Identification of Patterns in Stroke Care Transitions using OHDSI Pharmetrics+ Data

DS 5110
Alex Corcoran, Jaee Oh,
Sally Johnstone

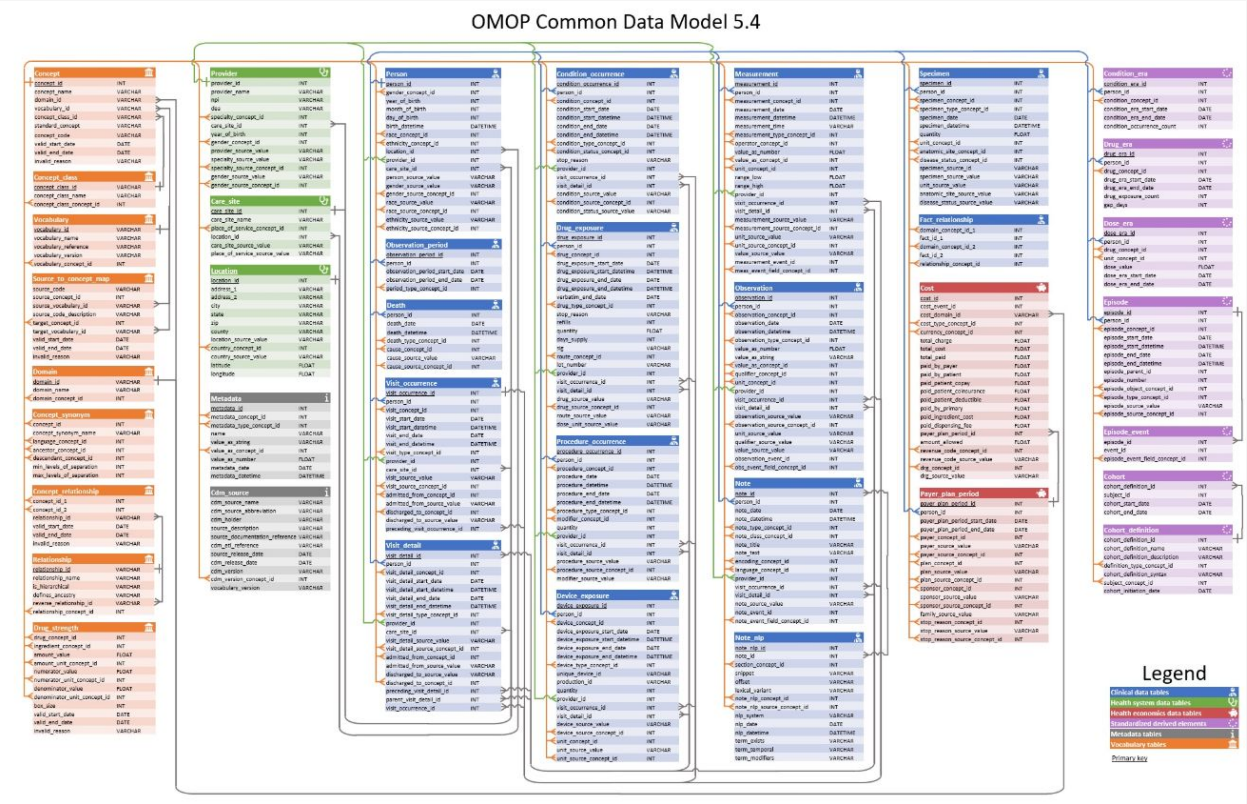
December 6, 2024

OHDSI
OBSERVATIONAL HEALTH DATA SCIENCES AND INFORMATICS

1. Accessing the data
2. An AGILE approach
3. Delivering an interim step

Accessing the data

OHDSI database is complicated

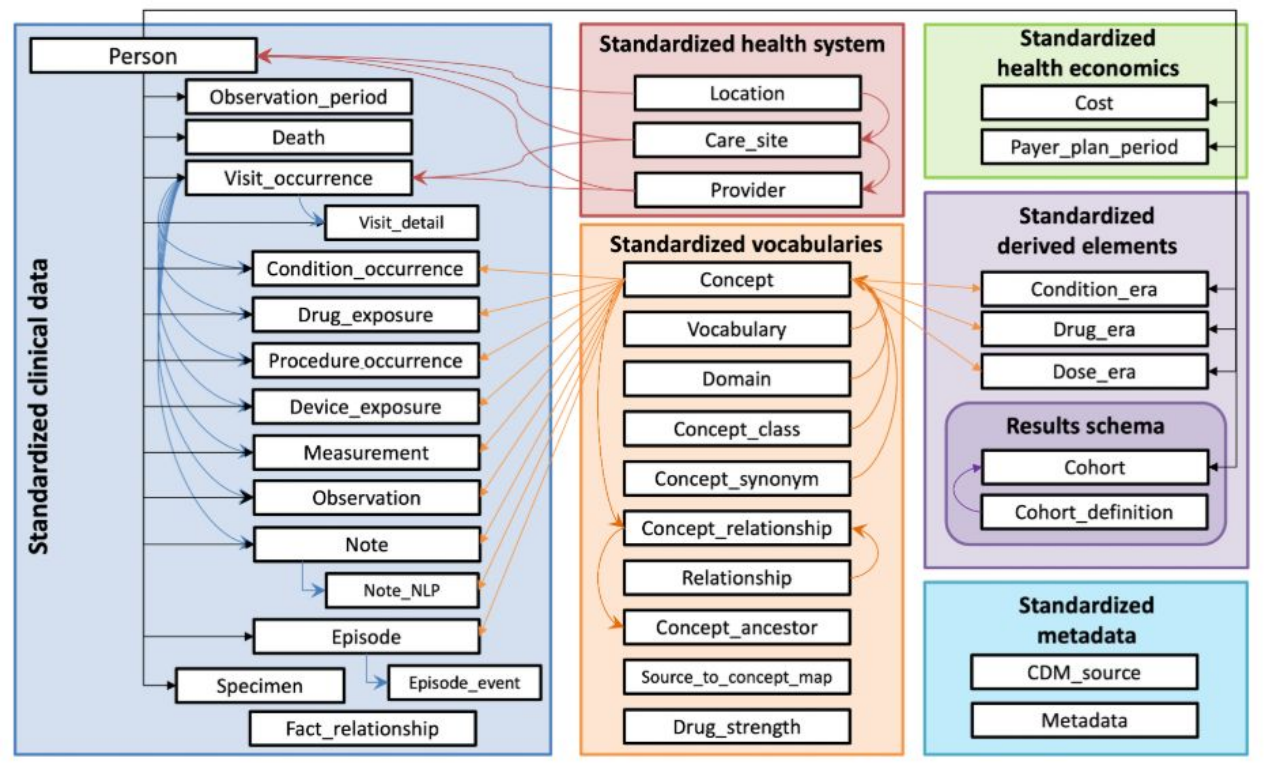


– Refer to OMOP Common Data Model github for details.

– Some columns are empty.

– Originally done with R

OHDSI database is complicated



– Refer to OMOP Common Data Model github for details.

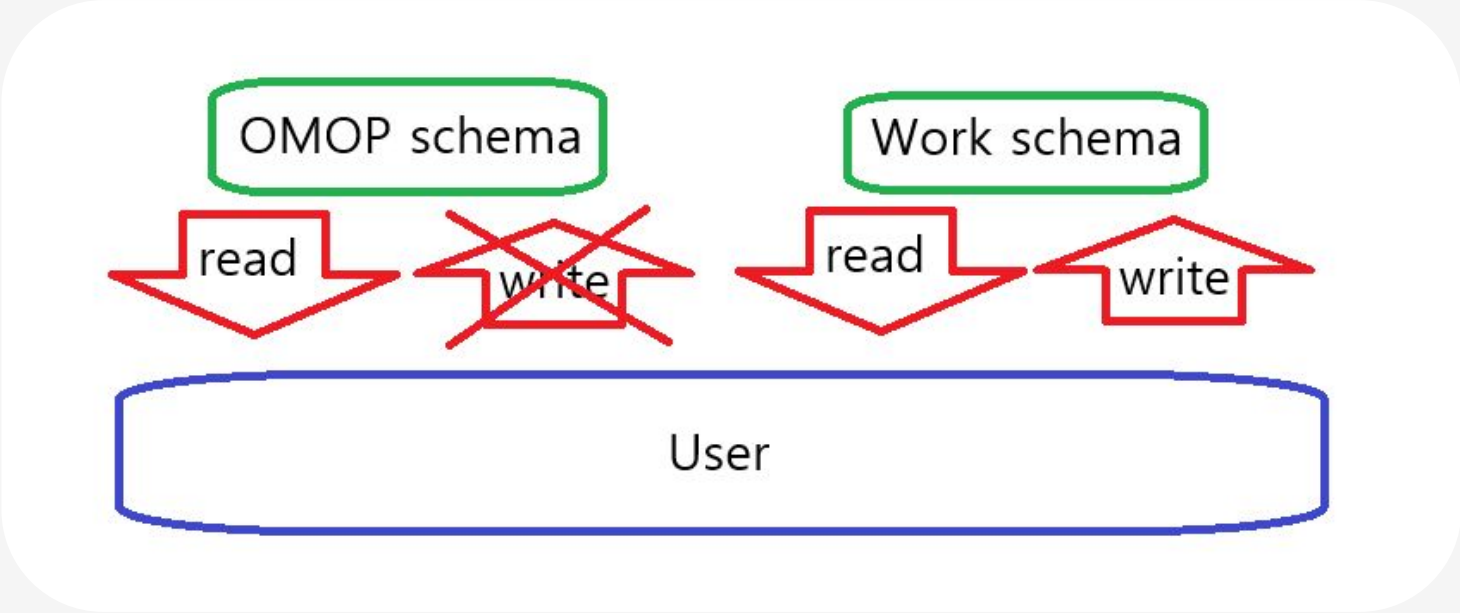
– Some columns are empty.

– Originally done with R

Good News

OMOP means the data has been cleaned and person_id has the same structure and format across tables. That's half the battle won, because real world databases are rarely so clean.

Concept of Schema



OMOP schema: Original tables. Only Read.

Work schema: Individual tables. Read and Write.

Rule of Thumb

If you are handling large data:

– Use SQL query directly to the database.

– Faster, but SQL query can be hard to read very quickly.

If you are handling small data:

– Use Pandas package

– Slower, due to converting tables into dataframes, but more intuitive and readable.

– You can even plot them or use machine learning!

