Executive Summary – April 11, 2021

Subject: GIS Data Quality Reporting Database Framework and Dashboard

From: Alex Zarley

Overview

Data quality reporting for the GIS team at the electric utility where I work is currently done on an ad-hoc basis. Each data quality project is created, maintained, and tracked separately. Therefore, there is no straightforward way for a manager to visualize, understand, and explore all the data quality issues that exist in our GIS currently or historically. Furthermore, analysts on my team must write new queries each time our managers need to provide progress reports for data quality cleanup to company executives.

To solve these issues, I created a database-focused data quality reporting framework for the GIS team at my company. The set of Python and SQL scripts I wrote automates nearly every step of the workflow for cataloging projects and stores a common set of spatial, attribute, and temporal data for each error in a centralized table. To visualize data quality errors, I created an ESRI ArcGIS Online Dashboard to allow managers to explore GIS data quality issues spatially and temporally.

Audience

GIS managers will use the dashboard to explore and track all our data quality cleanup projects from a centralized dashboard with an interactive map and several data visualizations. This should reduce the number of data quality report queries they need to request from analysts on the team. GIS Analysts on my team will also use the framework and database tables. Since errors are tracked with a specific set of spatial, temporal, and attribute information, we can also develop queries and views to produce standardized progress reports without having to write project-specific reporting queries. Having all identified data quality errors in our GIS in a single table will allow our team greater clarity into the overall health of our data and help us determine where to focus our cleanup efforts to have the greatest impact.

Deliverables

A version of the AGOL Dashboard I created with dummy data for the state of Wisconsin can be found here: https://uw-mad.maps.arcgis.com/apps/dashboards/47d451e1f13f4c5fb14587b8fba7c430. I included pertinent sections of Python and SQL code in the Appendix. All code for the project is on my company's servers.

Data Collection

Geospatial data was pulled from my company's enterprise geodatabase or generated from data quality analyses I had written.

Framework Design and Architecture

I created my relational database tables in MS SQL Server and the dashboard in ArcGIS Online using Dashboards Beta. The ETL process (Fig. 1) uses a combination of manual entry in Microsoft Excel and scheduled and manually run Python 2.7, Python 3.8, and Transact SQL scripts and statements.

I created five new database tables (Fig 2.) to store project and error data and created a SQL stored procedure to create the individual issues table for the project when its data is initially loaded from the Excel spreadsheet (Fig. 7). I created two tables in our Reference schema (REF) to store information about datasets. REF.DATASET includes basic information about the datasets in our GIS. We perform analyses in several replica databases, so REF.DATASET_DETAILS stores information about the specific version and location of the datasets used in data quality analyses. The other tables are all created in the Data Quality schema (DQ) and will catalog data quality project information and errors. DQ.PROJECT includes basic information about each project and provides a centralized location for anyone on our team to lookup all past and current data quality projects. It also stores the name of each project's individual issues table, which is used to load errors into the global issue table. DQ.TEST_INFO contains information about all data quality checks, including what project each test is associated with and what datasets are used. DQ.INDIVIDUAL_ISSUES defines the common set of attributes that will be recorded for every project. That table will be auto-created for each project with the columns defined in Figure 2. DQ.GLOBAL_ISSUES is the table where all errors from all data quality projects will be stored and is the dataset visualized in the AGOL Dashboard.

The first step in the workflow (Fig. 1) is for a user to create a copy of the Excel template. They then populate the three tabs of the Excel workbook with information about their project (Fig. 3), the datasets used in the project (Fig 4.), and the errors the project identifies (Fig 5.). After adding information to the Excel workbook, an analyst will run a Python script stored in the same directory as their workbook on a network drive. The Python script reads the Excel workbook into a Pandas dataframe, validates the data, checks if the information already exists in the database tables, inserts it if it does not (Fig. 6), and sends the analyst an email with record IDs for their project information in the database tables. It also calls a stored procedure to auto-create the individual project issues table using the project name supplied by the analyst (Fig. 7). I use Pandas for all data manipulations and analysis in the Python script. I use pyodbc and sqlalchemy for all interaction with the SQL Server database from Python.

Once the project information is catalogued in the database, the analyst can add the record IDs they received in the email to their existing Python script (Fig. 8), so they are included when errors are inserted from Pandas dataframes into the project issues table in the database. These Python scripts are usually scheduled to run weekly using Task Scheduler. The process to identify current errors each week, load them into a staging table in

the database, and merge the current errors into the individual project issue table is fully automated (Figs. 9, 10, 11). A second stored procedure (Fig. 12) is scheduled in the SQL Server database to run each Sunday and merge the errors from each project's individual issue table into the global issue table.

I manually published the initial hosted feature service of the global issue table to AGOL. I wrote a Python 3.6 script using arcpy and the ArcGIS API for Python (Fig. 13) to copy the global issue table from SQL Server to a file geodatabase and overwrite the feature service on AGOL using the feature class in the file geodatabase. I chose this method rather than GEOJSON because the size limit of 100MB for GEOJSON would quickly have been surpassed as more errors are loaded into the table. I scheduled this script using Task Scheduler so that it runs each Sunday after the completion of the stored procedure in Figure 12.

Dashboard Description & Summary

I created the ArcGIS Online Dashboard using Dashboard Beta. After feedback from user-testing in class, and several rounds of peer review at work, I arrived at the final design which attempts to answer the questions: how long will the cleanup take, where do we to focus our efforts, and how much cleanup have we been performing over time?

The data quality project I chose to visualize is an analysis I created to determine the distance between all our electric poles in our GIS and the location of the corresponding poles from a LiDAR dataset of our poles' locations. Our GIS data standards claim that all poles in GIS will be mapped within 20 feet of their actual location, so a pole is considered "correct" or "rectified" if it is 20 feet or less from its corresponding LiDAR pole. All poles greater than 20 feet from their corresponding LiDAR pole need to be rectified. I used Pandas and Geopandas to perform the analysis and calculate distances between the two pole datasets. Due to data sharing restrictions at my company, I created dummy pole data for the dashboard and aggregated it to the PLSS Wisconsin Townships feature class from the Wisconsin Department of Natural Resources Open Data site to visualize in the dashboard.

Figure 14 shows the initial view of the dashboard. All widgets within the red box will update as a new month is selected in the top right portion of the dashboard (teal box). The two widget groups on the right provide historical data summarized by month and do not update when the month filter is changed.

The **Hours to Rectify All Poles** widget in the top left (Fig. 14) attempts to answer the question "how long will the cleanup take?". I analyzed how many poles analysts rectified over the last several months and determined that an analyst can rectify, on average, 8 poles per hour. This widget visualizes that when this cleanup started at the end of November, it would take an estimated 13,000 analyst hours to rectify all poles. As of the end of March, that number is down to 11,200 hours.

