**Basic database structure**

* PostgreSQL database, version 15.2
  + Hosted on ERG server accessible by ERG VPN only.
  + Currently about 13GB in size.
* **State schemas** (xx\_ust and xx\_release)
  + Source data from the states is imported into state-specific “staging” schemas. Usually the state schemas will contain only the most recent dataset from the state (i.e. when we receive a new dataset we will wipe out the previous data from the schema and import the new data).
  + States submit data in a number of different ways, which affects how data are loaded into the state schemas.
    - If the state submits Excel, CVS, or other flat files (or makes such files available for download from a state website), the developer imports the files into the appropriate schema (this can be done manually or there is a Python script that automates it).
    - If the state uses an API, the developer will typically create a table structure in the state schema that mimics the format of the data in the API and write a script that calls the API and populates the table(s).
    - If the state submits an Access database or a backup file for some other database type, the developer will restore the database locally (in some cases this may be done by an ERG DBA on an ERG server) and will copy the state’s data from their original database into the ERG database, table by table.
      * In a few cases (such as Oregon UST), the state’s source database is so large that it is impractical to import the entire dataset into the ERG database. In these instances, ERG examines the source data and copies over just the tables that contain data that is mappable to the EPA template. (The Oregon UST database is particularly large because it contains a lot of images, which ERG ignores.)
    - There may arise a situation in which ERG is given direct access to a state database or portal. ERG will download/export the data in whatever method is available and upload it to the ERG database in a format that matches the original data source as closely as possible.
      * ERG has read-only access to the TrUSTD database at EPA. During the pilot, ERG wrote a script to save the tables they needed to CSV files, then imported the CSV files into the ERG database for processing.
      * California is an example of a state that has a portal to which ERG was granted access and from where data can be queried and downloaded on a county-by-county basis. ERG downloaded the data for each county as individual Excel files, then wrote a script to combine all Excel files into a single data source, then uploaded this source to the ERG database.
    - If the state submits the data in a populated EPA template, each template tab is imported into the state schema as a table.
      * Data submitted in the populated EPA template format is processed very similarly to any other state. ERG found it was impossible to automate the processing of populated templates much more than other datasets because states were inconsistent in their population and the templates needed to be thoroughly analyzed for adherence to the template format and compliance with the business rules.
  + If the developer needs to create any tables in the state schemas, for example, to generate ID columns, deaggregate state values into separate rows, concatenate multiple state columns into a single column, or for any other reason, they prefix the table name with “erg\_” so it is clear the table was ERG’s creation and not part of the original source data.
  + Developers will create views in the state schemas as part of the data processing. Views do not need to have “erg\_” in the name but should be named in such a way that it is obvious what their purpose is.
    - The automated programming process creates specific crosswalk views for each lookup table the state has values for; these crosswalk views utilize a specific naming convention (e.g., v\_facility\_type\_xwalk may be created to display the mapping from the state’s facility type values to EPA’s).
    - The developers write “data population views” that transform the state data into the EPA template format. These views are used to perform the inserts into the EPA data tables (see below).
* **“public” schema** 
  + This schema is the “EPA” schema, where the state data is compiled and stored in the EPA template format. It also includes many metadata, lookup, and mapping tables.
  + There are **comments** on most tables in this schema to explain their purpose/function. To see the comment, run the following query:
    - select get\_table\_comment('[TABLE\_NAME]'), where [TABLE\_NAME] is the name of the table.
      * Function **get\_table\_comment** takes two parameters; the first is the table or view name and the second is the schema name, with the latter defaulting to the “public” schema. It’s unlikely there are any comments on any objects in schemas other than public, so generally you can call it with a single parameter.
  + **Control tables** – one row per dataset; high-level information about the dataset such as organization\_id, date received, original data source, etc. The primary keys on these tables are used to uniquely relate a state dataset to the mapping and main data tables. The comment field of this table is also used to describe exclusions or other general technical information about the entire dataset that needs to be applied during processing; for example, if the source data contains both USTs and ASTs, the developer will enter a note in the comment field of the control table stating that ASTs will be excluded. The two control tables are:
    - **ust\_control** – primary key is ust\_control\_id
    - **release\_control** – primary key is release\_control\_id
  + Main **EPA data tables** – These tables are structured in the exact format of the EPA template, are used to populate the review spreadsheet, and will also be the source of the data uploaded to the UST Finder application.
    - Contains data from all states; datasets are identifiable by ust\_control\_id and release\_control\_id.
    - These tables contain many database constraints that enforce the business rules from the EPA template.
    - EPA UST tables (required tables marked with an asterisk):
      * ust\_facility \*
        + ust\_tank \*

