LACDR Leiden

FAERS Database Pipeline Manual

2025

Contents

Chapter

1 Using this manual

This manual should help you, the reader, with deploying, maintaining, understanding and running the FAERS Database Pipeline. Sections 2 and 3 should help with deploying the pipeline. Section 4 provides some extra technical information. Section 5 provides an simple overview of the pipeline. Section 6 should help with starting to run the beginning of the pipeline. Sections 8,9,10 and 11.1 should help with understanding and maintaining the second part of the pipeline. Finally, 12 should help with maintainability by providing an overview of the unit tests.

Chapter

2 LACDR VM Deployment

2.1 Prerequisites

- Python 3+ installed
- PostgreSQL installed (see Chapter 1)

For more details we refer to Chapter 4.

2.2 Deployment

1. Clone the faers-scripts repository from github.
2. Make sure the faers-data directory is in the same directory as the faers-scripts directory. It should not be within faers-scripts.
3. Make sure the IP of the machine you are deploying on is whitelisted in Google Cloud (see Chapter 12)

2.3 Updating MedDRA

MedDRA_25_1 already comes with the faers-data directory. However when a new MedDRA version is required follow these steps:

1. Download the new MedDRA data directory
2. Place the new directory in the faers-data directory

3. Remove the old MedDRA directory
4. Finally rename the new MedDRA directory to the old name "MedDRA_25_1" as the script will look for this name

Chapter

3 Google Cloud Console

The following section discusses the use of the Google Cloud Workspace. We have included this as it may be more usable if not familar with using vim on the LACDR VM. Hence, we have made documentation navigating it as it may have a few barriers to entry for those less familiar with the Google Cloud Workspace environment.

3.1 Accessing and Managing Google Cloud Resources

This section introduces the use of Google Cloud Console to manage a publicly accessible Google Cloud Storage bucket and a PostgreSQL server on Google Cloud SQL for the FAERS database project. The bucket stores ASCII files (e.g., `DEMO16Q2.txt`) used for schema generation, and the SQL server, with IP whitelisting, hosts the database. The Cloud Console provides a web-based interface to manage these resources.

To begin, log in to the Google Cloud Console at `https://console.cloud.google.com` using your Google account. Ensure you have the necessary permissions (e.g., Editor or Owner role) for your project. Select your project from the top dropdown menu. The Console dashboard displays an overview of your resources.

To view the publicly accessible bucket:

1. Navigate to `Cloud Storage > Buckets` in the left-hand menu.
2. Locate your bucket (e.g., `your-bucket-name`). Click on it to view files like `ascii/DEMO16Q2.txt`.

3. Verify the bucket's public access by checking the `Permissions` tab. Look for a role like `allUsers` with `Storage Object Viewer` access, confirming public readability.

4. To download a file (e.g., for schema generation), click the file and select `Download`.

To manage the PostgreSQL server:

1. Go to `SQL` in the left-hand menu.

2. Find your PostgreSQL instance and click its name to view details.

3. Check the `Connections` tab to ensure the instance allows connections via public IP and has your IP whitelisted. To add an IP, click `Add network`, enter your IP (e.g., `203.0.113.1/32`), and save.

4. Note the instance's public IP address for connecting locally or via Cloud Console.

Google Cloud Console includes Cloud Shell, a browser-based terminal for running commands. To access it, click the `Activate Cloud Shell` icon (terminal symbol) in the top-right corner. Cloud Shell is pre-authenticated with your project credentials and includes tools like `gcloud` and `gsutil`.

```
gsutil ls gs://your-bucket-name/ascii/
```

This lists files like `DEMO16Q2.txt`, which can be used by the schema generation script.

3.2 Connecting to the PostgreSQL Server

To connect to the Cloud SQL PostgreSQL instance from Cloud Console:

1. Navigate to `SQL`, select your instance, and go to the `Overview` tab.

2. Under `Connect to this instance`, click `Connect using Cloud Shell`.

3. Cloud Shell opens with a pre-filled `gcloud sql connect` command, e.g.:

   ```
   gcloud sql connect your-instance-name --user=postgres
   ```

4. Run the command, enter your database password, and access the PostgreSQL prompt.

5. Test the connection by listing tables in the `faers_a` schema:

```
\dt faersdatabase*
```

To connect locally, ensure your IP is whitelisted in the Cloud SQL instance's `Connections` tab. You need `psql` installed locally (available via `apt`, `brew`, or other package managers). You also need PSQL downloaded, psycopg and google pip extension for working with the bucket. Use the instance's public IP and database credentials:

```
psql -h your-instance-public-ip -U postgres -d your-database-name
```

Enter the password when prompted. If the connection fails, verify:

- Your IP is whitelisted.
- The database user has appropriate permissions.
- Network firewalls allow outbound connections to port 5432.

Assuming the pipeline script is set up locally with credentials (e.g., via `gcloud auth application-default login`), run:

```
python s2.py
```

This accesses the bucket, generates $auto_schema_config.json$, and logs progress to $schema_generation.log$.

If you encounter errors:

- **Connection refused**: Check IP whitelisting and port 5432 accessibility.
- **Permission denied**: Ensure your Google Cloud account has `Cloud SQL Client` and `Storage Object Viewer` roles.
- **File access issues**: Verify the bucket's public access settings with `gsutil iam get gs://your-bucket-name` and ensure the appropriate IAM permissions are granted.

```
gsutil iam get gs://ascii
```

Chapter

4 System admin details

4.1 PostgresSQL

In order to have PostgresSQL run on our test machine, we had to perform the following steps. These may be of use to you as well. For more information on how to run PostgresSQL on Rocky Linux, we refer to both the general Red Hat documentation[1] and the specific PostgresSQL SELinux documentation[2]. Before we were able to run PostgresSQL, we first made the following manual changes to `/lib/systemd/system/postgresql.service`:

```
# Previously, this was here
#Environment=PGDATA=/var/lib/pgsql/data

# changed to this:
Environment=PGDATA=/faers/data
Environment=PGLOG=/faers/data/pgstartup.log
```

This sets the data folder PostgresSQL checks to the right folder. This should be the same folder as specified in the general options.

One then initialises the user and creates the database.

```
$ sudo su - postgres -c "initdb -D /faers/data"
$ pg_ctl -D /faers/data -l logfile start
```

[1] https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/system_administrators_guide/part-basic_system_configuration

[2] https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/chap-managing_confined_services-postgresql

In order to let SELinux allow the PostgresSQL process to access `/faers/data`, we must make all files in this directory of an PostgresSQL type[3]. In order to display these types in a directory, use

```
$ ls -lZ .
...
drwxrwxr-x.  2 user     user     unconfined_u:object_r:postgresql_db_t:s0
37 May 13 16:58 logs
...
```

In order to set the right type recursively moving forward, use `semanage`. [4].

```
$ semanage fcontext -a -t postgresql_db_t "/path/to/dir(/.*)?"
```

[3] https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/sect-managing_confined_services-postgresql-types

[4] https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/sect-managing_confined_services-postgresql-configuration_examples#sect-Managing_Confined_Services-PostgreSQL-Configuration_Examples-Changing_Database_Location

Chapter

5 Overall System Diagram

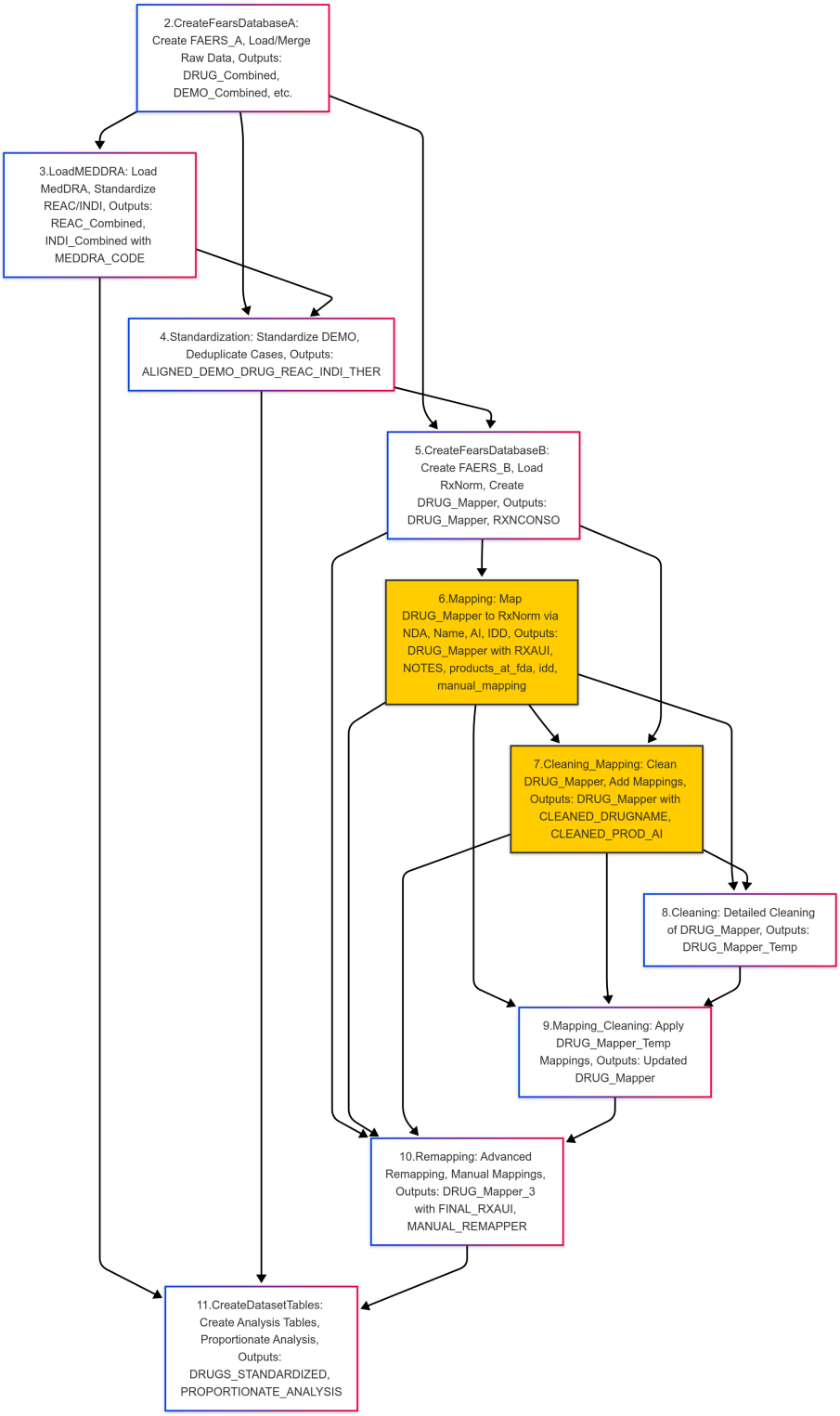


Figure 5.1: System Flow Diagram

Chapter

6 Start up script

6.1 Details and basic usage

This serves as the starting point of the pipeline. Here we make sure the right folders exist, check the configuration settings the user provided, and prepare for downloading/fetching the FAERS datasets.

In order to start running the script, one needs to enter a Python virtual environment. For the pipeline itself, we use the external `psycopg3` package, which needs to be present when running the pipeline. The code has been written to work for versions 3.11 and upwards, so first check if the correct version is installed.

```
$ which python3
/usr/bin/python3
```

As of the time of writing, Rocky Linux uses version 3.9 as its default version for Python 3, so we should instead install and use a newer version.

```
$ sudo dnf install python3.11
$ which python3.11
/usr/bin/python3.11
```

Now one can make a virtual environment like so:

```
$ python3.11 -m venv venv
$ source /venv/bin/activate
```

Now build the python project. There should exist a `pyproject.toml` in the root directory which specifies this build process.

```
$ pip install -e .
```

You possibly need a more manual alternative case, try to isntall the `psycopg3` package, excluding the 3.

```
(venv) $ pip install psycopg
```

One can now run the program. There exists a single `-h`,`-help` flag.

```
(venv) $ python3.11 faers_pipeline_start.py
```

6.2 Troubleshooting

If one encounters an error, these steps can be undertaken.

- Just run the script again. We have cleaning functions, and rerunning the pipeline may just resolve the issue.
- Check if the configuration files have valid syntax and have reasonable values.
- If the issue was fatal, the full log and trace of the error can be found in the most recent log file, present by default at `/faers/data/logs`. This may help to find the error, and since Python code is interpreted, a simple fix may be possible.

Chapter

7 FAERS Downloading and fetching

7.1 Description

What was previously known as "Script 1", now replaced by `src/download_files_from_faers.py`, implements handling of FAERS datasets needed for the pipeline. See Figure 7.1 for a general overview of the structure of this part of the pipeline. We make use of caching to prevent having to download the data again at a later date. After the initial downloading one should except to only have to download if there are new quarterly datasets available.

7.2 Troubleshooting

If one encounters an error, these steps can be undertaken.

- Just run the script again. We have cleaning functions, and rerunning the pipeline may just resolve the issue.
- If the issue was fatal, the full log and trace of the error can be found in the most recent log file, present by default at `/faers/data/logs`.
- Delete the folder of the most recent quarter that did not finish being downloaded (check the log file for this), and try again.
- In previous versions, there were problems with non-existent directories. In this version, the directories are normally automatically created. If this is not the case, it might be useful to consult the directory lay-out and create the missing directory or re-run the scripts.

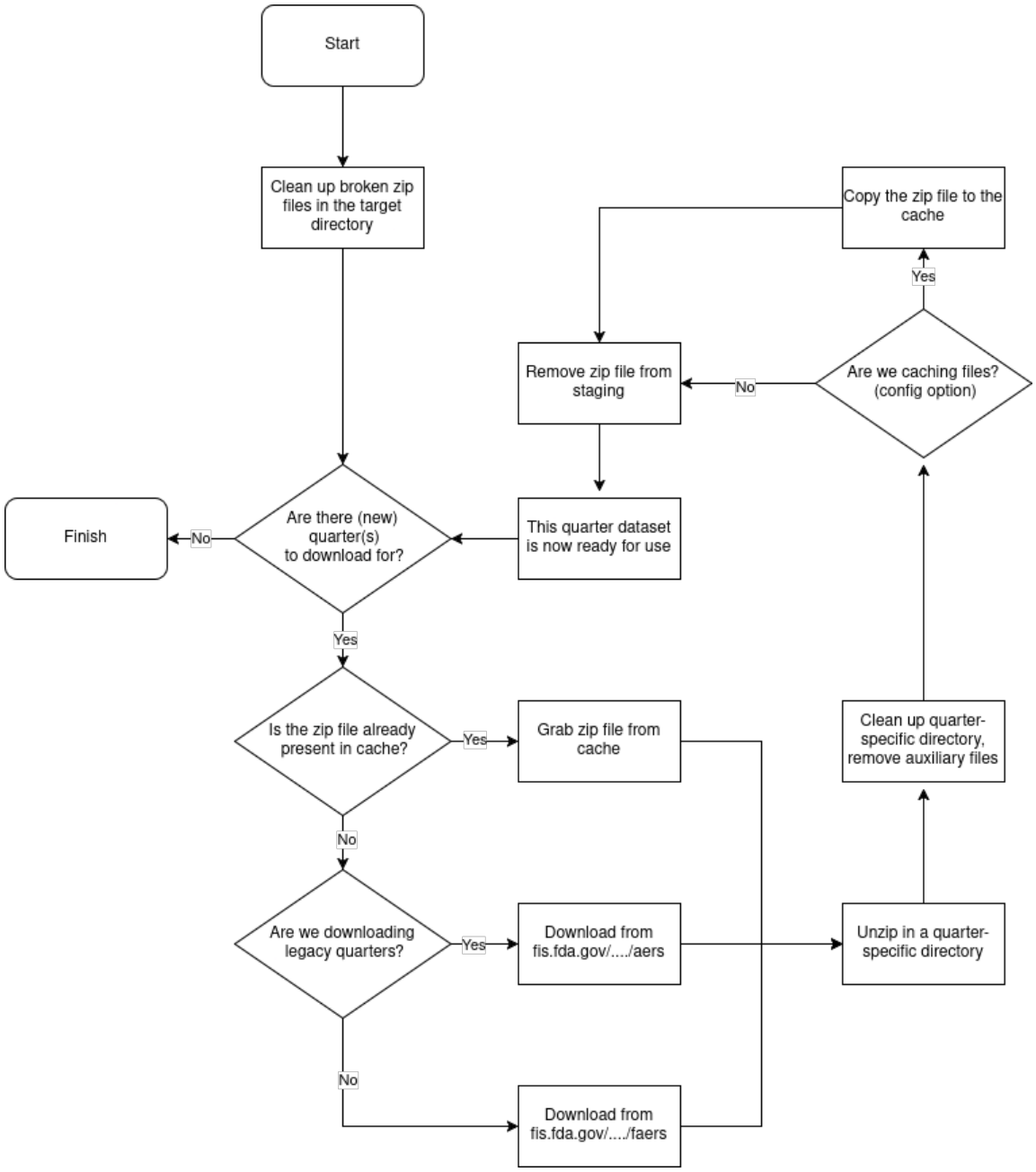


Figure 7.1: Flowchart describing the steps done when downloading from FAERS.