The **Total Cleanup Progress** pie chart in the bottom left (Fig. 14) provides a snapshot of how many poles have been rectified and how many are remaining to fix. The **Distribution** and **Transmission** tabs of this widget stack provide the same information, except filtered by each pole type.

Two web maps are embedded in the center of the dashboard (Fig. 14). Each map supports pan, zoom, bookmark, and search. Clicking a township will display a popup with information about the poles in that polygon (Fig. 21). **Estimated Hours to Complete Pole Rectification by Township** (Fig. 19) visualizes an estimate of how many analyst hours it will take to rectify all poles within a given township range grid. If all poles in a grid have been rectified, it will display as lime green. Otherwise, the lighter the color, the longer it will take to rectify all poles in that polygon. This map gives managers a quick visual summary of the cleanup and can allow them to see any clustering of particularly bad data in the service area. Clicking through each month can also show them where progress has been made on pole rectification. The **Time to Finish Township Summary** pie chart (Fig. 18) shows the breakdown of the townships from the map in each bucket for the month in the top right of the dashboard.

The **Pole Cleanup by Township** map (Fig. 20) is a bivariate choropleth map that attempts to provide greater insight into the status of the poles in each township. The two variables visualized are total pole count and percent of poles remaining to fix. I chose to do a bivariate map, rather than a simple choropleth visualizing percent of poles remaining to fix, to normalize the percentage by the total pole count in the township. This makes it is easier to differentiate between area with a high percent of poles to fix but only a few total poles and areas with a similar percent to fix but with more poles. Areas in white and yellow will require less work to fix, while areas in red and brown will require more work.

The list widgets below the maps (Fig. 14) provide information on the townships with the highest percentage of poles remaining to be fixed and the townships with the highest percentage of poles that have already been fixed. Clicking an item in these widgets will pan and zoom to that township on both maps. These widgets answer the question “which areas have the worst data?” and “which areas have the best data?”

The **Mean Distance by Pole Type between Our Poles and Lidar Poles** widget in the top right of the dashboard (Fig. 14) shows the average distance by pole type between our poles and their LiDAR locations at the end of each month. The shaded green area represents 20 feet and closer (my company’s GIS standard), so the goal is to get each line in the chart into the shaded green area. The **Mean Distance Improvement by Pole Type between Our Poles and Lidar Poles** widget (Fig. 15) shows the inverse of the previous widget and shows how much the mean distance improves each month.

The **Total Poles Remaining by Month** widget (Fig. 14) shows how many poles of each type need to be rectified at the end of each month, as well as the total poles remaining to rectify. The **Poles Corrected by Month** widget (Fig. 16) shows how many poles of each type analysts rectified each month. The **Poles Corrected Over Time** widget (Fig. 17) shows the total number of poles that have been rectified at the end of each month.

Feedback from my coworkers/manager was to wait to create the overview dashboard visualizing the global issues table until we load several projects into the framework, so the dashboard for this project is more specific to the project I loaded into the framework first. I discovered that other Dashboards, and many AGOL Items, can be nested inside a “parent” dashboard using the embedded content widget. In the future, this will

allow me to create a "landing page" Dashboard that will provide an overview of all projects loaded into the framework, and add any other project-specific Dashboards as well. Users will then be able to toggle between all the nested dashboards from the same URL by clicking on the tabs, like the stacked widgets in the dashboard I created.

Challenges and Personal Growth

The biggest challenge, along with the area I learned the most, was executing insert statements, stored procedure, and dynamic SQL from a Python environment. I quickly learned it was more complicated than copying the SQL I wrote in MS SQL Server Management Studio, pasting it into the Python scripts as strings, and executing those strings. In previous work, I found pandas' .to_sql() method to be sufficient for getting data from a pandas dataframe into a database environment. For this project, that method had three major shortcomings: column data types changed from dataframe to database table, dates lost timezone information, and it offered no way to insert geometries. I learned to connect to the SQL Server database with pyodbc, create sql strings with '?'s as placeholders for each column, and pass a pandas dataframe as a list of tuples to be inserted using the sql string with '?'s. By creating the database tables ahead of time and defining the specific data types I wanted, this method gave me full control over how the data was loaded from Python to the database. I also learned how to create geometries using this method by passing the WKT in the tuple (Fig. 9). Solving this challenge has been beneficial to my other work. I now have much greater control over how data is loaded into a database from a Python script and can perform more accurate ETL processes with much less Python code than I could four months ago.

I struggled with datetime data a lot too. Near the end of the project, I noticed all the dates were appearing wrong in my AGOL Dashboard. I then learned that all dates published to AGOL are assumed to be in the UTC time zone, and all my datetimes were in Pacific Daylight Time. This forced me to review all the code I had written and ensure all datetime stamps were transformed into the appropriate time zone. I now have a much better understanding of the Python datetime package, I learned how to use the pytz module to change time zones, and developed several code blocks I will reuse in other projects for changing time zones and getting datetime data into the proper formats when loading it from a pandas dataframe into a database. With SQL Server databases in particular, I learned that inserting datetime data as varchar datatype is the best way to maintain time zone offsets when inserting from Python.

I also greatly improved my ability to read API documentation and implement it into my specific situations. I had not used pyodbc to connect to databases before, and there was a steep learning curve. I now know how to use it to execute DDL and DML SQL, execute stored procedures while supplying parameters, and use dynamic SQL strings to convert data to specific formats, particularly geometry and datetime, in the same action as inserting it into the database. I also figured out how to optimize my code when inserting hundreds of thousands of rows of data by chunking my dataframes and using the fastexecutemany property of the pyodbc cursor object.

Summary/Conclusion

This project met the goal of creating a database framework to catalog data quality projects and their errors. It streamlined and centralized the project documentation process by loading data from an Excel workbook into a database, and standardizes the data tracked by analysts at my company in all data quality projects going forward. The framework is semi-automated and easy for my coworkers to use, and any project loaded into the database framework will automatically have its errors pulled into the global issues table. While the AGOL Dashboard is not in its final format of visualizing every data quality error, the dashboard I created tells a much more compelling story of our LiDAR pole cleanup project than the CSV's we used to track the errors previously. Furthermore, the layout and framework of this dashboard is scalable, and I have identified designs and strategies to visualize and quantify all data quality errors and cleanup across our service territory as more projects are loaded into this framework.