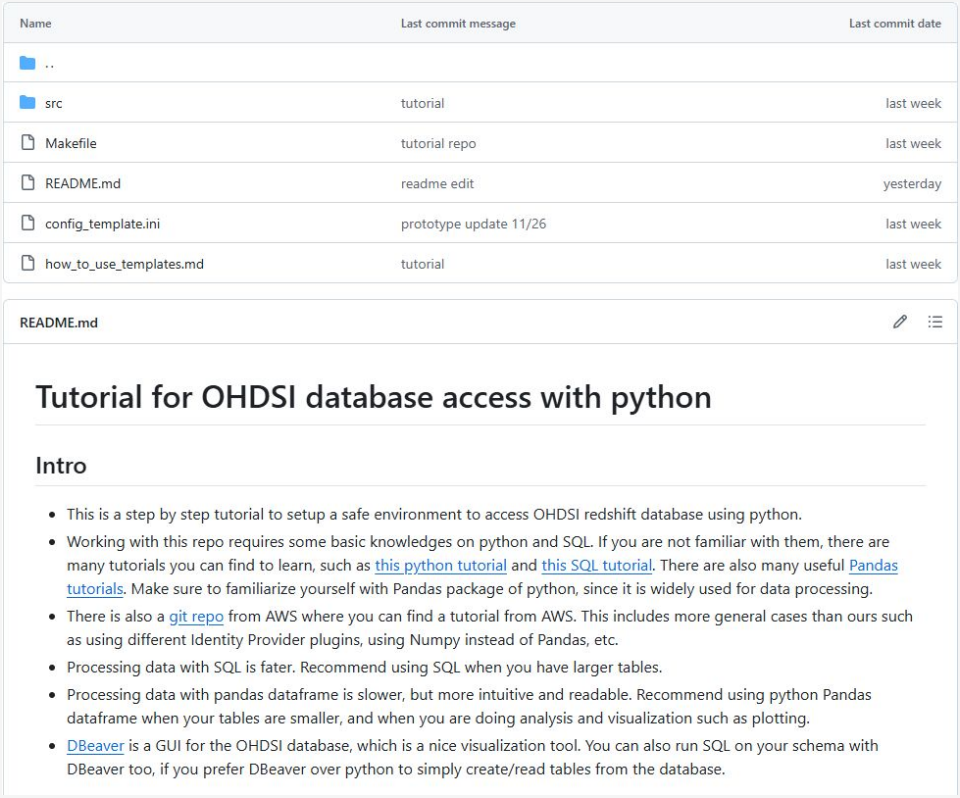
Reproducibility

– Unfortunately, AWS is setup as Windows Server 2019, and WSL cannot be installed.

– Use Anaconda Powershell Prompt as if it is your terminal from Linux.

– Utilize Make. If you setup properly, everything in our repo can be reproduced by a few make commands.

Tutorial of OHDSI for Python Users



Name	Last commit message	Last commit date
..		
src	tutorial	last week
Makefile	tutorial repo	last week
README.md	readme edit	yesterday
config_template.ini	prototype update 11/26	last week
how_to_use_templates.md	tutorial	last week

README.md

Tutorial for OHDSI database access with python

Intro

- This is a step by step tutorial to setup a safe environment to access OHDSI redshift database using python.
- Working with this repo requires some basic knowledges on python and SQL. If you are not familiar with them, there are many tutorials you can find to learn, such as this python tutorial and this SQL tutorial. There are also many useful Pandas tutorials. Make sure to familiarize yourself with Pandas package of python, since it is widely used for data processing.
- There is also a git repo from AWS where you can find a tutorial from AWS. This includes more general cases than ours such as using different Identity Provider plugins, using Numpy instead of Pandas, etc.
- Processing data with SQL is fater. Recommend using SQL when you have larger tables.
- Processing data with pandas dataframe is slower, but more intuitive and readable. Recommend using python Pandas dataframe when your tables are smaller, and when you are doing analysis and visualization such as plotting.
- DBeaver is a GUI for the OHDSI database, which is a nice visualization tool. You can also run SQL on your schema with DBeaver too, if you prefer DBeaver over python to simply create/read tables from the database.

– Tutorial folder within repo.

– Links to certification, AWS github, python, sql, etc.

– Guide for ‘configparser’ and ‘redshift_connector’

Built-in Function: config()

```
# Configuration setup.
def config():
    # Read the config.ini file.
    config = configparser.ConfigParser()
    config.read("config.ini")

    # Connect to Redshift using your credentials.
    con = redshift_connector.connect(
        host=config["redshift"]["host"],
        database=config["redshift"]["database"],
        port=int(config["redshift"]["port"]),
        user=config["redshift"]["user"],
        password=config["redshift"]["password"],
        # timeout=60,
    )
    work_schema = config["redshift"]["schema"]
    print(f"Connection is created. Your work schema is '{work_schema}'")
    return con, work_schema
```

Create connection using ‘config.ini’:

```
[redshift]
host=<endpoint>
database=<databae>
port=<port>
user=<redshift user>
passwor=<redshift password>
schema=<your works schema>
```

Info above is given by email from OHDSI Lab Admin <ohdsilab@northeastern.edu>

Built-in Function: run_query()

```
# Run SQL query.
def run_query(con, query):
    # Create a cursor.
    cursor = con.cursor()
    try:
        # Execute a query.
        cursor.execute(query)
    except Exception as e:
        print(f"Error: {e}")
    else:
        # Print executed query.
        print(f"Executed query: \n {query}")
        # Commit the changes to the database.
        con.commit()
    finally:
        # Close the connection even if error occurs.
        con.close()
        print("Connection is closed.")
    return
```

Run a SQL query.

Input: connection, SQL query

Output: print()

Built-in Function: write_df()

```python
# Write pandas dataframe into your schema.
def write_df(con, df, work_schema, table):
    # Create a cursor.
    cursor = con.cursor()
    try:
        # Write the pandas dataframe to a table in your schema. Table must already exists.
        cursor.write_dataframe(df, f"{work_schema}.{table}")
    except Exception as e:
        print(f"Error: {e}")
    else:
        # Commit the changes to the database.
        con.commit()
    finally:
        # Close the connection even if error occurs.
        con.close()
        print("Connection is closed.")
    return
```

Write a df into work schema.

Input: connection, SQL query, work schema name, table name

Output: print()

Built-in Function: read_df()

```python
# Read a table and convert it to a pandas dataframe.
def read_df(con, query):
    # Create a cursor.
    cursor = con.cursor()
    try:
        # Execute a query.
        cursor.execute(query)
    except Exception as e:
        print(f"Error: {e}")
    else:
        # Print executed query.
        print(f"Executed query: \n {query}")
        # Create a pandas dataframe of the table read.
        df = cursor.fetch_dataframe()
        # Commit the changes to the database.
        con.commit()
        return df
    finally:
        # Close the connection even if error occurs.
        con.close()
        print("Connection is closed.")
```

SQL table **read** as df.

