import streamlit as st

import pandas as pd

import numpy as np

from datetime import datetime, date, timezone

import os

from typing import Dict, List, Optional, Union, Any

import logging

from backend.data\_reader import (

    read\_csv\_dat,

    read\_sql,

    read\_stored\_proc,

    read\_teradata,

    read\_api,

    read\_parquet,

    read\_zipped\_files

)

from backend.mapping\_utils import (

    auto\_map\_columns,

    validate\_join\_columns,

    validate\_mapping,

    apply\_column\_mapping,

    get\_excluded\_columns

)

from backend.report\_generator import (

    generate\_datacompy\_report,

    generate\_ydata\_profile,

    generate\_regression\_report,

    generate\_difference\_report,

    create\_consolidated\_report,

    create\_individual\_reports\_zip

)

from backend.utils import check\_file\_size, setup\_logger

# Set page config

st.set\_page\_config(

    page\_title="Data Comparison Framework",

    page\_icon="📊",

    layout="wide"

)

# Set up logging

logger = logging.getLogger(\_\_name\_\_)

if not logger.handlers:

    logger = setup\_logger()

# Define the type mapping dictionary

TYPE\_MAPPING = {

    # Integer types

    'int': 'int64',

    'int32': 'int64',

    'int64': 'int64',

    'integer': 'int64',

    'numeric': 'int64',

    'bigint': 'int64',

    'smallint': 'int64',

    'tinyint': 'int64',

    # Float types

    'Float': 'float64',

    'float': 'float64',

    'float32': 'float64',

    'float64': 'float64',

    'decimal': 'float64',

    'double': 'float64',

    'real': 'float64',

    # String types

    'varchar': 'string',

    'nvarchar': 'string',

    'char': 'string',

    'nchar': 'string',

    'text': 'string',

    'ntext': 'string',

    'string': 'string',

    'object': 'string',

    # Date/Time types

    'date': 'datetime64[ns]',

    'datetime': 'datetime64[ns]',

    'datetime64': 'datetime64[ns]',

    'timestamp': 'datetime64[ns]',

    # Boolean types

    'Boolean': 'bool',

    'bool': 'bool',

    'bit': 'bool',

    # Default type

    'unknown': 'string'

}

def main():

    # Add header with image

    st.image("https://images.pexels.com/photos/414612/pexels-photo-414612.jpeg",

             caption="Data Comparison Dashboard",

             use\_column\_width=True)

    st.title("Data Comparison Framework")

    # Create sidebar for source and target selection

    with st.sidebar:

        st.header("Configuration")

        # Source selection

        st.subheader("Source Configuration")

        source\_type = st.selectbox(

            "Select Source Type",

            ["CSV file", "DAT file", "SQL Server", "Stored Procs",

             "Teradata", "API", "Parquet file", "Zipped Flat files"],

            key="source\_type"

        )

        # Target selection

        st.subheader("Target Configuration")

        target\_type = st.selectbox(

            "Select Target Type",

            ["CSV file", "DAT file", "SQL Server", "Stored Procs",

             "Teradata", "API", "Parquet file", "Zipped Flat files"],

            key="target\_type"

        )

        # Delimiter selection for file types

        if source\_type in ["CSV file", "DAT file", "Zipped Flat files"]:

            source\_delimiter = st.text\_input("Source Delimiter", value=",", key="source\_delimiter")

        if target\_type in ["CSV file", "DAT file", "Zipped Flat files"]:

            target\_delimiter = st.text\_input("Target Delimiter", value=",", key="target\_delimiter")

    # Main panel

    col1, col2 = st.columns(2)

    # Source data input

    with col1:

        st.header("Source Data")

        source\_data = handle\_data\_input("source", source\_type)

    # Target data input

    with col2:

        st.header("Target Data")

        target\_data = handle\_data\_input("target", target\_type)

    # If both source and target data are loaded

    if source\_data is not None and target\_data is not None:

        # Auto-map columns

        column\_mapping = auto\_map\_columns(source\_data.columns, target\_data.columns)

        st.header("Column Mapping")

        # Create initial mapping DataFrame

        mapping\_df = pd.DataFrame({

            'Source Column': list(column\_mapping.keys()),

            'Target Column': list(column\_mapping.values()),

            'Exclude from Comparison': [False] \* len(column\_mapping)

        })

        # Allow manual editing of mappings

        st.write("You can edit the mappings below. Select target columns from the dropdown and check boxes to exclude columns from comparison.")

