Semantic Mapping Format (SMAP): A Context Protocol for Semantic Augmentation of Tabular Data

Scott Affens

1. Abstract

The proliferation of open data and machine learning, especially LLMs has intensified the need for datasets to carry clear, machine-actionable semantics. However, data products such as CSVs and relational tables rarely include standardized metadata explaining the meaning of each field. This omission hinders data integration, analytics, and compliance. This paper introduces the **Semantic Mapping Format (SMAP)**, designed to provide machine-readable, ontology-based semantic context for tabular data.

It presents the format, its specification, and a full mapping of an FDIC bank location dataset to leading ontologies including FIBO, OGC GeoSPARQL, GeoNames, Schema.org, etc.

2. The SMAP Standard

Background

Recent years have seen a welcome proliferation of Semantic Types as defined by various Ontologies, promising significant benefits to automated processing of data. However there is no widely accepted standard for assigning semantic types to data products. Further, it is unrealistic to require that data providers alter their products to facilitate Semantic Typing, or even to assume that the data provider should always be responsible for determining which Ontologies should be applied.

To solve the problem of assigning Semantic Types to real-world data products, the paper proposes a new mapping standard, where a new file type is created that specifies the link between each element in a data product to externally defined ontologies.Our goal is to provide a lightweight representation of this arrangement which standardizes how this mapping can be performed. Once a standard is in place, interested parties will be able to easily specify Semantic Types via a machine-readable file.

For context, this paper treats semantic types as a label on data instead of a hard compile time type. It is also focused on type alignment vs full ontological definition and recognize that external ontologies may need to be referenced for this purpose.

Goals

- This paper defines a format to enable the mapping of multiple ontologies to schemas to enable the semantic type and enrichment of data. It defines a two layer mapping scheme which enables the mapping of an ontology into an intermediate format, then a second layer to map to a schema. This decoupling enables the use of multiple ontologies as well as the ability to define internal mappings for a given community of interest.
- Enable LLM mapping with minimal manual work.

Non-Goals

It is not a goal to seek to define a new format for ontologies/reasoning languages in this paper nor couple the format to ontological definition. It is important to recognize the existence of other mechanisms to do this work such as R2RML which are purpose built for these tasks. It is important to also recognize that there are other standards for more principled ontological mapping such as R2RML however see the need for simplified format for pure, type oriented mapping onto data.

3. Semantic Mapping Format (SMAP): Specification

3.1 Format Overview

SMAP is a JSON-serializable format with three main sections:

- **ontology**: Declares the ontologies referenced.
- **type_map**: Assigns type labels to one or more ontology URIs.
- **datastore**: Maps schema columns to type_map labels.

3.2 File Extension

- `.smap.json`

A type map file defines an alignment between a schema and its semantic types. The goal is to enable a reference to public and private types. It consists of three parts, an ontology map, a type map to reference the ontological specification and a mapping to a schema. Column Spanning Types can have a structure which enables the type to be applied to more than one column. Subfields are permitted.

Use an internal representation to decouple the external ontologies from the names used in the mapping stage.

Mapping Steps

1) Declare a manifest to define the mapping files to source alignment

2) Declare the ontologies in use
3) Mapping from ontology to internal name
4) Mapping from ontological class to internal name
5) Mapping declaration for the schema
6) Mapping declaration on each field in the schema

The type_map can reference either public or private types.

- An external reference can be any named URI such as a FIBO type.
- An internal reference can be an internal reference to a URI.
- A hard reference can be a named type without a URI.

Mapping File

JSON

```
{
       ontology: {
             ONTOLOGY_NAME_A: “URI”,
             ONTOLOGY_NAME_B: “URI”,
       },
       data_product_description:"This is a description of the data product"
},
{
       type_map:
       {
             SEMANTIC_TYPE_NAME_A: EXTERNAL_REF,
             SEMANTIC_TYPE_NAME_B: INTERNAL_REF,
             SEMANTIC_TYPE_NAME_C: [ SEMANTIC_TYPE_NAME_A,
       SEMANTIC_TYPE_NAME_B],
             SEMANTIC_TYPE_NAME_D:: “HARD_REF”

       }
},
{
       datastore:
       {
             data_table_a:
             {
               data_col1:[  SEMANTIC_TYPE_A, SEMANTIC_TYPE_B]
               data_col2:[  SEMANTIC_TYPE_C.SEMANTIC_TYPE_NAME_A ]
               data_col3:[  SEMANTIC_TYPE_C.SEMANTIC_TYPE_NAME_B ]
             },
             cata_table_b:
             {
               data_col1:[SEMANTIC_TYPE_D]
             }
```

```
        }
}
```

4. Contest Mapping for the Semantic Augmentation Challenge

Our goal was to enable LLM based mapping of data types with minimal code and minimal prompting This mapping uses Anthropic claude-code model: Sonet 4. For an accurate answer the second two prompt steps were required. With some refinement this could be reduced to a one shot prompt.

```
You are an expert ontologist and data engineer.  Given the following
semantic mapping format, map all columns in the following dataset header to
the following ontologies.  Ensure all fields are mapped and make sure that
the complete URI is used in the ontological specification.   Think step by
step and be specific.  Output only the SMAP format.

Given this mapping format map all columns in this data set to the following
ontologies:

https://opengeospatial.github.io/ogc-geosparql/geosparql11/geo.ttl
https://spec.edmcouncil.org/fibo/ontology/FND/Places/NorthAmerica/USPostalSe
rviceAddresses/ZIPCode

There is only one data table the type_map should have a vector [a,b] for the
multiple mappings remember to do these mappings as a vector in the mapping
section and use the type_map the data store should use elements of the type
map, also include multiple mappings Use the uri in the type map.


This is the SMAP format:

{
      ontology: {
            ONTOLOGY_NAME_A: “URI”,
            ONTOLOGY_NAME_B: “URI”,
      },
      data_product_description:"This is a description of the data product"
},
{
      type_map:
      {
            SEMANTIC_TYPE_NAME_A: EXTERNAL_REF,
            SEMANTIC_TYPE_NAME_B: INTERNAL_REF,
            SEMANTIC_TYPE_NAME_C: [ SEMANTIC_TYPE_NAME_A,
      SEMANTIC_TYPE_NAME_B],
            SEMANTIC_TYPE_NAME_D:: "HARD_REF"

      }
},
{
      datastore:
      {
            data_table_a:
            {
              data_col1:[  SEMANTIC_TYPE_A, SEMANTIC_TYPE_B]
              data_col2:[  SEMANTIC_TYPE_C.SEMANTIC_TYPE_NAME_A ]
              data_col3:[  SEMANTIC_TYPE_C.SEMANTIC_TYPE_NAME_B ]
            },
            cata_table_b:
            {
              data_col1:[SEMANTIC_TYPE_D]
            }
      }
}

This is the header:
            X      Y      OBJECTID     ACQDATE      ADDRESS      ADDRESS2
BKCLASS      CBSA   CBSA_DIV     CBSA_DIV_FLG CBSA_DIV_NO  CBSA_METRO
CBSA_METRO_FLG     CBSA_METRO_NAME    CBSA_MICRO_FLG     CBSA_NO      CERT   CITY
      COUNTY CSA    CSA_FLG      CSA_NO ESTYMD FI_UNINUM    ID      LATITUDE
LONGITUDE    MAINOFF      MDI_STATUS_CODE    MDI_STATUS_DESC    NAME   OFFNAME
      OFFNUM RUNDATE      SERVTYPE     SERVTYPE_DESC      STALP  STCNTY STNAME
UNINUM ZIP
```