ust\_tank\_substance

ust\_tank\_dispenser

ust\_compartment \*

ust\_compartment\_substance (is also a child of ust\_tank\_substance)

ust\_piping

ust\_compartment\_dispenser

ust\_tank\_dispenser

* + - * + ust\_facility\_dispenser
    - EPA Release tables (required tables marked with an asterisk):
      * ust\_release \*
        + ust\_release\_substance
        + ust\_release\_source
        + ust\_release\_cause
        + ust\_release\_corrective\_action\_strategy
  + **Lookup tables**
    - Usually have table names that are plural/end in “s” (e.g. “substances”)
      * Lookup tables are documented in the ust\_element\_lookup\_tables and release\_element\_lookup\_tables tables.
    - Relate directly to an element and business rule in the EPA template.
    - Most contain an ID and description column, with the ID column being an incremental number assigned by the database.
      * The substances lookup table contains additional columns related to grouping, federally regulated substances (this column needs EPA revision), and hazardous substances (this column also needs EPA revision).
      * The compartment\_statuses lookup table contains additional columns related to the hierarchy that should be used to roll conflicting compartment statuses up to a tank status if the state does not provide a separate tank status.
    - Ideally we’d like to add a sort\_order column to the lookup tables in the future. (EPA templates are generated by a Python script that queries the database; it would be beneficial to be able to order the allowed values appropriately as we add additional values over time.)
    - When doing the element mapping, the ID column name for the epa\_column\_name (e.g. “substance\_id”) is inserted into the element mapping table (ust\_element\_mapping or release\_element\_mapping), even if the column in the source data being mapped is a description/text column, not an ID column.
  + **Mapping tables** (and views)
    - UST
      * ust\_element\_mapping – maps source data elements to EPA template data elements.
      * ust\_element\_value\_mapping – maps *values* for specific data elements from source data to EPA template format.
      * v\_ust\_element\_mapping – a view that combines the two tables above for easy querying
    - Releases
      * release\_element\_mapping– maps source data elements to EPA template data elements.
      * release\_element\_value\_mapping– maps *values* for specific data elements from source data to EPA template format.
      * v\_release\_element\_mapping- a view that combines the two tables above for easy querying
  + **Metadata/informational tables** 
    - Often used by the Python scripts to generate code.
    - Can be used by the developers to understand the database structure and processes.
    - There are also a lot of views written on these tables for specific purposes, both for the Python scripts and developer queries.
    - See table comments (database metadata) for additional information about their use.
    - UST-specific metadata tables:
      * ust\_elements
      * ust\_elements\_tables
      * ust\_element\_table\_sort\_order
      * ust\_element\_allowed\_values
      * ust\_element\_lookup\_tables
      * ust\_template\_data\_tables
      * ust\_template\_lookup\_tables
      * ust\_view\_key\_columns
      * ust\_required\_view\_columns
      * generated\_table\_sort\_order
    - Release-specific metadata tables:
      * release\_elements
      * release\_elements\_tables
      * release\_element\_table\_sort\_order
      * release\_element\_allowed\_values
      * release\_element\_lookup\_tables
      * release\_template\_data\_tables
      * release\_template\_lookup\_tables
      * release\_view\_key\_columns
    - **Performance measure tables** – stores performance measure data that is extracted from a quarterly publication from EPA; *used for informational purposes only in the review materials*. Most developers will not need to use these tables for any reason. There is a Python script (UST/ust/python/util/refresh\_performance\_measures.py that assists in updating the numbers in the database tables below, however, it is a bit complex to run. Eventually we’d like to create a better system for importing/querying the performance measure numbers.
      * performance\_measure\_ust
      * performance\_measure\_release
    - **CUI “stopword” table** – ERG developed a partially automated/partially manual process for identifying possible CUI in facility names (prevention side) and site names (releases side). As part of the automated process, words or phrases that are very likely to be used in a name that is NOT CUI, or “stopwords”, are stored in this table (with a “maybe\_flag” marking a few words that ARE likely to be CUI). Additional stopwords are manually added to this table each time CUI is processed, so the CUI identification procedure continuously improves over time, with fewer and fewer rows needing a close manual inspection.
      * cui\_exclusions
    - **Temp tables** – Developers may occasionally create a table for a specific, ad hoc purpose. These will generally use the convention of a “temp\_” prefix in the name (and/or a datestamp suffix) and can be ignored/dropped. They do not store permanent data.