        # Create a selection widget for each source column

        edited\_mappings = []

        for idx, row in mapping\_df.iterrows():

            col1, col2, col3 = st.columns([2, 2, 1])

            with col1:

                st.text(row['Source Column'])  # Source column (read-only)

            with col2:

                # Dropdown for target column selection

                selected\_target = st.selectbox(

                    f"Map to target column",

                    options=[''] + list(target\_data.columns),

                    index=0 if row['Target Column'] not in target\_data.columns

                          else list(target\_data.columns).index(row['Target Column']) + 1,

                    key=f"target\_col\_{idx}",

                    help="Select the corresponding target column"

                )

            with col3:

                # Checkbox for exclusion

                exclude = st.checkbox(

                    "Exclude",

                    value=row['Exclude from Comparison'],

                    key=f"exclude\_{idx}",

                    help="Check to exclude this column from comparison"

                )

            edited\_mappings.append({

                'Source Column': row['Source Column'],

                'Target Column': selected\_target if selected\_target else None,

                'Exclude from Comparison': exclude

            })

        # Convert edited mappings to DataFrame

        edited\_mapping = pd.DataFrame(edited\_mappings)

        # Create valid mappings dictionary (excluding None/empty values and excluded columns)

        valid\_mappings = {

            row['Source Column']: row['Target Column']

            for \_, row in edited\_mapping.iterrows()

            if row['Target Column'] and not pd.isna(row['Target Column']) and not row['Exclude from Comparison']

        }

        # Show mapping summary

        if valid\_mappings:

            st.success(f"✅ {len(valid\_mappings)} columns mapped successfully")

        else:

            st.warning("⚠️ No valid column mappings. Please map at least one column.")

        # Join columns selection

        st.subheader("Join Columns")

        # Create list of available join columns from valid mappings

        available\_columns = [(f"{src} → {tgt}", src)

                           for src, tgt in valid\_mappings.items()

                           if not edited\_mapping[edited\_mapping['Source Column'] == src]['Exclude from Comparison'].iloc[0]]

        if available\_columns:

            selected\_join\_columns = st.multiselect(

                "Select Join Columns",

                options=[col[0] for col in available\_columns],

                help="Select one or more columns to join the datasets"

            )

        else:

            st.warning("No valid column mappings available for join selection. Please map at least one column.")

            selected\_join\_columns = []

        # Data type mapping

        st.subheader("Data Type Mapping")

        dtype\_mapping = {}

        for col in source\_data.columns:

            # Get current column type

            current\_type = str(source\_data[col].dtype)

            # Map numpy/pandas types to our type mapping

            if 'int' in current\_type:

                base\_type = 'int'

            elif 'float' in current\_type:

                base\_type = 'Float'

            elif 'datetime' in current\_type:

                base\_type = 'datetime'

            elif 'bool' in current\_type:

                base\_type = 'Boolean'

            elif 'object' in current\_type:

                # Try to infer type from data

                sample = source\_data[col].dropna().head(100)

                if len(sample) > 0:

                    if all(isinstance(x, (int, np.integer)) for x in sample):

                        base\_type = 'int'

                    elif all(isinstance(x, (float, np.floating)) for x in sample):

                        base\_type = 'Float'

                    elif all(isinstance(x, bool) for x in sample):

                        base\_type = 'Boolean'

                    elif all(isinstance(x, (date, datetime, pd.Timestamp)) for x in sample):

                        base\_type = 'datetime'

                    else:

                        base\_type = 'varchar'

                else:

                    base\_type = 'varchar'

            else:

                base\_type = 'varchar'

            # Get mapped type from TYPE\_MAPPING

            mapped\_type = TYPE\_MAPPING.get(base\_type, TYPE\_MAPPING['varchar'])

            # Create options list from TYPE\_MAPPING values

            type\_options = sorted(set(TYPE\_MAPPING.values()))

            # Create the selectbox with proper type selection

            dtype\_mapping[col] = st.selectbox(

                f"Type for {col}",

                options=type\_options,

                index=type\_options.index(mapped\_type),

                help=f"Current type: {current\_type}"

            )

        # Validate mappings and show comparison button

        if valid\_mappings:

            # Validate data type compatibility

            is\_valid, error\_message = validate\_mapping(

                source\_data, target\_data,

                valid\_mappings,

                dtype\_mapping

            )

            if not is\_valid:

                st.error(f"❌ Mapping validation failed: {error\_message}")

                st.info("💡 Please review your column mappings and data types.")

