "RIDRETH3": "ethnicity_id",
5.                             "DMDHRBR4": "country_id",
6.                             "INDHHIN2": "income_id"}, inplace = True)
7. patients.fillna(0, inplace = True)
```

Transform Medications Tables

We first create the medications table by selecting columns we need, then we dropped duplicated and null values and renamed the columns.

```
1. medications =
    meds[['RXDDRGID', 'RXDDRUG']].drop_duplicates().dropna().reset_index(drop=True).
    rename(columns={"RXDDRGID": "medication_id", "RXDDRUG": "medication_name"})
```

We then moved on to the patient_medications and prescription_reasons tables. First we created an incrementing counter to keep track of whether a given patient has been prescribed the same medication multiple times -- this is the prescription_number attribute.

Then we created the prescription_reason table. We isolated the relevant columns, converted the original table from a wide one to a long format, then we converted the reason number to be integer.

```
1. # Isolate relevant columns and rename id variable
2. prescription_reasons_t1 = med_transform[['SEQN',
    'RXDDRGID', 'RXDRSD1', 'RXDRSD2', 'RXDRSD3',
    'prescription_number']].rename(columns = {'SEQN': 'patient_id', 'RXDDRGID':
    'medication_id', 'RXDRSD1': '1', 'RXDRSD2': '2', 'RXDRSD3': '3'})
3. # Convert from wide to long
4. prescription_reasons_t2 = pd.melt(prescription_reasons_t1,
    id_vars=['patient_id', 'medication_id',
    'prescription_number'], var_name='reason_number',
    value_name='reason_for_prescription')
5. # Convert reason_number column to integer, drop
6. prescription_reasons_t2['reason_number'] =
    prescription_reasons_t2['reason_number'].astype(int)
```

```
7. prescription_reasons =
    prescription_reasons_t2.drop_duplicates(keep='first').dropna()
```

Afterwards, we created the patient_medications table. Similarly, we selected the columns we need, drop null values and rename the columns we chose.

```
1. Patient_Medications_t1 = med_transform[['SEQN', 'RXDDRGID', 'RXDDAYS',
    'prescription_number']]
2. Patient_Medications_t2 = Patient_Medications_t1.dropna().reset_index(drop=True)
3. patient_medications =
    Patient_Medications_t2.rename(columns={'SEQN': 'patient_id', 'RXDDAYS': 'days_take
    n', 'RXDDRGID': 'medication_id'})
```

Transform Examinations Tables

We started by converting the table from wide to long format, and then removing null values. We also removed irrelevant duplicate examinations_id values from the examinations data dictionary.

```
1. examinations = pd.melt(exams, id_vars=['SEQN'], var_name='examination_id',
    value_name='value')
2. examinations = examinations.dropna().reset_index(drop=True)
3. # there are a number of duplicated 'examination_id' rows in our dictionary
4. # these rows arent relevant to our examinations data, but we'll remove them,
    anyway
5. examination_types_t2 =
    examination_types_t1.drop_duplicates(subset='examination_id', keep = 'first')
```

Then we isolate a list of examinations whose definition we need, and join that list with the definition from the dictionary.

```
1. relevant_exams = list(set(examinations.examination_id))
2. relevant_exams = pd.DataFrame(relevant_exams, columns=['examination_id'])
3. examination_types_t3 = pd.merge(left = relevant_exams, right =
    examination_types_t2, how = 'inner')
```

Then we realized that in the examination there are examination_id codes in the examination table that are not in the data dictionary, therefore we'll have to remove them or add them into dictionary manually. In this case, we are going to remove for simplicity.

```
1. key_diff =
    set(examinations.examination_id).difference(examination_types_t3.examination_id
    )
```

```

2. examinations['where_diff'] = examinations.examination_id.isin(key_diff)
3. examinations = examinations[examinations['where_diff'] == False]
4. examinations.drop(columns = 'where_diff', inplace = True)

```

After that we create a table for the broader examination_categories and add id into the table, and merge the examination_types with new examination_category_id column.