Input: connection, SQL query

Output: print(), pandas dataframe

Example with Built-in Functions

Use **f-string** to write SQL query.

```
from utils import config, read_df, write_df, run_query
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


con, work_schema = config()
omop_schema = "omop_cdm_53_pmtx_202203"
omop_table = "concept"
# work_table = "stroke_cohort_w_aphasia"
query = f"""
SELECT *
FROM {omop_schema}.{omop_table}
LIMIT 5
"""
# run_query(con, query)
df = read_df(con, query)

print(df.info())
print(df)
```

Interactive make commands: read_table

Make **read_table**

You can input table name to read.

```
from utils import config, read_df, write_df, run_query
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


con, work_schema = config()
omop_schema = "omop_cdm_53_pmtx_202203"
omop_table = "concept"
print("Enter the table to read info: ")
work_table = input()
query = f"""
SELECT *
FROM {work_schema}.{work_table}
;
"""
df = read_df(con, query)
print(df.info())
print(df)
```

Interactive make commands: drop_table

Make **drop_table**

You can input table name to drop.

```
from utils import config, read_df, write_df, run_query
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


con, work_schema = config()
omop_schema = "omop_cdm_53_pmtx_202203"
omop_table = "concept"
print("Enter the table to drop: ")
work_table = input()
query = f"""
DROP TABLE {work_schema}.{work_table}
"""
run_query(con, query)
```

Confidentiality

NEVER create local files containing sensitive data. Use your work schema instead.

NEVER put your credential file on a public repo (.gitignore them).

ALWAYS work on AWS, when direct access to the database is required.

An AGILE approach

Waterfall project management

Requirements

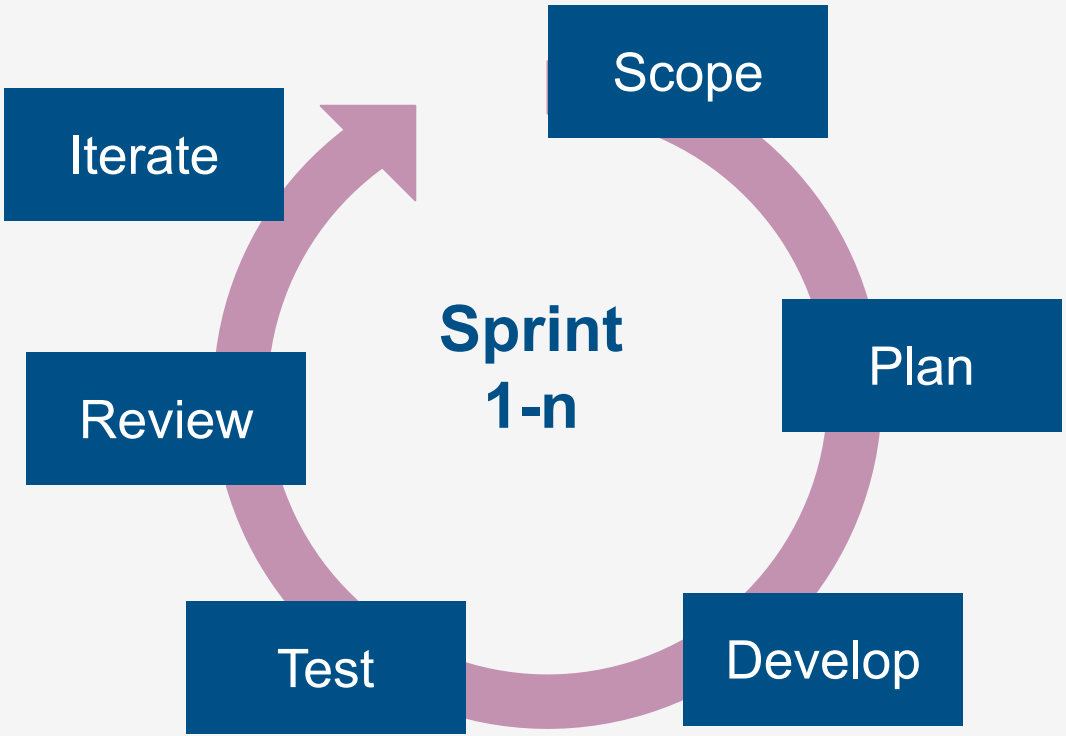
Planning

Execution

Testing

Deployment

AGILE development



Why AGILE?

#1 Create a cohort

#2 Track patient locations

#3 Append therapies?

#4 Potentially 147 different cohorts!