Appendix

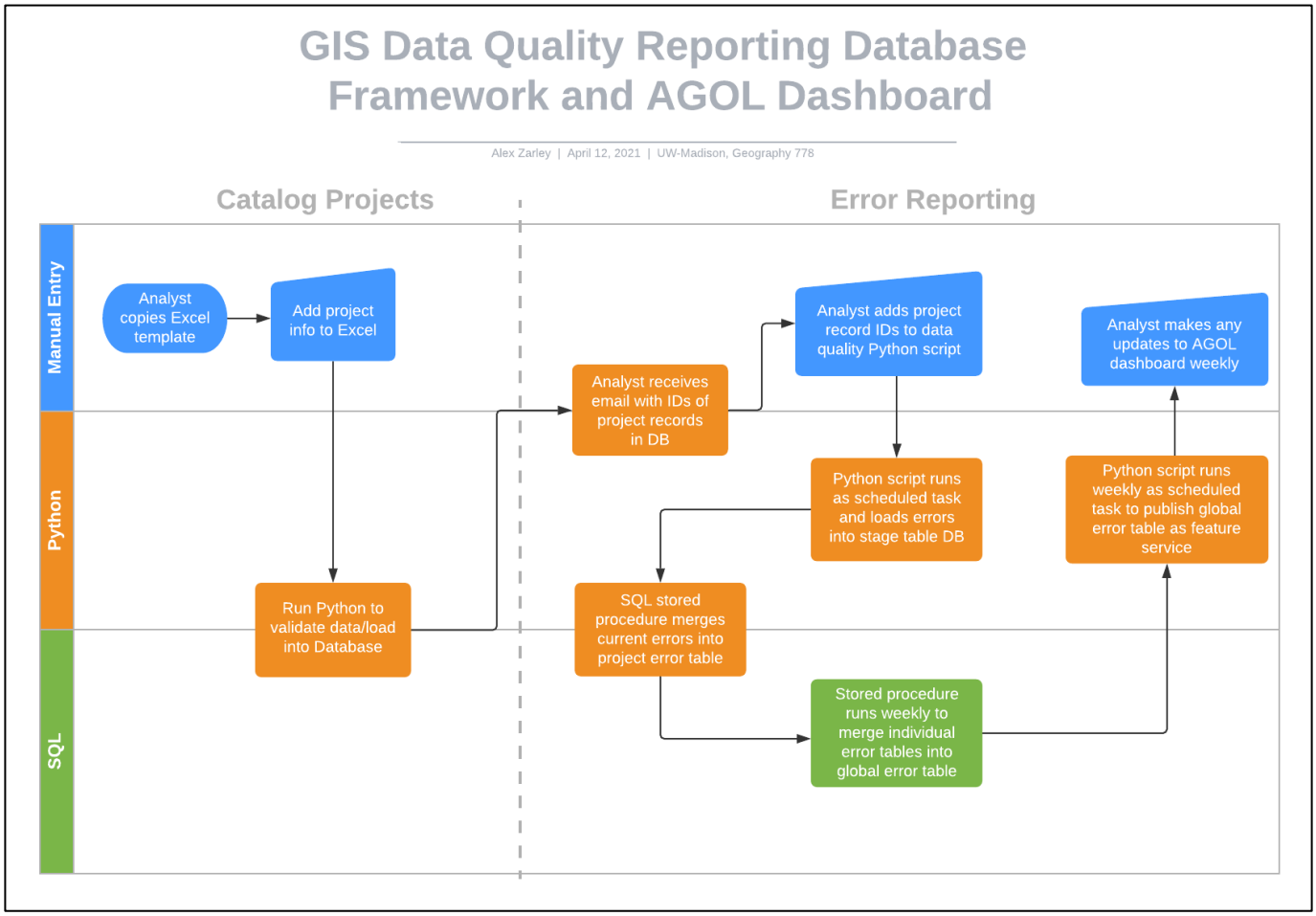


Figure 1: Data quality framework workflow diagram. Oval is starting point, rectangles are scripts (either scheduled or manually run), and rectangles with a diagonal top require manual work.

