Workflow Dependency Diagram

Ben Gelman
Data Engineering Fellow

Project Description and Goals

My project goal to expand the Workflow Dependency Diagram: which illustrates the dependency relationships between different stages of our ETL workflows (tasks, tables). This enables us to visualize workflows, which is helpful for both engineers and analysts, who may not have as much visibility into the workflows that the engineers maintain.

These diagrams are important in order to understand which aspects of the Analytics Team's work rely upon each other, and are especially useful when one table or piece of code breaks, as we can then easily see which other files rely upon it, and will therefore not work as well. It is also useful for analysts as a visual representation of the workflows, rather than just code.

I was attracted to the project as an opportunity to explore a new type of data visualization and application (Neo4J), as well as gain more experience in python, specifically developing and testing functions and scripts, and in writing efficient code.

Script Structure

Step 1

Parse YAML (workflow configuration) of workflow into python-readable data.

Step 2

Identify and parse tasks within workflow.

Step 3

Write nodes and relationships to Civis tables, visualize in Neo4J.

Functions part 1

```
# Maps script template to function that parses it
template_dict = {
    'Util: Email': parse_email_task,
    'Util: Data Unit Test': parse_dut_task,
    'Util: Microsoft Word': parse_word_task,
    'Export: Arcgis Feature Layer': parse_agol_feature_layer_task,
    'Export: Arcgis Update Feature Layer Definition': parse_agol_feature_layer_definition_task,
    'Export: Ckan Update Resource': parse_ckan_update_resource_task,
    'Export: Custom Civis': parse_export_custom_civis_task,
    'Export: Custom Database': parse_custom_database_export_task,
    'Export: Table to CSV': parse_table_to_csv_task,
    'Import: Arcgis Feature Layer': parse_agol_feature_layer_import_task,
    'Import: Custom Database Import': parse_custom_database_task,
    'Import: Custom URL': parse_url_task,
    'Import: Knack': parse_knack_task,
    'Import: Knack V2': parse_knack_v2_task,
    'Transform: Civis Query': parse_civis_query_task,
    'Transform: Custom Civis': parse_transform_task,
    'Transform: Geocode': parse_geocode_task,
    'Word to PDF Converter': parse_word_to_pdf_task,
    'Multi from S3 to Civis': parse_s3_to_civis_task,
    'Import Multi from Civis Job': parse_multi_from_civis_task,
    'Transform: Arcgis Buffer': parse_arcgis_buffer_transform_task,
    'Import: Smartsheet': parse_smartsheet_import_task,
    'Import: Salesforce': parse_salesforce_import_task,
    'Export: File to Google Drive': parse_file_to_gdrive_export_task,
    'Export: Civis to AWS S3': parse_civis_to_aws_s3_export_task,
    'Import: BigQuery': parse_bigquery_import_task,
    'Export: Table to XLSX': parse_table_to_xlsx_export_task,

    # Built-In Components
    'Import from URL': parse_url_task
}
```

My main task was to edit the python script parse_workflow_dependencies.py, within the scripts/audit folder. The code works by parsing through each YAML workflow, dividing it into tasks, and processing its dependencies.

Functions Part 2

The first step was to get an understanding of the problem: I searched each of our existing workflows in civis to see how many used each template, and I then knew which templates to prioritize.

Each template requires its own function, so most of my work consisted of writing these functions so that tasks that utilized these templates could be included in the diagram.

This specific function works to parse google sheets exported via URL.

```
def parse_url_task(task_name, task, df_dict):
    arguments = task['input']['arguments']
    index_dict = get_latest_index(['task', 'url'], df_dict)

    # Nodes
    df_dict['url_df'] = df_dict['url_df'].append(
        {'index': index_dict['url_id'], 'url': arguments['URL']}, ignore_index=True)
    df_dict['task_df'] = df_dict['task_df'].append(
        {'index': index_dict['task_id'], 'task_name': task_name, 'name': task['input']['name'],
**arguments}, ignore_index=True)

    # Relationships
    df_dict['imported_to_df'] = df_dict['imported_to_df'].append(
        {'source_id': index_dict['url_id'], 'task_id': index_dict['task_id']}, ignore_index=True)

    # This parser handles both the custom and civis url import and the table parameter name is
different
    table_parameter_name = 'DEST_TABLE' if 'DEST_TABLE' in arguments else 'TABLE_NAME'
    df_dict['exported_to_df'] = df_dict['exported_to_df'].append(
        {'destination_id': df_dict['table_df'][df_dict['table_df']['full_name'] ==
arguments[table_parameter_name]]['index'].values[0], 'task_id': index_dict['task_id']},
ignore_index=True)
```

New Code

I added 13 new Civis tasks, to be able to parse all of the component scripts commonly used in workflows.