#5 Reduced scope

#6 Final deliverable confirmed

Final deliverable: Interim step

Stroke patients with aphasia can be identified

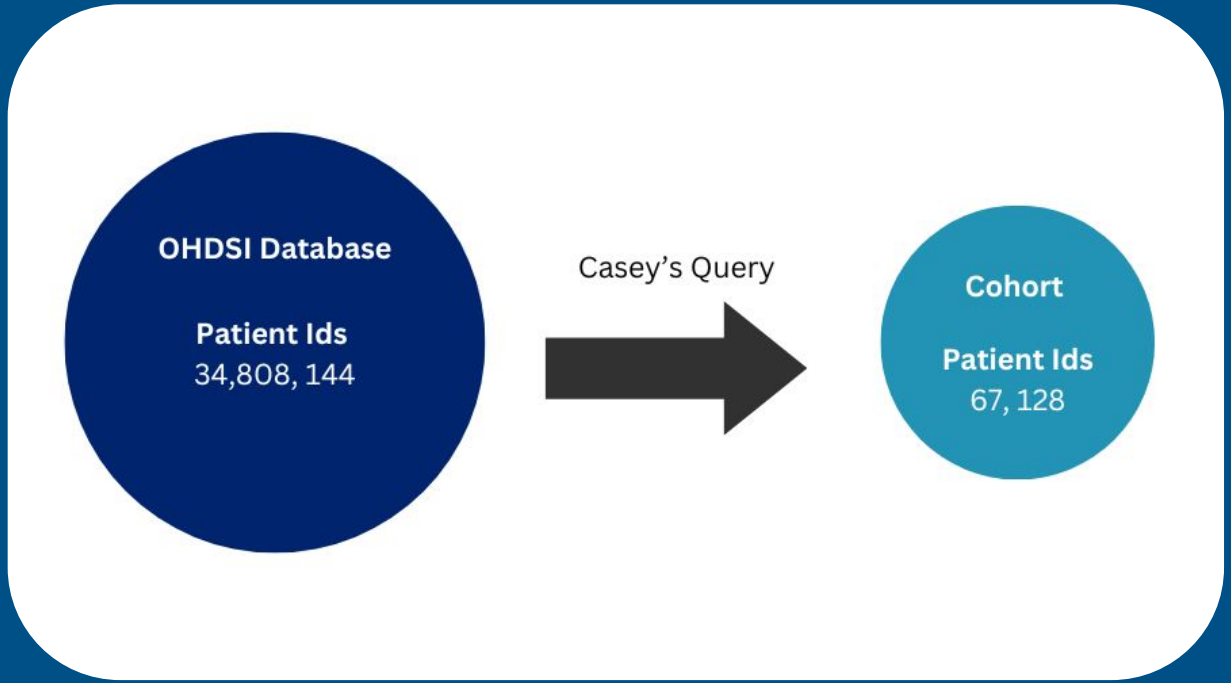
Different locations, with dates attached, can be tracked for individual patients

Speech therapy procedure occurrences, with dates, can be appended to patient records

Delivering an interim step

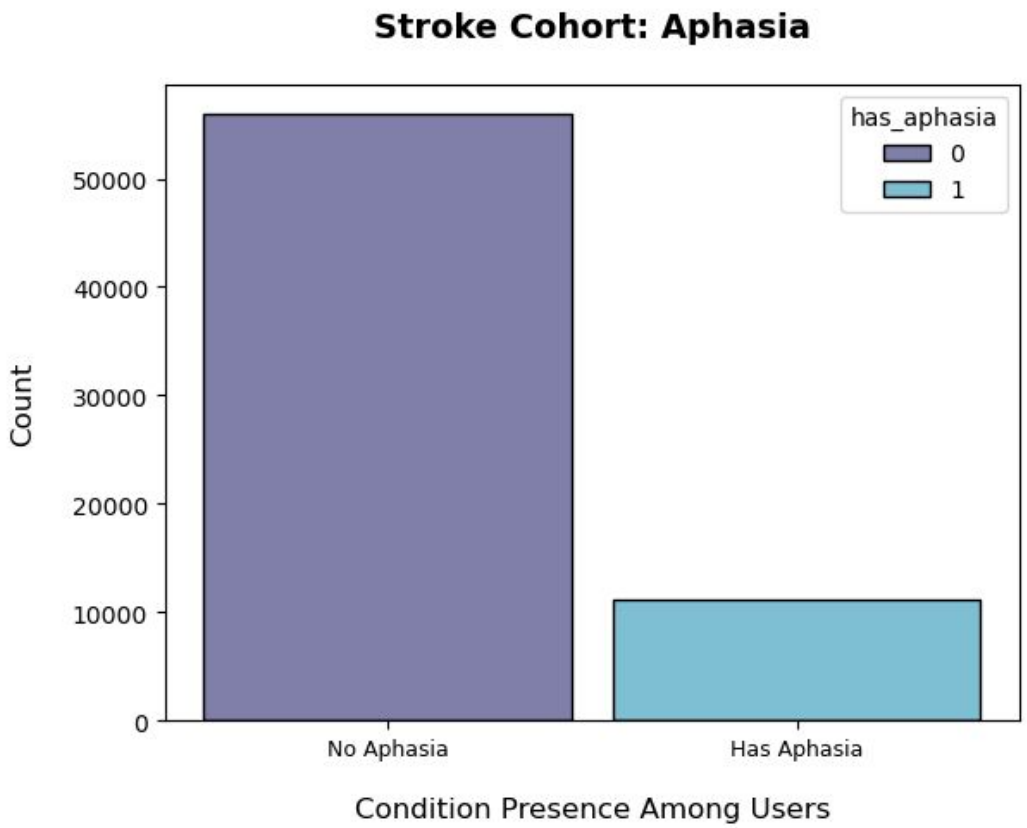
1. SQL Query to Create Cohort

- We used Casey Tilton's cohort query to cut the database down to size
- Casey worked with Rob, our stakeholder, on selecting pertinent stroke codes
- Allowed us to focus on care pathways after the first stroke occurrence