```

1. exam_cat = list(set(examination_types_t3['examination_type']))
2. examination_categories = pd.DataFrame(exam_cat, columns =
    ['examination_category_name'])
3. # create id for examination_categories
4. examination_categories['examination_category_id'] =
    range(1,len(examination_categories)+1)
5. # merge examinaion_types with new examination_category_id column
6. examination_types = pd.merge(left = examination_types_t3, left_on =
    'examination_type', right = examination_categories, right_on =
    'examination_category_name', how = 'inner')
7. examination_types.drop(columns = ['examination_type',
    'examination_category_name'], inplace = True)

```

Transform Lab Tables

We went through a very similar process for the Labs tables. First we selected needed variables, removed null and duplicate values.

```

1. labs = pd.melt(labs,id_vars=['SEQN'],var_name='lab_id', value_name='value')
2. labs = labs.dropna().reset_index(drop=True)
3. lab_types_t2 = lab_types_t1.drop_duplicates(subset='lab_id', keep = 'first')

```

Then we isolate a list of labs whose definition we need, and join that list with the definition from the dictionary.

```

1. relevant_labs = list(set(labs.lab_id))
2. relevant_labs = pd.DataFrame(relevant_labs, columns=['lab_id'])
3. lab_types_t3 = pd.merge(left = relevant_labs, right = lab_types_t2, how =
    'inner')

```

Similarly, in the labs there are lab_id codes in the lab table that are not in the data dictionary, therefore we'll have to remove them or add them into dictionary manually. In this case, we choose to remove them for simplicity.

```

1. key_diff = set(labs.lab_id).difference(lab_types_t3.lab_id)
2. labs['where_diff'] = labs.lab_id.isin(key_diff)

```

```
3. labs = labs[labs['where_diff'] == False]
4. labs.drop(columns = 'where_diff', inplace = True)
```

After that we create a table for the broader lab_categories and add id into the table, then merge the lab_types with new lab_category_id column.

```
1. lab_cat = list(set(lab_types_t3['lab_category']))
2. lab_categories = pd.DataFrame(lab_cat, columns = ['lab_category_name'])
3. lab_categories['lab_category_id'] = range(1, len(lab_categories)+1)
4. lab_types = pd.merge(left = lab_types_t3, left_on = 'lab_category', right =
    lab_categories, right_on = 'lab_category_name', how = 'inner')
5. lab_types.drop(columns = ['lab_category', 'lab_category_name'], inplace = True)
```

Transform Diet Tables

In a similar pattern, first we selected needed variables, removed null and duplicate values.

```
1. diet = pd.melt(diet, id_vars=['SEQN'], var_name='diet_id', value_name='value')
2. diet = diet.dropna().reset_index(drop=True)
3.
4. diet_types_t2 = diet_types_t1.drop_duplicates(subset='diet_id', keep = 'first')
```

Then we isolate a list of diets whose definition we need, and join that list with the definition from the dictionary.

```
1. relevant_diet = list(set(diet.diet_id))
2. relevant_diet = pd.DataFrame(relevant_diet, columns=['diet_id'])
3. diet_types_t3 = pd.merge(left = relevant_diet, right = diet_types_t2, how =
    'inner')
```

In the diets there are diet_id codes in the diet table that are not in the data dictionary, therefore we'll have to remove them or add them into dictionary manually. In this case, we chose to remove them for simplicity.

```
1. key_diff = set(diet.diet_id).difference(diet_types_t3.diet_id)
2. diet['where_diff'] = diet.diet_id.isin(key_diff)
3. diet = diet[diet['where_diff'] == False]
4. diet.drop(columns = 'where_diff', inplace = True)
```

After that we create a table for the broader diet_categories and add id into the table, then merge the diet_types with new diet_category_id column.