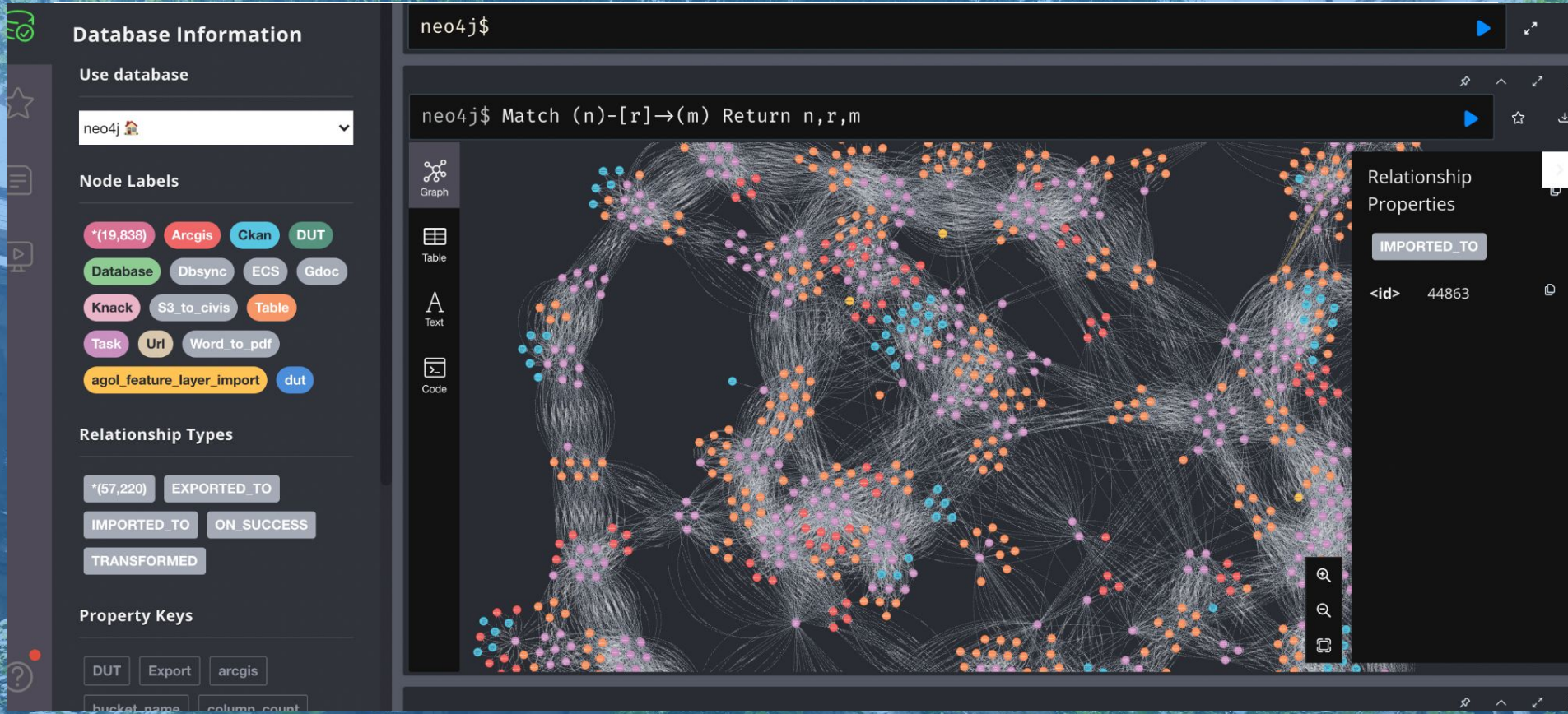
The next step was to write these nodes and relationships to Civis. These can then be downloaded as csvs onto a user's computer to be displayed within Neo4J.