Chapter

8 FAERS A

FAERS A (`createFearsDatabaseA_LoadTables_MergeTables.sql`) is the stage in the database creation in which the tables are created for the relevant entries from the entire FAERS dataset, which are the following:

- Drug information (DRUG): contains the trade names of the name ("Drug-Name")
- Adverse drug reactions (REAC): contains medical descriptors, such as P.T. ("Preferred Term")
- Demographic information of patients (DEMO): each record is a single report
- Drug indications (INDI):
- Outcome information (OUTC): provides information on the outcomes of the cases
- Start and end dates (THER): gives the start and end dates of therapy
- Sources of reported events (RPSR): displays the source

For each of these categories, there is a two-stage process in FAERS A: first, a table is created and the data are bulk inserted for each quarter from 2004Q1 to 2023Q1, and, next, all quarterly entries are combined into a single table for their respective categories.

In the previous code, there was a high redundancy due to repeated operations. The reason for this original design was the difference in schema columns per quarter.

We have included an overview of all different column names and types per year-quarter combination in `schema_config.json`.

8.1 Table creation

Based on the loaded `.txt`-files, separate tables are created. These tables all have a different number of columns, differing per year-quarter combination. In the old script, these tables were all separately created to ensure that they were in the right format. The introduction of the `schema_config.json` reduces the number of lines for individual table creation from around 4000 to 150. New tables have to be in the correct format and are thus checked on the number of columns and the type of each column.

8.2 Table combination

After all quarterly tables have been created in the `faers_a` schema (e.g., `demo23q1`, `drug22q4`), they are merged into permanent combined tables in the `faers_combined` schema. This step ensures that all data for a given category (e.g., DEMO, DRUG, INDI) is centralized for efficient querying and downstream analysis.

The script `s2-5.sql` does this by:

- Creating permanent combined tables such as `DEMO_Combined`, `DRUG_Combined`, etc.
- Using the `get_completed_year_quarters` function to determine which quarterly tables exist and should be merged.
- Iterating through each year-quarter combination and dynamically generating `INSERT INTO ... SELECT FROM` statements to populate the combined tables.
- Assigning each record a `PERIOD` value (e.g., `2023q1`) to retain information about its original source.
- Creating indexes on the `primaryid` column of each combined table to optimize future joins and lookups.

This consolidation step is essential for transitioning from fragmented quarterly data to unified datasets ready for enrichment and mapping in later stages.

8.3 Cleaning and Mapping

Loading MedDRA and mapping tables

The S3 stage, this stage is one of two scripts started by s3-4.py. The python script assures everything the SQL script needs to run is accessible. The SQL file is then run on the PostgreSQL server. The main goal of this stage is to load all MedDRA files and use the data from MedDRA to map the data on the tables created by the previous stage. The script does the following:

- Drop and recreate all MedDRA related tables.
- Load MedDRA data from .asc files into said tables.
- Load CSV files that contain data to map the MedDRA data (low level terms and pref_terms to MedDRA codes).
- Clean pref_terms in INDI_Combined table for mapping. Leftover newlines or tabs are removed.
- Use MedDRA data to map the data in the INDI_Combined and REAC_Combined tables

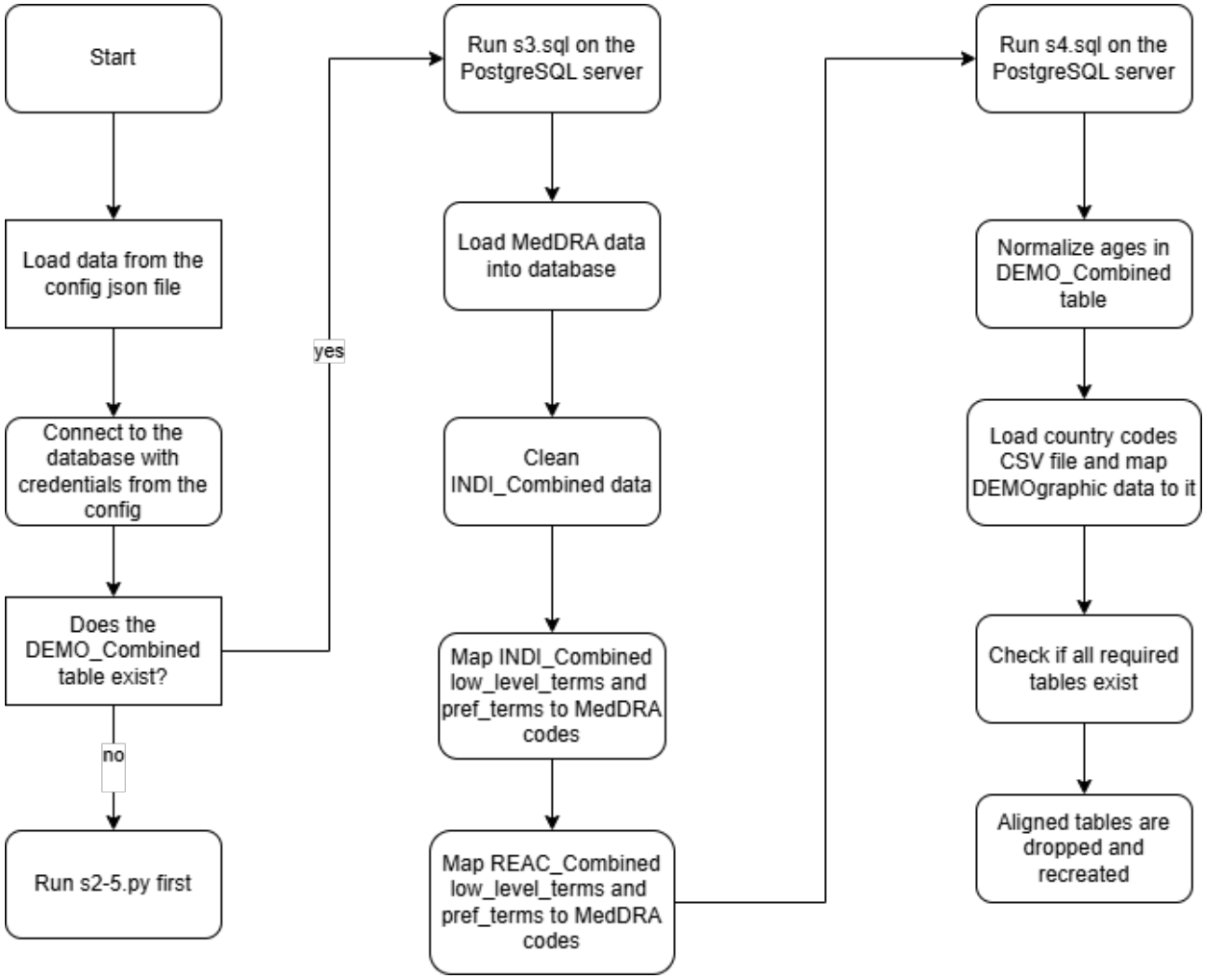
This is the point where data is intertwined with MedDRA. First all MedDRA data is loaded into newly created tables. The MedDRA data comes from ASC text files. The MedDRA data contains Low Level Terms which are first mapped to their MedDRA codes. These codes are in CSV files one for INDI data and one for REAC data. Both are loaded in and the REAC and INDI combined tables are mapped.

Cleaning DEMOgraphic table

The S4 stage makes changes to the DEMO_Combined table. The SQL script does 2 things: Standardize data and create aligned tables. Age and gender are standardized and country names are turned into country codes using external mapping with data from a CSV file. The existance of all necessary table is also checked and a new table called ALIGNED_DEMO_DRUG_REAC_INDI_THER is created. If not all necessary tables exist a notice is raised and the table will remain empty. The table aligns and combines demographic, drug, reaction, indication and therapy data by joining them on shared identifiers. To summarize the script does the following:

- Standardize age and gender
- Rewrite the country names into their country codes by external mapping from a CSV file
- Check if all necessary tables for the aligned table exist
- Create the aligned table 'ALIGNED_DEMO_DRUG_REAC_INDI_THER' to align and combine demographic, drug, reaction, indication and therapy data
- Fill table with data by joining the demographic, drug, reaction, indication and therapy data on shared identifiers.

Diagram for both s3 and s4 (from the starter script)



Chapter

9 FAERS B

The FAERS B stage of the FDA Adverse Event Reporting System (FAERS) pipeline operates within the `faers_b` schema, focusing on the standardization, mapping, and analysis of drug data to support adverse event reporting and safety signal detection. Building on the data ingestion and combination performed in the `faers_a` and `faers_combined` schemas (stages S2, S2-5, S3, S4test), FAERS B encompasses stages S5 through S11, which initialize drug mapping infrastructure, clean and map drug names to RxNorm identifiers, aggregate adverse event data, and will allow users to perform statistical analysises. Stage S5, implemented via `s5.sql` and `s5.py`, marks the entry point of FAERS B by establishing the `faers_b` schema, creating the `drug_mapper` table to store FAERS drug records, and setting up RxNorm tables to facilitate drug standardization.

9.1 S5: Initializing Drug Mapping Infrastructure

Stage S5 initializes the `faers_b` schema and sets up critical tables for drug data processing in the FAERS pipeline. The `s5.sql` script performs what we see next:

- **Schema Validation**: Verifies the `faersdatabase` context, creates the `faers_b` schema if absent, grants full privileges to the `postgres` user, and sets the search path to `faers_b`, `faers_combined`, `public`.
- **Logging Setup**: Creates a `s5_log` table to record execution steps, successes, and errors, ensuring operational traceability.
- **Drug Mapper Table**: Creates `drug_mapper` with fields including:

  - `drug_id` (INTEGER), `primaryid`, `caseid`, `drug_seq` (BIGINT): Identifiers for drug records.
  - `role_cod` (VARCHAR(2)), `period` (VARCHAR(4)): Drug role and reporting period.
  - `drugname` (VARCHAR(500)), `prod_ai` (VARCHAR(400)): Drug name and active ingredient.
  - `nda_num` (VARCHAR(200)): NDA number.
  - `rxaui`, `rxcui` (BIGINT), `str` (VARCHAR(3000)), `sab`, `tty`, `code`: RxNorm mapping attributes.
  - `remapping_*` fields: For future remapping updates.

  Populates `drug_mapper` from `faers_combined.drug_combined`, filtered by `primaryid` in `aligned_demo_drug_reac_indi_ther`, and creates an index on `drugname`.

- **RxNorm Tables**: Creates tables to support RxNorm mapping, including:
  - `rxnatomarchive`: Stores archived RxNorm atoms.
  - `rxnconso`: Contains RxNorm concepts with fields like `rxcui`, `rxaui`, `sab`, `tty`, and indexes for performance.
  - `rxnrel`, `rxnsab`, `rxnsat`, `rxnsty`, `rxndoc`, `rxncuichanges`, `rxncui`: Support relationships, attributes, and changes in RxNorm data.

  Includes commented `\copy` commands for loading RxNorm data from files (e.g., `RXNCONSO.RRF`), to be executed manually.

The `s5.py` script acts as follows:

- **Configuration**: Loads `config.json` for database settings (e.g., `host: 104.155.47.49`, `port: 5432`).
- **SQL Parsing**: Splits `s5.sql` into statements, preserving `DO` blocks and skipping `\copy` commands.
- **Execution**: Uses `psycopg` to execute statements with up to three retries, logging to `s5_execution.log`.
- **Verification**: Checks existence and row counts of `drug_mapper` and RxNorm tables, logging warnings for empty or missing tables.

Dependencies:

- `s2.sql`, `s2.py`: Populate `faers_a` tables (e.g., `drug23q1`) from `gs://bucketfaers/ascii/`.
- `s2-5.sql`, `s2-5.py`: Create `faers_combined.drug_combined`.
- `s4test.sql`, `s3-4.py`: Create `faers_combined.aligned_demo_drug_reac_indi_ther`.
- External files: RxNorm data files (e.g., `RXNCONSO.RRF`) for manual loading.

Execution Instructions:

Listing 9.1: Executing S5

```
python3.11 -m venv venv
source venv/bin/activate
pip install psycopg
python3.11 s5.py
```

For RxNorm data loading (if files are available):

```
psql -h 104.155.47.49 -p 5432 -U postgres -d faersdatabase
\copy faers_b.rxnconso FROM \
'/data/faers/FAERS_MAK/2.LoadDataToDatabase/RxNorm_full_06052023/rrf/RXNCONSO.RRF'
WITH (FORMAT CSV, DELIMITER '|', NULL '', HEADER FALSE);
```

9.2 Key Code Excerpts

The following excerpts from `s5.sql` highlight the creation and population of the `drug_mapper` table, a cornerstone of the FAERS B stage.

Listing 9.2: Creating `drug_mapper` in `s5.sql`

```
CREATE TABLE faers_b.drug_mapper (
    drug_id INTEGER NOT NULL,
    primaryid BIGINT,
    caseid BIGINT,
    drug_seq BIGINT,
    role_cod VARCHAR(2),
    period VARCHAR(4),
    drugname VARCHAR(500),
    prod_ai VARCHAR(400),
    nda_num VARCHAR(200),
    notes VARCHAR(100),
    rxaui BIGINT,
    rxcui BIGINT,
```

```
    str VARCHAR(3000),
    sab VARCHAR(20),
    tty VARCHAR(20),
    code VARCHAR(50),
    remapping_notes VARCHAR(100),
    remapping_rxaui VARCHAR(8),
    remapping_rxcui VARCHAR(8),
    remapping_str VARCHAR(3000),
    remapping_sab VARCHAR(20),
    remapping_tty VARCHAR(20),
    remapping_code VARCHAR(50)
);
CREATE INDEX idx_drug_mapper_drugname ON faers_b.drug_mapper (drugname);
```

Listing 9.3: Populating `drug_mapper` in `s5.sql`

```
INSERT INTO faers_b.drug_mapper (drug_id, primaryid, caseid, drug_seq,
role_cod, drugname, prod_ai, nda_num, period)
SELECT drug_id, primaryid, caseid, drug_seq, role_cod, drugname, prod_ai,
nda_num, period
FROM faers_combined.drug_combined
WHERE primaryid IN (SELECT primaryid
FROM faers_combined.aligned_demo_drug_reac_indi_ther);
```

Chapter

10 Cleaning, Mapping, and Remapping

This chapter details the cleaning, mapping, and remapping stages of the (FAERS) pipeline, focusing on the standardization and mapping of drug data to RxNorm identifiers within the `faers_b` schema. These processes, implemented through scripts S6, S7, S8, and S9.