```
1. diet_cat = list(set(diet_types_t3['diet_category']))
```

```

2. diet_categories = pd.DataFrame(diet_cat, columns = ['diet_category_name'])
3. diet_categories['diet_category_id'] = range(1,len(diet_categories)+1)
4. diet_types = pd.merge(left = diet_types_t3, left_on = 'diet_category', right =
    diet_categories, right_on = 'diet_category_name', how = 'inner')
5. diet_types.drop(columns = ['diet_category', 'diet_category_name'], inplace =
    True)

```

However, we found that there is only one diet_category left in our diet table, after cleaning, hence there is no need for the diet_category table. We also dropped the diet_category_id column, which correlates to the diet_category table.

```

1. diet_types.drop(columns=['diet_category_id'], inplace = True)

```

Data Analysis

Data

A GitHub repository of our analytic sql queries can be found [here](#).

1. How many patients are being treated for Type 2 Diabetes?

```

1. SELECT COUNT(patient_id) AS diabetes_patients, (SELECT COUNT(patient_id) FROM
    patients) AS all_patients
2. FROM patients p
3. WHERE patient_id IN ( SELECT patient_id
4. FROM prescription_reasons
5. WHERE reason_for_prescription ILIKE '%Type 2 Diabetes%'
    );

```

2. What are the most commonly prescribed medications for patients with Type 2 Diabetes?

```

1. SELECT *
2. FROM (SELECT medication_id, COUNT(*)
3. FROM prescription_reasons p
4. WHERE reason_for_prescription ILIKE '%Type 2 Diabetes%'
5. GROUP BY medication_id) AS prescription_count
6. NATURAL JOIN medications
7. ORDER BY count DESC

```

3. What is the average mg of Sodium (diet_types: DR1TSODI) consumed by patients with Type 2 Diabetes?

```
1. SELECT AVG(value)
2. FROM diet
3. WHERE diet_id = 'DR1TSODI'
4. AND patient_id IN (SELECT patient_id
5.                     FROM prescription_reasons
6.                     WHERE reason_for_prescription ILIKE '%Type 2 Diabetes%');
```

4. What percent of patients are married/divorced/separated etc. ?

```
1. select maritalstatus_value, count(patient_id)/((select count(*) from
   patients)/100) as percentage
2. from maritalstatus_lookup m left join patients p on
   m.maritalstatus_id=p.maritalstatus_id
3. where m.maritalstatus_id=3 or m.maritalstatus_id=4 or m.maritalstatus_id=1
4. group by m.maritalstatus_id
```

5. What's the ethnicity of patients who have diabetes?

```
1. select el.ethnicity_value, count(el.ethnicity_id)
2. from ethnicity_lookup el join patients on el.ethnicity_id=
   patients.ethnicity_id
3. join prescription_reasons pr on patients.patient_id= pr.patient_id
4. where reason_for_prescription ilike '%type 2 diabetes%'
5. group by el.ethnicity_id
```

6. What are the names and values of all examination information for patient 75262?

```
1. SELECT examination_id, examination_name, value
2. FROM examinations NATURAL JOIN examination_types
3. WHERE patient_id = 75262
```

7. For the 2nd most prescribed medication in the dataset, what are the top reasons for prescription?

```

1. SELECT reason_for_prescription, COUNT(*)
2. FROM prescription_reasons
3. WHERE medication_id = (SELECT medication_id
4.                        FROM (SELECT medication_id,
5.                                RANK () OVER (
6.                                    ORDER BY c.med_count DESC) as top_med
7.                                FROM (SELECT medication_id, COUNT(*) AS
med_count
8.                                FROM patient_medications pm
9.                                GROUP BY medication_id) AS c) AS rr
10.                        WHERE rr.top_med = 2)
11. GROUP BY reason_for_prescription
12. ORDER BY count DESC
13. LIMIT 5;

```

8. For those patients who have taken medicine for 100 days, how many are married?