**ERG data processing workflow (followed by developers)**

1. Set Jira ticket status to “Data Mapping in Progress” and assign yourself as the Assignee.
2. Set up the repo.
   1. In your local repo, on the main branch, do a git pull to pull down the newest code updates.
   2. Create a branch for your ticket and switch to it.
   3. If it does not exist, create a new folder in the repo for your processing scripts: UST/ust/sql/states/XX/UST or UST/ust/sql/states/XX/Releases, where XX is the state code.
   4. Copy the relevant SQL processing template (UST/ust/sql/templates/UST.sql or UST/ust/sql/templates/releases.sql) and paste it into the folder above. Name the copy of the script XX\_UST.sql or XX\_Releases.sql.
   5. Follow the instructions at the top of the processing template to do a global replace of “XX” in the processing template with the state code and save it.
3. Follow the steps in the SQL processing script.
   1. The scripts are heavily commented. Read the instructions for each step carefully.
   2. Some of the steps instruct you to run a Python script. Some of the Python scripts generate additional SQL scripts that are exported to UST/ust/sql/states/XX/UST or UST/ust/sql/states/XX/Releases, which you will then open in DBeaver, possibly edit, and then run.
   3. Do NOT commit changes to the Python scripts to the repo after setting the variables at the top of them.
      1. If you need to modify a script in some way to make it work for a specific state, COPY the script, save it to UST/ust/sql/states/XX/UST or UST/ust/sql/states/XX/Releases, and make the changes in the copy. You can commit the copy to the repo.
      2. If you find a bug in the script, let Renae know.
4. When all steps of the SQL processing script have been completed, the data are in the EPA tables, and the review materials have been exported, request a peer review for the ticket.
5. Address any issues raised in the peer review and repeat until the reviewer approves the ticket to be sent to OUST.
6. Victoria will open the review with OUST.
7. OUST will make comments in the review spreadsheets and will share their feedback during the bi-weekly Tuesday morning meetings. Do your best to attend any meeting where they are discussing a ticket you worked on.
8. Address any issues raised by the OUST review and repeat until OUST approves the ticket to go to the state for review.
9. Address any issues raised by the state during their review and repeat until the state approves the data for UST Finder.
10. If necessary, ERG will geocode the data before it is ready to be exported to the UST Finder app.

**Broad overview of the SQL processing script**

***For the most detailed explanation of each of the processing steps, see the detailed comments in the two “processing templates”: UST.sql and release.sql*** *(found in the repo at ust/sql/templates).*

1. Import source data into ERG database
   1. Source data goes into state-specific schemas (xx\_ust and xx\_release, where xx is the lowercase state code)
2. Make an entry for the dataset in the appropriate control table (ust\_control or release\_control).
3. Examine the source data and map the source data elements to the EPA template data elements.
4. Where appropriate, map the data element values to the EPA template lookup table values.
5. Deaggregate any data that has been rolled up into multiple values in a single row in the source data so we end up with a single value per row.
6. If any required ID fields do not exist in the source data, generate those fields and map them.
7. Write views in the state schema that manipulate the source data to the EPA template format.
8. Run the Python script that QAs the views to ensure they are valid.
9. Insert the data into the EPA data tables.
10. Export the review materials:
    1. Control table summary
    2. QAQC spreadsheet
    3. Populated EPA template
11. Upload review materials to EPA Teams site.