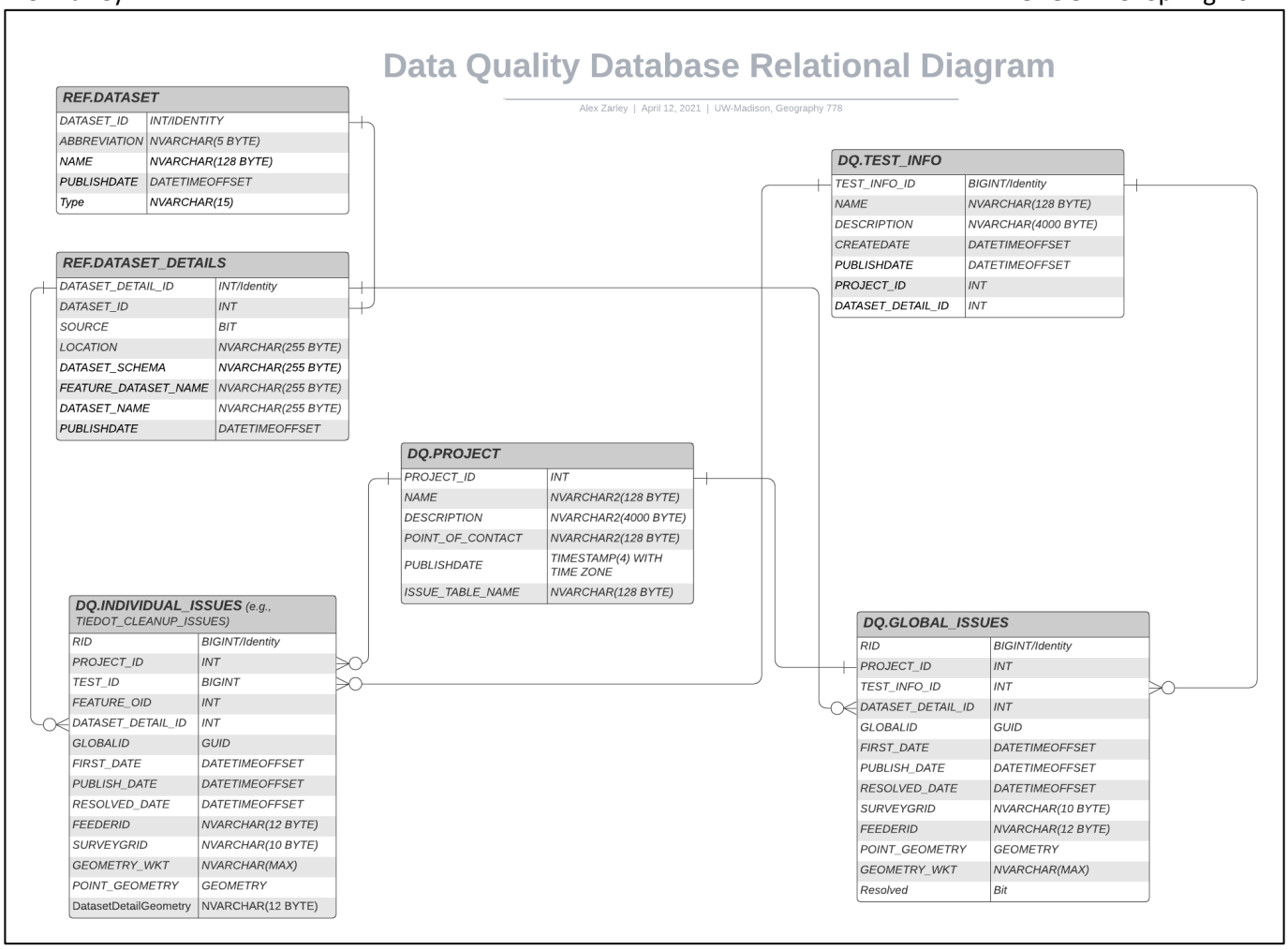


Figure 2: Database Relational Diagram

Project_Name	Project_Description	Your_E_Number
Lidar Pole Correction	This project measures the distance between each support and transmission structure and its corresponding LiDAR pole in the datasets we received from QSI. The goal is to move each PGE pole within 20 feet of its corresponding LiDAR pole.	E12345

Figure 3: Project Information tab of Excel Workbook template. Each cell has help text on hover and basic data validation.

Dataset_ID	Location	Dataset_Schema	Feature_Dataset_Name	Dataset_Name	Source_of_Truth
68	GISS	REPL		SUPPORTSTRUCTURE	No
80	GISS	REPL		TRANSMISSIONSTRUCTURE	No
86	GISP001P	EXTRNL		QSI_LidarStructures	Yes

Figure 4: Dataset tab of Excel Workbook template. Information about version and location of each dataset used in analysis. Each cell has help text on hover and basic data validation.

Location	Dataset_Schema	Feature_Dataset_Name	Dataset_Name	Test_Name	Test_Description
GISS	REPL		SUPPORTSTRUCTURE	Support Structure within 20 feet of LiDAR pole	Checks if distance between PGE pole and QSI LiDAR pole is 20 feet or less
GISS	REPL		TRANSMISSIONSTRUCTU	Transmission Structure within 20 feet of LiDAR pole	Checks if distance between PGE pole and QSI LiDAR pole is 20 feet or less

Figure 5: Dataset tab of Excel Workbook template. Information about DQ error checks in analysis. Each cell has help text on hover and basic data validation.

```
# create pyodbc connection to GISS and a cursor object for inserting df into GISS
cnxn = create_sqlserver_con(GISS['E06782'])
cursor = cnxn.cursor()

# create components of queries
sql_cols = ", ".join([col for col in [project_cols_dict[col] for col in df.columns]])
val_str = ''
for i in range(0, len(df.columns)):
    val_str += '?, '
# construct sql statements; final ? is for inserting datetime as string to maintain timezone info
insert_sql = "INSERT INTO {s}.{t} ({c}) values ({v}, CAST(? AS VARCHAR(50)))".format(s=schema, t=table_name,
                                                                                     c=sql_cols, v=val_str[:-5])

cursor.executemany(insert_sql, list(df.itertuples(index=False, name=None)))
```

Figure 6: Python to insert project information from Excel template into database tables. Very similar code used to insert dataset and test information from Excel template.

```
ALTER PROCEDURE DQ.createProjectIssueTable @issue_table_name nvarchar(max)
AS

DECLARE @CreateIssueTableSql NVARCHAR(MAX);

SET @CreateIssueTableSql = ''
SET @CreateIssueTableSql = @CreateIssueTableSql + 'USE [GISS] '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'SET ANSI_NULLS ON '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'SET QUOTED_IDENTIFIER ON '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'CREATE TABLE DQ.' +  @issue_table_name
SET @CreateIssueTableSql = @CreateIssueTableSql + ' ([RID] BIGINT IDENTITY(1,1),'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [PROJECT_ID] INT NOT NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [TEST_ID] INT NOT NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [FEATURE_OID] INT NOT NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [DATASET_DETAIL_ID] INT NOT NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [GLOBALID] NVARCHAR(50) NOT NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [FIRST_DATE] DATETIMEOFFSET NOT NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [PUBLISH_DATE] DATETIMEOFFSET NOT NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [RESOLVED_DATE] DATETIMEOFFSET NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [SURVEYGRID] [nvarchar](10) NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [FEEDERID] [nvarchar](12) NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [DATASET_DETAIL_GEOMETRY] [nvarchar](7) NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [POINT_GEOMETRY] [geometry] NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' [GEOMETRY_WKT] [nvarchar](max) NULL,'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' CONSTRAINT [PK_' + @issue_table_name + '] PRIMARY KEY CLUSTERED ([RID] ASC))'
SET @CreateIssueTableSql = @CreateIssueTableSql + 'ALTER TABLE [DQ].[' + @issue_table_name + ']'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' WITH CHECK ADD CONSTRAINT [FK_' + @issue_table_name + '_PROJECT] FOREIGN KEY([PROJECT_ID])'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' REFERENCES [DQ].[PROJECT] ([PROJECT_ID]) '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'ALTER TABLE [DQ].[' + @issue_table_name + ']'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' CHECK CONSTRAINT [FK_' + @issue_table_name + '_PROJECT] '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'ALTER TABLE [DQ].[' + @issue_table_name + ']'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' WITH CHECK ADD CONSTRAINT [FK_' + @issue_table_name + '_DATASET_DETAIL] FOREIGN KEY([DATASET_DETAIL_ID])'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' REFERENCES [REF].[DATASET_DETAIL] ([DATASET_DETAIL_ID]) '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'ALTER TABLE [DQ].[' + @issue_table_name + ']'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' CHECK CONSTRAINT [FK_' + @issue_table_name + '_DATASET_DETAIL] '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'ALTER TABLE [DQ].[' + @issue_table_name + ']'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' WITH CHECK ADD CONSTRAINT [FK_' + @issue_table_name + '_TEST_INFO] FOREIGN KEY([TEST_ID])'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' REFERENCES [DQ].[TEST_INFO] ([TEST_INFO_ID]) '
SET @CreateIssueTableSql = @CreateIssueTableSql + 'ALTER TABLE [DQ].[' + @issue_table_name + ']'
SET @CreateIssueTableSql = @CreateIssueTableSql + ' CHECK CONSTRAINT [FK_' + @issue_table_name + '_TEST_INFO]'


--print @CreateIssueTableSql
EXECUTE sp_executesql @CreateIssueTableSql
```