            # Compare button

            compare\_button = st.button(

                "Compare",

                type="primary",

                disabled=not (selected\_join\_columns and is\_valid),

                help="Generate comparison reports for the mapped columns"

            )

            if compare\_button:

                try:

                    with st.spinner("Generating comparison reports..."):

                        progress\_text = st.empty()

                        progress\_bar = st.progress(0)

                        # Get the actual join columns and their mappings

                        progress\_text.text("Preparing join columns...")

                        join\_cols = []

                        join\_mappings = {}

                        for col\_pair in selected\_join\_columns:

                            src\_col = col\_pair.split(" → ")[0]

                            tgt\_col = valid\_mappings[src\_col]

                            join\_cols.append(src\_col)

                            join\_mappings[src\_col] = tgt\_col

                        progress\_bar.progress(10)

                        # Apply data type conversions

                        progress\_text.text("Converting data types...")

                        for col, dtype in dtype\_mapping.items():

                            # Skip if column is not in valid mappings

                            if col not in valid\_mappings:

                                continue

                            target\_col = valid\_mappings[col]

                            try:

                                if dtype == 'int64':

                                    source\_data[col] = pd.to\_numeric(source\_data[col], errors='coerce').astype('Int64')

                                    target\_data[target\_col] = pd.to\_numeric(target\_data[target\_col], errors='coerce').astype('Int64')

                                elif dtype == 'float64':

                                    source\_data[col] = pd.to\_numeric(source\_data[col], errors='coerce')

                                    target\_data[target\_col] = pd.to\_numeric(target\_data[target\_col], errors='coerce')

                                elif dtype == 'datetime64[ns]':

                                    source\_data[col] = pd.to\_datetime(source\_data[col], errors='coerce')

                                    target\_data[target\_col] = pd.to\_datetime(target\_data[target\_col], errors='coerce')

                                elif dtype == 'bool':

                                    source\_data[col] = source\_data[col].astype('boolean')

                                    target\_data[target\_col] = target\_data[target\_col].astype('boolean')

                                else:  # string type

                                    source\_data[col] = source\_data[col].astype(str)

                                    target\_data[target\_col] = target\_data[target\_col].astype(str)

                            except Exception as e:

                                st.error(f"❌ Error converting {col} to {dtype}: {str(e)}")

                                logger.error(f"Error converting column {col} to {dtype}: {str(e)}")

                                raise Exception(f"Data type conversion failed for column {col}: {str(e)}")

                        progress\_bar.progress(20)

                        # Generate reports with progress updates

                        progress\_text.text("Generating DataCompy report...")

                        datacompy\_excel, datacompy\_html = generate\_datacompy\_report(

                            source\_data, target\_data, join\_cols, edited\_mapping, join\_mappings

                        )

                        progress\_bar.progress(40)

                        progress\_text.text("Generating Y-Data Profiles...")

                        try:

                            ydata\_source\_report, ydata\_target\_report, ydata\_comparison\_report = generate\_ydata\_profile(

                                source\_data, target\_data, edited\_mapping

                            )

                        except Exception as e:

                            st.warning("⚠️ Y-Data profile generation failed, falling back to basic comparison")

                            logger.warning(f"Y-Data profile generation failed: {str(e)}")

                            # Create basic HTML reports as fallback

                            source\_output = BytesIO()

                            target\_output = BytesIO()

                            comparison\_output = BytesIO()

                            source\_report = f"""

                            <html><head><title>Source Data Profile</title></head>

                            <body><h1>Source Data Profile</h1>{source\_data.describe().to\_html()}</body></html>

                            """

                            target\_report = f"""

                            <html><head><title>Target Data Profile</title></head>

                            <body><h1>Target Data Profile</h1>{target\_data.describe().to\_html()}</body></html>

                            """

                            comparison\_report = f"""

                            <html>

                            <head><title>Data Comparison Report</title></head>

                            <body>

                            <h1>Data Comparison Report</h1>

                            <h2>Source Data Summary</h2>{source\_data.describe().to\_html()}

                            <h2>Target Data Summary</h2>{target\_data.describe().to\_html()}

                            </body></html>

                            """