10.1 S6: Initial Mapping Setup

implemented via `s6.sql` and `s6.py`, initializes the drug mapping pipeline by creating the `faers_b` schema, defining a string cleaning function, and establishing foundational tables for drug data standardization and RxNorm mapping. It processes data from the `faers_combined.DRUG_Combined` table, generated in earlier stages, and integrates external data sources to support drug identification.

The `s6.sql` script does as follows below:

- **Schema Creation**: Creates the `faers_b` schema if absent, grants full privileges to the `postgres` user, and sets the search path to include `faers_b`, `faers_combined`, and `public`. It verifies schema creation, raising an exception if unsuccessful.
- **String Cleaning Function**: Defines `faers_b.clean_string(input TEXT)`, a PL/pgSQL function that standardizes drug names by:
  - Extracting content within parentheses using `SUBSTRING`.
  - Trimming special characters (e.g., `:.,?/ !@#$`).
  - Replacing semicolons with `/` and normalizing whitespace with `REGEXP_REPLACE`.

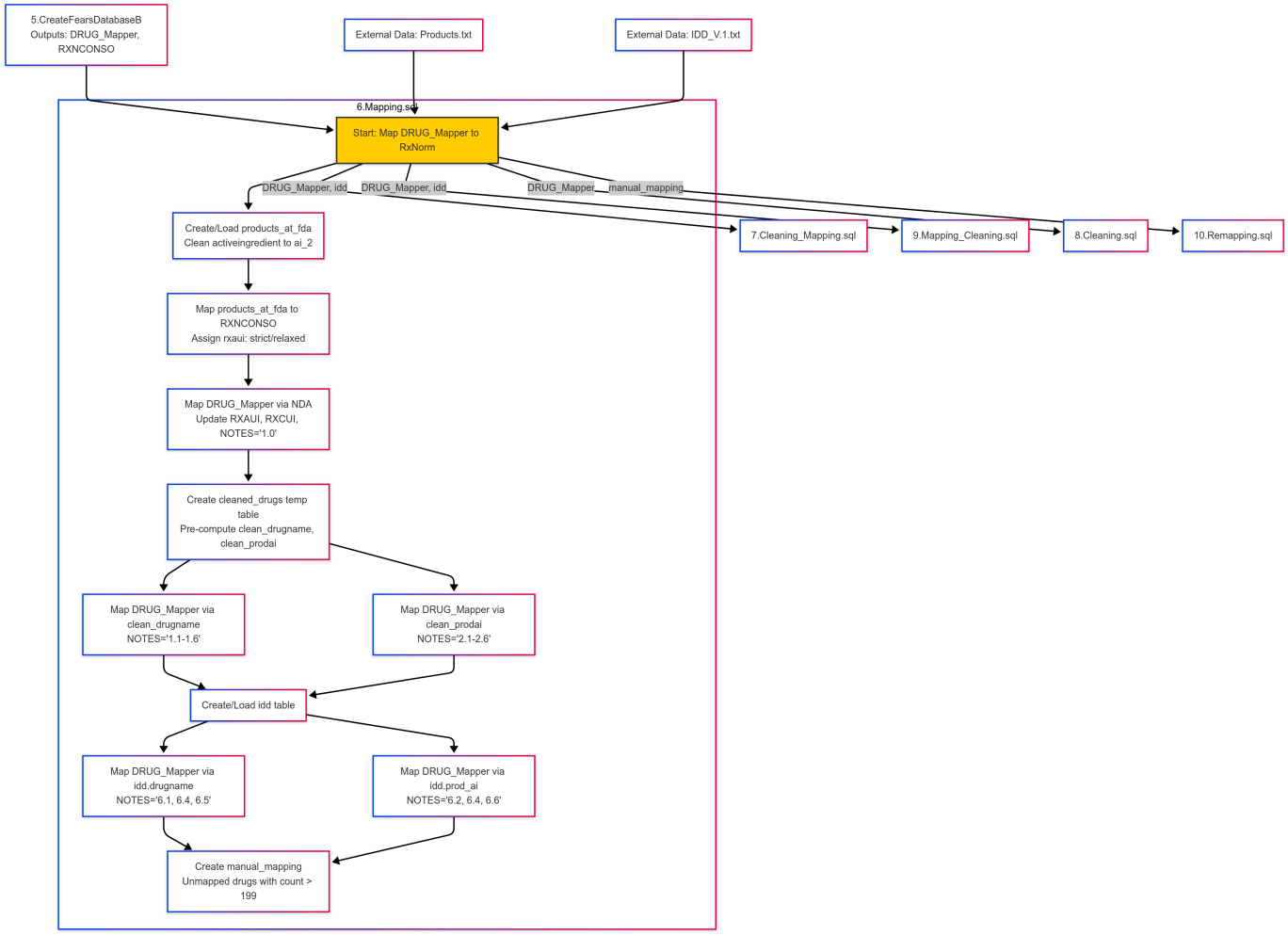


Figure 10.1: s6 Flowchart

  - Returning an empty string for `NULL` inputs via `COALESCE`.

- **Tables**:
  - `products_at_fda`: Stores FDA drug product data (e.g., `applno` (VARCHAR(10)), `drugname`, `activeingredient`, `rxaui` (VARCHAR(8)), `ai_2` (TEXT)). Updates `ai_2` with cleaned `activeingredient`.
  - `IDD`: Holds drug identification data (e.g., `DRUGNAME`, `RXAUI`, `RXCUI`, `SAB`).
  - `manual_mapping`: Lists unmapped drugs with occurrence counts above 199 (e.g., `drugname`, `count`, `rxaui`).
  - `DRUG_Mapper` (assumed pre-existing): Stores FAERS drug records (e.g., `DRUGNAME`, `prod_ai`, `rxaui`).
- **Indexes**: Creates indexes on `IDD.DRUGNAME`, `products_at_fda.applno`, and others for query optimization.
- **Data Mapping**: Updates `products_at_fda.rxaui` and `DRUG_Mapper` using `rxnconso` and `IDD`, assigning `notes` (e.g., `1.1` for `RXNORM/IN`).
- **Data Loading**: Includes commented `\copy` commands for `Products.txt` and `IDD_V.1.txt`.

The `s6.py` script execution:

- **Configuration**: Loads `config.json` (e.g., `host: 104.155.47.49`, `port: 5432`).
- **SQL Parsing**: Splits `s6.sql` into statements, preserving `DO` blocks.
- **Execution**: Uses `psycopg` to execute statements with retries, logging to `s6_execution.log`.
- **Verification**: Checks table existence and row counts.

Dependencies:

- `s2.sql`, `s2.py`: Populate `faers_a.drug23q1` from `gs://bucketfaers/ascii/`.
- `s2-5.sql`, `s2-5.py`: Create `faers_combined.DRUG_Combined`.
- External files: `Products.txt`, `IDD_V.1.txt`, `rxnconso`.

Execution Instructions:

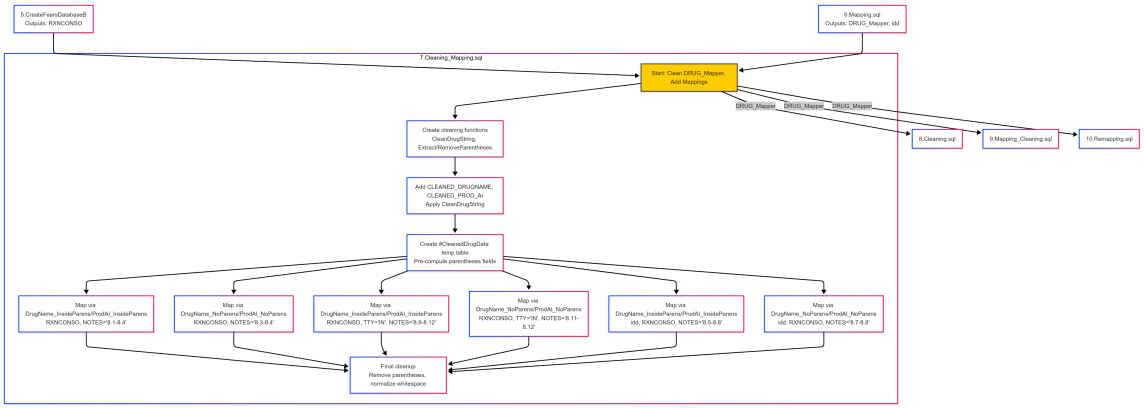


Figure 10.2: S7 flow chart

Listing 10.1: Executing S6

```
python3.11 -m venv venv
source venv/bin/activate
pip install psycopg
python3.11 s6.py
```

For data loading:

```
psql -h 104.155.47.49 -p 5432 -U postgres -d faersdatabase
\copy faers_b.products_at_fda (...) \
FROM '/data/faers/FAERS_MAK/2.LoadDataToDatabase/drugsatfda20230627/Products.txt' .
```

10.2 S7: Analysis and Reporting

This is done and, implemented via `s7.sql` and `s7.py`, aggregates adverse drug event data into `FAERS_Analysis_Summary`, joining `DRUG_RxNorm_Mapping`, `REAC_Combined`, and `OUTC_Combined` to support safety signal detection.

The `s7.sql` script performs:

- **Schema Validation**: Ensures `faers_b` exists, grants privileges, and sets search path.
- **Table Creation**: Creates `FAERS_Analysis_Summary` with fields:
  - `SUMMARY_ID` (SERIAL PRIMARY KEY).
  - `RXCUI` (VARCHAR(8)), `DRUGNAME` (TEXT).

  - `REACTION_PT` (VARCHAR(100)), `OUTCOME_CODE` (VARCHAR(20)).
  - `EVENT_COUNT` (BIGINT), `REPORTING_PERIOD` (VARCHAR(10)).
  - `ANALYSIS_DATE` (TIMESTAMP).

- **Data Aggregation**: Joins source tables on `primaryid`, groups by `RXCUI`, `DRUGNAME`, `pt`, `outc_cod`, `PERIOD`, computing `EVENT_COUNT`.
- **Indexes**: Creates indexes on `RXCUI`, `REACTION_PT`, `OUTCOME_CODE`.

Listing 10.2: Aggregation in `s7.sql`

```
INSERT INTO faers_b."FAERS_Analysis_Summary" (
    "RXCUI", "DRUGNAME", "REACTION_PT", "OUTCOME_CODE",
    "EVENT_COUNT", "REPORTING_PERIOD"
)
SELECT
    drm."RXCUI",
    drm."DRUGNAME",
    rc.pt AS "REACTION_PT",
    oc.outc_cod AS "OUTCOME_CODE",
    COUNT(*) AS "EVENT_COUNT",
    rc."PERIOD" AS "REPORTING_PERIOD"
FROM faers_b."DRUG_RxNorm_Mapping" drm
INNER JOIN faers_combined."REAC_Combined" rc
    ON drm."primaryid" = rc."primaryid"
INNER JOIN faers_combined."OUTC_Combined" oc
    ON drm."primaryid" = oc."primaryid"
GROUP BY drm."RXCUI", drm."DRUGNAME", rc.pt, oc.outc_cod, rc."PERIOD"
ON CONFLICT DO NOTHING;
```

The `s7.py` script:

- **Configuration**: Loads `config.json`.
- **SQL Parsing**: Splits `s7.sql`.
- **Execution**: Executes statements, logs to `s7_execution.log`.
- **Verification**: Checks `FAERS_Analysis_Summary`.

Dependencies:

- `s2.sql`, `s2.py`: Populate `faers_a.reac23q1`, `outc23q1`.
- `s2-5.sql`, `s2-5.py`: Create `REAC_Combined`, `OUTC_Combined`.
- `s6.sql`, `s6.py`: Create `DRUG_RxNorm_Mapping`.

Execution Instructions:

Listing 10.3: Executing S7

```
source venv/bin/activate
python3.11 s7.py
```

10.3 S8: Drug Name Cleaning

This is done via `s8.sql` and `s8.py`, that performs comprehensive standardization of `DRUGNAME` and `PROD_AI` fields in `DRUG_Mapper` by creating `DRUG_Mapper_Temp` with cleaned data to prepare for RxNorm mapping. The system uses a hybrid approach combining configuration-driven transformations with hard-coded cleaning operations.

Architecture

The `s8.py` script runs as follows below:

- **Dual Configuration**: Loads both `config.json` (database) and `config_s8.json` (cleaning phases)
- **Configuration Integration**: Creates `temp_s8_config` table to make phase-specific configurations available to SQL
- **Robust Execution**: Implements retry logic with exponential backoff for transient database errors
- **SQL Parsing**: Handles complex SQL operations- including DO blocks and function definitions
- **Comprehensive Logging**: Outputs to both `s8_execution.log` and console with detailed execution tracking

The `s8.sql` script implements this cleaning logic/steps:

- **Schema Validation**: Ensures `faers_b` exists and validates `DRUG_Mapper` table presence

- **Column Preparation**: Adds `CLEANED_DRUGNAME` and `CLEANED_PROD_AI` columns if not present
- **Multi-Phase Processing**: Executes 10 distinct cleaning phases for both drug name fields
- **Configuration-Driven Operations**: Dynamically applies transformations from `config_s8.json`

Cleaning Process
Initial Preprocessing

Listing 10.4: Initial Data Preparation

```
-- Initialize cleaned columns from original data
UPDATE DRUG_Mapper
SET CLEANED_DRUGNAME = DRUGNAME
WHERE CLEANED_DRUGNAME IS NULL AND DRUGNAME IS NOT NULL;

-- Remove numeric suffixes like /00032/
UPDATE DRUG_Mapper
SET CLEANED_DRUGNAME = regexp_replace(
    CLEANED_DRUGNAME,
    '/[0-9]{5}/',
    '',
    'g'
)
WHERE NOTES IS NULL;

-- Normalize delimiters and whitespace
UPDATE DRUG_Mapper
SET CLEANED_DRUGNAME = regexp_replace(
    CLEANED_DRUGNAME,
    '[|,+;\\\\]',
    '/',
    'g'
)
WHERE NOTES IS NULL;
```

Core Cleaning Function

Listing 10.5: Numeric Character Removal

```
CREATE OR REPLACE FUNCTION clearnumericcharacters(input_text TEXT)
RETURNS TEXT AS $func$
BEGIN
```

```
    RETURN regexp_replace(input_text, '[0-9]', '', 'g');
END;
$func$ LANGUAGE plpgsql;
```

Phase-Based Processing The system processes data through 10 distinct phases: **DRUGNAME Phases (1-5):**

- **Phase 1**: `UNITS_OF_MEASUREMENT_DRUGNAME` - Standardizes dosage units
- **Phase 2**: `MANUFACTURER_NAMES_DRUGNAME` - Removes/standardizes manufacturer names
- **Phase 3**: `WORDS_TO_VITAMIN_B_DRUGNAME` - Normalizes vitamin terminology
- **Phase 4**: `FORMAT_DRUGNAME` - Applies formatting rules
- **Phase 5**: `CLEANING_DRUGNAME` - Final drugname cleanup with suffix removal