**Github repository structure**

<https://github.com/Eastern-Research-Group/UST>

* **arcgis** – [Jim]
* **ust**
  + **python**
    - **adhoc** – standalone scripts used for a specific one-time purpose that have been saved as they may be useful again later.
    - **backups** – The ERG database is officially backed up in its entirety once a week (to a PostgreSQL backup file). The scripts in this directory export database objects locally on an ad hoc basis.
      * **backup\_tables**.py – Exports certain tables – (optionally) including their contents – to CSV files. Categorizes tables in the public schema as mapping, data, metadata, lookup, and performance\_measures and allows categories of tables to be exported at a time.
      * **create\_indexes**.py – Creates indexes on ID columns in the public schema. (This was used as a one-time script but saved as it could be repurposed for other uses.)
      * **save\_ddl**.py – Exports the DDL of tables, views, and functions to individual .sql files (does NOT export data, only object structure) in the repository (or to a specified directory). By default, running this script saves the DDL for all tables, views, and functions in the public schema.
      * **save\_source\_ddl**.py – Calls the save\_ddl.py script for each state schema (instead of the public schema).
    - **data\_analysis**
      * **element\_row\_counts**.py – Generates the “coverage” spreadsheet (row counts for each data element in the EPA template, by state) and emails it to Victoria, who forwards it to Fran. Script expects Outlook to be installed and connected on the computer its being run on; to skip the automated email, set the send\_email parameter to False. A senior developer is scheduled to run this script every Thursday morning.
      * **haz\_subs.**py – Used in the past to assist in ad hoc analyses from Fran regarding hazardous substances; compiles all state CAS Numbers into a single table for easier querying.
    - **example\_schema** – Originally intended as a training tool for new developers. Can be ignored for now.
    - **exports** – This is the directory to which any of the scripts in the repo will export files they generate (other than generated SQL scripts, which are exported to a different location; see below). The scripts will create subdirectories such as control\_table\_summaries, cui, epa\_templates, mapping, other, QAQC, and source\_data, each of which will be further organized by individual state folders. This directory is included in the .gitignore file which means files that are exported to this location will not get pushed to the repo unless overridden. (This is to prevent unnecessary binary files from clogging up the repo.)
    - **imports** – This directory is simply a place files can be dropped so they can be read and acted upon by scripts. This directory is included in the .gitignore file which means files that are exported to this location will not get pushed to the repo unless overridden. (This is to prevent unnecessary binary files from clogging up the repo.)
    - **state\_processing** – Most of the scripts called by the main processing templates (UST.sql and releases.sql) are located in this directory.
      * **archive** – Directory containing scripts from the UST Finder pilot, which were saved as a starting point for UST Finder 2. These can all probably be deleted at this point.
      * **control\_table\_summary**.py – Exports the control table summary review spreadsheet that is part of the OUST review package.
      * **create\_missing\_id\_columns**.py – Checks the state source data for required ID columns that are not used by the state and creates these columns by creating system-generated IDs. For example, if a state does not report at the compartment level, they will not have CompartmentIDs in their data, however, CompartmentID is a required field, so this script will generate IDs and assign them to facility/tank combinations.
      * **create\_view\_sql**.py – This script is under construction and does not currently work as intended. The plan is for it to automatically generate the SQL required to write the views that populate the EPA data tables, by reading the ust\_element\_mapping and release\_element\_mapping tables.
      * **cui\_check**.py – This script performs Step 1 of the automated part of the CUI identification process, which exports a spreadsheet marking as many rows as possible as TRUE or FALSE for possible CUI. Victoria reviews the results and assigns any rows without a value a TRUE or FALSE, then sends the final version back to Renae to import into the database.
      * **cui\_import**.py - This script reads a final CUI file from Victoria (all rows now have a TRUE or FALSE value) and saves it back to the state schema in the database.