                            source\_output.write(source\_report.encode('utf-8'))

                            target\_output.write(target\_report.encode('utf-8'))

                            comparison\_output.write(comparison\_report.encode('utf-8'))

                            source\_output.seek(0)

                            target\_output.seek(0)

                            comparison\_output.seek(0)

                            ydata\_source\_report = source\_output

                            ydata\_target\_report = target\_output

                            ydata\_comparison\_report = comparison\_output

                        progress\_bar.progress(60)

                        progress\_text.text("Generating Regression report...")

                        regression\_report = generate\_regression\_report(

                            source\_data, target\_data, edited\_mapping, dtype\_mapping

                        )

                        progress\_bar.progress(80)

                        progress\_text.text("Generating Difference report...")

                        difference\_report = generate\_difference\_report(

                            source\_data, target\_data, join\_cols, edited\_mapping, join\_mappings

                        )

                        progress\_bar.progress(90)

                        progress\_text.text("Creating final reports...")

                        consolidated\_report = create\_consolidated\_report(

                            datacompy\_report,

                            ydata\_source\_report,

                            ydata\_target\_report,

                            ydata\_comparison\_report,

                            regression\_report,

                            difference\_report

                        )

                        progress\_bar.progress(100)

                        progress\_text.text("✅ Comparison completed!")

                        # Display reports in UI

                        st.success("✅ Comparison completed successfully!")

                        # Create tabs for different reports

                        report\_tabs = st.tabs(["Y-Data Profiles", "DataCompy Report", "Downloads"])

                        with report\_tabs[0]:

                            ydata\_subtabs = st.tabs(["Source Profile", "Target Profile", "Comparison"])

                            with ydata\_subtabs[0]:

                                st.components.v1.html(ydata\_source\_report.getvalue().decode(), height=800, scrolling=True)

                            with ydata\_subtabs[1]:

                                st.components.v1.html(ydata\_target\_report.getvalue().decode(), height=800, scrolling=True)

                            with ydata\_subtabs[2]:

                                st.components.v1.html(ydata\_comparison\_report.getvalue().decode(), height=800, scrolling=True)

                        with report\_tabs[1]:

                            datacompy\_subtabs = st.tabs(["Excel View", "HTML View"])

                            with datacompy\_subtabs[0]:

                                # Show Excel report in tables

                                excel\_data = pd.read\_excel(datacompy\_excel, sheet\_name=None)

                                for sheet\_name, df in excel\_data.items():

                                    st.write(f"### {sheet\_name}")

                                    st.dataframe(df, use\_container\_width=True)

                                    st.markdown("---")

                            with datacompy\_subtabs[1]:

                                # Show HTML report

                                datacompy\_excel, datacompy\_html = generate\_datacompy\_report(

                                    source\_data, target\_data, join\_cols, edited\_mapping, join\_mappings

                                )

                                st.components.v1.html(datacompy\_html.getvalue().decode(), height=800, scrolling=True)

                        with report\_tabs[2]:

                            # Provide download links in columns

                            col1, col2 = st.columns(2)

                            with col1:

                                st.download\_button(

                                label="📥 Download Consolidated Report",

                                data=create\_consolidated\_report(

                                    datacompy\_excel,

                                    datacompy\_html,

                                    ydata\_source\_report,

                                    ydata\_target\_report,

                                    ydata\_comparison\_report,

                                    regression\_report,

                                    difference\_report

                                ),

                                file\_name=f"consolidated\_report\_{datetime.utcnow().strftime('%Y%m%d\_%H%M%S')}.zip",

                                mime="application/zip",

                                help="Download all reports in a single ZIP file"

                            )

                            with col2:

                                st.download\_button(

                                label="📊 Download Individual Reports",

                                data=create\_individual\_reports\_zip(

                                    datacompy\_excel,

                                    datacompy\_html,

                                    ydata\_source\_report,

                                    ydata\_target\_report,

                                    ydata\_comparison\_report,

                                    regression\_report,

                                    difference\_report

                                ),

                                file\_name=f"individual\_reports\_{datetime.utcnow().strftime('%Y%m%d\_%H%M%S')}.zip",

                                mime="application/zip",

                                help="Download reports as separate files in a ZIP"

                            )

                except Exception as e:

                    # Clear progress indicators

                    if 'progress\_text' in locals():

                        progress\_text.empty()

                    if 'progress\_bar' in locals():

                        progress\_bar.empty()

                    # Show detailed error message

                    error\_msg = str(e)

                    if "Failed to read file" in error\_msg:

                        st.error("❌ Error reading file. Please check the file format and delimiter.")