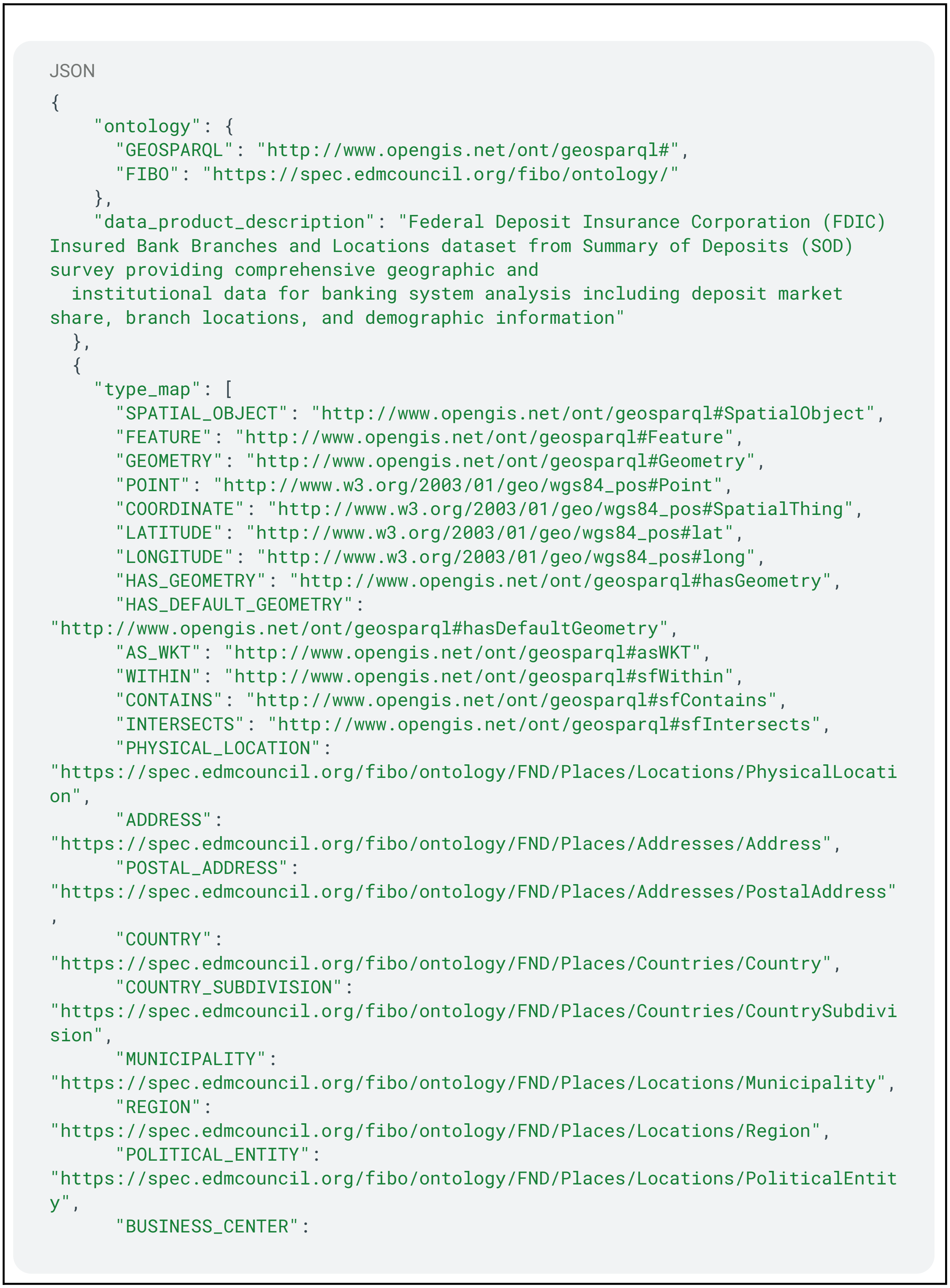
The entirety of the url contents was also then provided to the context [truncated here to save space]. Note that claude would also download this on its own if asked as well as all dependencies.
: [*https://hub.arcgis.com/datasets/geoplatform::fdic-insured-banks/about*](https://hub.arcgis.com/datasets/geoplatform::fdic-insured-banks/about)

The LLM was then asked:

Confirm the FIBO mappings using https://spec.edmcouncil.org/fibo/ontology/ as the root.

Improve the geo mappings.
https://opengeospatial.github.io/ogc-geosparql/geosparql11/geo.html

Here is the mapping output:



JSON

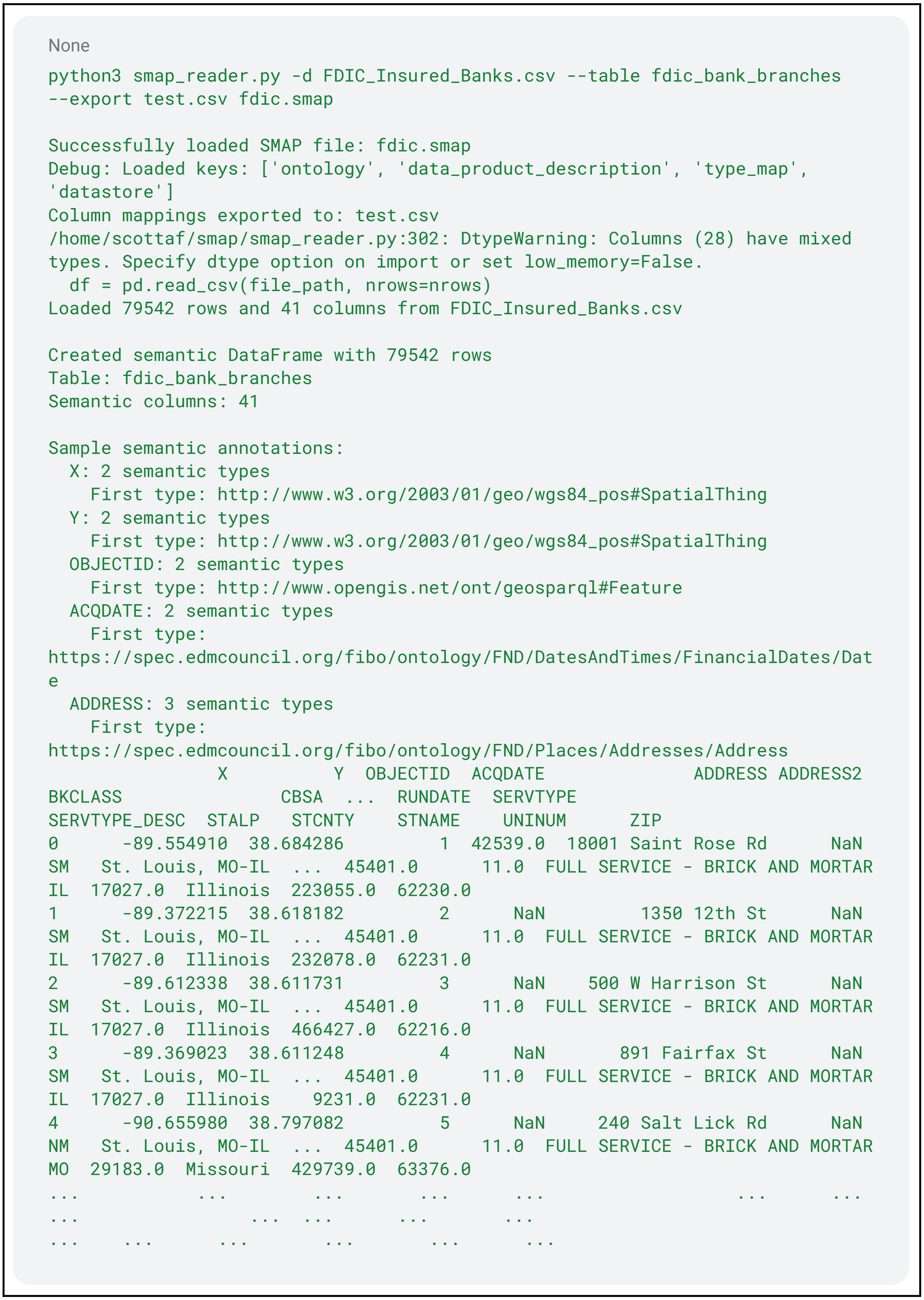
```json
{
    "ontology": {
      "GEOSPARQL": "http://www.opengis.net/ont/geosparql#",
      "FIBO": "https://spec.edmcouncil.org/fibo/ontology/"
    },
    "data_product_description": "Federal Deposit Insurance Corporation (FDIC)
Insured Bank Branches and Locations dataset from Summary of Deposits (SOD)
survey providing comprehensive geographic and
  institutional data for banking system analysis including deposit market
share, branch locations, and demographic information"
  },
  {
    "type_map": [
      "SPATIAL_OBJECT": "http://www.opengis.net/ont/geosparql#SpatialObject",
      "FEATURE": "http://www.opengis.net/ont/geosparql#Feature",
      "GEOMETRY": "http://www.opengis.net/ont/geosparql#Geometry",
      "POINT": "http://www.w3.org/2003/01/geo/wgs84_pos#Point",
      "COORDINATE": "http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing",
      "LATITUDE": "http://www.w3.org/2003/01/geo/wgs84_pos#lat",
      "LONGITUDE": "http://www.w3.org/2003/01/geo/wgs84_pos#long",
      "HAS_GEOMETRY": "http://www.opengis.net/ont/geosparql#hasGeometry",
      "HAS_DEFAULT_GEOMETRY":
"http://www.opengis.net/ont/geosparql#hasDefaultGeometry",
      "AS_WKT": "http://www.opengis.net/ont/geosparql#asWKT",
      "WITHIN": "http://www.opengis.net/ont/geosparql#sfWithin",
      "CONTAINS": "http://www.opengis.net/ont/geosparql#sfContains",
      "INTERSECTS": "http://www.opengis.net/ont/geosparql#sfIntersects",
      "PHYSICAL_LOCATION":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Locations/PhysicalLocati
on",
      "ADDRESS":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Addresses/Address",
      "POSTAL_ADDRESS":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Addresses/PostalAddress"
,
      "COUNTRY":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Countries/Country",
      "COUNTRY_SUBDIVISION":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Countries/CountrySubdivi
sion",
      "MUNICIPALITY":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Locations/Municipality",
      "REGION":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Locations/Region",
      "POLITICAL_ENTITY":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Locations/PoliticalEntit
y",
      "BUSINESS_CENTER":
```

```
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/Locations/BusinessCenter
",
      "ZIP_CODE":
"https://spec.edmcouncil.org/fibo/ontology/FND/Places/NorthAmerica/USPostalSer
viceAddresses/ZIPCode",
      "FINANCIAL_SERVICE_PROVIDER":
"https://spec.edmcouncil.org/fibo/ontology/FBC/FunctionalEntities/FinancialSer
vicesEntities/FinancialServiceProvider",
      "DEPOSITORY_INSTITUTION":
"https://spec.edmcouncil.org/fibo/ontology/FBC/FunctionalEntities/FinancialSer
vicesEntities/DepositoryInstitution",
      "BANK":
"https://spec.edmcouncil.org/fibo/ontology/FBC/FunctionalEntities/FinancialSer
vicesEntities/Bank",
      "FINANCIAL_INSTITUTION":
"https://spec.edmcouncil.org/fibo/ontology/FBC/FunctionalEntities/FinancialSer
vicesEntities/FinancialInstitution",
      "BRANCH_OFFICE":
"https://spec.edmcouncil.org/fibo/ontology/FBC/ProductsAndServices/ClientsAndA
ccounts/BranchOffice",
      "BUSINESS_ENTITY":
"https://spec.edmcouncil.org/fibo/ontology/BE/LegalEntities/LegalPersons/Busin
essEntity",
      "ORGANIZATION":
"https://spec.edmcouncil.org/fibo/ontology/FND/Organizations/Organizations/Org
anization",
      "FORMAL_ORGANIZATION":
"https://spec.edmcouncil.org/fibo/ontology/FND/Organizations/Organizations/For
malOrganization",
      "LEGAL_PERSON":
"https://spec.edmcouncil.org/fibo/ontology/BE/LegalEntities/LegalPersons/Legal
Person",
      "IDENTIFIER":
"https://spec.edmcouncil.org/fibo/ontology/FND/Relations/Relations/Identifier"
,
      "DATE":
"https://spec.edmcouncil.org/fibo/ontology/FND/DatesAndTimes/FinancialDates/Da
te",
      "CLASSIFICATION_SCHEME":
"https://spec.edmcouncil.org/fibo/ontology/FND/Arrangements/ClassificationSche
mes/ClassificationScheme",
      "CLASSIFIER":
"https://spec.edmcouncil.org/fibo/ontology/FND/Arrangements/ClassificationSche
mes/Classifier",
      "SERVICE_PROVIDER":
"https://spec.edmcouncil.org/fibo/ontology/FND/ProductsAndServices/ProductsAnd
Services/ServiceProvider",
      "SERVICE":
"https://spec.edmcouncil.org/fibo/ontology/FND/ProductsAndServices/ProductsAnd