Neo4J

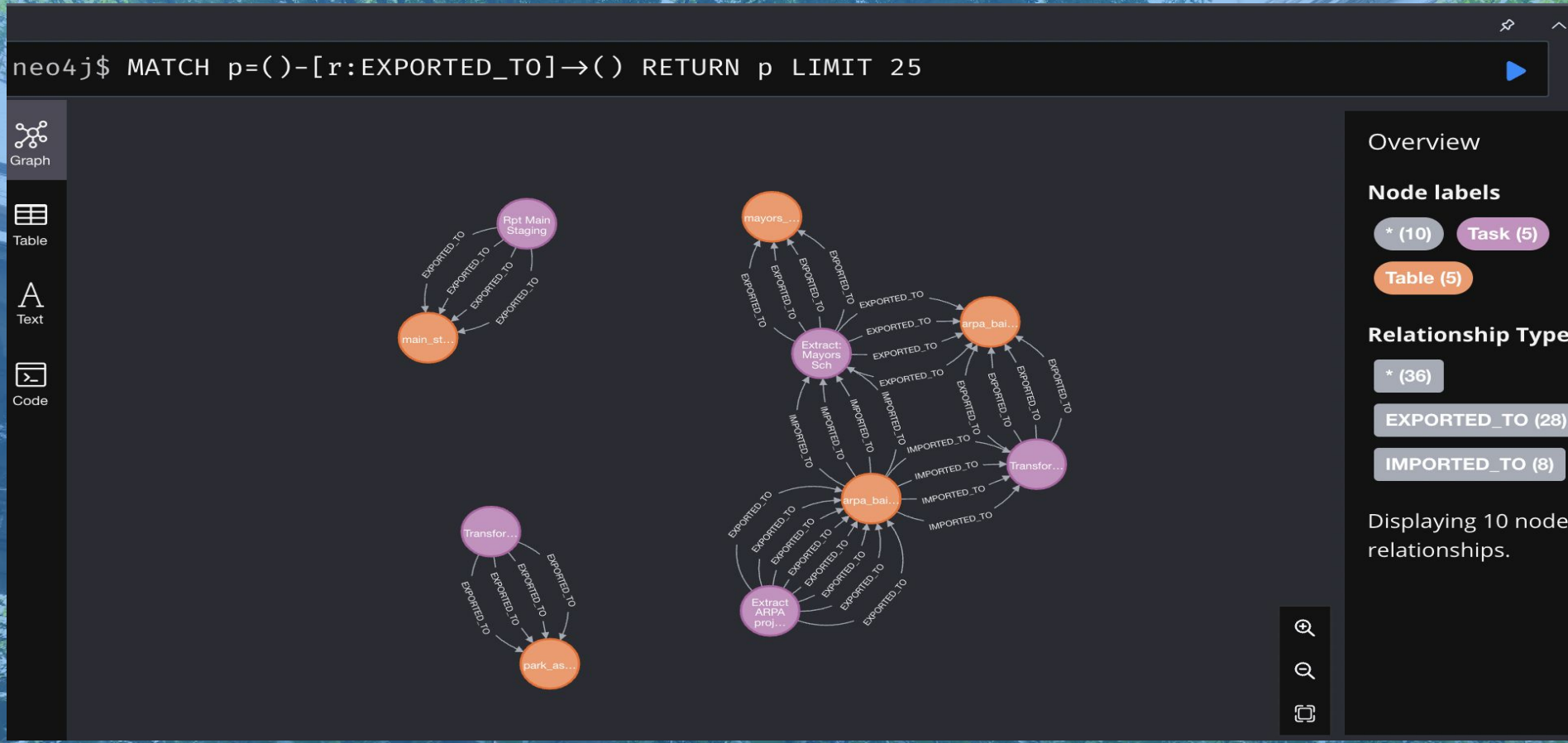
I then visualized the dependencies using Neo4J, a graph DBMS, used for storing and processing graph images.

I created a gitbooks entry on Neo4J on how to generate these graphs, based on Dan’s instructions, so that the Analytics Team can expand on this dependency diagram and make similar projects in the future.