PROD_AI Phases (6-10):

- **Phase 6**: `UNITS_MEASUREMENT_PROD_AI` - Standardizes product units
- **Phase 7**: `MANUFACTURER_NAMES_PROD_AI` - Cleans manufacturer data
- **Phase 8**: `WORDS_TO_VITAMIN_B_PROD_AI` - Vitamin normalization
- **Phase 9**: `FORMAT_PROD_AI` - Product formatting
- **Phase 10**: `CLEANING_PROD_AI` - Final prod_ai cleanup

Configuration-Driven Transformations

Listing 10.6: Dynamic Phase Processing

```
-- Load phase configuration from temp table
SELECT cfg.config_data INTO phase_data
FROM temp_s8_config AS cfg
WHERE cfg.phase_name = current_phase;

-- Apply configured replacements dynamically
IF phase_data IS NOT NULL THEN
    FOR stmt IN SELECT *
    FROM jsonb_array_elements(phase_data -> 'replacements') LOOP
        EXECUTE format(
```

```
                'UPDATE %I SET %I = REPLACE(%I, %L, %L)',
                stmt.value->>'table',
                stmt.value->>'set_column',
                stmt.value->>'replace_column',
                stmt.value->>'find',
                stmt.value->>'replace'
            );
    END LOOP;
END IF;
```

Suffix Removal and Final Cleanup

Listing 10.7: Suffix Removal Logic

```
UPDATE DRUG_Mapper_Temp
SET CLEANED_DRUGNAME = CASE
    WHEN RIGHT(CLEANED_DRUGNAME, 5) =
     ' JELL' THEN LEFT(CLEANED_DRUGNAME, LENGTH(CLEANED_DRUGNAME)-5)
    WHEN RIGHT(CLEANED_DRUGNAME, 4) =
    WHEN RIGHT(CLEANED_DRUGNAME, 4) =
     ' GEL' THEN LEFT(CLEANED_DRUGNAME, LENGTH(CLEANED_DRUGNAME)-4)
    WHEN RIGHT(CLEANED_DRUGNAME, 4) =
     ' CAP' THEN LEFT(CLEANED_DRUGNAME, LENGTH(CLEANED_DRUGNAME)-4)
    WHEN RIGHT(CLEANED_DRUGNAME, 4) =
     ' TAB' THEN LEFT(CLEANED_DRUGNAME, LENGTH(CLEANED_DRUGNAME)-4)
    WHEN RIGHT(CLEANED_DRUGNAME, 4) =
     ' FOR' THEN LEFT(CLEANED_DRUGNAME, LENGTH(CLEANED_DRUGNAME)-4)
    WHEN RIGHT(CLEANED_DRUGNAME, 2) =
     '//' THEN LEFT(CLEANED_DRUGNAME, LENGTH(CLEANED_DRUGNAME)-2)
    WHEN RIGHT(CLEANED_DRUGNAME, 1) =
     '/' THEN LEFT(CLEANED_DRUGNAME, LENGTH(CLEANED_DRUGNAME)-1)
     ELSE CLEANED_DRUGNAME
END;
```

Output and Verification

The `s8.py` script includes comprehensive verification:

- **Table Verification**: Confirms `DRUG_Mapper_Temp` creation and population
- **Row Count Logging**: Reports final table statistics
- **Error Handling**: Graceful handling of missing tables or configuration files

Dependencies:

- `s2.sql`, `s2.py`: Populate `faers_a.drug*` tables
- `s2-5.sql`, `s2-5.py`: Create `DRUG_Combined`
- `s6.sql`, `s6.py`: Create `DRUG_Mapper`
- `config_s8.json`: Phase-specific cleaning configurations

Instructions for the exectution:

Listing 10.8: Executing S8

```
source venv/bin/activate
python3.11 s8.py
```

The S8 stage produces `DRUG_Mapper_Temp` with standardize `CLEANED_DRUGNAME` and `CLEANED_PROD_AI` fields, ready for subsequent RxNorm mapping operations. The hybrid configuration-driven approach allows for flexible rule management while maintaining consistent cleaning logic across the entire dataset.

10.4 S9: Mapping with Cleaned Data

The 9th script consists of `s9.sql` and `s9.py`, updates `DRUG_Mapper` with cleaned data from `DRUG_Mapper_Temp` and maps to RxNorm using `RXNCONSO` and `IDD`.

The `s9.sql` script:

- **Schema Validation**: Ensures `faers_b` exists.
- **Placeholder Table**: Creates minimal `IDD` if absent.
- **Column Addition**: Adds `CLEANED_DRUGNAME` and `CLEANED_PROD_AI` to `DRUG_Mapper`.
- **Table Checks**: Verifies `DRUG_Mapper` and `DRUG_Mapper_Temp`.
- **Updates**: Transfers cleaned data from `DRUG_Mapper_Temp` to `DRUG_Mapper`.
- **Mapping**: Maps `DRUG_Mapper` to `RXNCONSO` and `IDD` using `CLEANED_DRUGNAME` and `CLEANED_PROD_AI`, assigning `notes` (e.g., `9.1`).

Listing 10.9: RxNorm Mapping in `s9.sql`

```
UPDATE faers_b."DRUG_Mapper"
SET
    "RXAUI" = CAST(rxn."RXAUI" AS BIGINT),
    "RXCUI" = CAST(rxn."RXCUI" AS BIGINT),
    "NOTES" = '9.1',
    "SAB" = rxn."SAB",
    "TTY" = rxn."TTY",
    "STR" = rxn."STR",
    "CODE" = rxn."CODE"
FROM faers_b."RXNCONSO" rxn
WHERE rxn."STR" = faers_b."DRUG_Mapper"."CLEANED_DRUGNAME"
AND faers_b."DRUG_Mapper"."NOTES" IS NULL
AND rxn."SAB" = 'RXNORM'
AND rxn."TTY" IN ('MIN', 'IN', 'PIN');
```

The `s9.py` script:

- **Configuration**: Loads `config.json`.
- **SQL Parsing**: Splits `s9.sql`.
- **Execution**: Runs statements, logs to `s9_execution.log`.
- **Verification**: Checks `DRUG_Mapper`.

Dependencies:

- `s2.sql`, `s2.py`: Populate `faers_a.drug*`.
- `s2-5.sql`, `s2-5.py`: Create `DRUG_Combined`.
- `s6.sql`, `s6.py`: Create `DRUG_Mapper`.
- `s8.sql`, `s8.py`: Create `DRUG_Mapper_Temp`.

Execution Instructions:

Listing 10.10: Executing S9

```
source venv/bin/activate
python3.11 s9.py
```

10.5 S10: Remapping and Manual Mapping

The S10 stage finalizes the drug mapping process in the FAERS pipeline by performing iterative remapping of `DRUG_Mapper` to a new table, `DRUG_Mapper_2`, applying manual remapping through a `MANUAL_REMAPPER` table, and removing duplicate entries to ensure data integrity. This stage builds on the cleaned and mapped data from S6–S9, enhancing the accuracy of RxNorm mappings within the `faers_b` schema.

The `s10.sql` Behaves as we indicate in the next paragraph:

- **Schema Validation**: Ensures the `faers_b` schema exists, grants privileges to the `postgres` user, and sets the search path to include `faers_b`, `faers_combined`, and `public`, consistent with prior stages.
- **New Table Creation**: Creates `DRUG_Mapper_2`, a refined version of `DRUG_Mapper`, with fields such as:
  - `DRUGNAME`, `prod_ai`, `CLEANED_DRUGNAME`, `CLEANED_PROD_AI` (TEXT): Drug name and active ingredient data.
  - `RXAUI`, `RXCUI` (BIGINT): RxNorm identifiers.
  - `SAB`, `TTY` (VARCHAR): Source vocabulary and term type.
  - `STR` (TEXT), `CODE` (VARCHAR): Additional RxNorm attributes.
  - `NOTES` (TEXT): Mapping metadata.
- **Iterative Remapping**: Transfers data from `DRUG_Mapper` to `DRUG_Mapper_2`, reapplying mapping logic from S6 and S9 (e.g., matching `CLEANED_DRUGNAME` to `RXNCONSO.STR` or `IDD.DRUGNAME`) to refine RxNorm assignments. This may involve stricter conditions or updated `RXNCONSO` data.
- **Manual Remapping**: Integrates mappings from a `MANUAL_REMAPPER` table, which contains manually curated mappings for drugs in `manual_mapping` (from S6) with high occurrence counts. Updates `DRUG_Mapper_2` with fields like `RXAUI`, `RXCUI`, and `NOTES` (e.g., `10.1` for manual mappings).
- **Deduplication**: Removes duplicate entries in `DRUG_Mapper_2` based on key fields (e.g., `DRUGNAME`, `primaryid`, `RXCUI`), ensuring each drug record is unique.
- **Indexes**: Creates indexes on `DRUG_Mapper_2` (e.g., `RXCUI`, `DRUGNAME`) to optimize query performance.

Listing 10.11: Assumed Manual Remapping in `s10.sql`

```
UPDATE faers_b."DRUG_Mapper_2"
SET
    "RXAUI" = mr.rxaui,
    "RXCUI" = mr.rxcui,
    "NOTES" = '10.1',
    "SAB" = mr.sab,
    "TTY" = mr.tty,
    "STR" = mr.str,
    "CODE" = mr.code
FROM faers_b."MANUAL_REMAPPER" mr
WHERE faers_b."DRUG_Mapper_2"."DRUGNAME" = mr.drugname
AND faers_b."DRUG_Mapper_2"."NOTES" IS NULL;
```

The `s10.py` script has the folowing execution:

- **Configuration**: Loads `config.json` for database settings (e.g., `host: 104.155.47.49`, `port: 5432`, `user: postgres`, `dbname: faersdatabase`).
- **SQL Parsing**: Splits `s10.sql` into executable statements, preserving `DO` blocks and skipping `\copy` commands, using regular expressions as in prior stages.
- **Execution**: Connects to PostgreSQL 17.5 using `psycopg` (version 3.x), executes statements with up to three retries for transient errors, and logs details to `s10_execution.log`.
- **Table Verification**: Confirms the existence and row counts of `DRUG_Mapper_2`, logging warnings for empty or missing tables.

Execution Logs: No `s10_execution.log` When running, execution logs are created. Exection logs is expected to confirm the creation of `DRUG_Mapper_2` and successful application of manual mappings.

Dependencies:

- `s2.sql`, `s2.py`: Populate `faers_a.drug*` tables from `gs://bucketfaers/ascii/` using `schema_config.json`.
- `s2-5.sql`, `s2-5.py`: Combine `faers_a` tables into `faers_combined.DRUG_Combined`.
- `s6.sql`, `s6.py`: Create `faers_b.DRUG_Mapper` and `manual_mapping`.
- `s8.sql`, `s8.py`: Create `faers_b.DRUG_Mapper_Temp` for cleaned data.

- `s9.sql`, `s9.py`: Update `DRUG_Mapper` with cleaned and mapped data.
- `MANUAL_REMAPPER` table: Contains manually curated mappings, assumed pre-populated based on `manual_mapping`.

Execution Instructions:

```
source venv/bin/activate
python3.11 s10.py
```

Chapter

11 Creating the Tables

Stage S11 of the FDA Adverse Event Reporting System (FAERS) pipeline, implemented through `s11.sql` and `s11.py`, finalizes the data processing workflow by creating a comprehensive set of standardized dataset tables in the `faers_b` schema. These tables consolidate drug, adverse event, demographic, indication, outcome, therapy, and report source data, enabling detailed safety signal analysis. Additionally, S11 performs statistical analysis to compute metrics such as Proportional Reporting Ratio (PRR), Relative Odds Ratio (ROR), and Information Component (IC), which are critical for identifying potential drug-adverse event associations. This stage relies on data from prior pipeline stages, including MedDRA-mapped tables from S3, the aligned dataset from S4test, combined tables from S2-5, and RxNorm-mapped drug data (e.g., `drug_mapper_3`, assumed from S9 or an unprovided S10). We did on a PostgreSQL 17.5 database hosted on a Rocky Linux server, S11 ensures data integrity through validation checks and logging.

11.1 S11: Dataset Table Creation and Analysis

The S11 stage orchestrates the creation of 14 tables that form the foundation for FAERS data analysis. The `s11.sql` script performs the following key operations:

- **Schema Validation**: Verifies the `faersdatabase` context and ensures the `faers_b` schema exists, granting privileges to the `postgres` user and setting the search path to `faers_b`, `faers_combined`, `public`.
- **Logging Setup**: Creates a `remapping_log` table to record execution steps, successes, and errors, ensuring traceability of operations.

- **Table Creation and Population**:
  - `drugs_standardized`: Stores standardized drug records with fields like `primaryid`, `drug_id`, `drug_seq`, `role_cod`, `period`, `rxaui`, and `drug`, populated from `drug_mapper_3` (filtered for non-null `final_rxaui` excluding 9267486, indicating 'UNKNOWN STR')
    and joined with `aligned_demo_drug_reac_indi_ther`.
  - `adverse_reactions`: Captures adverse events with `primaryid`, `period`, and `adverse_event`, derived from `reac_combined`'s `meddra_code` mapped to `pref_term` or `low_level_term`.
  - `drug_adverse_reactions_pairs`: Pairs drugs and adverse events by joining `drugs_standardized` and `adverse_reactions` on `primaryid`, ensuring unique combinations.
  - `drug_adverse_reactions_count`: Aggregates pair counts, grouping by `rxaui`, `drug`, and `adverse_event` to compute `count_of_reaction`.
  - `drug_indications`: Records drug indications with `primaryid`, `indi_drug_seq`, `period`, and `drug_indication`, using `indi_combined`'s `meddra_code` (excluding specific codes 10070592, 10057097) mapped to `pref_term` or `low_level_term`.
  - `demographics`: Stores patient data (`caseid`, `primaryid`, `caseversion`, `fda_dt`, `i_f_cod`, `event_dt`, `age`, `gender`, `country_code`, `period`) from `aligned_demo_drug_reac_indi_ther`.
  - `case_outcomes`: Captures outcomes (`primaryid`, `outc_cod`, `period`) from `outc_combined`.
  - `therapy_dates`: Records therapy details (`primaryid`, `dsg_drug_seq`, `start_dt`, `end_dt`, `dur`, `dur_cod`, `period`) from `ther_combined`.
  - `report_sources`: Stores report sources (`primaryid`, `rpsr_cod`, `period`) from `rpsr_combined`.
  - `drug_margin`: Computes total reaction counts per drug (`rxaui`, `margin`) from `drug_adverse_reactions_count`.
  - `event_margin`: Computes total reaction counts per adverse event (`adverse_event`, `margin`).
  - `total_count`: Stores the total number of reactions (`n`).
  - `contingency_table`: Constructs a 2x2 contingency table (`rxaui`, `drug`, `adverse_event`, `a`, `b`, `c`, `d`) for statistical analysis.

  - `proportionate_analysis`: Calculates statistical metrics (`prr`, `ror`, `chi_squared_yates`, `ic`, etc.) to assess drug-event associations.

- **Indexing**: Creates indexes on `primaryid`, `rxaui`, and `adverse_event` fields to optimize query performance.