Services/Service",
      "COORDINATE_PAIR": ["COORDINATE", "LATITUDE", "LONGITUDE"],
      "GEOGRAPHIC_POINT": ["POINT", "LATITUDE", "LONGITUDE", "GEOMETRY"],
      "COMPLETE_POSTAL_ADDRESS": ["POSTAL_ADDRESS", "ADDRESS",
"PHYSICAL_LOCATION", "ZIP_CODE"],
      "GEOGRAPHIC_HIERARCHY": ["COUNTRY_SUBDIVISION", "MUNICIPALITY",
"REGION", "POLITICAL_ENTITY"],
      "SPATIAL_FEATURE": ["FEATURE", "SPATIAL_OBJECT", "HAS_GEOMETRY",
"PHYSICAL_LOCATION"],
      "BANKING_ENTITY": ["BANK", "DEPOSITORY_INSTITUTION",
"FINANCIAL_INSTITUTION", "FINANCIAL_SERVICE_PROVIDER"],
      "ORGANIZATIONAL_STRUCTURE": ["BUSINESS_ENTITY", "FORMAL_ORGANIZATION",
"LEGAL_PERSON"],
      "SPATIAL_RELATIONSHIPS": ["WITHIN", "CONTAINS", "INTERSECTS"]
    ]
  },
  {
    "datastore": [
      "fdic_bank_branches": {
        "X": ["COORDINATE", "COORDINATE_PAIR.COORDINATE",
"SPATIAL_FEATURE.HAS_GEOMETRY"],
        "Y": ["COORDINATE", "COORDINATE_PAIR.COORDINATE",
"SPATIAL_FEATURE.HAS_GEOMETRY"],
        "OBJECTID": ["IDENTIFIER", "SPATIAL_FEATURE.FEATURE"],
        "ACQDATE": ["DATE", "BANKING_ENTITY.BANK"],
        "ADDRESS": ["COMPLETE_POSTAL_ADDRESS.ADDRESS",
"SPATIAL_FEATURE.PHYSICAL_LOCATION", "BRANCH_OFFICE"],
        "ADDRESS2": ["COMPLETE_POSTAL_ADDRESS.ADDRESS",
"SPATIAL_FEATURE.PHYSICAL_LOCATION", "BRANCH_OFFICE"],
        "BKCLASS": ["CLASSIFIER", "CLASSIFICATION_SCHEME",
"BANKING_ENTITY.BANK"],
        "CBSA": ["GEOGRAPHIC_HIERARCHY.REGION",
"SPATIAL_FEATURE.PHYSICAL_LOCATION", "BUSINESS_CENTER"],
        "CBSA_DIV": ["GEOGRAPHIC_HIERARCHY.REGION",
"SPATIAL_FEATURE.PHYSICAL_LOCATION", "SPATIAL_RELATIONSHIPS.WITHIN"],
        "CBSA_DIV_FLG": ["CLASSIFIER", "GEOGRAPHIC_HIERARCHY.REGION"],
        "CBSA_DIV_NO": ["IDENTIFIER", "GEOGRAPHIC_HIERARCHY.REGION"],
        "CBSA_METRO": ["GEOGRAPHIC_HIERARCHY.REGION", "BUSINESS_CENTER",
"SPATIAL_FEATURE.PHYSICAL_LOCATION"],
        "CBSA_METRO_FLG": ["CLASSIFIER", "GEOGRAPHIC_HIERARCHY.REGION"],
        "CBSA_METRO_NAME": ["GEOGRAPHIC_HIERARCHY.REGION", "BUSINESS_CENTER"],
        "CBSA_MICRO_FLG": ["CLASSIFIER", "GEOGRAPHIC_HIERARCHY.REGION"],
        "CBSA_NO": ["IDENTIFIER", "GEOGRAPHIC_HIERARCHY.REGION"],
        "CERT": ["IDENTIFIER", "BANKING_ENTITY.DEPOSITORY_INSTITUTION"],
        "CITY": ["COMPLETE_POSTAL_ADDRESS.MUNICIPALITY",
"GEOGRAPHIC_HIERARCHY.MUNICIPALITY", "BRANCH_OFFICE"],
        "COUNTY": ["GEOGRAPHIC_HIERARCHY.COUNTRY_SUBDIVISION",
"GEOGRAPHIC_HIERARCHY.POLITICAL_ENTITY", "SPATIAL_RELATIONSHIPS.CONTAINS"],
        "CSA": ["GEOGRAPHIC_HIERARCHY.REGION", "BUSINESS_CENTER",
"SPATIAL_RELATIONSHIPS.CONTAINS"],
        "CSA_FLG": ["CLASSIFIER", "GEOGRAPHIC_HIERARCHY.REGION"],
        "CSA_NO": ["IDENTIFIER", "GEOGRAPHIC_HIERARCHY.REGION"],
        "ESTYMD": ["DATE", "BRANCH_OFFICE"],
        "FI_UNINUM": ["IDENTIFIER", "BANKING_ENTITY.FINANCIAL_INSTITUTION"],
        "ID": ["IDENTIFIER", "BRANCH_OFFICE"],
        "LATITUDE": ["GEOGRAPHIC_POINT.LATITUDE", "GEOGRAPHIC_POINT.POINT",
"SPATIAL_FEATURE.HAS_GEOMETRY", "BRANCH_OFFICE"],
        "LONGITUDE": ["GEOGRAPHIC_POINT.LONGITUDE", "GEOGRAPHIC_POINT.POINT",
"SPATIAL_FEATURE.HAS_GEOMETRY", "BRANCH_OFFICE"],
        "MAINOFF": ["CLASSIFIER", "BRANCH_OFFICE"],
        "MDI_STATUS_CODE": ["CLASSIFIER", "CLASSIFICATION_SCHEME",
"BANKING_ENTITY.DEPOSITORY_INSTITUTION"],
        "MDI_STATUS_DESC": ["CLASSIFIER", "CLASSIFICATION_SCHEME",
"BANKING_ENTITY.DEPOSITORY_INSTITUTION"],
        "NAME": ["ORGANIZATIONAL_STRUCTURE.FORMAL_ORGANIZATION",
"BANKING_ENTITY.FINANCIAL_INSTITUTION"],
        "OFFNAME": ["ORGANIZATIONAL_STRUCTURE.FORMAL_ORGANIZATION",
"BRANCH_OFFICE"],
        "OFFNUM": ["IDENTIFIER", "BRANCH_OFFICE"],
        "RUNDATE": ["DATE", "BANKING_ENTITY.FINANCIAL_INSTITUTION"],
        "SERVTYPE": ["CLASSIFIER", "SERVICE", "BRANCH_OFFICE"],
        "SERVTYPE_DESC": ["CLASSIFIER", "SERVICE", "BRANCH_OFFICE"],
        "STALP": ["COMPLETE_POSTAL_ADDRESS.COUNTRY_SUBDIVISION",
"GEOGRAPHIC_HIERARCHY.COUNTRY_SUBDIVISION", "BRANCH_OFFICE"],
        "STCNTY": ["IDENTIFIER", "GEOGRAPHIC_HIERARCHY.COUNTRY_SUBDIVISION"],
        "STNAME": ["COMPLETE_POSTAL_ADDRESS.COUNTRY_SUBDIVISION",
"GEOGRAPHIC_HIERARCHY.COUNTRY_SUBDIVISION", "BRANCH_OFFICE"],
        "UNINUM": ["IDENTIFIER", "BRANCH_OFFICE"],
        "ZIP": ["ZIP_CODE", "COMPLETE_POSTAL_ADDRESS.ZIP_CODE",
"BRANCH_OFFICE"]
      }
    ]
  }
```