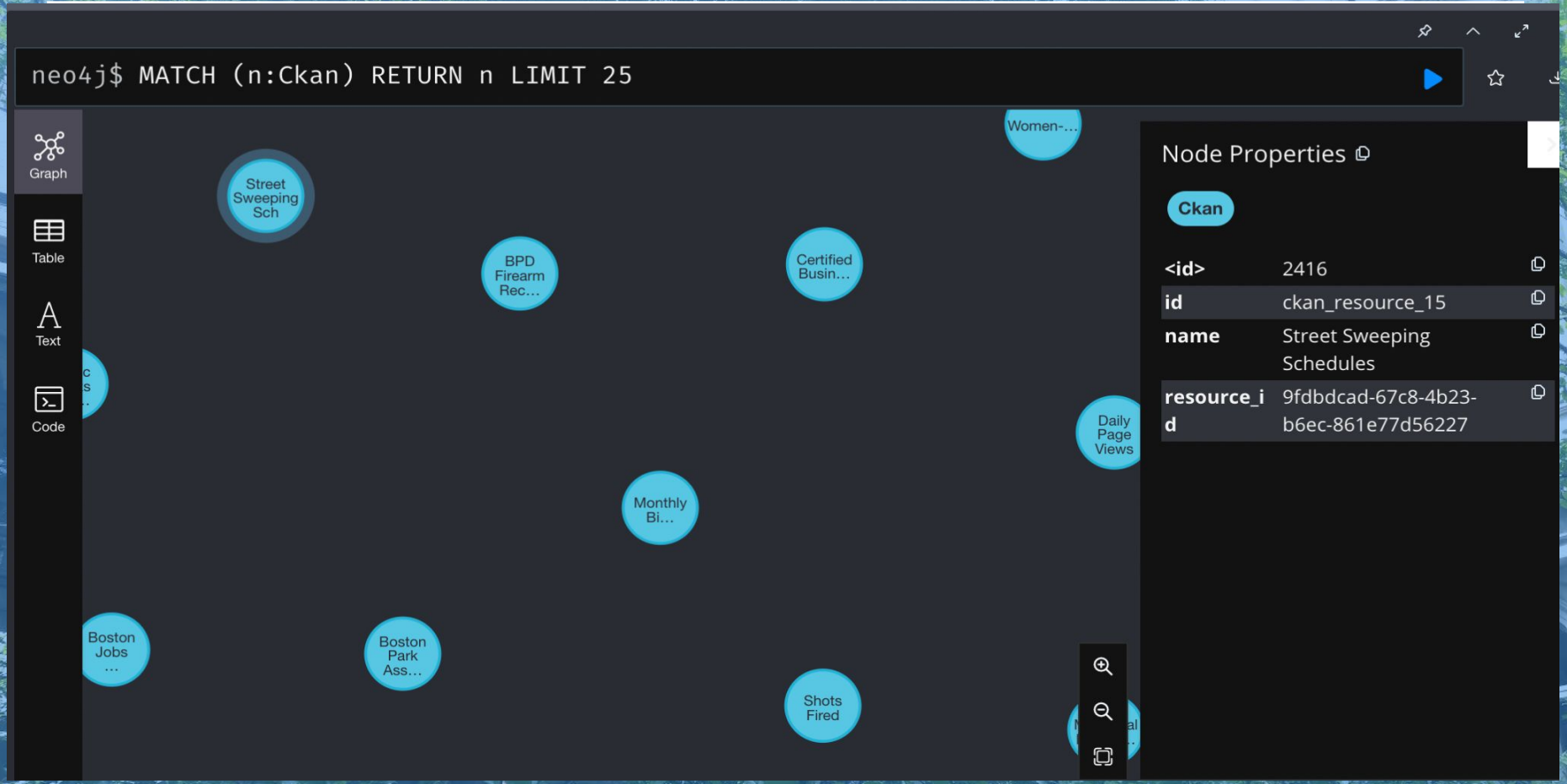
Bird's Eye View



Zooming in one one type of relationship: Exports



One type of node: Ckan



Further Steps

Some of the workflows have YAQL language that has references to tables rather than hard coded table names. We need to figure out how to access the actual name rather than the reference.

I am also working on writing some loops within the code that streamline many of the processes and makes the script more efficient.