Figure 7: Stored procedure to automatically create project issue table when data loaded from Excel workbook. Stored procedure is called from same Python script as Figure 6, and supplies the project name the user entered in the workbook to construct and execute dynamic SQL string shown above.

```
# dq framework IDs
project_id = 1
dist_test_id = 2
trans_test_id = 3
dist_ds_id = 1
trans_ds_id = 2
ds_geom = 'Point'

# add dq framework columns
dist_distance['TEST_ID'] = dist_test_id
dist_distance['DATASET_DETAIL_ID'] = dist_ds_id
trans_distance['TEST_ID'] = trans_test_id
trans_distance['DATASET_DETAIL_ID'] = trans_ds_id

logger.info('concatenating transmission and distribution distance dataframes')
all_distance = pd.concat([dist_distance, trans_distance], axis=0)
all_distance['PROJECT_ID'] = project_id
all_distance['PUBLISH_DATE'] = timestamp
all_distance['PUBLISH_DATE'] = all_distance['PUBLISH_DATE'].astype(str)
all_distance['DATASET_DETAIL_GEOMETRY'] = ds_geom
```

Figure 8: Python showing how to add the projects/dataset/test record IDs received in email to existing Python script so errors loaded into database individual issues table are associated with proper project, dataset, and test records.

```
schema = 'DQ'
table_name = 'STAGE_ISSUES_LIDAR_POLE_CORRECTION'

delete_sql = 'DELETE FROM {s}.{t}'.format(s=schema, t=table_name)
logger.info('Deleting all records from {0}.{1}'.format(schema, table_name))
cursor.execute(delete_sql)

# create components of queries
sql_cols = ", ".join([col for col in all_distance.columns])
val_str = ''
for i in range(0, len(all_distance.columns)):
    val_str += '?, '

# construct sql statements
insert_sql = "INSERT INTO {s}.{t} ({c}) values ({v}, CAST(? AS VARCHAR(50)), geometry::STGeomFromText(?, 2913))"\
    .format(s=schema, t=table_name, c=sql_cols, v=val_str[:-8])  # removing last ? because I want it added with STGeomFromText

logger.info('SQL to insert records into {0}.{1}: {2}'.format(schema, table_name, insert_sql))

start = datetime.datetime.now()
# number of records to be inserted at a time to save memory
chunk_size = 5000
# turn on fast_executemany
cursor.fast_executemany = True
for row_count in range(0, all_distance.shape[0], chunk_size):

    logger.info('Inserting rows {0} to {1}'.format(row_count, row_count + chunk_size))
    # subset rows, choose all columns, convert each row's values to a list
    chunk = all_distance.iloc[row_count:row_count + chunk_size, :].values.tolist()

    # convert row value lists to tuples and place them in a tuple
    tuple_of_tuples = tuple(tuple(x) for x in chunk)
    try:
        cursor.executemany(insert_sql, tuple_of_tuples)
    except:
        print(traceback.format_exc())
        logger.info(traceback.format_exc())
        sys.exit(0)
```

Figure 9: Python deleting errors from stage table and inserting current errors in chunks of 5000 to prevent memory errors. The publish date column is inserted as VARCHAR to maintain time zone information and the geometry is created from WKT in the *insert_sql* variable.

```
cursor.fast_executemany = False
confirm_insert_sql = 'SELECT CONVERT(nvarchar(max), CONVERT(datetime2(0), MAX(PUBLISH_DATE))), COUNT(*) FROM DQ.STAGE_ISSUES_LIDAR_POLE_CORRECTION'
logger.info('SQL to confirm records inserted: {0}'.format(confirm_insert_sql))
# retrieve most recent time stamp from stage table to confirm data successfully loaded
cursor.execute(confirm_insert_sql)

res = cursor.fetchone()
rowcount = res[1]
pub_date = res[0]
timestamp_chk = timestamp.strftime('%Y-%m-%d %H:%M:%S')


# check that rowcount and date in staging table match those from current run before executing merge
if rowcount == all_distance.shape[0] and pub_date == timestamp_chk:
    print('merging')
    # logger.info('rowcount in {0}.{1} matches insert dataframe and publish date in {0}.{1} matches current Python run, merging with DQ.merge_issues
    cursor.execute('EXEC DQ.merge_issues_lidar')
    logger.info('merge complete')
    cursor.close()
    cnxn.close()
else:
    print('staging table doesnt match current run, not merging')
    logger.info('staging table doesnt match current run, not merging')
    logger.info('{0}.{1} rowcount={2}, current Python run rowcount={3}'.format(schema, table_name, rowcount, all_distance.shape[0]))
    logger.info('{0}.{1} publish_date={2}, current Python run timestamp={3}'.format(schema, table_name, pub_date, timestamp_chk))
    sys.exit(0)
```