                        st.info("💡 Try selecting a different delimiter or check if the file is properly formatted.")

                    elif "SQL Server" in error\_msg:

                        st.error("❌ Database connection error. Please check your connection details.")

                        st.info("💡 Verify your server address, credentials, and ensure the database is accessible.")

                    elif "data type conversion" in error\_msg:

                        st.error("❌ Data type conversion error. Please review your data type mappings.")

                        st.info("💡 Some columns may contain values incompatible with the selected data types.")

                    else:

                        st.error(f"❌ Error during comparison: {error\_msg}")

                    # Log the full error

                    logger.error(f"Comparison error: {error\_msg}", exc\_info=True)

def handle\_data\_input(prefix: str, data\_type: str) -> Optional[pd.DataFrame]:

    """Handle different types of data input based on the selected type."""

    try:

        if data\_type in ["CSV file", "DAT file", "Parquet file"]:

            uploaded\_file = st.file\_uploader(

                f"Upload {data\_type}",

                type=["csv", "dat", "parquet"],

                key=f"{prefix}\_file"

            )

            if uploaded\_file:

                if data\_type == "Parquet file":

                    return read\_parquet(uploaded\_file)

                else:

                    delimiter = st.session\_state.get(f"{prefix}\_delimiter", ",")

                    return read\_csv\_dat(uploaded\_file, delimiter)

        elif data\_type in ["SQL Server", "Teradata", "Stored Procs"]:

            with st.expander(f"{data\_type} Connection Details"):

                server = st.text\_input("Server", key=f"{prefix}\_server")

                database = st.text\_input("Database", key=f"{prefix}\_database")

                auth\_method = st.radio(

                    "Authentication Method",

                    ["Windows Authentication", "SQL Server Authentication"],

                    key=f"{prefix}\_auth\_method",

                    help="Choose Windows Authentication to use your Windows credentials"

                )

                username = None

                password = None

                if auth\_method == "SQL Server Authentication":

                    username = st.text\_input("Username", key=f"{prefix}\_username")

                    password = st.text\_input("Password", type="password", key=f"{prefix}\_password")

                if data\_type == "Stored Procs":

                    proc\_name = st.text\_input("Stored Procedure Name", key=f"{prefix}\_proc")

                    if server and database and proc\_name and (auth\_method == "Windows Authentication" or (username and password)):

                        return read\_stored\_proc(server, database, username, password, proc\_name)

                else:

                    query = st.text\_area("SQL Query", key=f"{prefix}\_query")

                    if server and database and query and (auth\_method == "Windows Authentication" or (username and password)):

                        if data\_type == "SQL Server":

                            return read\_sql(server, database, username, password, query)

                        else:

                            return read\_teradata(server, database, username, password, query)

        elif data\_type == "API":

            with st.expander("API Details"):

                api\_url = st.text\_input("API URL", key=f"{prefix}\_api\_url")

                method = st.selectbox("Method", ["GET", "POST"], key=f"{prefix}\_api\_method")

                headers = st.text\_area("Headers (JSON)", key=f"{prefix}\_api\_headers")

                body = st.text\_area("Body (JSON)", key=f"{prefix}\_api\_body")

                if api\_url:

                    return read\_api(api\_url, method, headers, body)

        elif data\_type == "Zipped Flat files":

            uploaded\_zip = st.file\_uploader(

                "Upload ZIP file",

                type=["zip"],

                key=f"{prefix}\_zip"

            )

            if uploaded\_zip:

                delimiter = st.session\_state.get(f"{prefix}\_delimiter", ",")

                return read\_zipped\_files(uploaded\_zip, delimiter)

    except Exception as e:

        st.error(f"Error loading {prefix} data: {str(e)}")

        logger.error(f"Data loading error for {prefix}: {str(e)}")

        return None

if \_\_name\_\_ == "\_\_main\_\_":

    main()