```

1. select count(*) from patient_medications m left join patients p on
m.patient_id=p.patient_id
2. where m.days_taken>100 and p.maritalstatus_id=1

```

9. What is the income level for patients who are being treated for diabetes?

```

1. select p.patient_id,i.income_value
2. from income_lookup as i left join patients as p on i.income_id=p.income_id
3. where i.income_value!='Missing' and p.patient_id in (select patient_id from
prescription_reasons where reason_for_prescription LIKE 'Type 2 diabetes' )
4. order by i.income_id

```

10. What are the top 10 labs patients who developed diabetes conducted?

```

1. select count(t.lab_name) count,t.lab_name
2. from lab_types t left join labs l on t.lab_id=l.lab_id
3. where l.patient_id in (select patient_id from prescription_reasons where
reason_for_prescription ilike '%type 2 diabetes%')
4. group by t.lab_name

```



```
5. order by count desc limit 10
```

Interacting with our database

Our primary goal for the database is to make it more intelligible for both technical and non-technical people to access the health data. The whole dataset has been normalized and missing, duplicate, and unnecessary data has been mitigated, as well. We want to make sure that individuals can interact with our dataset effectively and accurately. To this end, we have also added lookup tables and included the relevant data dictionaries as elements in the database, in order to streamline the process of interacting with the information.

We expect technical users to interact with our database directly using SQL code. As it currently stands, these data were collected over the course of a single year, and this study has concluded - no new data are forthcoming. It is therefore not necessary to implement processes for the inclusion of new data into the database. However, envisioning a situation in which a new phase of data collection took place, a separate operational database would be created in order to provide analysts with a safe way to interact with the information.

When even more advanced analytics are required, analysts will use higher-level analytic programming languages, such as Python and R, to interact with the information. Analysts will implement SQL code in these language environments in order to perform the database operations necessary to extract information from the database into local memory (or a distributed computing environment, such as spark). Using a Python or R (or scala) environment to execute SQL code allows for the best of both worlds, the efficient database operations made possible through SQL, and the more robust analytic capabilities offered by Python/R.

Less technical users, such as C-suite executives needing high level overviews of the information, will likely interact with the database through the use of business intelligence platforms, such as the Metabase dashboards demonstrated below. The new database structure is designed to interact favorably with these BI tools, and powerful analyses are possible. Technical employees will likely be charged with designing and building dashboards, as requested by upper level management.