Figure 10: Python confirming all errors from dataframe *all_distance* were inserted into database and publish date in stage table matches date from Python script before executing this project's stored procedure to merge errors that were just inserted into stage table into the existing Individual project Issue table.

```
ALTER PROCEDURE DQ.merge_issues_lidar AS

DECLARE @cur_date datetimeoffset;
SELECT @cur_date = MAX(PUBLISH_DATE) FROM DQ.STAGE_ISSUES_LIDAR_POLE_CORRECTION;


MERGE DQ.ISSUES_LIDAR_POLE_CORRECTION trgt
USING DQ.STAGE_ISSUES_LIDAR_POLE_CORRECTION src
    ON      trgt.DATASET_DETAIL_ID = src.DATASET_DETAIL_ID
        AND trgt.FEATURE_OID = src.FEATURE_OID
    WHEN MATCHED                                                    -- this captures all poles fixed since last run; pole now withi
                AND src.CURRENT_DISTANCE <= 20.0
                AND trgt.RESOLVED_DATE IS NULL -- pole has been fixed
        THEN
        UPDATE SET
                trgt.PUBLISH_DATE = @cur_date
               ,trgt.RESOLVED_DATE = @cur_date
               ,trgt.CURRENT_DISTANCE = src.CURRENT_DISTANCE
    WHEN NOT MATCHED BY TARGET  --this captures any pole that has never previously been more than 20 feet away; rows in source table th
        AND src.CURRENT_DISTANCE > 20.0
    THEN
    INSERT (  PROJECT_ID, TEST_ID, FEATURE_OID, DATASET_DETAIL_ID, GLOBALID, FIRST_DATE, PUBLISH_DATE, SURVEYGRID, FEEDERID
            , DATASET_DETAIL_GEOMETRY, POINT_GEOMETRY, GEOMETRY_WKT, ORIGINAL_DISTANCE, CURRENT_DISTANCE, QSI_POLE_TYPE, QSI_OID
            , QSI_STR_NUM, QSI_PGE_GUID, PGE_FACILITYID, PGE_LABELTEXT)
    VALUES (  src.PROJECT_ID, src.TEST_ID, src.FEATURE_OID, src.DATASET_DETAIL_ID, src.GLOBALID, src.PUBLISH_DATE, src.PUBLISH_DATE
            , src.SURVEYGRID, src.FEEDERID, src.DATASET_DETAIL_GEOMETRY, src.POINT_GEOMETRY, src.GEOMETRY_WKT, src.CURRENT_DISTANCE
            , src.CURRENT_DISTANCE, src.QSI_POLE_TYPE, src.QSI_OID, src.QSI_STR_NUM, src.QSI_PGE_GUID, src.FACILITYID, src.LABELTEXT)
    WHEN NOT MATCHED BY SOURCE -- delete rows if pole no longer exists
    THEN DELETE;
;

MERGE DQ.ISSUES_LIDAR_POLE_CORRECTION trgt
USING DQ.STAGE_ISSUES_LIDAR_POLE_CORRECTION src
    ON      trgt.DATASET_DETAIL_ID = src.DATASET_DETAIL_ID
        AND trgt.FEATURE_OID = src.FEATURE_OID

        WHEN MATCHED -- this caputres all unresolved poles; pole not yet within 20feet; update publish date and current distance (cur_d
                AND src.CURRENT_DISTANCE > 20.0
                AND trgt.RESOLVED_DATE IS NULL -- pole has not been fixed yet (res_date null), current distance has changed but error
    THEN
        UPDATE SET
                trgt.PUBLISH_DATE = @cur_date
               ,trgt.CURRENT_DISTANCE = src.CURRENT_DISTANCE;
```

Figure 11: SQL stored procedure called from Python script to merge current errors into project issue table. It will update the RESOLVED_DATE column if an error has been fixed or update the PUBLISH_DATE and CURRENT_DISTANCE columns if it still exists. New errors will be inserted and features with errors that were deleted from GIS will have their record deleted from project issue table.

```
CREATE PROCEDURE DQ.merge_project_into_global AS


DECLARE @cur_date datetimeoffset;
-- variable to hold name of table to be merged into global errors in while loop
DECLARE @tbl_name nvarchar(max);
-- dynamic sql to retrieve most recent publish date in each project error table
DECLARE @cur_date_sql nvarchar(max);
-- dynamic sql to execute merge of each project error table into global error table
DECLARE @merge_sql nvarchar(max);

-- initially set variable to first ISSUE_TABLE_NAME in DQ.PROJECT, alphabetically
SELECT @tbl_name = min(ISSUE_TABLE_NAME) from DQ.PROJECT;
WHILE @tbl_name is not null -- execute line of code above after each merge to move to next ISSUE_TABLE_NAME

BEGIN

    SET @cur_date_sql = ''
    SET @cur_date_sql = @cur_date_sql + 'MAX(PUBLISH_DATE) FROM DQ.@cur_tbl';
    EXEC sp_executesql @cur_date_sql, N'@cur_tbl nvarchar(max) out', @cur_date out;

    SET @merge_sql = ''
    SET @merge_sql = @merge_sql + 'MERGE DQ.GLOBAL_ISSUES trgt'
    SET @merge_sql = @merge_sql + 'USING DQ.' + @tbl_name + ' src'
```

Figure 12: SQL stored scheduled in SQL Server database to run each Sunday. This crawls through the issue table for each project listed in DQ.PROJECT and merges the project's errors into global errors table. Merge logic is similar to Figure 11. This figure demonstrates the "while loop" to crawl through each distinct value in DQ.PROJECT.ISSUE_TABLE_NAME column.

```
14  print('Creating AGOL GIS object . . .')
15  agol = GIS(agol_url, agol_user_name, keyring.get_password('AGOL', agol_user_name))
16  print('Connected to AGOL')
17
18  print('Retrieving existing feature layer collection by item id: {0}'.format(item_id))
19  flyr_coll_item = agol.content.get()
20  print('Creating feature layer collection. . .')
21  flc2 = FeatureLayerCollection.fromitem(flyr_coll_item)
22  print('Overwriting feature service with fgdb feature class')
23  success = flc2.manager.overwrite(fgdb)
24  print(success)
```

Figure 13: Python script to overwrite existing Global Issue hosted feature service with new data that has been exported to file geodatabase.