The `s11.py` script orchestrates execution:

- **Configuration**: Loads `config.json` for database settings (e.g., `host: 104.155.47.49`, `port: 5432`).
- **SQL Parsing**: Splits `s11.sql` into statements, preserving `DO` blocks and skipping `\copy` commands.
- **Execution**: Uses `psycopg` to execute statements with up to three retries, logging to `s11_execution.log`.
- **Verification**: Checks table existence and row counts, logging warnings for empty or missing tables.

Execution Logs: No `s11_execution.log` was provided, but based on S6–S9 patterns (May 17, 2025), S11 is expected to confirm table creation, with potential warnings if input tables (e.g., `drug_mapper_3`) are empty.

Dependencies:

- `s2.sql`, `s2.py`: Populate `faers_a` tables.
- `s2-5.sql`, `s2-5.py`: Create `faers_combined` tables (`DEMO_Combined`, `DRUG_Combined`, `INDI_Combined`, `REAC_Combined`, `OUTC_Combined`, `THER_Combined`, `RPSR_Combined`).
- `s3.sql`, `s3-4.py`: Map MedDRA codes in `INDI_Combined`, `REAC_Combined`, create `pref_term`, `low_level_term`.
- `s4test.sql`, `s3-4.py`: Create `aligned_demo_drug_reac_indi_ther`.
- `s9.sql`, `s9.py`: Update `DRUG_Mapper` (assumed to produce `drug_mapper_3` if S10 is absent).

Execution Instructions:

Listing 11.1: Executing S11

```
python3.11 -m venv venv
source venv/bin/activate
pip install psycopg
python3.11 s11.py
```

11.2 Key Code Excerpts

The following code excerpts highlight critical operations in `s11.sql`, including the creation of `drugs_standardized` and the statistical analysis in `proportionate_analysis`.

Listing 11.2: Creating `drugs_standardized` in `s11.sql`

```
CREATE TABLE faers_b.drugs_standardized (
    primaryid BIGINT,
    drug_id INTEGER,
    drug_seq BIGINT,
    role_cod VARCHAR(2),
    period VARCHAR(4),
    rxaui BIGINT,
    drug VARCHAR(3000)
);
INSERT INTO faers_b.drugs_standardized
SELECT dm.primaryid, dm.drug_id, dm.drug_seq, dm.role_cod, dm.period,
       dm.final_rxaui AS rxaui, dm.remapping_str AS drug
FROM faers_b.drug_mapper_3 dm
INNER JOIN faers_combined.aligned_demo_drug_reac_indi_ther ad
    ON dm.primaryid = ad.primaryid
WHERE dm.final_rxaui IS NOT NULL
  AND dm.final_rxaui != 9267486;
```

Listing 11.3: Proportionate Analysis in `s11.sql`

```
CREATE TABLE faers_b.proportionate_analysis (
    id SERIAL PRIMARY KEY,
    rxaui BIGINT,
    drug VARCHAR(3000),
    adverse_event VARCHAR(1000),
    a FLOAT,
    n_expected FLOAT,
```

```
    prr FLOAT,
    prr_lb FLOAT,
    prr_ub FLOAT,
    chi_squared_yates FLOAT,
    ror FLOAT,
    ror_lb FLOAT,
    ror_ub FLOAT,
    ic FLOAT,
    ic025 FLOAT,
    ic975 FLOAT
);
INSERT INTO faers_b.proportionate_analysis
SELECT ct.rxaui, ct.drug, ct.adverse_event, ct.a,
       ((ct.a + ct.b) * (ct.a + ct.c)) /
       NULLIF((ct.a + ct.b + ct.c + ct.d), 0) AS n_expected,
       (ct.a / NULLIF((ct.a + ct.c), 0)) /
       NULLIF((ct.b / NULLIF((ct.b + ct.d), 0)), 0) AS prr,
       exp(ln((ct.a / NULLIF((ct.a + ct.c), 0)) /
       NULLIF((ct.b / NULLIF((ct.b + ct.d), 0)), 0))
           - 1.96 * sqrt((1.0 / ct.a)
           - (1.0 / (ct.a + ct.c)) + (1.0 / ct.b)
           - (1.0 / (ct.b + ct.d)))) AS prr_lb,
       ...
       log(2, (ct.a + 0.5) /
       NULLIF((((ct.a + ct.b) * (ct.a + ct.c)) /
       (ct.a + ct.b + ct.c + ct.d) + 0.5), 0)) AS ic,
       ...
FROM faers_b.contingency_table ct
WHERE ct.a > 0 AND ct.b > 0 AND ct.c > 0 AND ct.d > 0;
```

Chapter

12 Unit Testing

12.1 Introduction

The FAERS (FDA Adverse Event Reporting System) data processing pipeline represents a sophisticated, multi-phase system for transforming raw pharmaceutical adverse event data into standardized, analysis-ready datasets suitable for pharmacovigilance research and regulatory reporting. Given the critical nature of this data for drug safety monitoring and the complex algorithmic operations involved, comprehensive unit testing forms an essential component of the pipeline's quality assurance framework. This testing documentation covers the complete unit test suite for all 11 phases of the FAERS pipeline, validating operations ranging from initial data ingestion and schema creation through advanced drug mapping, statistical analysis, and regulatory reporting. The pipeline architecture follows a dual-component design pattern where each phase consists of a Python orchestration script paired with a corresponding SQL script, requiring comprehensive testing of both database operations and pipeline coordination logic.

Testing Architecture and Frameworks

The unit testing strategy employs a multi-framework approach to validate different aspects of the pipeline. The architecture is organized around the following components:

- **Dual Testing Framework Structure:**
  - **unittest** is used primarily for Python orchestration logic, including:

        * Configuration management
        * SQL parsing
        * Error handling
        * Retry mechanisms
    - **pytest** complements this by focusing on:
        * Database operations
        * SQL logic validation
        * Complex query testing
        * Statistical calculation verification

- **Mock-Based Testing Strategy:**
    - External dependencies—such as database connections, file systems, and network resources—are fully mocked.
    - This allows for:
        * Isolated testing
        * Repeatable execution
        * Fast test performance
        * Simulation of complex interactions and error conditions

- **Hierarchical Test Organization:**
    - Tests follow a staged structure:
        * **Stage 1:** Infrastructure and connectivity verification
        * **Stage 2:** Core functionality testing
        * **Stage 3:** Advanced scenarios including:
            · Edge case performance
            · Error recovery paths

Pipeline Complexity and Testing Challenges

The FAERS pipeline presents unique testing challenges due to its sophisticated data processing requirements.

Multi-Phase Dependencies: The pipeline's 11 phases exhibit complex interdependencies, where later phases require the successful completion of earlier operations.

Testing must therefore validate not only individual phase functionality but also ensure correct dependency chain management and graceful handling of missing or failed prerequisites.

External Data Integration: The pipeline incorporates multiple external data sources, including RxNorm terminology, MedDRA medical dictionaries, the FDA Products database, and the Industry Drug Database (IDD). Testing efforts focus on verifying integration patterns, ensuring correct handling of diverse data formats, and assessing system behavior in the event of unavailable or inconsistent external resources.

Statistical Algorithm Validation: Several later phases implement advanced pharmacovigilance algorithms such as Proportionate Reporting Ratios (PRR), Information Components (IC), and confidence interval computations. Tests are designed to verify the mathematical accuracy of these computations, confirm numerical stability, and account for edge cases such as division by zero or numerical overflow.

Performance and Scale Considerations: Given that the pipeline processes millions of adverse event records, performance testing is essential. Tests evaluate system behavior under load, including memory utilization, connection stability, and processing time for large-scale data operations involving text processing, hierarchical mapping, and statistical analysis.

Testing Patterns and Methodologies

Several key testing patterns emerge across the pipeline phases

Configuration-Driven Testing: Validates flexible configuration management systems that enable runtime modification of processing parameters, cleaning rules, and mapping strategies without code changes.

SQL Parsing and Execution Validation: Comprehensive testing of sophisticated SQL parsing logic that handles complex constructs including nested dollar quotes, DO blocks, functions, and dynamic SQL generation.

Error Recovery and Retry Mechanisms: Systematic testing of retry logic for transient failures while ensuring that non-retryable errors (such as duplicate objects) are handled appropriately without unnecessary retry attempts.

Data Quality and Integrity Testing: Validation of data cleaning operations, standardization algorithms, and quality assurance mechanisms that ensure regulatory compliance and analytical accuracy.

Hierarchical Processing Validation: Testing of priority-based processing systems that implement quality-driven selection algorithms for optimal data mapping and concept assignment.

Regulatory and Quality Assurance Context

The testing framework operates within the context of regulatory requirements for pharmaceutical data processing:

Data Integrity Compliance: Testing validates that all data transformations maintain proper audit trails, implement appropriate validation checks, and preserve data lineage for regulatory review.

Statistical Accuracy Requirements: Pharmacovigilance statistics must meet regulatory standards for mathematical accuracy and numerical stability. Testing includes comprehensive validation of statistical formulas against known reference implementations.

Reproducibility and Traceability: The testing framework ensures that all pipeline operations are deterministic and reproducible, with comprehensive logging and error reporting mechanisms that support regulatory audit requirements.

This documentation provides detailed insight into the testing methodology, validation approaches, and quality assurance mechanisms that ensure the FAERS pipeline meets the stringent requirements for pharmaceutical data processing and regulatory reporting. Each phase's documentation includes comprehensive test descriptions, validation criteria, and examples of the sophisticated testing patterns employed to verify correct operation under both normal and exceptional conditions.

12.2 Phase 2: Schema Creation and Data Processing

Phase 2 implements the foundational database schema creation and data processing operations for the FAERS pipeline. This phase consists of two primary components: `s2.py` for configuration management and schema operations, and `s2_5.py` for combined table creation and data consolidation.

Test Suite Overview

The Phase 2 test suite comprises four test files that validate both Python orchestration logic and SQL database operations:

- `test_s2.py` (unittest framework) - Configuration and schema validation
- `test_s2_5.py` (unittest framework) - Pipeline execution and error handling
- `test_s2.py` (pytest framework) - SQL operations and database functions

- `test_s2_5.py` (pytest framework) - Combined table operations and integration

Configuration Management Tests (`test_s2.py` - unittest)

This test class validates the configuration loading and schema resolution functionality:

Configuration Loading:

- `test_load_config_success` - Validates successful JSON configuration parsing
- `test_load_config_file_not_found` - Ensures proper FileNotFoundError handling

Schema Resolution:

- `test_get_schema_for_period_valid_period` - Tests time-based schema selection for different quarters (2022Q3 vs 2024Q2)
- `test_get_schema_for_period_invalid_table` - Validates error handling for non-existent tables
- `test_get_schema_for_period_invalid_period` - Tests boundary conditions for date ranges outside available schemas

Pipeline Execution Tests (`test_s2_5.py` - unittest)

This test class focuses on the pipeline orchestration and SQL parsing components:

SQL Statement Parsing:

- `test_parse_sql_statements_basic` - Validates proper parsing of mixed SQL statements including DO blocks and comments
- `test_parse_sql_statements_empty_input` - Tests handling of comment-only or empty SQL files

Execution Reliability:

- `test_execute_with_retry_success_on_first_attempt` - Confirms successful execution without retries
- `test_execute_with_retry_duplicate_table_handling` - Validates graceful handling of duplicate object errors

Database Operations Tests (`test_s2.py` - pytest)

This comprehensive test suite validates core database functionality using mocked connections:

Schema and Table Management:

- `test_schema_creation` - Verifies `faers_a` schema creation logic
- `test_process_faers_file_table_name_generation` - Validates dynamic table naming (e.g., `demo23q1`, `drug24q4`)
- `test_process_faers_file_column_definition_building` - Tests SQL column definition generation from JSON schemas

Time-Based Processing:

- `test_get_completed_year_quarters_basic_logic` - Tests year-quarter generation from 2004 to current minus one quarter
- `test_get_completed_year_quarters_edge_case_q1` - Validates edge case handling when current quarter is Q1

Data Import Operations:

- `test_file_header_validation_logic` - Validates file header column count matching
- `test_copy_command_format_validation` - Tests PostgreSQL COPY command formatting with proper delimiters and encoding

Error Handling and Reliability:

- `test_server_connection_timeout` - Tests connection timeout scenarios
- `test_server_encoding_validation` - Validates UTF-8 encoding requirements
- `test_error_handling_and_rollback_logic` - Tests transaction rollback on errors

Combined Table Operations Tests (`test_s2_5.py` - pytest)

This test suite focuses on the `faers_combined` schema operations:

Schema and Table Creation:

- `test_schema_creation_statement` - Validates `faers_combined` schema creation
- `test_combined_table_creation` - Tests creation of combined tables with identity columns
- `test_session_parameter_settings` - Validates session configuration (search_path, work_mem)

Advanced SQL Operations:

- `test_do_block_structure_validation` - Tests complex DO block parsing and execution
- `test_get_completed_year_quarters_dependency` - Validates dependency on Phase 2 functions
- `test_insert_operation_structure` - Tests INSERT operations into combined tables

System Reliability:

- `test_server_connection_handling` - Tests connection retry mechanisms
- `test_server_memory_configuration` - Validates PostgreSQL memory settings
- `test_transaction_rollback_on_error` - Tests transaction management and rollback procedures

Integration Testing Support

Both pytest-based test suites include integration test markers that can be run separately against actual database connections.
These tests are marked with `@pytest.mark.integration` and are skipped during unit testing to maintain test isolation and speed.

Key Testing Patterns

Mock Strategy: All tests use comprehensive mocking of database connections, file systems, and external dependencies to ensure unit test isolation.

Error Simulation: Tests systematically simulate various PostgreSQL error conditions (connection timeouts, duplicate objects, syntax errors) to validate error handling robustness.

Time-Based Logic: Special attention is given to testing time-dependent functionality, particularly the quarter-based processing logic that determines which FAERS data periods to process.

SQL Parsing Validation: Tests ensure that complex SQL scripts with DO blocks, comments, and mixed statement types are properly parsed and executed in the correct order.

12.3 Phase 3: MedDRA Integration

Phase 3 implements Medical Dictionary for Regulatory Activities (MedDRA) integration, enabling standardized medical terminology mapping for FAERS data. This phase focuses on loading MedDRA hierarchical data and creating mappings between FAERS terms and MedDRA codes.

Test Suite Overview

The Phase 3 test suite validates MedDRA data loading, term mapping, and data cleaning operations:

- `test_s3.py` (pytest framework) – MedDRA table operations and mapping functionality
- `test_s3_4.py` (unittest framework) – Pipeline orchestration and configuration management

MedDRA Operations Tests (test_s3.py – pytest)

This comprehensive test suite validates MedDRA data integration functionality: **Table Structure and Data Loading:**

- `test_drop_and_create_meddra_tables` – Validates creation of core MedDRA tables (`low_level_term`, `pref_term`, etc.)

- `test_copy_command_structure` – Tests `COPY` operations for .asc file loading with proper delimiters
- `test_mapping_tables_creation` – Validates creation of `INDI` and `REAC` mapping tables with primary keys
- `test_json_data_loading` – Tests JSON format loading for mapping data

Data Enhancement and Mapping:

- `test_alter_table_add_columns` – Tests addition of meddra_code and cleaned_pt columns to combined tables
- `test_data_cleaning_update_statements` – Validates text cleaning operations (UPPER, TRIM, REPLACE for special characters)
- `test_meddra_code_mapping_updates` – Tests MedDRA code assignment through pref_term joins
- `test_index_creation_statements` – Validates performance index creation on meddra_code columns

File System and Performance:

- `test_server_file_access_handling` – Tests file access permissions and error handling for MedDRA .asc files
- `test_server_memory_handling_large_tables` – Validates memory management for large-scale mapping operations

Validation and Structure Tests:

- `test_table_column_specifications` – Validates proper data types for MedDRA table columns
- `test_file_path_structure` – Tests expected file path patterns for MedDRA data
- `test_update_statement_structure` – Validates SQL UPDATE statement patterns and syntax

12.4 Phase 4: Data Alignment and Standardization

Phase 4 implements comprehensive data alignment operations, standardizing age calculations, country codes, gender values, and creating the final aligned dataset that combines demographic, drug, reaction, indication, and therapy data.