      * **cui\_import\_update\_export**.py – This script calls cui\_import.py and cui\_update.py, in that order, and then exports a new marked-up EPA template, to automate Step 2 of the automated part of the CUI identification process.
      * **cui\_update**.py – This script copies the verified CUI TRUE/FALSE flags from the state schema and sets the CUI flag on the EPA data table.
      * **deagg**.py – When a state submits data with multiple values in a single row, such as multiple substances in a single facility/tank row, we need to create a lookup table of the unique allowed values (for example, a list of unique substances used by the state, with one substance per row). This script will create a new table in the state schema and populate it with a single row per unique value.
      * **deagg\_rows**.py – This script should be run AFTER deagg.py. It creates a new table in the state schema and inserts the unique combination of primary key columns plus individual values. For example, if a state submits multiple substances per FacilityID/TankID, this script will create a table with one row per FacilityID/TankID/SubstanceID.
      * **element\_mapping\_to\_excel**.py – Called by other scripts to build an Excel worksheet that contains the element value mapping performed by ERG for review by OUST and the state. Not intended as a standalone script. (This script should probably be moved to the utils directory.)
      * **exclude\_unregulated**.py – Generates (and optionally executes) new view SQL that excludes unregulated heating oil tanks. Optionally calls find\_unregulated.py first to identify the unregulated tanks.
      * **export\_all\_review\_materials**.py – Calls control\_table\_summary.py, export\_template.py, and qa\_check.py, which produce all three review materials that are sent to OUST, from a single script.
      * **export\_source\_data**.py – Exports the source state data from the ERG database and saves it to CSV file(s). This script is run when the state data was submitted in a format other than a database, Excel, or flat file (for example, through an API), making the source data in its original format unavailable to OUST during their review. So that OUST can see the source data, ERG exports the tables containing the source data to CSV files and uploads it to the EPA Teams site. ERG only runs this script for states where the original source data is unavailable to OUST.
      * **export\_substance\_mapping**.py – Creates an Excel spreadsheet that contains the substance value mapping and (when run on a machine with Outlook running and connected) automatically emails it to John Wilhelmi to perform a review of possible hazardous substances.
      * **export\_template**.py – Generates an EPA template. Depending on how the variables are set, this can be an empty, unpopulated template to be distributed to the states, a completely populated template with state data and mapping, or a “data only” template with state data but no reference tabs.
      * **find\_unregulated**.py – Identifies unregulated heating oil tanks. Must be run after the state schema data population views are written (v\_ust\_facility, etc.); see the processing templates for more information.
      * **generate\_deagg\_code**.py – Checks source state data for columns that need to be deaggregated because the state put multiple values into a single row. Generates a state-specific SQL file the developer can run to determine whether deaggregation is necessary, with variable values they can copy and paste into the appropriate Python scripts (deagg.py and deagg\_rows.py) to perform the deaggregation.
      * **generate\_value\_mapping\_sql**.py – Generates insert statements to populate ust\_element\_value\_mapping and release\_element\_value\_mapping and saves them to a SQL file. The developer then maps the state value in each insert statement to the appropriate EPA value and then executes the insert statements.
      * **import\_data\_from\_files**.py – Reads a directory from a variable and uploads all .xlsx and .csv files it finds as tables in the appropriate state schema of the database. Used to automate uploading state source data to the ERG database.
      * **insert\_control**.py – Inserts a new row into ust\_control or release\_control when beginning processing of a new state dataset. This insert statement can also be written and run manually if the developer prefers.
      * **org\_mapping\_xwalks**.py – After the developer has performed all of the element value mapping, this scripts creates individual crosswalk views between the state values and EPA values for all lookup tables. These crosswalk views are joined to when writing the data population views (e.g., v\_ust\_facility, etc.)
      * **populate\_epa\_data\_tables**.py – Uses the data population views in the state schemas to insert data into the main EPA data tables in the public schema. Optionally deletes existing data from the EPA tables first.