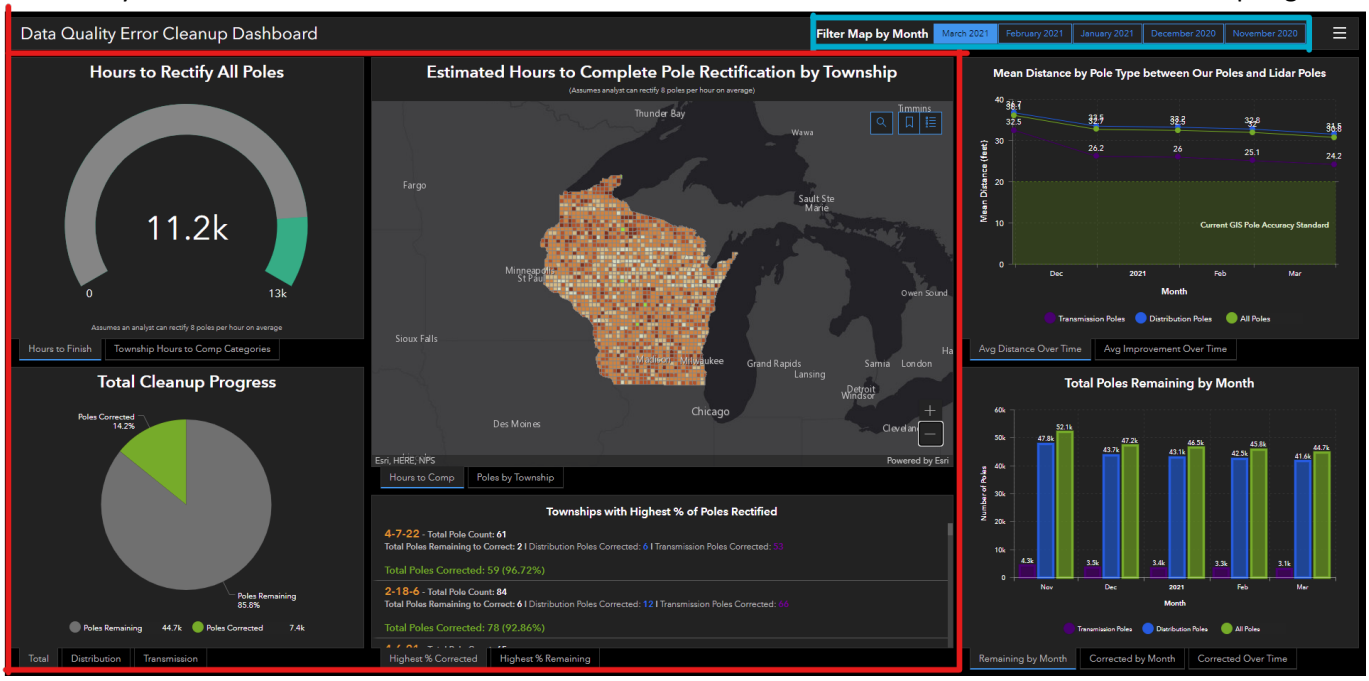


Figure 14: Default view of AGOL Dashboard. Use Months in teal box to filter data in widgets inside red box.

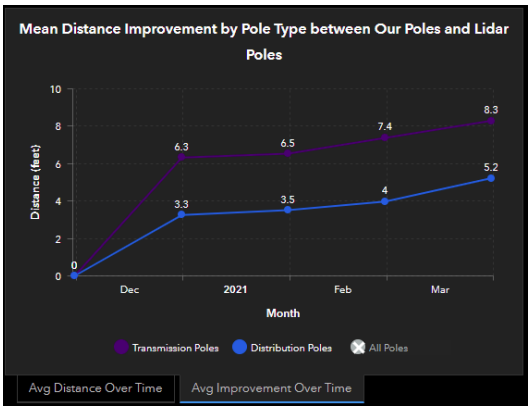
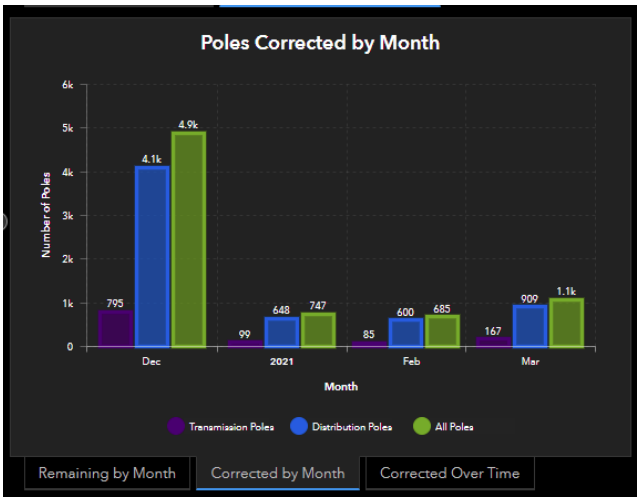


Figure 15: Widget showing how much mean pole accuracy by pole type improved by month.



Widget 16: Widget showing how many poles were rectified during each month.

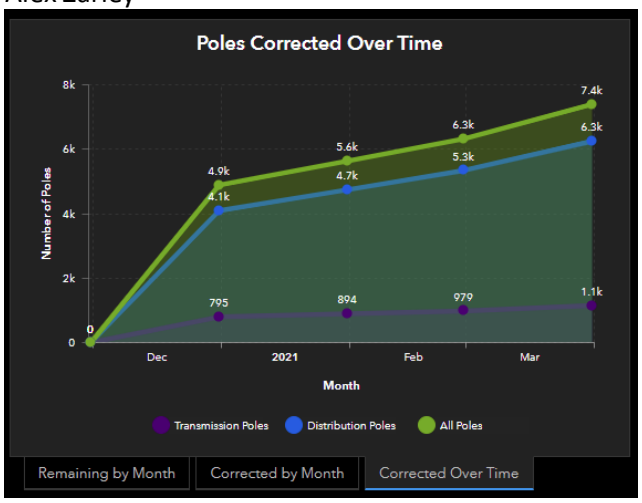


Figure 17: Widget showing running total of poles correct at end of each month.

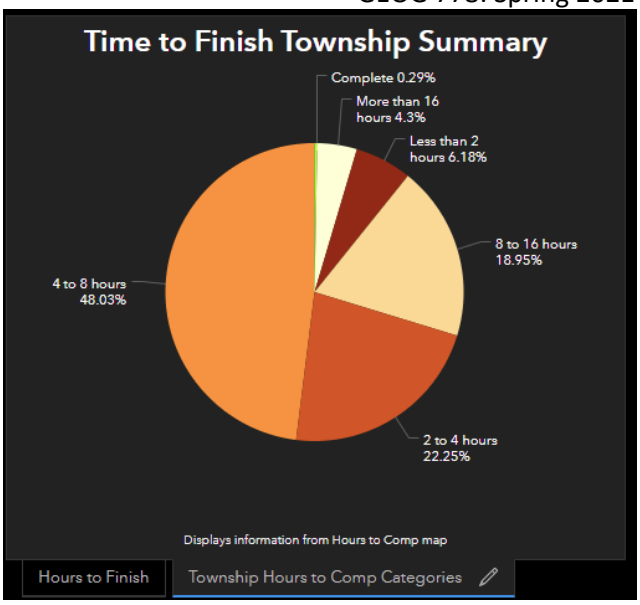


Figure 18: Pie chart showing breakdown of townships in Hours to Comp map tab. Filtered by month toggles.