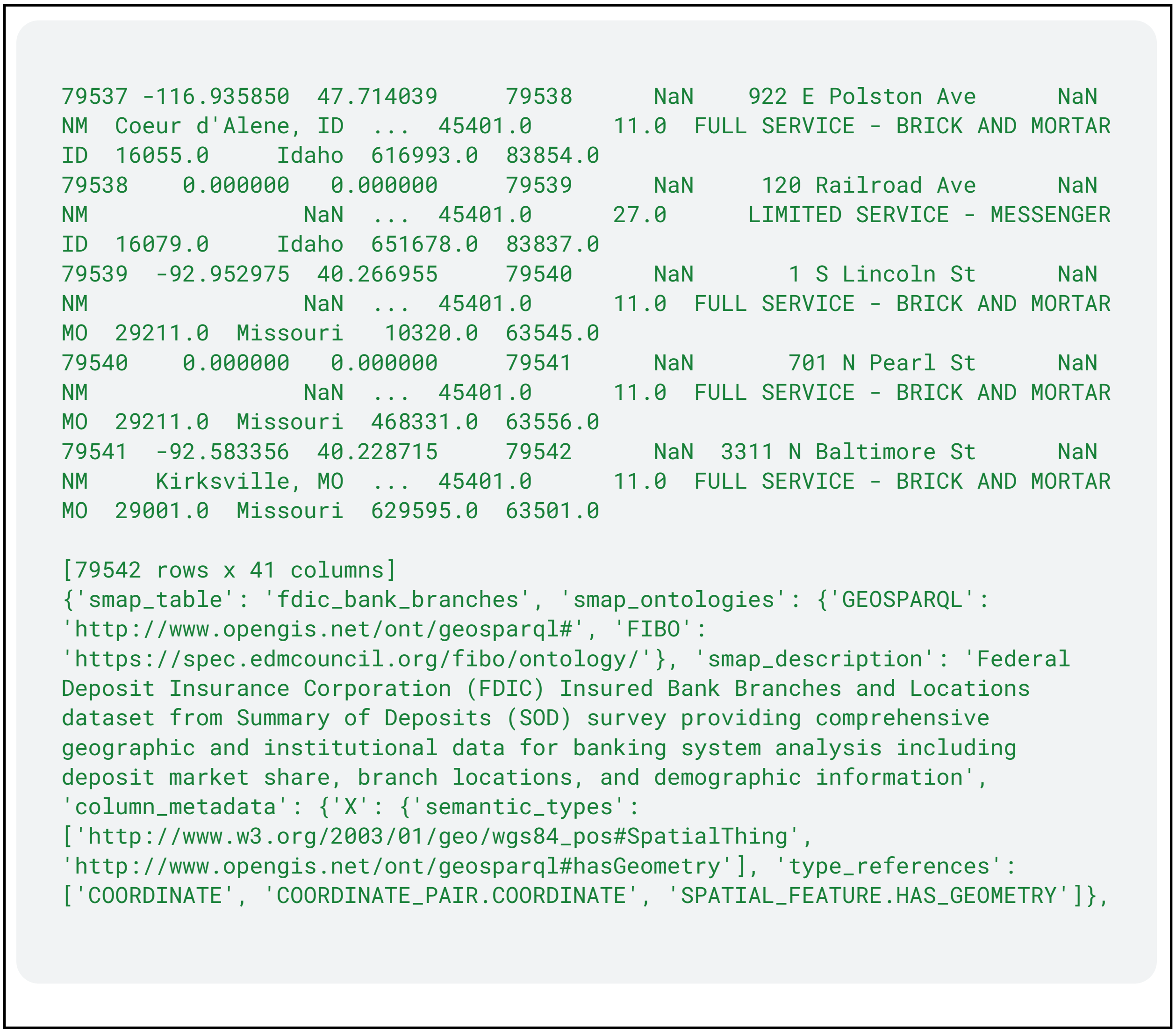
5. Mapper to Pandas

Here is a python script to demonstrate a mapping from SMAP to pandas.

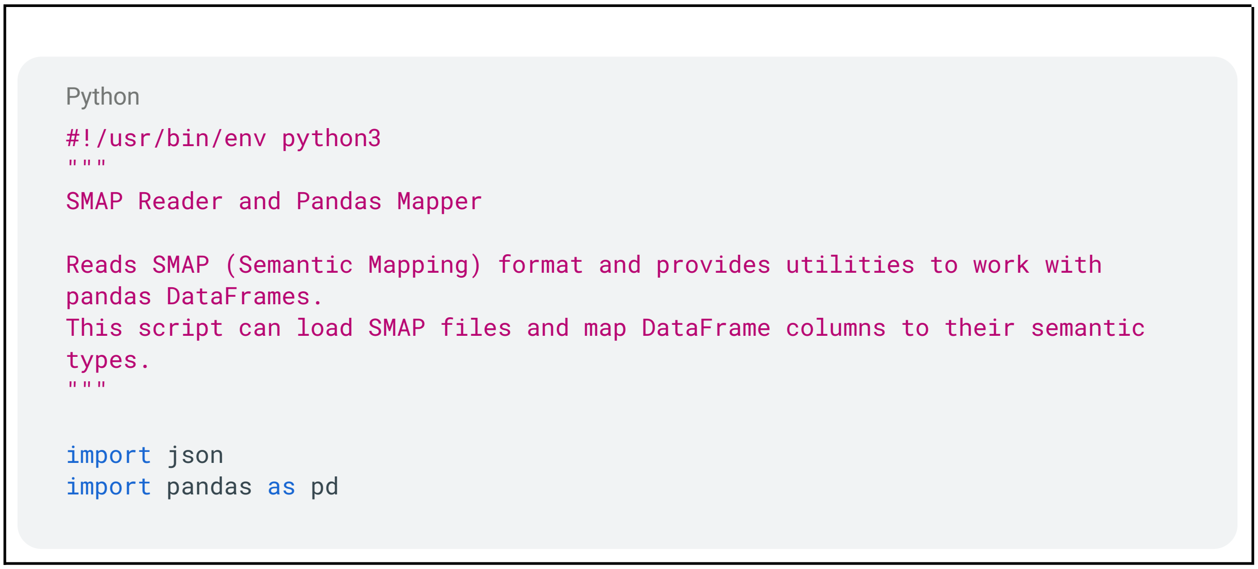
```
python3 smap_reader.py -d FDIC_Insured_Banks.csv --table fdic_bank_branches --export
test.csv fdic.smap
```



```
python3 smap_reader.py -d FDIC_Insured_Banks.csv --table fdic_bank_branches
--export test.csv fdic.smap

Successfully loaded SMAP file: fdic.smap
Debug: Loaded keys: ['ontology', 'data_product_description', 'type_map',
'datastore']
Column mappings exported to: test.csv
/home/scottaf/smap/smap_reader.py:302: DtypeWarning: Columns (28) have mixed
types. Specify dtype option on import or set low_memory=False.
  df = pd.read_csv(file_path, nrows=nrows)
Loaded 79542 rows and 41 columns from FDIC_Insured_Banks.csv

Created semantic DataFrame with 79542 rows
Table: fdic_bank_branches
Semantic columns: 41

Sample semantic annotations:
  X: 2 semantic types
    First type: http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
  Y: 2 semantic types
    First type: http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing
  OBJECTID: 2 semantic types
    First type: http://www.opengis.net/ont/geosparql#Feature
  ACQDATE: 2 semantic types
    First type:
https://spec.edmcouncil.org/fibo/ontology/FND/DatesAndTimes/FinancialDates/Dat
e
  ADDRESS: 3 semantic types
    First type:
https://spec.edmcouncil.org/fibo/ontology/FND/Places/Addresses/Address
                X          Y  OBJECTID  ACQDATE              ADDRESS ADDRESS2
BKCLASS              CBSA  ...  RUNDATE  SERVTYPE
SERVTYPE_DESC  STALP   STCNTY    STNAME    UNINUM      ZIP
0      -89.554910  38.684286         1  42539.0  18001 Saint Rose Rd      NaN
SM   St. Louis, MO-IL  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
IL  17027.0  Illinois  223055.0  62230.0
1      -89.372215  38.618182         2      NaN         1350 12th St      NaN
SM   St. Louis, MO-IL  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
IL  17027.0  Illinois  232078.0  62231.0
2      -89.612338  38.611731         3      NaN    500 W Harrison St      NaN
SM   St. Louis, MO-IL  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
IL  17027.0  Illinois  466427.0  62216.0
3      -89.369023  38.611248         4      NaN       891 Fairfax St      NaN
SM   St. Louis, MO-IL  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
IL  17027.0  Illinois    9231.0  62231.0
4      -90.655980  38.797082         5      NaN     240 Salt Lick Rd      NaN
NM   St. Louis, MO-IL  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
MO  29183.0  Missouri  429739.0  63376.0
...           ...        ...       ...      ...                   ...      ...
...                ...  ...      ...       ...
...    ...      ...       ...       ...      ...
```



```
79537 -116.935850  47.714039     79538      NaN    922 E Polston Ave      NaN
NM  Coeur d'Alene, ID  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
ID  16055.0     Idaho  616993.0  83854.0
79538    0.000000   0.000000     79539      NaN     120 Railroad Ave      NaN
NM                 NaN  ...  45401.0      27.0      LIMITED SERVICE - MESSENGER
ID  16079.0     Idaho  651678.0  83837.0
79539  -92.952975  40.266955     79540      NaN       1 S Lincoln St      NaN
NM                 NaN  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
MO  29211.0  Missouri   10320.0  63545.0
79540    0.000000   0.000000     79541      NaN       701 N Pearl St      NaN
NM                 NaN  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
MO  29211.0  Missouri  468331.0  63556.0
79541  -92.583356  40.228715     79542      NaN  3311 N Baltimore St      NaN
NM     Kirksville, MO  ...  45401.0      11.0  FULL SERVICE - BRICK AND MORTAR
MO  29001.0  Missouri  629595.0  63501.0

[79542 rows x 41 columns]
{'smap_table': 'fdic_bank_branches', 'smap_ontologies': {'GEOSPARQL':
'http://www.opengis.net/ont/geosparql#', 'FIBO':
'https://spec.edmcouncil.org/fibo/ontology/'}, 'smap_description': 'Federal
Deposit Insurance Corporation (FDIC) Insured Bank Branches and Locations
dataset from Summary of Deposits (SOD) survey providing comprehensive
geographic and institutional data for banking system analysis including
deposit market share, branch locations, and demographic information',
'column_metadata': {'X': {'semantic_types':
['http://www.w3.org/2003/01/geo/wgs84_pos#SpatialThing',
'http://www.opengis.net/ont/geosparql#hasGeometry'], 'type_references':
['COORDINATE', 'COORDINATE_PAIR.COORDINATE', 'SPATIAL_FEATURE.HAS_GEOMETRY']},
```