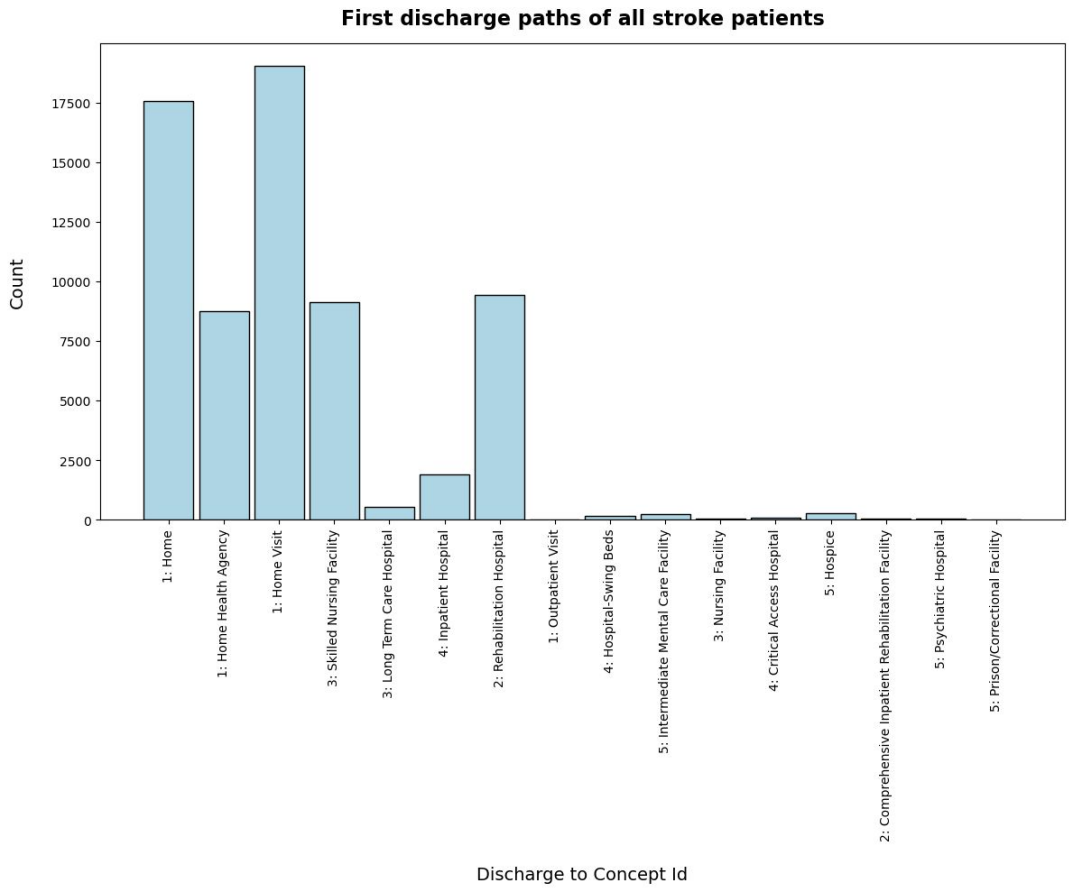
2. Append Aphasia Diagnosis

- A flag was added for aphasia, which is a particular focus for our stakeholder
- Aphasia is a language disorder than can occur after a stroke



3.1 First Discharge Path for All Cohort Patients

- We mapped each of the discharge_to_concept_ids to a concept name

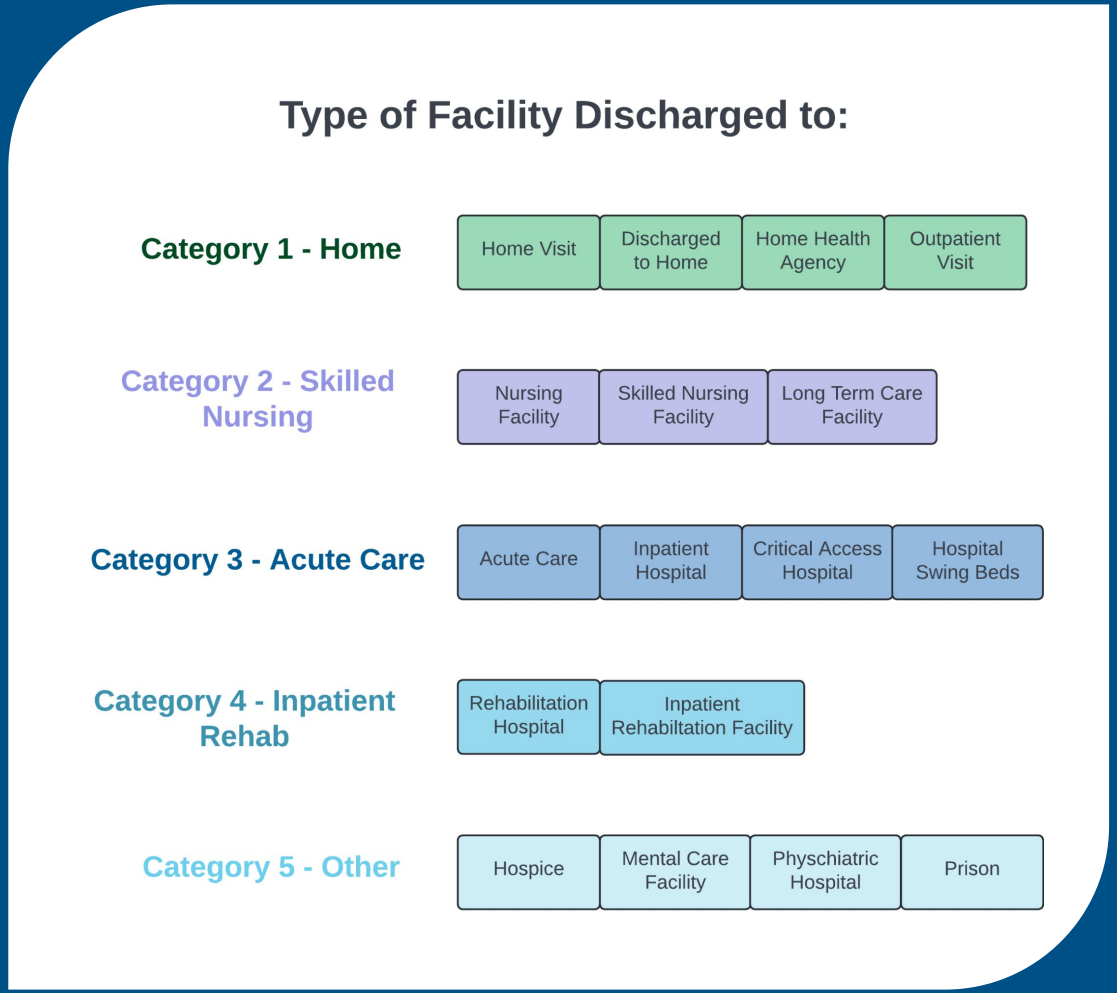


3.2 Categories for Discharge Facilities

- 17 distinct discharge_to_concept_ids:

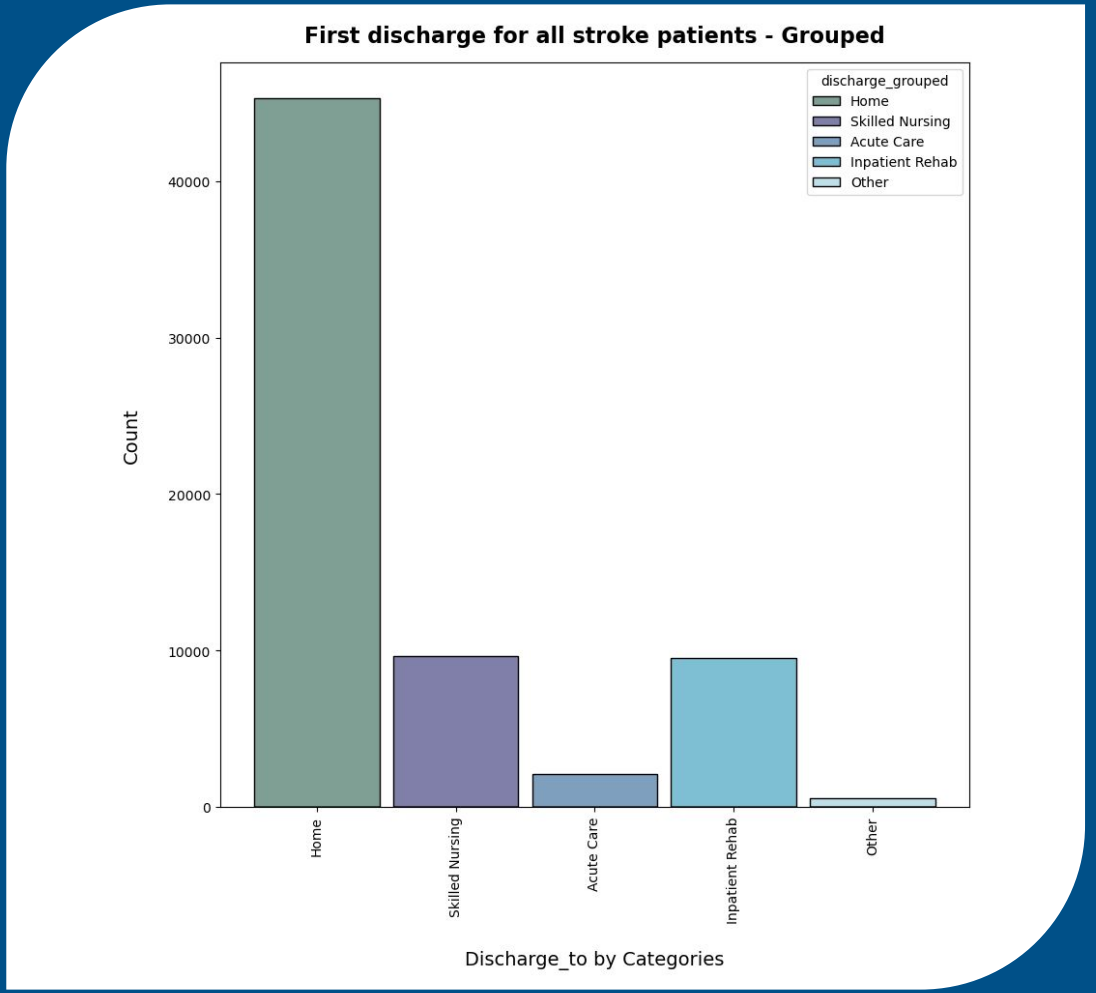
Discharge_to_concept_id / Concept_name	
0	Home
38004519	Home Health Agency

- Further grouped into 5 different categories specified by Rob and Casey's model to more easily see the trends when plotting

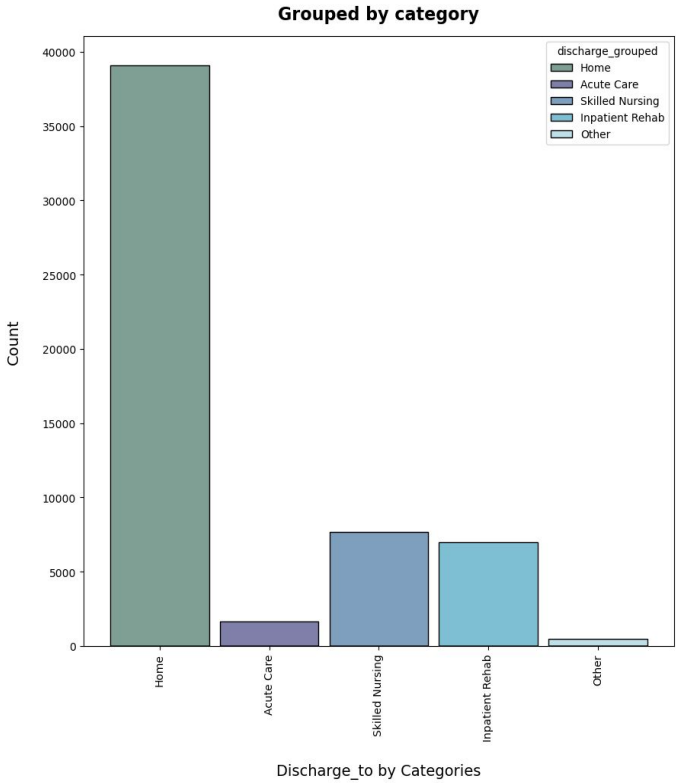


3.3 First Discharge Path for All Cohort Patients

- Grouped by category:
  - Home
  - Skilled Nursing
  - Acute Care
  - Inpatient Rehab
  - Other