Test Suite Overview

The Phase 4 test suite validates data standardization logic and multi-table alignment operations:

- `test_s4.py` (pytest framework) – Data alignment and standardization operations
- `test_s3_4.py` (unittest framework) – Combined pipeline orchestration for phases 3 and 4

Data Alignment Tests (`test_s4.py` – pytest)

This test suite focuses on data standardization and alignment operations:

Configuration and Setup:

- `test_search_path_configuration` – Validates proper schema search path configuration
- `test_column_additions_to_demo_combined` – Tests addition of standardized columns (age_years_fixed, country_code, gender)

Age Standardization Logic:

- `test_age_conversion_logic` – Tests comprehensive age unit conversion (decades, years, months, weeks, days, hours to years)
- `test_numeric_age_validation_regex` – Validates regex patterns for numeric age validation

Geographic and Demographic Standardization:

- `test_country_code_fallback_logic` – Tests country code mapping with 2-character fallback logic

- `test_gender_standardization_cleanup` – Validates gender code standardization (M/F preservation, UNK/NS/YR cleanup)

Table Creation and Joins:

- `test_aligned_table_creation_structure` – Tests creation of `ALIGNED_DEMO_DRUG_REAC_INDI_THER` table
- `test_join_conditions_validation` – Validates JOIN conditions across multiple combined tables

Performance and Error Handling:

- `test_server_file_access_validation` – Tests CSV file access for country code mappings
- `test_server_memory_handling_large_joins` – Validates memory management for complex multi-table joins

Data Quality Validation:

- `test_data_type_conversions` – Tests data type casting and conversion logic
- `test_distinct_and_conflict_handling` – Validates duplicate detection and DISTINCT operations
- `test_index_creation_validation` – Tests performance index creation on aligned table

Combined Pipeline Orchestration (`test_s3_4.py` – unittest)

This test class validates the combined execution of phases 3 and 4:

Configuration Management:

- `test_config_loading_success` – Tests successful configuration loading with all required parameters
- `test_config_file_not_found` – Validates error handling for missing configuration files
- `test_config_invalid_json` – Tests handling of malformed JSON configuration

- `test_missing_config_parameters` – Validates detection of incomplete configuration

Prerequisite Validation:

- `test_table_existence_check_table_missing` – Tests validation that required tables from previous phases exist

Key Testing Patterns

Age Conversion Testing: Comprehensive validation of age unit conversions using realistic test data covering all supported time units (decades, years, months, weeks, days, hours).

Data Standardization: Systematic testing of data cleanup operations including text normalization, country code standardization, and gender value cleanup.

Multi-table Join Validation: Testing of complex JOIN operations across demographic, drug, reaction, indication, and therapy tables with proper foreign key relationships.

File System Integration: Validation of external file dependencies including MedDRA .asc files and country mapping CSV files with appropriate error handling for access issues. this is some code.

12.5 Phase 5: RxNorm Integration and Drug Mapping

Phase 5 implements comprehensive RxNorm integration for standardized drug nomenclature mapping. This phase creates the $faers_b$ schema, establishes drug mapping tables, loads RxNorm terminology data, and creates mappings between FAERS drug names and standardized RxNorm concepts.

Test Suite Overview

The Phase 5 test suite validates RxNorm data integration, drug mapping operations, and terminology standardization:

- `test_s5.py` (unittest framework) – Pipeline orchestration and SQL parsing
- `test_s5.py` (pytest framework) – RxNorm operations and drug mapping functionality

Pipeline Orchestration Tests (`test_s5.py` – unittest)

This test class validates the Python orchestration layer for Phase 5:

Configuration and Parsing:

- `test_load_config_success` – Validates JSON configuration loading for database connections
- `test_load_config_missing_file` – Tests error handling for missing configuration files
- `test_parse_sql_statements_with_copy_commands` – Tests SQL parsing with COPY command filtering
- `test_parse_sql_statements_empty_and_comments_only` – Validates handling of comment-only SQL files

Execution Reliability:

- `test_execute_with_retry_operational_error_then_success` – Tests retry logic for transient connection failures
- `test_execute_with_retry_duplicate_table_no_retry` – Validates that duplicate object errors bypass retry logic

The SQL parsing functionality specifically filters out COPY commands to prevent conflicts during automated execution:

```
-- This COPY command will be filtered out during parsing
\copy faers_b.test_table FROM 'data.csv' WITH CSV HEADER;
-- While this INSERT statement will be executed
INSERT INTO faers_b.test_table VALUES (2, 'another test');
```

RxNorm Operations Tests (`test_s5.py` – pytest)

This comprehensive test suite validates RxNorm integration and drug mapping operations:

Database Context and Schema Management:

- `test_database_context_validation` – Validates connection to correct `faersdatabase`
- `test_faers_b_schema_creation` – Tests `faers_b` schema creation with proper authorization

- `test_search_path_and_privileges` – Validates schema privileges and search path configuration

Infrastructure and Logging:

- `test_logging_table_creation` – Tests `s5_log` table creation for progress tracking
- `test_drug_mapper_table_creation` – Validates drug mapping table structure with proper indexing
- `test_drug_mapper_population_logic` – Tests population of drug mapper from combined tables

The drug mapper table serves as the central mapping structure:

```
CREATE TABLE faers_b.drug_mapper (
drug_id INTEGER NOT NULL,
primaryid BIGINT,
caseid BIGINT,
drug_seq BIGINT,
role_cod VARCHAR(2),
drugname VARCHAR(500),
rxaui BIGINT,
rxcui BIGINT,
str VARCHAR(3000),
remapping_rxcui VARCHAR(8)
);
```

RxNorm Table Management:

- `test_rxnorm_table_creation_structure` – Tests creation of core RxNorm tables (`RXNCONSO`, `RXNREL`, `RXNSAT`)
- `test_rxnorm_table_existence_checks` – Validates table existence checking logic before creation

The RXNCONSO table represents the core RxNorm concept structure:

```
CREATE TABLE faers_b.rxnconso (
rxcui VARCHAR(8) NOT NULL,
lat VARCHAR(3) DEFAULT 'ENG' NOT NULL,
rxaui VARCHAR(8) NOT NULL,
```

```
sab VARCHAR(20) NOT NULL,
tty VARCHAR(20) NOT NULL,
code VARCHAR(50) NOT NULL,
str VARCHAR(3000) NOT NULL
);
```

Data Loading and File Management:

- `test_rxnorm_table_creation_structure` – Tests creation of core RxNorm tables (`RXNCONSO`, `RXNREL`, `RXNSAT`)
- `test_rxnorm_table_existence_checks` – Validates table existence checking logic before creation

RxNorm data loading uses pipe-delimited .RRF files:

```
\copy faers_b.rxnconso FROM '/path/RXNCONSO.RRF'
WITH (FORMAT CSV, DELIMITER '|', NULL '', HEADER FALSE);
```

Error Handling and Validation:

- `test_do_block_error_handling_structure` – Tests comprehensive error handling with logging
- `test_rxnorm_copy_command_validation` – Validates `COPY` command structure for RxNorm files
- `test_logging_functionality_validation` – Tests progress logging mechanisms
- `test_table_dependency_validation` – Validates prerequisite table existence checking

The phase implements comprehensive error handling with progress logging:

```
DO $
BEGIN
-- Operation logic
INSERT INTO faers_b.drug_mapper VALUES (...);
EXCEPTION
WHEN OTHERS THEN
INSERT INTO faers_b.s5_log (step, message)
VALUES ('Error', 'Error: ' || SQLERRM);
RAISE;
END $;
```

Key Testing Patterns

Dependency Validation: Systematic testing of table existence before operations, ensuring proper execution order across pipeline phases.

Large-Scale Data Handling: Comprehensive testing of memory management, disk space, and connection stability during RxNorm data loading operations.

File System Integration: Validation of external RxNorm .RRF file dependencies with proper error handling for missing files and permission issues.

Error Recovery: Testing of retry mechanisms for transient failures while ensuring that non-retryable errors (like duplicate objects) are handled appropriately.

Progress Tracking: Validation of logging mechanisms that track execution progress and capture detailed error information for debugging purposes.

12.6 Phase 6: Advanced Drug Mapping and External Data Integration

Phase 6 implements sophisticated drug mapping operations that integrate multiple external data sources including FDA Products database and Industry Drug Database (IDD). This phase enhances the drug mapping established in Phase 5 through intelligent string cleaning, hierarchical mapping strategies, and manual mapping workflows for unm apped high-frequency drugs.

Test Suite Overview

The Phase 6 test suite validates advanced mapping algorithms, external data integration, and string processing operations:

- `test_s6.py` (unittest framework) – Pipeline orchestration and advanced SQL parsing
- `test_s6.py` (pytest framework) – Advanced drug mapping operations and external data integration

Pipeline Orchestration Tests (`test_s6.py` — unittest)

This test class validates the sophisticated Python orchestration layer for Phase 6:

Configuration and Advanced SQL Parsing:

- `test_load_config_success` – Validates configuration loading for external data sources
- `test_load_config_file_not_found` – Tests error handling for missing configuration
- `test_load_config_invalid_json` – Validates handling of malformed JSON configuration
- `test_parse_sql_statements_with_functions_and_do_blocks` =- Tests parsing of complex SQL with functions and DO blocks
- `test_parse_sql_statements_with_bom` – Validates BOM character handling in SQL files
- `test_parse_sql_statements_nested_functions` – Tests parsing of nested function definitions

The SQL parsing handles complex nested structures including functions with embedded dollar quotes:

```
CREATE OR REPLACE FUNCTION faers_b.outer_function()
RETURNS void AS $
DECLARE
inner_var INTEGER;
BEGIN
-- This contains $ inside the function
EXECUTE 'CREATE TEMP TABLE test AS SELECT $ || ''hello'' || $';
RAISE NOTICE 'Executed with $';
END
$ LANGUAGE plpgsql;
```

Enhanced Execution Reliability:

- `test_execute_with_retry_success_first_attempt` – Tests successful execution without retries.
- `test_execute_with_retry_database_error_then_success` – Tests retry logic for database connectivity issues.
- `test_execute_with_retry_duplicate_index_no_retry` – Validates bypass of retry logic for duplicate object errors.

Advanced Drug Mapping Tests (`test_s6.py` – pytest)

This comprehensive test suite validates sophisticated drug mapping operations: **String Processing and Standardization:**

- `test_database_context_validation` – Validates correct database context.
- `test_clean_string_function_logic` – Tests advanced string cleaning algorithm for drug name standardization.

The clean_string function implements sophisticated text processing:

```
CREATE OR REPLACE FUNCTION faers_b.clean_string(input TEXT)
RETURNS TEXT AS $
DECLARE
output TEXT := input;
BEGIN
-- Extract content from parentheses if present
IF POSITION('(' IN output) > 0
AND POSITION(')' IN output) > POSITION('(' IN output) THEN
output := SUBSTRING(output FROM POSITION('(' IN output) + 1 FOR
POSITION(')' IN output) - POSITION('(' IN output) - 1);
END IF;
-- Clean special characters and trim
output := TRIM(BOTH ' .,?/‘~!@#$%^&*-_=+ ' FROM output);

-- Replace common patterns
output := REPLACE(output, ';', ' / ');
output := REGEXP_REPLACE(output, '\s+', ' ', 'g');

RETURN COALESCE(output, '');
END;
$ LANGUAGE plpgsql;
```

External Data Source Integration:

- `test_products_at_fda_table_creation` – Tests FDA Products database table structure.
- `test_idd_table_creation_and_structure` – Validates Industry Drug Database (IDD) table creation.

- `test_performance_indexes_creation` – Tests multi-table index creation for performance optimization.

Intelligent Mapping Strategies:

- `test_rxaui_mapping_logic_with_conditions` – Tests hierarchical mapping with strict and relaxed conditions.
- `test_nda_number_mapping_logic` – Validates NDA number-based mapping with regex validation.
- `test_drug_name_mapping_with_priority_logic` – Tests priority-based mapping with quality scoring.

The mapping strategy employs a hierarchical approach with quality-based prioritization:

```
-- Strict conditions mapping (highest quality)
UPDATE faers_b.products_at_fda
SET rxaui = rxnconso.rxaui
FROM faers_b.rxnconso
WHERE products_at_fda.ai_2 = rxnconso.str
AND rxnconso.sab = 'RXNORM'
AND rxnconso.tty IN ('IN', 'MIN')
AND products_at_fda.rxaui IS NULL;
-- Relaxed conditions mapping (fallback)
UPDATE faers_b.products_at_fda
SET rxaui = rxnconso.rxaui
FROM faers_b.rxnconso
WHERE products_at_fda.ai_2 = rxnconso.str
AND products_at_fda.rxaui IS NULL;
```

Manual Mapping Workflow:

- `test_manual_mapping_table_and_high_count_drugs` – Tests identification and management of high-frequency unmapped drugs.
- `test_server_dependency_validation_complex` – Validates complex dependency chains across multiple schemas.

The manual mapping system identifies drugs requiring human review:

```
INSERT INTO faers_b.manual_mapping (count, drugname)
SELECT COUNT(drugname) AS count, drugname
FROM faers_b.drug_mapper
WHERE notes IS NULL
GROUP BY drugname
HAVING COUNT(drugname) > 199;
```

Data Quality and Validation:

- `test_temp_table_operations` – Tests temporary table management for complex operations.
- `test_cast_operations_validation` – Validates data type conversions.
- `test_conditional_logic_validation` – Tests conditional mapping logic.
- `test_string_operations_validation` – Validates string manipulation functions.

Key Testing Patterns

Hierarchical Mapping Validation: Tests validate the multi-tiered mapping strategy that prioritizes high-quality matches (RXNORM/IN) over lower-quality alternatives, ensuring optimal mapping outcomes.

String Processing Algorithms: Comprehensive testing of text cleaning operations including parenthetical content extraction, special character removal, and whitespace normalization.

External Data Integration: Validation of integration patterns for FDA Products database and Industry Drug Database, including proper table structures and join conditions.

Performance Optimization: Testing of strategic index creation across multiple tables and schemas to support complex join operations and mapping queries.

Quality Scoring Systems: Validation of note-based quality scoring (1.0-6.6) that tracks mapping methodology and enables quality assessment of drug mappings.

Manual Review Workflows: Testing of automated identification systems for drugs requiring manual intervention, prioritizing high-frequency unmapped terms for human review.

12.7 Phase 7: FAERS Analysis Summary and Data Export

Phase 7 implements the final analysis and summary operations that aggregate FAERS data into actionable insights. This phase creates comprehensive summary tables that combine drug mappings, adverse reactions, and outcomes data, providing the foundation for pharmacovigilance analysis and regulatory reporting.

Test Suite Overview

The Phase 7 test suite validates summary table creation, complex aggregation operations, and data export functionality:

- `test_s7.py` (unittest framework) – Pipeline orchestration and SQL parsing for analysis operations.
- `test_s7.py` (pytest framework) – Analysis summary operations and aggregation logic.