Python

```python
#!/usr/bin/env python3
"""
SMAP Reader and Pandas Mapper

Reads SMAP (Semantic Mapping) format and provides utilities to work with
pandas DataFrames.
This script can load SMAP files and map DataFrame columns to their semantic
types.
"""

import json
import pandas as pd
import argparse
from typing import Dict, List, Any, Optional, Tuple
from pathlib import Path


class SMAPReader:
    def __init__(self, smap_file: str = None):
        """Initialize SMAP Reader with optional SMAP file."""
        self.smap_data = None
        self.ontologies = {}
        self.type_map = {}
        self.datastore = {}
        self.data_product_description = ""

        if smap_file:
            self.load_smap(smap_file)

    def load_smap(self, smap_file: str) -> None:
        """Load SMAP file and parse its components."""
        try:
            with open(smap_file, 'r') as f:
                content = f.read().strip()

            # Parse SMAP format which contains comma-separated JSON objects
            self.smap_data = self._parse_smap_format(content)

            self._parse_smap_components()
            print(f"Successfully loaded SMAP file: {smap_file}")
            print(f"Debug: Loaded keys: {list(self.smap_data.keys())}")

        except FileNotFoundError:
            raise FileNotFoundError(f"SMAP file not found: {smap_file}")
        except (json.JSONDecodeError, ValueError) as e:
            raise ValueError(f"Invalid SMAP format: {e}")

    def _parse_smap_format(self, content: str) -> Dict[str, Any]:
        """Parse SMAP format which contains comma-separated JSON objects."""
        # SMAP format is comma-separated JSON objects, so wrap in array and
parse
        try:
            # Try wrapping in array brackets
            array_content = f"[{content}]"
            objects = json.loads(array_content)

            # Merge all objects into a single dictionary
            merged = {}
            for obj in objects:
                if isinstance(obj, dict):
                    merged.update(obj)

            return merged
        except json.JSONDecodeError:
            # Fall back to manual parsing if array approach fails
            return self._manual_parse_smap(content)

    def _manual_parse_smap(self, content: str) -> Dict[str, Any]:
        """Manually parse SMAP format by splitting on },{ pattern."""
        # Split on the pattern "},\n{" to separate objects
        import re

        # Add array brackets and fix the comma-separated structure
        parts = re.split(r'},\s*{', content.strip())

        objects = []
        for i, part in enumerate(parts):
            # Add missing braces
            if i == 0:
                part = part + '}'
            elif i == len(parts) - 1:
                part = '{' + part
            else:
                part = '{' + part + '}'

            try:
                obj = json.loads(part.strip())
                objects.append(obj)
            except json.JSONDecodeError as e:
                print(f"Warning: Skipping malformed JSON object: {e}")

        # Merge all objects
        merged = {}
        for obj in objects:
            if isinstance(obj, dict):
                merged.update(obj)

        return merged

    def _parse_smap_components(self) -> None:
        """Parse the main components of the SMAP file."""
        # Parse ontologies
        ontologies_data = self.smap_data.get('ontology', {})
        if isinstance(ontologies_data, dict):
            self.ontologies.update(ontologies_data)
        elif isinstance(ontologies_data, list):
            for item in ontologies_data:
                if isinstance(item, dict):
                    self.ontologies.update(item)

        # Parse type map
        type_map_data = self.smap_data.get('type_map', {})
        if isinstance(type_map_data, dict):
            self.type_map.update(type_map_data)
        elif isinstance(type_map_data, list):
            for item in type_map_data:
                if isinstance(item, dict):
                    self.type_map.update(item)

        # Parse datastore
        datastore_data = self.smap_data.get('datastore', [])
        if isinstance(datastore_data, list):
            for item in datastore_data:
                if isinstance(item, dict):
                    self.datastore.update(item)
        elif isinstance(datastore_data, dict):
            self.datastore.update(datastore_data)

        # Get description
        self.data_product_description =
self.smap_data.get('data_product_description', "")

    def get_ontologies(self) -> Dict[str, str]:
        """Get all ontology mappings."""
        return self.ontologies

    def get_type_definitions(self) -> Dict[str, Any]:
        """Get all type definitions from the type map."""
        return self.type_map

    def get_tables(self) -> List[str]:
        """Get list of all table names in the datastore."""
        return list(self.datastore.keys())

    def get_table_columns(self, table_name: str) -> Dict[str, List[str]]:
        """Get column mappings for a specific table."""
        return self.datastore.get(table_name, {})

    def resolve_type_reference(self, type_ref: str) -> List[str]:
        """Resolve a type reference to its full URI(s)."""
        if '.' in type_ref:
            # Handle composite references like
"COMPLETE_POSTAL_ADDRESS.ZIP_CODE"
            base_type, sub_type = type_ref.split('.', 1)
            if base_type in self.type_map:
                base_definition = self.type_map[base_type]
                if isinstance(base_definition, list):
                    # Find the sub_type in the list
                    for item in base_definition:
                        if item == sub_type and sub_type in self.type_map:
                            return [self.type_map[sub_type]]
                    # If not found as direct reference, return the sub_type if
it exists
                    if sub_type in self.type_map:
                        return [self.type_map[sub_type]]
            return []
        else:
            # Direct reference
            if type_ref in self.type_map:
                definition = self.type_map[type_ref]
                if isinstance(definition, str):
                    return [definition]
                elif isinstance(definition, list):
                    # Recursively resolve list items
                    resolved = []
                    for item in definition:
                        resolved.extend(self.resolve_type_reference(item))
                    return resolved
        return []

    def get_column_semantic_types(self, table_name: str, column_name: str) ->
List[str]:
        """Get all semantic type URIs for a specific column."""
        table_data = self.get_table_columns(table_name)
        if column_name in table_data:
            all_uris = []
            for type_ref in table_data[column_name]:
                uris = self.resolve_type_reference(type_ref)
                all_uris.extend(uris)
            return list(set(all_uris))  # Remove duplicates
        return []

    def create_semantic_dataframe(self, df: pd.DataFrame, table_name: str) ->
pd.DataFrame:
        """Create a DataFrame with semantic type annotations."""
        if table_name not in self.datastore:
            raise ValueError(f"Table '{table_name}' not found in SMAP
datastore")

        # Create a copy of the DataFrame
        semantic_df = df.copy()

        # Add semantic metadata as DataFrame attributes
        semantic_df.attrs['smap_table'] = table_name