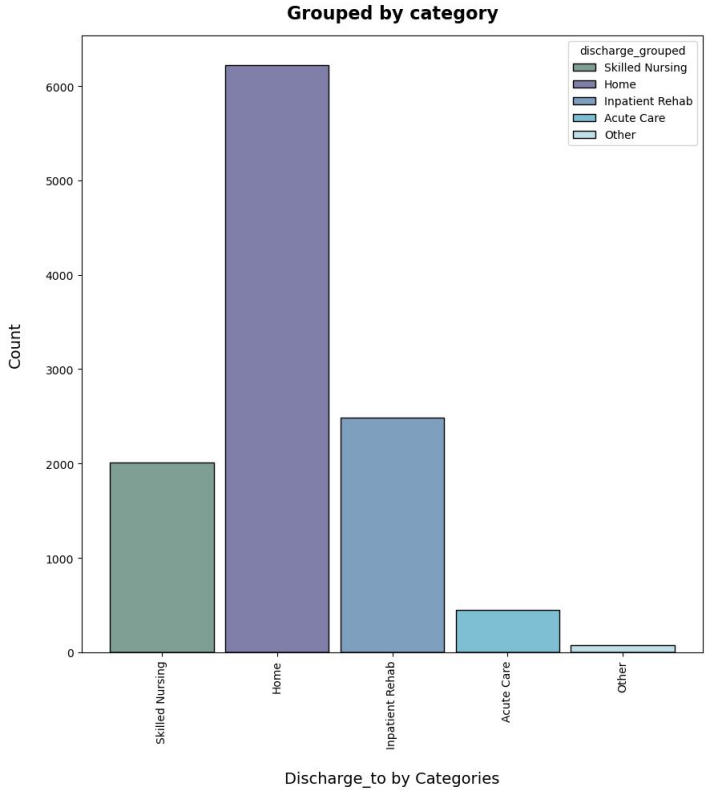


3.3 No Aphasia



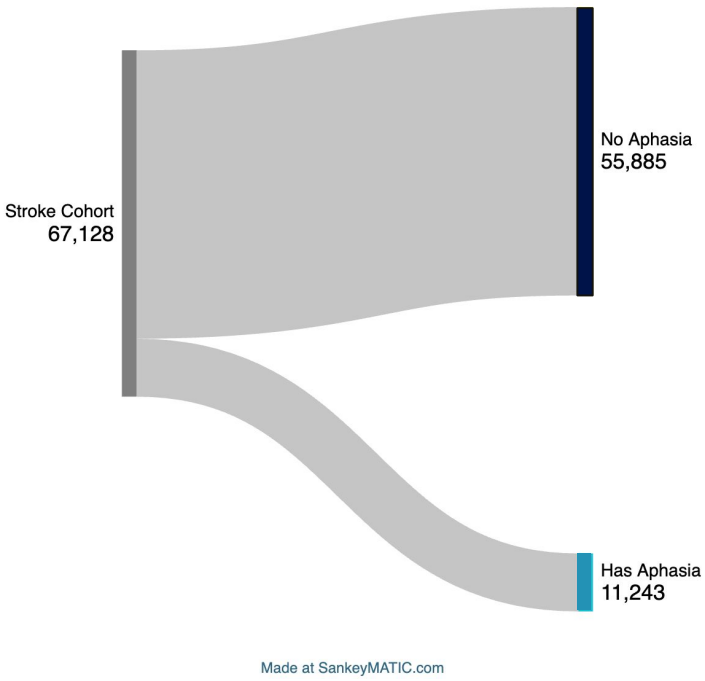
Home > Skilled Nursing > Inpatient Rehab > Acute Care > Other

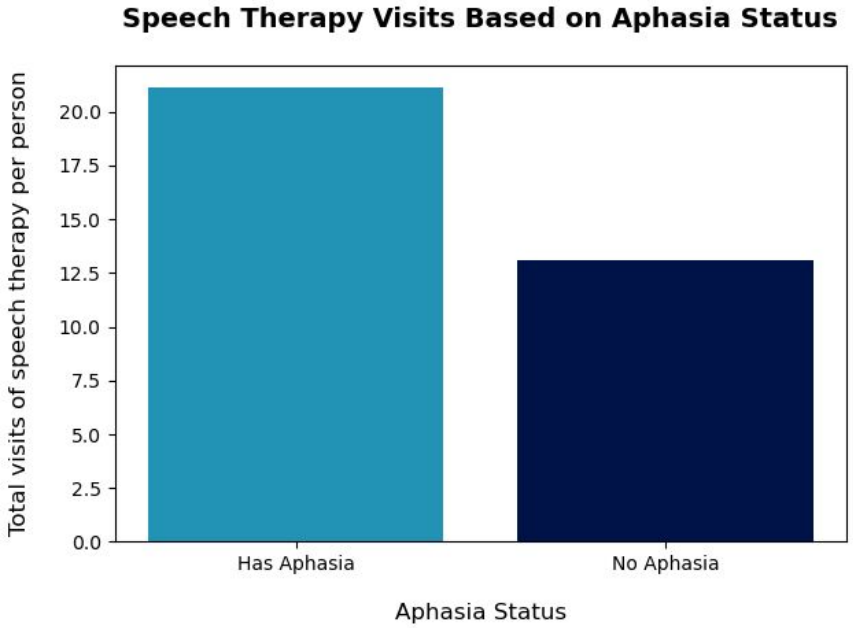
3.4 Aphasia



Home > Inpatient Rehab > Skilled Nursing > Acute Care > Other

5. Speech Therapy





Questions