We organized the medication data, as well as the prescription data, therefore it is easier for analysts to draw correlations between medications and prescription information. Technical users could directly write SQL queries, then use those queries to obtain data they need. For later analysis, they could interact with Python packages to generate optimal results. From there, they could use the metabase to visualize the analysis for members or researches in the National Center for Health Statistics to view.

We implemented our analysis with Python, since Python is efficient and suitable for large database analysis. Since SQL on its own performs really well with update or retrieving data, but it has a lot of limitations when it comes to data analysis. In contrast, Programming languages like Python and R have more features and functions for data analysis and visualization, hence when we combine the two, we are able to apply complicated analysis to the database, while also experiencing the convenience of SQL queries. Non-technical personeels can interact with our database with Business Intelligence tools such as Tableau.

Redundancy and Performance

For the redundancy, our database is built based on the 3NF which largely eliminate the redundant and useless data. Normalization process is the key step for our project, since we are able to reduce the complexity of the data and set essential relationship between each table with foreign keys and make sure the logic and dependency all make sense. We spend a long time to revise our tables with correlative columns and data, and follow the rules to reach out the Third Normal Form.

Host On-Premises or On the Cloud

Our client host should on the cloud:

Firstly, it's safer and more efficient. Moving from local disks to the cloud can avoid repeated backup, and increase the data security level. Especially some cloud developing platforms have created handful security management tools, which is easy to use.

Secondly, working on the cloud can scale up and scale down, and only need to pay for the resources they used, which can help save money and resources. For example, there is no need to buy expensive hardware to build the data infrastructure, and the resources can be saved to develop analytics capabilities.

Finally, the client can access to other data for training models and developing our analytics tools. For example, if the host kept on-premises, the client can only use local data resources, but if it's on the cloud, the client may be able to use global data resources to train model and develop BI tools.

Metabase Dashboards

We've created 3 metabase dashboards which help demonstrate the power of our new database schema.

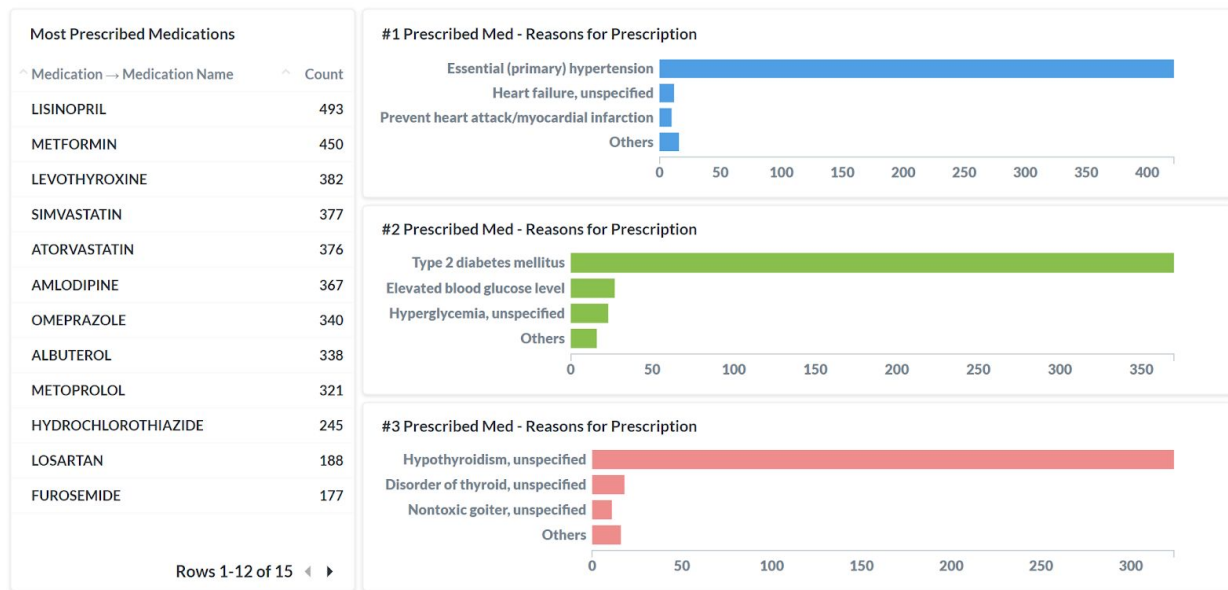
1. [Demographics Dashboard](#)

From the Demographics dashboard, we have an overall review of the demographic composition of our dataset. This information is important in understanding how balanced the dataset is, and determining whether a given analysis is feasible given the sample.



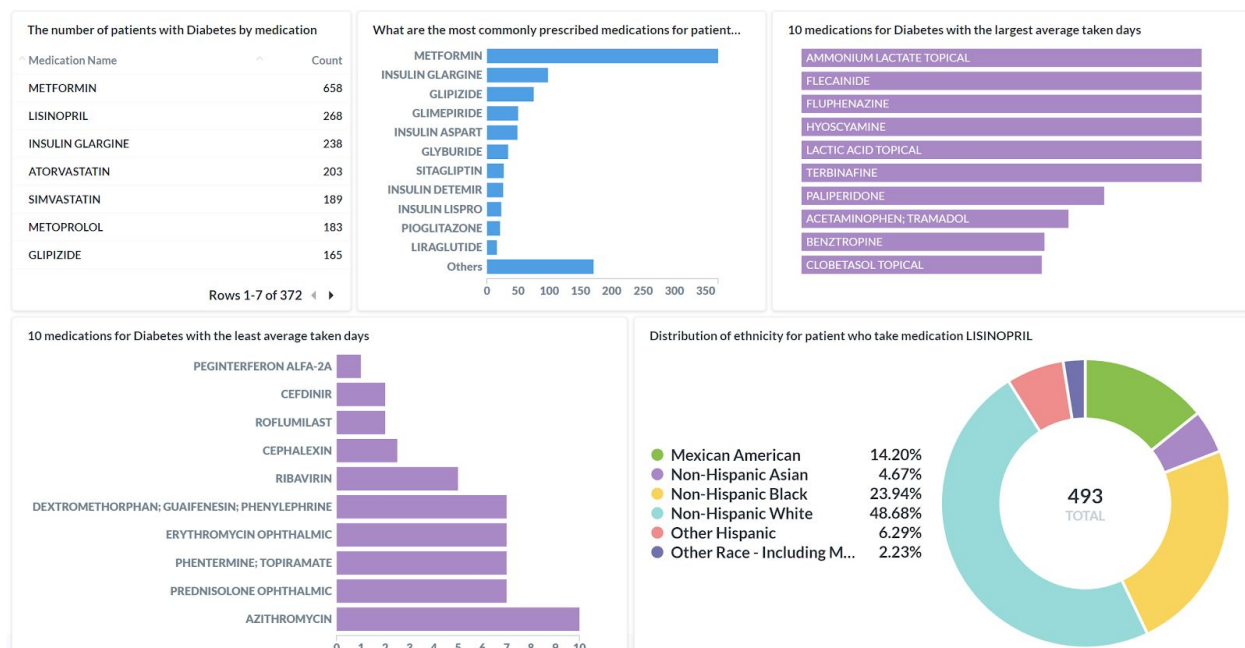
2. [Prescription Activity](#)