      * **qa\_check**.py – Analyzes all data population views in a state schema to see if they conform to the EPA template format. In theory, once the qa\_check script returns no “bad” rows for all views, the execution of the populate\_epa\_data\_tables.py script will be successful because no database constraints were violated. Developers are instructed to run the qa\_check script, address any errors it identifies, then run the script again, and repeat until the script returns no errors. The script generates the QAQC spreadsheet that is one of the three review materials sent to OUST.
      * **repopulate\_and\_reexport**.py – Allows the developer to find unregulated tanks, populate the EPA data tables, export all three OUST review materials, and run the automated peer review, or any combination of these four steps, from a single script. Useful after addressing OUST or state comments, just before re-opening the review.
    - **ust\_finder** – [Jim]
    - **utils** – Contains useful functions and classes.
      * **add\_template\_field\_to\_db**.py – If EPA decides to add a new data element to the EPA template, this script should be used to do so, as several metadata tables, etc. need to be updated in addition to the DDL changes. Contains a comment with instructions on how to do the same thing manually in the database.
      * **config**.py – Contains database connection information and email address settings for certain scripts that send emails using Outlook.
      * **database\_importer**.py – Class used by the functions and scripts that upload source data files to the ERG database.
      * **dataset**.py – Class used by many of the scripts to describe a source dataset.
      * **delete\_data**.py – Deletes all data from the EPA data tables for a specific dataset.
      * **emailer**.py – Class used to send emails from Outlook.
      * **export\_query**.py – Exports the results of any valid SQL query to an Excel file.
      * **export\_table**.py – Exports a single database table to an Excel file. Will accept a list of column names to limit the export to.
      * **find\_unpopulated\_columns**.py – Counts the number of rows without null values for all columns in a table (or all tables in a schema). This script is used when processing a state-populated template. States will not populate all columns of the template, and developers are instructed to only perform mapping for the data elements that contain data. This script identifies those data elements that should not be mapped because they contain no data.
      * **import\_factory**.py - Class used by the functions and scripts that upload source data files to the ERG database.
      * **import\_service**.py - Class used by the functions and scripts that upload source data files to the ERG database.
      * **logger\_factory**.py – Class the builds the logging function of the Python scripts.
      * **peer\_review**.py – Compares the number of rows in the state schema data population views with the number of rows in the EPA data tables for a specific dataset, and if there are mismatches, generates SQL queries that identify the missing rows for further investigation. This is just one step of a peer review; eventually we’d like to add more automated review steps to this script.
      * **refresh\_performance\_measures**.py – Extracts certain data from the quarterly performance measures document published by EPA and uploads it to the performance measure tables in the database. Because we have to extract tables from a PDF file first, this is not a smooth process; there is more information about the process in the comments in this script.
      * **upload\_general\_file**.py – Uploads any Excel or CSV file to a specified schema/table(s) in the database.
      * **utils**.py – A collection of useful functions called by other scripts in this library.
    - **utm\_conversion** 
      * **utm\_conversion**.py – Converts UTM coordinates in a CSV file to lat/longs. Ideally this will be converted to a database function and/or re-written and moved into utils.
  + **resources** – contains the EPA templates and some documentation.
  + **sql** – Directory that contains the processing SQL templates, as well as all other SQL scripts written by developers or automatically generated by the Python scripts.
    - **ddl** –The save\_ddl.py script exports all DDL scripts to this location, organized by schema then object type (table, view, or function).
    - **states** – Developers can save any state-specific scripts in these directories. Additionally, the Python scripts that generate customized SQL statements/queries export them to .sql files in this directory.
    - **templates** – This is where the UST.sql and releases.sql processing templates are stored.