        semantic_df.attrs['smap_ontologies'] = self.ontologies
        semantic_df.attrs['smap_description'] = self.data_product_description

        # Create column metadata dictionary
        column_metadata = {}
        table_columns = self.get_table_columns(table_name)

        for column in semantic_df.columns:
            if column in table_columns:
                semantic_types = self.get_column_semantic_types(table_name,
column)
```

```python
                type_refs = table_columns[column]
                column_metadata[column] = {
                    'semantic_types': semantic_types,
                    'type_references': type_refs
                }
            else:
                column_metadata[column] = {
                    'semantic_types': [],
                    'type_references': []
                }

        semantic_df.attrs['column_metadata'] = column_metadata

        return semantic_df

    def print_mapping_summary(self, table_name: str = None) -> None:
        """Print a summary of the semantic mappings."""
        print(f"\n=== SMAP Summary ===")
        print(f"Description: {self.data_product_description}")
        print(f"\nOntologies ({len(self.ontologies)}):")
        for name, uri in self.ontologies.items():
            print(f"  {name}: {uri}")

        print(f"\nType Definitions ({len(self.type_map)}):")
        for type_name, definition in list(self.type_map.items())[:10]:  # Show first 10
            if isinstance(definition, str):
                print(f"  {type_name}: {definition}")
            else:
                print(f"  {type_name}: {definition}")
        if len(self.type_map) > 10:
            print(f"  ... and {len(self.type_map) - 10} more")

        if table_name:
            if table_name in self.datastore:
                print(f"\nTable: {table_name}")
                table_data = self.datastore[table_name]
                print(f"Columns ({len(table_data)}):")
                for col, mappings in table_data.items():
                    semantic_types = self.get_column_semantic_types(table_name, col)
                    print(f"  {col}: {mappings} -> {len(semantic_types)} URIs")
            else:
                print(f"\nTable '{table_name}' not found")
        else:
            print(f"\nTables ({len(self.datastore)}):")
            for table in self.datastore.keys():
                print(f"  {table}: {len(self.datastore[table])} columns")
    def export_column_mappings(self, table_name: str, output_file: str = None) -> pd.DataFrame:
        """Export column mappings to a CSV file for review."""
        if table_name not in self.datastore:
            raise ValueError(f"Table '{table_name}' not found in SMAP datastore")

        mappings_data = []
        table_data = self.get_table_columns(table_name)

        for column, type_refs in table_data.items():
            semantic_types = self.get_column_semantic_types(table_name, column)
            mappings_data.append({
                'column_name': column,
                'type_references': ', '.join(type_refs),
                'semantic_uris': ', '.join(semantic_types),
                'uri_count': len(semantic_types)
            })

        mappings_df = pd.DataFrame(mappings_data)

        if output_file:
            mappings_df.to_csv(output_file, index=False)
            print(f"Column mappings exported to: {output_file}")

        return mappings_df

    def validate_dataframe_columns(self, df: pd.DataFrame, table_name: str) -> Tuple[List[str], List[str]]:
        """Validate that DataFrame columns match SMAP table definition."""
        if table_name not in self.datastore:
            raise ValueError(f"Table '{table_name}' not found in SMAP datastore")

        smap_columns = set(self.get_table_columns(table_name).keys())
        df_columns = set(df.columns)

        missing_in_df = list(smap_columns - df_columns)
        missing_in_smap = list(df_columns - smap_columns)

        if missing_in_df:
            print(f"Warning: Columns in SMAP but not in DataFrame: {missing_in_df}")
        if missing_in_smap:
            print(f"Warning: Columns in DataFrame but not in SMAP: {missing_in_smap}")

        return missing_in_df, missing_in_smap
```

```
def load_sample_data(file_path: str, nrows: int = None) -> pd.DataFrame:
    """Load sample data from CSV file."""
    try:
        if file_path.endswith('.csv'):
            df = pd.read_csv(file_path, nrows=nrows)
        elif file_path.endswith('.json'):
            df = pd.read_json(file_path, nrows=nrows)
        else:
            # Try CSV as default
            df = pd.read_csv(file_path, nrows=nrows)

        print(f"Loaded {len(df)} rows and {len(df.columns)} columns from
{file_path}")
        return df

    except Exception as e:
        print(f"Error loading data file: {e}")
        return None


def main():
    parser = argparse.ArgumentParser(description='Read SMAP format and work
with pandas DataFrames')
    parser.add_argument('smap_file', help='SMAP JSON file')
    parser.add_argument('-d', '--data', help='Data file (CSV/JSON) to map')
    parser.add_argument('-t', '--table', help='Table name in SMAP to use for
mapping')
    parser.add_argument('-s', '--summary', action='store_true', help='Show
mapping summary')
    parser.add_argument('-e', '--export', help='Export column mappings to CSV
file')
    parser.add_argument('-n', '--nrows', type=int, help='Number of rows to
load from data file')

    args = parser.parse_args()

    # Load SMAP file
    try:
        reader = SMAPReader(args.smap_file)
    except Exception as e:
        print(f"Error loading SMAP file: {e}")
        return 1

    # Show summary if requested
    if args.summary:
        reader.print_mapping_summary(args.table)

    # Export mappings if requested
    if args.export and args.table:
        try:
            reader.export_column_mappings(args.table, args.export)
        except Exception as e:
            print(f"Error exporting mappings: {e}")

    # Load and map data if provided
    if args.data:
        if not args.table:
            print("Error: --table is required when using --data")
            return 1

        df = load_sample_data(args.data, args.nrows)
        if df is not None:
            try:
                # Validate columns
                missing_in_df, missing_in_smap =
reader.validate_dataframe_columns(df, args.table)

                # Create semantic DataFrame
                semantic_df = reader.create_semantic_dataframe(df, args.table)

                print(f"\nCreated semantic DataFrame with {len(semantic_df)}
rows")
                print(f"Table: {semantic_df.attrs['smap_table']}")
                print(f"Semantic columns: {len([c for c in
semantic_df.attrs['column_metadata'].keys() if
semantic_df.attrs['column_metadata'][c]['semantic_types']])}")

                # Show sample of semantic annotations
                print(f"\nSample semantic annotations:")
                for col in list(semantic_df.columns)[:5]:
                    metadata = semantic_df.attrs['column_metadata'][col]
                    if metadata['semantic_types']:
                        print(f"  {col}: {len(metadata['semantic_types'])}
semantic types")
                        print(f"    First type: {metadata['semantic_types'][0]
if metadata['semantic_types'] else 'None'}")
                print(semantic_df)
                print(semantic_df.attrs)

            except Exception as e:
                print(f"Error creating semantic DataFrame: {e}")

    return 0


if __name__ == "__main__":
    exit(main())
```