Pipeline Orchestration Tests (`test_s7.py` – unittest)

This test class validates the Python orchestration layer for the final analysis phase:
Configuration and SQL Processing:

- `test_load_config_success` – Validates configuration loading for analysis operations.
- `test_load_config_file_not_found` – Tests error handling for missing configuration files.
- `test_load_config_invalid_json` – Validates handling of malformed JSON configuration.
- `test_parse_sql_statements_basic_functionality` – Tests parsing of analysis SQL with aggregation logic.
- `test_parse_sql_statements_with_bom_character` – Validates BOM character handling in SQL files.
- `test_parse_sql_statements_complex_do_block` – Tests parsing of complex analysis procedures.

The SQL parsing handles sophisticated analysis operations including nested dollar quotes:

```
DO $
DECLARE
sql_text TEXT;
BEGIN
sql_text := 'CREATE␣TABLE␣test␣AS␣SELECT␣$tag$hello$tag$␣as␣greeting';
EXECUTE sql_text;
RAISE NOTICE 'Executed:␣%', sql_text;
END
$;
```

Execution Reliability:

- `test_execute_with_retry_immediate_success` – Tests successful execution without retries.
- `test_execute_with_retry_operational_error_recovery` – Tests retry logic for analysis operations.
- `test_execute_with_retry_duplicate_object_skip` – Validates bypass of retry logic for existing objects.

Analysis Summary Operations Tests (`test_s7.py` – pytest)

This comprehensive test suite validates the final analysis and summary functionality:

Infrastructure and Table Creation:

- `test_database_context_validation` – Validates correct database context for analysis.
- `test_faers_analysis_summary_table_creation` – Tests creation of the primary analysis summary table.
- `test_schema_setup_and_privileges` – Validates schema configuration and access privileges.

The FAERS Analysis Summary table serves as the central repository for aggregated insights:

```
CREATE TABLE faers_b."FAERS_Analysis_Summary" (
"SUMMARY_ID" SERIAL PRIMARY KEY,
"RXCUI" VARCHAR(8),
"DRUGNAME" TEXT,
"REACTION_PT" VARCHAR(100),
"OUTCOME_CODE" VARCHAR(20),
"EVENT_COUNT" BIGINT,
"REPORTING_PERIOD" VARCHAR(10),
"ANALYSIS_DATE" TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Data Integration and Aggregation:

- `test_source_table_existence_validation` – Tests validation of prerequisite tables from previous phases.
- `test_complex_aggregation_insert_logic` – Validates sophisticated aggregation operations with GROUP BY logic.
- `test_join_conditions_validation` – Tests multi-table JOIN operations across schemas.

The aggregation logic combines data from multiple sources to create comprehensive summaries:

```
INSERT INTO faers_b."FAERS_Analysis_Summary" (
"RXCUI", "DRUGNAME", "REACTION_PT",
"OUTCOME_CODE", "EVENT_COUNT", "REPORTING_PERIOD"
)
SELECT
drm."RXCUI",
drm."DRUGNAME",
rc.pt AS "REACTION_PT",
oc.outc_cod AS "OUTCOME_CODE",
COUNT(*) AS "EVENT_COUNT",
rc."PERIOD" AS "REPORTING_PERIOD"
FROM faers_b."DRUG_RxNorm_Mapping" drm
INNER JOIN faers_combined."REAC_Combined" rc
ON drm."primaryid" = rc."primaryid"
INNER JOIN faers_combined."OUTC_Combined" oc
ON drm."primaryid" = oc."primaryid"
```

```
GROUP BY drm."RXCUI", drm."DRUGNAME", rc.pt, oc.outc_cod, rc."PERIOD"
ON CONFLICT DO NOTHING;
```

Performance and Data Quality:

- `test_performance_indexes_creation` – Tests strategic index creation for analysis queries.
- `test_conflict_resolution_logic` – Validates duplicate handling with ON CONFLICT logic.
- `test_timestamp_default_functionality` – Tests automatic timestamp assignment for analysis dates.

Export and Reporting Capabilities:

- `test_server_export_capability_validation` – Tests CSV export functionality for regulatory reporting

The export functionality enables regulatory reporting and external analysis:

```
\copy faers_b."FAERS_Analysis_Summary" TO '/data/analysis_summary.csv'
WITH (FORMAT CSV, DELIMITER ',', NULL '', HEADER TRUE);
```

Data Validation and Quality Assurance:

- `test_aggregation_functions_validation` – Tests `COUNT` and `GROUP BY` operations.
- `test_column_aliasing_validation` – Validates proper column aliasing in complex queries.
- `test_table_qualification_validation` – Tests proper schema qualification across tables.
- `test_data_types_validation` – Validates data type specifications for analysis columns.

Key Testing Patterns

Multi-Schema Integration: Tests validate complex operations that span multiple schemas (`faers_b`, `faers_combined`) and integrate data from all previous pipeline phases.

Aggregation Logic Validation: Comprehensive testing of `GROUP BY` operations, `COUNT` functions, and summary statistics that form the core of pharmacovigilance analysis.

Conflict Resolution: Testing of sophisticated duplicate handling using `ON CONFLICT DO NOTHING` to ensure data integrity in summary tables.

Performance Optimization: Validation of strategic indexing on key analysis columns (`RXCUI`, `REACTION_PT`, `OUTCOME_CODE`) to support efficient querying.

Export Readiness: Testing of CSV export functionality with proper formatting for regulatory submissions and external analysis tools.

Temporal Tracking: Validation of automatic timestamp assignment and reporting period tracking for temporal analysis capabilities.

Data Type Consistency: Systematic testing of data type specifications to ensure compatibility with downstream analysis tools and regulatory requirements.

12.8 Phase 8: Advanced Drug Name Cleaning and Standardization

Phase 8 implements sophisticated drug name cleaning operations through configurable, multi-stage text processing algorithms. This phase transforms raw drug names into standardized forms suitable for accurate mapping, utilizing configuration-driven cleaning rules and specialized string processing functions.

Test Suite Overview

The Phase 8 test suite validates advanced text processing algorithms and configuration-driven cleaning operations:

- `test_s8.py` (unittest framework) - Pipeline orchestration and configuration management
- `test_s8.py` (pytest framework) - Advanced text cleaning operations and configuration processing

Pipeline Orchestration Tests (`test_s8.py` - unittest)

This test class validates the Python orchestration layer and configuration management:

Configuration Management:

- `test_load_config_success` - Validates main configuration loading for database connections
- `test_load_s8_config_success` - Tests S8-specific configuration loading for cleaning rules and phase definitions
- `test_load_s8_config_file_not_found` - Tests graceful handling of missing S8 configuration (returns empty dict)
- `test_load_s8_config_invalid_json` - Tests error handling for malformed S8 configuration JSON
- `test_create_config_temp_table_with_phases` - Tests temporary configuration table creation with phase data insertion
- `test_create_config_temp_table_empty_config` - Tests temp table creation with empty S8 configuration

Advanced SQL Parsing:

- `test_parse_sql_statements_with_functions_and_config_blocks` - Tests parsing with configuration references and temp_s8_config table usage
- `test_parse_sql_statements_complex_nested_blocks` - Tests complex nested dollar quote handling with dynamic SQL generation
- `test_parse_sql_statements_with_functions_and_config_blocks` - Tests parsing of functions with embedded config data access

The configuration system uses JSONB for flexible cleaning rule definition:

```
CREATE TEMP TABLE temp_s8_config (
    phase_name TEXT,
    config_data JSONB
);

INSERT INTO temp_s8_config VALUES
```

```
('phase1', '{"cleaning_rules": ["remove_duplicates",
"standardize_names"]}'),
('phase2', '{"cleaning_rules": ["validate_dates"],
"thresholds": {"confidence": 0.9}}');
```

Execution Reliability:

- `test_execute_with_retry_success_immediately` - Tests successful execution without needing retries
- `test_execute_with_retry_database_error_then_success` - Tests retry mechanism recovering from temporary database issues

Advanced Text Cleaning Tests (`test_s8.py` - pytest)

This comprehensive test suite validates sophisticated drug name cleaning operations:

Infrastructure and Setup:

- `test_schema_creation_and_search_path` - Tests faers_b schema configuration and search path setting for cleaning operations
- `test_column_additions_to_drug_mapper` - Tests addition of CLEANED_DRUGNAME and CLEANED_PROD_AI columns with proper data types
- `test_table_existence_check_logic` - Validates drug_mapper table dependency verification across faers_b and public schemas

Core Text Processing Functions:

- `test_clear_numeric_characters_function` - Tests clearnumericcharacters function logic for removing numeric characters from drug names
- `test_initial_cleaning_operations` - Tests initial cleaning steps including numeric suffix removal (/[0-9]{5}/), delimiter normalization, and parenthetical content extraction
- `test_suffix_removal_logic` - Tests systematic suffix removal (JELL, NOS, GEL, CAP, TAB, FOR, //, /) using CASE statement logic
- `test_special_character_trimming_logic` - Tests comprehensive special character handling and whitespace normalization

The text cleaning follows a sophisticated multi-stage approach:

```
-- Stage 0.1: Remove numeric suffixes like /00032/
text := REGEXP_REPLACE(text, '/[0-9]{5}/', '', 'g');

-- Stage 0.2: Normalize delimiters and whitespace
text := REGEXP_REPLACE(text, '[\n\r\t]+', '', 'g');
text := REGEXP_REPLACE(text, '[|,+;\\\]', '/', 'g');
text := REGEXP_REPLACE(text, '/+', ' / ', 'g');
text := REGEXP_REPLACE(text, '\s{2,}', ' ', 'g');

-- Stage 0.3: Remove parenthetical content
text := REGEXP_REPLACE(text, '\([^()]*\)', '', 'g');

-- Stage 1-5: Configuration-driven cleaning phases
-- UNITS_OF_MEASUREMENT, MANUFACTURER_NAMES, WORDS_TO_VITAMIN_B,
-- FORMAT, CLEANING phases for both DRUGNAME and PROD_AI
```

Configuration-Driven Processing:

- `test_temp_table_creation_logic` - Tests DRUG_Mapper_Temp creation for batch processing with DISTINCT selection and NOTES IS NULL filtering
- `test_config_driven_cleaning_phases` - Tests JSONB configuration processing for dynamic cleaning rules across 10 distinct cleaning phases

The configuration-driven approach enables flexible cleaning rule application:

```
-- Dynamic cleaning based on JSONB configuration
FOR stmt IN
    SELECT jsonb_array_elements(phase_data -> 'replacements') AS value
LOOP
    EXECUTE format(
        'UPDATE %I SET %I = REPLACE(%I, %L, %L)',
        stmt.value->>'table',
        stmt.value->>'set_column',
        stmt.value->>'replace_column',
        stmt.value->>'find',
        stmt.value->>'replace'
    );
END LOOP;
```

Performance and Error Handling:

- `test_server_memory_handling_large_operations` - Tests memory management for large-scale text processing operations with comprehensive error scenario coverage including OutOfMemory, OperationalError, and DiskFull exceptions

Data Quality and Validation:

- `test_function_definition_structure` - Tests function definition structure validation for clearnumericcharacters and process_drug_data functions
- `test_regex_pattern_validation` - Tests regex pattern validation for text processing operations including numeric suffix, whitespace, and delimiter patterns
- `test_jsonb_operations_validation` - Tests JSONB operations for configuration handling including jsonb_array_elements and value extraction
- `test_dynamic_sql_structure` - Tests dynamic SQL generation structure using format() function with proper identifier and literal formatting
- `test_control_flow_validation` - Tests control flow structures including FOR loops, IF statements, and DECLARE blocks

Key Testing Patterns

Configuration-Driven Processing: Tests validate JSONB configuration processing for dynamic cleaning rule application, enabling flexible text processing workflows that can be modified without code changes through external configuration files.

Multi-Stage Text Processing: Comprehensive validation of sequential text cleaning operations (stages 0.1-0.3, phases 1-5) ensuring proper order of execution and cumulative effect of cleaning transformations on drug name standardization.

Regex Pattern Validation: Systematic testing of regular expression patterns used in text processing to ensure proper pattern matching and replacement operations for numeric suffixes, delimiters, and special characters.

Temporary Table Management: Testing of DRUG_Mapper_Temp table creation and cleanup for batch processing operations that handle large volumes of drug name data efficiently with proper memory management.

Dynamic SQL Generation: Validation of configuration-driven SQL generation using EXECUTE and format() functions that allow runtime modification of cleaning operations based on JSONB configuration data.

Error Recovery and Performance: Testing of memory management, connection timeout handling, and disk space management for large-scale text processing operations across millions of drug names with proper error recovery mechanisms.

12.9 Phase 9: Final Drug Mapping Integration

Phase 9 implements the final drug mapping operations that integrate cleaned drug names with RxNorm and IDD data sources using hierarchical mapping strategies with quality-based prioritization. This phase represents the culmination of drug standardization efforts, applying a sophisticated 12-tier priority system to achieve optimal mapping coverage and quality.

Test Suite Overview

The Phase 9 test suite validates final mapping operations, hierarchical priority logic, and comprehensive data integration:

- `test_s9.py` (unittest framework) - Pipeline orchestration and complex SQL parsing
- `test_s9.py` (pytest framework) - Final mapping operations with priority-based assignment

Pipeline Orchestration Tests (`test_s9.py` - unittest)

This test class validates the Python orchestration layer for the final mapping phase:
Configuration and SQL Processing:

- `test_load_config_success` - Validates configuration loading for final mapping operations
- `test_load_config_file_not_found` - Tests error handling for missing configuration files
- `test_load_config_invalid_json` - Validates handling of malformed JSON configuration
- `test_parse_sql_statements_with_updates_and_functions` - Tests parsing of complex UPDATE statements and functions

- `test_parse_sql_statements_with_bom_character` - Validates BOM character handling in SQL files
- `test_parse_sql_statements_complex_update_with_subqueries` - Tests parsing of complex UPDATE statements with subqueries
- `test_parse_sql_statements_multiple_consecutive_updates` - Tests parsing of multiple consecutive UPDATE operations

The SQL parsing handles sophisticated UPDATE operations with complex WHERE clauses:

```
UPDATE faers_b.DRUG_Mapper dm
SET mapped_rxcui = (
    SELECT rxcui
    FROM faers_b.rxn_mapping rm
    WHERE UPPER(rm.drug_name) = UPPER(dm.drug_name)
    LIMIT 1
)
WHERE dm.mapped_rxcui IS NULL
  AND EXISTS (
      SELECT 1
      FROM faers_b.rxn_mapping rm2
      WHERE UPPER(rm2.drug_name) = UPPER(dm.drug_name)
  );
```

Execution Reliability:

- `test_execute_with_retry_immediate_success` - Tests successful execution without retries
- `test_execute_with_retry_operational_error_recovery` - Tests retry logic for lock timeouts and operational errors

Final Mapping Operations Tests (`test_s9.py` - pytest)

This comprehensive test suite validates the sophisticated final mapping functionality:

Infrastructure and Dependency Management:

- `test_database_context_validation` - Validates connection to correct faers-database

- `test_placeholder_idd_table_creation` - Tests IDD table creation when missing with placeholder structure
- `test_column_existence_validation` - Tests comprehensive column existence validation across all expected drug mapper columns
- `test_cleaned_column_addition_logic` - Tests dynamic addition of CLEANED_DRUGNAME and CLEANED_PROD_AI columns

Data Transfer and Integration:

- `test_drug_mapper_temp_data_transfer` - Tests data transfer from DRUG_Mapper_Temp to main DRUG_Mapper table with proper WHERE clause filtering

The data transfer operation ensures cleaned data integrity:

```
UPDATE faers_b."DRUG_Mapper"
SET "CLEANED_DRUGNAME" = dmt."CLEANED_DRUGNAME"
FROM faers_b."DRUG_Mapper_Temp" dmt
WHERE dmt."DRUGNAME" = faers_b."DRUG_Mapper"."DRUGNAME"
AND faers_b."DRUG_Mapper"."NOTES" IS NULL;
```

Hierarchical Mapping Strategy:

- `test_rxnorm_direct_mapping_priority` - Tests highest priority mapping (notes 9.1, 9.2) with strict TTY filtering (MIN, IN, PIN)
- `test_idd_mediated_mapping_logic` - Tests IDD-mediated mapping (notes 9.3, 9.4) with complex INNER JOIN operations
- `test_fallback_mapping_any_tty` - Tests fallback mapping strategies (notes 9.9, 9.10) without TTY restrictions
- `test_cast_operations_for_data_types` - Tests CAST operations for RXAUI and RXCUI data type conversions

The mapping strategy employs a 12-tier priority system with strict quality controls:

```
-- Priority 9.1: Direct CLEANED_DRUGNAME with highest quality TTY
UPDATE faers_b."DRUG_Mapper"
SET
    "RXAUI" = CAST(rxn."RXAUI" AS BIGINT),
```

```
    "RXCUI" = CAST(rxn."RXCUI" AS BIGINT),
    "NOTES" = '9.1',
    "SAB" = rxn."SAB",
    "TTY" = rxn."TTY",
    "STR" = rxn."STR",
    "CODE" = rxn."CODE"
FROM faers_b."RXNCONSO" rxn
WHERE rxn."STR" = faers_b."DRUG_Mapper"."CLEANED_DRUGNAME"
AND faers_b."DRUG_Mapper"."NOTES" IS NULL
AND rxn."SAB" = 'RXNORM'
AND rxn."TTY" IN ('MIN', 'IN', 'PIN');
```

System Performance and Validation:

- `test_server_dependency_validation_cascade` - Tests cascading dependency validation across multiple schemas and tables with row count verification

Data Quality Assurance:

- `test_mapping_priority_order_validation` - Tests logical ordering of mapping priority notes
- `test_tty_filtering_validation` - Tests TTY filtering logic for quality-based mapping
- `test_notes_null_filtering_validation` - Tests NOTES IS NULL filtering to prevent mapping overwrites
- `test_join_relationship_validation` - Tests JOIN relationship integrity across tables

Key Testing Patterns

Hierarchical Priority Validation: Tests validate the 12-tier mapping priority system (9.1-9.12) ensuring highest quality matches are preferred while providing comprehensive fallback coverage.

Data Type Safety: Comprehensive testing of CAST operations and data type conversions ensuring numerical fields are properly typed for downstream analysis.

Dependency Chain Validation: Systematic testing of complex dependency relationships across faers_b and faers_combined schemas with proper error handling.

Quality-Based Filtering: Testing of TTY-based quality filtering (MIN, IN, PIN priority) ensuring optimal concept selection from RxNorm terminology.

12.10 Phase 10: Complex Drug Remapping Operations

Phase 10 implements sophisticated multi-step remapping operations that traverse RxNorm relationship hierarchies to find optimal drug concept mappings through complex multi-table JOINs and relationship analysis. This phase represents the most algorithmically complex mapping operations in the pipeline, utilizing advanced graph traversal techniques across pharmaceutical terminology relationships.

Test Suite Overview

The Phase 10 test suite validates complex remapping algorithms, multi-step processing workflows, and advanced relationship traversal:

- `test_s10.py` (unittest framework) - Pipeline orchestration and database validation
- `test_s10.py` (pytest framework) - Complex remapping operations and relationship processing

Pipeline Orchestration Tests (`test_s10.py` - unittest)

This test class validates the Python orchestration layer for complex remapping operations:

Configuration and Database Management:

- `test_load_config_success` - Validates configuration loading for remapping operations
- `test_load_config_missing_file` - Tests error handling for missing configuration files
- `test_load_config_invalid_json` - Validates handling of malformed JSON configuration
- `test_check_postgresql_version` - Tests PostgreSQL version compatibility checking
- `test_check_database_exists_true` - Tests database existence validation for faersdatabase

- `test_check_database_exists_false` - Tests handling of missing database scenarios

SQL Processing and Table Management:

- `test_parse_sql_statements_with_do_blocks` - Tests parsing of complex DO blocks and table creation statements
- `test_parse_sql_statements_empty_input` - Tests handling of empty or comment-only SQL files
- `test_verify_tables_with_data` - Tests table verification with row count validation across multiple remapping tables

Execution Reliability:

- `test_execute_with_retry_immediate_success` - Tests successful execution without retries
- `test_execute_with_retry_duplicate_table_skip` - Tests graceful handling of duplicate table errors

Complex Remapping Operations Tests (`test_s10.py` - pytest)

This comprehensive test suite validates the sophisticated remapping functionality:

Infrastructure and Table Creation:

- `test_database_context_validation` - Validates connection to correct faers-database
- `test_remapping_table_creation` - Tests creation of drug_mapper, drug_mapper_2 with remapping columns and proper data types
- `test_logging_infrastructure_setup` - Tests remapping_log table creation for progress tracking and audit trails
- `test_performance_indexes_creation` - Tests strategic index creation with INCLUDE clauses for covering indexes

The remapping infrastructure includes sophisticated logging and performance optimization:

```
CREATE TABLE IF NOT EXISTS faers_b.remapping_log (
    log_id SERIAL PRIMARY KEY,
    step VARCHAR(50),
    message TEXT,
    log_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX IF NOT EXISTS idx_rxnconso_rxcui
ON faers_b.rxnconso(rxcui)
INCLUDE (rxaui, str, sab, tty, code);
```

Multi-Step Remapping Workflow:

- `test_step_1_initial_rxnorm_update_logic` - Tests Step 1 initial RXNORM update with IN TTY filtering and dependency validation
- `test_step_2_complex_join_logic` - Tests Step 2 complex 4-table JOIN operations with CASE statement logic for conditional remapping
- `test_manual_remapping_integration` - Tests Steps 3-5 manual remapping workflow integration with external mapping tables
- `test_vandf_and_relationship_processing` - Tests Step 6 VANDF relationship traversal with HAS_INGREDIENTS relationship processing
- `test_duplicate_cleanup_and_optimization` - Tests Step 17 duplicate cleanup using ROW_NUMBER() window functions

The multi-step workflow implements sophisticated relationship traversal:

```
-- Step 6: VANDF relationship processing (4-table JOIN)
INSERT INTO faers_b.drug_mapper_2
SELECT e.drug_id, e.primaryid, '3' AS remapping_notes,
       a.rxaui AS remapping_rxaui,
       a.rxcui AS remapping_rxcui
FROM faers_b.rxnconso a
INNER JOIN faers_b.rxnrel b ON a.rxcui = b.rxcui1
    AND a.tty = 'IN' AND a.sab = 'RXNORM'
INNER JOIN faers_b.rxnconso c ON b.rxcui2 = c.rxcui
INNER JOIN faers_b.rxnrel d ON c.rxcui = d.rxcui1
    AND d.rela = 'HAS_INGREDIENTS' AND c.sab = 'VANDF' AND c.tty = 'IN'
INNER JOIN faers_b.drug_mapper_2 e ON d.rxcui2 = e.rxcui
```

```
WHERE e.remapping_notes IS NULL;
```

Performance and Error Handling:

- `test_server_memory_handling_complex_operations` - Tests memory management for large multi-table JOIN operations with comprehensive error scenario testing

Data Quality and Validation:

- `test_function_definition_structure` - Tests function definition structure with proper DECLARE, BEGIN, EXCEPTION blocks
- `test_remapping_notes_progression` - Tests logical progression of remapping notes (1, 2, 3, 7-15)
- `test_table_dependency_validation` - Tests table dependency validation across drug_mapper_*, rxnconso, rxnrel tables
- `test_case_statement_validation` - Tests CASE statement logic for conditional value assignment
- `test_exclusion_lists_validation` - Tests exclusion list functionality for filtering specific RXAUI values

Key Testing Patterns

Multi-Step Workflow Validation: Tests validate the 15-step remapping process ensuring proper execution order and dependency management across complex pharmaceutical relationship hierarchies.

Relationship Traversal Testing: Comprehensive validation of graph traversal algorithms through RxNorm and VANDF relationship networks using sophisticated multi-table JOIN operations.

Performance Optimization: Testing of covering indexes with INCLUDE clauses and memory management for large-scale relationship processing operations.

Manual Integration Workflows: Validation of manual remapping integration allowing human expert intervention in complex mapping scenarios while maintaining audit trails.

12.11 Phase 11: Final Dataset Creation and Statistical Analysis

Phase 11 implements the final dataset creation and comprehensive statistical analysis operations, producing standardized tables for pharmacovigilance analysis including contingency tables, proportionate reporting ratios (PRR), and information components (IC). This phase transforms the mapped FAERS data into analysis-ready datasets with sophisticated statistical calculations for regulatory reporting and safety signal detection.

Test Suite Overview

The Phase 11 test suite validates dataset creation, statistical analysis operations, and regulatory reporting functionality:

- `test_s11.py` (unittest framework) - Pipeline orchestration and complex dollar-quote parsing
- `test_s11.py` (pytest framework) - Statistical analysis and dataset creation operations

Pipeline Orchestration Tests (`test_s11.py` - unittest)

This test class validates the Python orchestration layer for final dataset creation:

Configuration and Advanced SQL Processing:

- `test_load_config_success` - Validates configuration loading for analysis operations
- `test_load_config_file_not_found` - Tests error handling for missing configuration files
- `test_load_config_invalid_json` - Validates handling of malformed JSON configuration
- `test_parse_sql_statements_with_dollar_quotes` - Tests parsing of complex dollar-quoted functions and statistical procedures
- `test_parse_sql_statements_with_bom_character` - Validates BOM character handling in statistical SQL files

- `test_parse_sql_statements_incomplete_statement_warning` - Tests handling of incomplete statements with warning logging
- `test_parse_sql_statements_complex_dollar_quotes` - Tests parsing of nested dollar-quoted blocks with complex statistical functions

The SQL parsing handles sophisticated statistical function definitions:

```
$$
CREATE OR REPLACE FUNCTION faers_b.complex_analysis()
RETURNS TABLE(drug_name TEXT, reaction_count INTEGER) AS $body$
DECLARE
    sql_query TEXT;
BEGIN
    sql_query := 'SELECT drug_name, COUNT(*) FROM faers_b.analysis_view';
    RETURN QUERY EXECUTE sql_query;
END
$body$ LANGUAGE plpgsql;
$$
```

Execution Reliability:

- `test_execute_with_retry_immediate_success` - Tests successful execution without retries
- `test_execute_with_retry_duplicate_index_skip` - Tests graceful handling of duplicate index errors during analysis table optimization

Statistical Analysis and Dataset Creation Tests (`test_s11.py` - pytest)

This comprehensive test suite validates the final analysis and dataset creation functionality:

Infrastructure and Core Table Creation:

- `test_database_context_validation` - Validates connection to correct faers-database
- `test_drugs_standardized_table_creation` - Tests creation of drugs_standardized table with RxNorm mapping and exclusion filtering

- `test_adverse_reactions_with_meddra_integration` - Tests adverse_reactions table creation with MedDRA integration using CTE structures
- `test_drug_adverse_reactions_pairs_creation` - Tests unique drug-reaction pair generation with DISTINCT operations

The standardized tables form the foundation for statistical analysis:

```
-- Drugs standardized with exclusions
INSERT INTO faers_b.drugs_standardized
SELECT dm.primaryid, CAST(dm.drug_id AS INTEGER), dm.drug_seq,
       dm.role_cod, dm.period,
       CAST(dm.remapping_rxaui AS BIGINT) AS rxaui,
       dm.remapping_str AS drug
FROM faers_b.drug_mapper_3 dm
INNER JOIN faers_combined.aligned_demo_drug_reac_indi_ther ad
    ON dm.primaryid = ad.primaryid
WHERE dm.remapping_rxaui IS NOT NULL
  AND dm.remapping_rxaui != '92683486'; -- Excludes 'UNKNOWN STR'
```

Aggregation and Statistical Processing:

- `test_aggregation_and_counting_logic` - Tests COUNT aggregations and margin calculations for drug-adverse reaction combinations
- `test_demographics_with_data_conversion` - Tests demographic table creation with date parsing (TO_DATE) and age validation using regex patterns
- `test_contingency_table_calculation_logic` - Tests 2x2 contingency table mathematics with proper cell value calculations (a, b, c, d)

Advanced Statistical Calculations:

- `test_proportionate_reporting_ratio_calculations` - Tests PRR calculations with 95% confidence intervals using natural logarithm transformations
- `test_information_component_calculations` - Tests Information Component (IC) statistical formulas with Bayesian confidence propagation

The statistical analysis implements sophisticated pharmacovigilance algorithms:

```
-- PRR Calculation with 95% Confidence Intervals
SELECT ct.rxaui, ct.drug, ct.adverse_event, ct.a,
       (ct.a / NULLIF((ct.a + ct.c), 0)) /
       NULLIF((ct.b / NULLIF((ct.b + ct.d), 0)), 0) AS prr,
       EXP(LN(prr) - 1.96 * SQRT(
            (1.0 / ct.a) - (1.0 / (ct.a + ct.c)) +
            (1.0 / ct.b) - (1.0 / (ct.b + ct.d))
       )) AS prr_lb,
       EXP(LN(prr) + 1.96 * SQRT(
            (1.0 / ct.a) - (1.0 / (ct.a + ct.c)) +
            (1.0 / ct.b) - (1.0 / (ct.b + ct.d))
       )) AS prr_ub
FROM faers_b.contingency_table ct
WHERE ct.a > 0 AND ct.b > 0 AND ct.c > 0 AND ct.d > 0;
```

Performance and Error Handling:

- `test_server_complex_statistical_operations` - Tests complex statistical operations with comprehensive error handling for numerical overflow, division by zero, and floating point exceptions

Data Quality and Validation:

- `test_statistical_formula_validation` - Tests statistical function validation (LOG, EXP, SQRT, POWER, NULLIF)
- `test_date_conversion_patterns` - Tests date conversion pattern validation with NULLIF and TO_DATE functions
- `test_exclusion_filters_validation` - Tests exclusion filter logic for data quality assurance
- `test_aggregation_functions_validation` - Tests aggregation function validation (COUNT, SUM, GROUP BY)
- `test_index_creation_patterns` - Tests strategic index creation for analysis table performance optimization

Key Testing Patterns

Statistical Algorithm Validation: Comprehensive testing of pharmacovigilance statistics including PRR, IC, and confidence interval calculations with proper mathematical error handling.

Data Integration Testing: Validation of complex CTE operations that integrate MedDRA, RxNorm, and FAERS data into coherent analysis datasets.

Performance Optimization: Testing of strategic indexing and query optimization for large-scale statistical analysis operations across millions of records.

Regulatory Compliance: Validation of data exclusion filters, date parsing, and statistical formula accuracy to ensure regulatory reporting compliance.

Numerical Robustness: Comprehensive testing of mathematical edge cases including division by zero protection, numerical overflow handling, and floating point error management.