For the prescription activity dashboard, we've demonstrated how this new structure allows for an analysis of medications according to their subscription frequency. Additionally the top three subscribed medications are listed on the right with their reasons for subscription. This information would be useful for an analysis of prescription behavior, and could be filtered by additional dimensions for a more in-depth look.



3. [Medications](#)

The medications dashboard demonstrates how a more in-depth analysis of a specific subset of medications or conditions is possible. This dashboard in particular focuses on prescription behavior among patients suffering from diabetes. This view of the data already raises interesting questions about why certain medications might be prescribed over others in different situations.



Conclusion

The NHANES database at the center of this project represents a comprehensive and powerful resource for research and analysis in the medical field. The structure in which it was originally organized, however, made it extremely difficult to interact with in an analytical capacity. Even the process of understanding what information was available for a given patient was an undertaking. Furthermore, the database contained no internal mechanism for interpreting the hundreds of unintelligible variable codes, or categorizing them into relevant groupings. An analyst working in this structure would be forced to constantly reference external dictionaries to assign meaning, gather information tediously from numerous unlinked tables, and perform complex data manipulations in order to achieve meaningful aggregations of data.

Our restructuring of the schema associated with this information has resolved these issues, and ultimately made these data more accessible, more easily understandable, and as a result, more ready for both high-level summarizations and involved technical analysis. Our more technical users will find that this structure enables a streamlined assessment of what information is available for a given subset of patients, and an easier path for extracting this information into a format that makes sense for deeper analysis. Our less technical users will find that the database itself now provides meaningful tools for categorizing, understanding, and making sense of the information contained within.

Despite all the work done to get the database to this point, there are yet more opportunities to continue improving and expanding on this structure. Primarily, both the questionnaire table and a large subset of the available demographic information were omitted from this project in order to keep our scope in a manageable place. The analytic potential of the database would be further enhanced by the inclusion of these data at a later time.

Nevertheless, our updated schema as it currently stands has already opened up interesting analytic pathways within our own group. We are excited about the potential for this revised database schema to streamline medical analysis of these data, and look forward to seeing how this tool is used in the future.