Appendix A: Other Examples

Example with FIBO Mapping to a basic CSV

In this example there is a map of a simple CSV file to a set of external and internal ontologies.

```
CSV Schema
FinQualData:EquityDailyInfoProduct file1.csv
id, fqdedi, ticker, close_price, date
1,   , AAPL, 175.10, 2023-09-15
```

Mapping File

`prices.smap`

```
JSON
{
       {
             data_product_description: "FinQualData:EquityDailyInfoProduct"
             ontology_map: [
             fibo: ”https://spec.edmcouncil.org/fibo/ontology/”
             acme_type: OLHCPrice.close,
             figi: ”https://www.omg.org/spec/FIGI/20150501”
           ]
       },
       {
             type_map:
             {
                  vendor_id : vendor_id
                  figi_id: figi:GlobalIndustryIdentifer,
                  fibo_ticker: fibo:fibo-sec-sec-id;TickerSymbol,
                  fibo_close_price: fibo:fibo-fbc-fi-ip;AdjustedClosingPrice,
                  internal_close_price : acme_type,
                  close_price_date : fibo:fibo-fnd-dt-fd;SpecifiedDate

             }
       },
       {
             datastore:
             {
                    csv: {
                      id:[ vendor_id ],
                      fqdedi:figi: [ figi_id ],
                      ticker:[ fibo_ticker ],
                      close_price:[ fibo_close_price, internal_close_price ],
                      date : [ close_price_date ]
                    }
             }
       }
}
```

Database Example

In this example there is a map of a set of ontologies to a simple database schema.

```
database : etf_database
table etf_prices_table
id, ticker, price, date
table etf_constituents
id, ticker, etf_ticker, date_joined, date_left
```

Mapping File

`etf.smap`

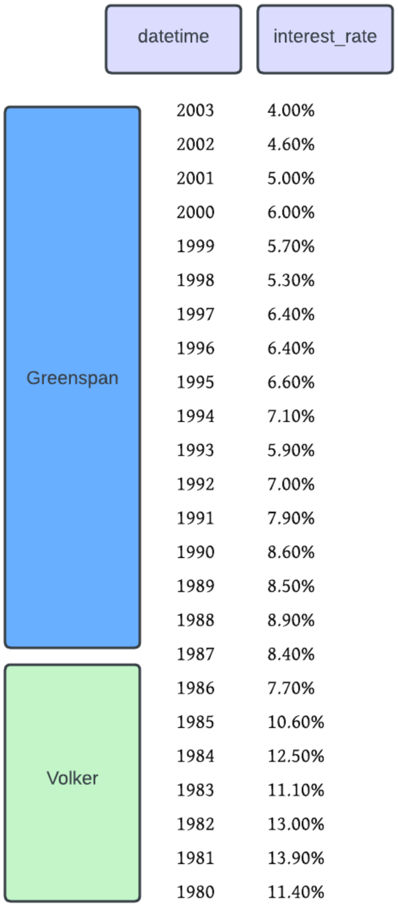
```
JSON

{
       {
             ontology_map: [
             Fibo: "https://spec.edmcouncil.org/fibo/ontology/"
             acme_type: OLHCPrice.close,
             figi: "https://www.omg.org/spec/FIGI/20150501"
       ],
             data_product_description: "FinQualData:ETFProduct"
},
{
       type_map:
       {
            vendor_id : vendor_id
            figi_id: figi:GlobalIndustryIdentifer,
            fibo_ticker: fibo:fibo-sec-sec-id;TickerSymbol,
            fibo_close_price: fibo:fibo-fbc-fi-ip;AdjustedClosingPrice,
            internal_close_price : acme_type,
            etf_date: fibo:fibo-fnd-dt-fd;SpecifiedDate,
             etf_date_joined :  fibo:fibo-fnd-dt-fd;SpecifiedDate,
			etf: fibo-sec-fund-fund:ExchangeTradedFund


		}
},
{
		datastore: {
			table1: {
				etf_prices_table:{
				id:[ vendor_id ],
				ticker:[ etf, figi_id ],
				close_price:[ fibo_close_price, internal_close_price ],
				date : [ close_price_date ]
			},

			etf_constituents : {
				id:[ vendor_id ],
				ticker:[ fibo_ticker ],
				eft_ticker:[ figi_id , etf ],
				date_joined : [ etf_date_joined ]
				date_left : [ etf_date_left ]
			}
		]

}
```

Appendix B: Optional Features

Optional: Column Content Context (For enrichment)

Semantic types can be useful in defining row based content labels which can assist users in understanding regime shift within data sets. Consider US interest rates over time; looking historically there are periods where there are different conditions impacting the rates. Those conditions could be time aligned if there was a specific row label indicating this knowledge on the rate. There are many other possible alignments where labeling data in this way can help such as if measurement changes occurred or even if there are simple shifts in units.

A user can add an SMAP optional field ‘applied_date’ and repeat the pattern if this is required for a given data set.



	datetime	interest_rate
Greenspan	2003	4.00%
	2002	4.60%
	2001	5.00%
	2000	6.00%
	1999	5.70%
	1998	5.30%
	1997	6.40%
	1996	6.40%
	1995	6.60%
	1994	7.10%
	1993	5.90%
	1992	7.00%
	1991	7.90%
	1990	8.60%
	1989	8.50%
	1988	8.90%
	1987	8.40%
Volker	1986	7.70%
	1985	10.60%
	1984	12.50%
	1983	11.10%
	1982	13.00%
	1981	13.90%
	1980	11.40%