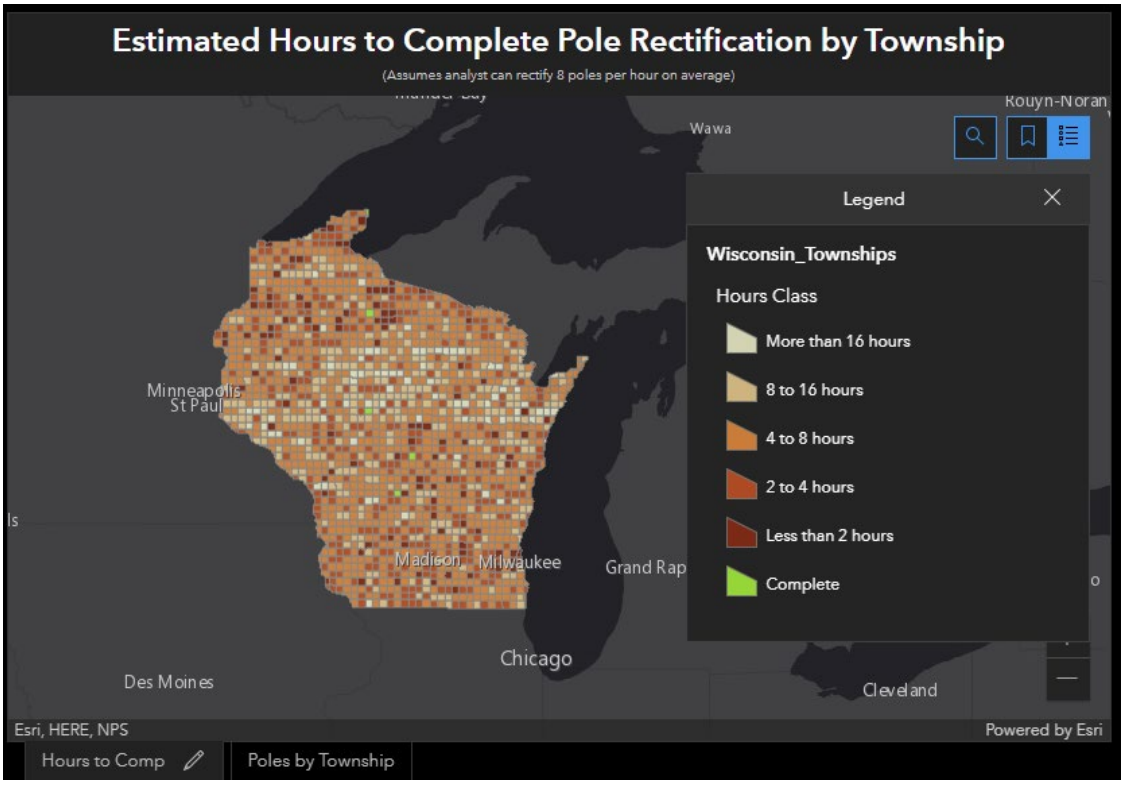


Figure 19: Choropleth web map visualizing estimated number of hours to rectify all poles in township. Filtered by month toggles.

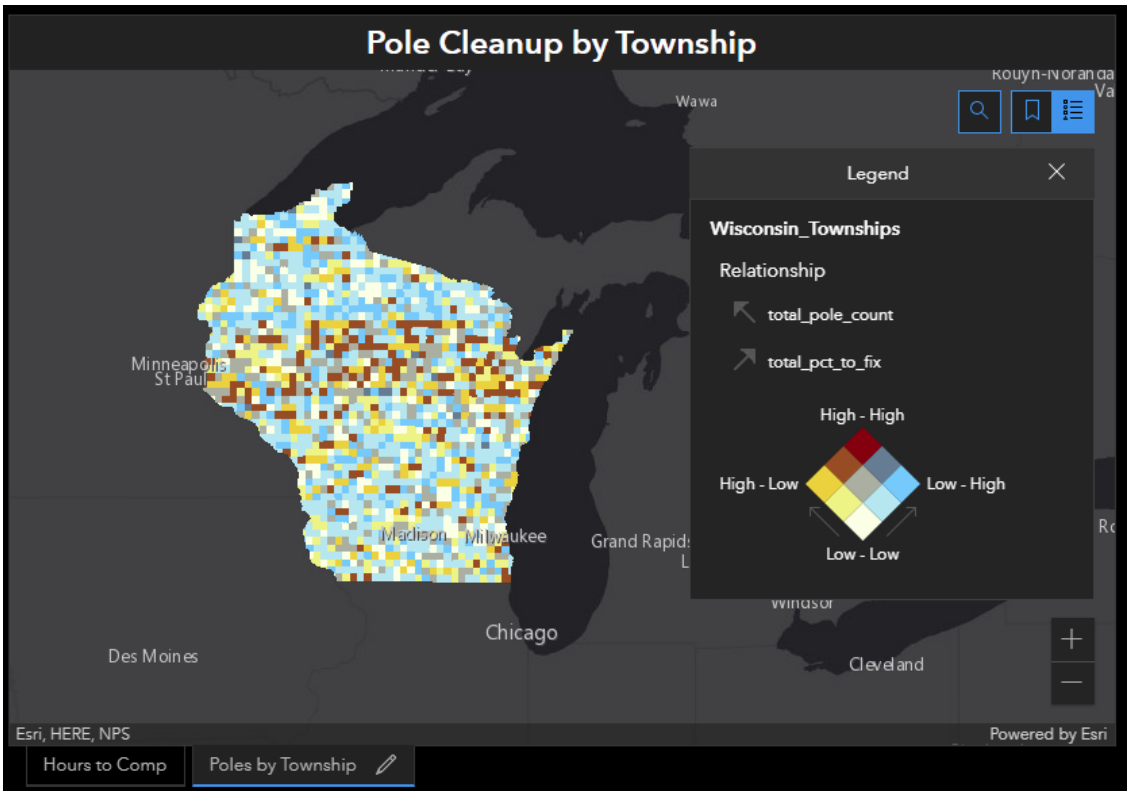


Figure 20: Bivariate choropleth web map visualizing total pole count vs percent of all poles remaining to be fixed in township. Filtered by month toggles.

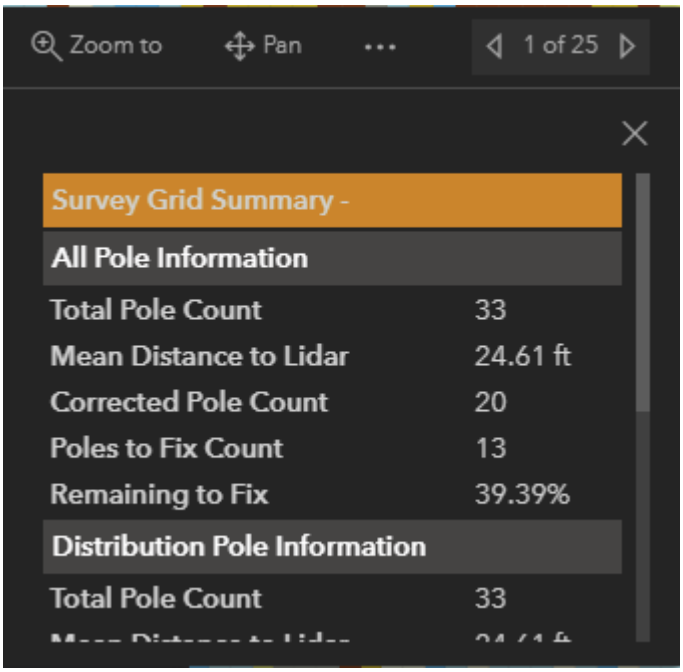


Figure 21: Popup for Web Maps. Displays information for each pole type.