import pandas as pd

import numpy as np

import datacompy

from ydata\_profiling import ProfileReport

import xlsxwriter

from datetime import datetime

import os

import zipfile

from io import BytesIO

import logging

from typing import Dict, List, Optional, Tuple

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

def generate\_datacompy\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                            join\_columns: List[str], mapping\_df: pd.DataFrame,

                            join\_mappings: Dict[str, str]) -> BytesIO:

    """

    Generate a DataCompy comparison report.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        join\_columns: List of source columns to join on

        mapping\_df: DataFrame containing column mapping information

        join\_mappings: Dictionary mapping source join columns to target join columns

    Returns:

        BytesIO object containing the report

    """

    try:

        # Get excluded columns

        excluded\_columns = mapping\_df[mapping\_df['Exclude from Comparison']]['Source Column'].tolist()

        # Create mapping dictionary from mapping\_df

        column\_mapping = dict(zip(

            mapping\_df['Source Column'],

            mapping\_df['Target Column']

        ))

        # Filter out unmapped and excluded columns

        valid\_columns = {

            src: tgt for src, tgt in column\_mapping.items()

            if tgt and not pd.isna(tgt) and src not in excluded\_columns

        }

        # Prepare DataFrames for comparison

        source\_cols = list(valid\_columns.keys())

        target\_cols = [valid\_columns[src] for src in source\_cols]

        source\_compare = source\_df[source\_cols].copy()

        target\_compare = target\_df[target\_cols].copy()

        # Rename target columns to match source columns for comparison

        target\_compare.columns = source\_cols

        # Get the target join column names

        target\_join\_columns = [join\_mappings[src] for src in join\_columns]

        # Rename join columns in target DataFrame to match source

        join\_column\_mapping = dict(zip(target\_join\_columns, join\_columns))

        target\_compare.rename(columns=join\_column\_mapping, inplace=True)

        # Ensure datetime columns are properly converted

        for col in source\_compare.columns:

            if source\_compare[col].dtype == 'datetime64[ns]':

                target\_compare[col] = pd.to\_datetime(target\_compare[col], errors='coerce')

            elif target\_compare[col].dtype == 'datetime64[ns]':

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

        # Ensure all columns are strings to prevent type comparison issues

        source\_compare.columns = source\_compare.columns.astype(str)

        target\_compare.columns = target\_compare.columns.astype(str)

        # Create comparison object

        try:

            comparison = datacompy.Compare(

                df1=source\_compare,

                df2=target\_compare,

                join\_columns=join\_columns,

                df1\_name='Source',

                df2\_name='Target',

                on\_index=False

            )

        except Exception as e:

            logger.error(f"Error creating comparison object: {str(e)}")

            # Try alternative comparison with converted data types

            source\_compare = source\_compare.astype(str)

            target\_compare = target\_compare.astype(str)

            comparison = datacompy.Compare(

                df1=source\_compare,

                df2=target\_compare,

                join\_columns=join\_columns,

                df1\_name='Source',

                df2\_name='Target',

                on\_index=False

            )

        # Generate report

        output = BytesIO()

        with pd.ExcelWriter(output, engine='xlsxwriter') as writer:

            # Write summary

            # Create summary DataFrame

            summary\_data = {

                'Metric': [

                    'Rows in Source',

                    'Rows in Target',

                    'Rows in Common',

                    'Rows Only in Source',

                    'Rows Only in Target',

                    'Columns Match',

                    'All Row Values Match'

                ],

                'Value': [

                    len(source\_compare),

                    len(target\_compare),

                    comparison.intersect\_rows,

                    len(comparison.df1\_unq\_rows) if hasattr(comparison, 'df1\_unq\_rows') else 0,

                    len(comparison.df2\_unq\_rows) if hasattr(comparison, 'df2\_unq\_rows') else 0,

                    comparison.all\_columns\_match(),

                    comparison.matches()

                ]

            }

            pd.DataFrame(summary\_data).to\_excel(writer, sheet\_name='Summary', index=False)

            # Write column stats

            if hasattr(comparison, 'column\_stats') and comparison.column\_stats is not None:

                if isinstance(comparison.column\_stats, pd.DataFrame):

                    comparison.column\_stats.to\_excel(writer, sheet\_name='Column Stats', index=True)

                else:

                    pd.DataFrame(comparison.column\_stats).to\_excel(writer, sheet\_name='Column Stats', index=True)

            # Write sample mismatches

            try:

                # Get mismatched rows

                df1\_unq = comparison.df1\_unq\_rows if hasattr(comparison, 'df1\_unq\_rows') else pd.DataFrame()

                df2\_unq = comparison.df2\_unq\_rows if hasattr(comparison, 'df2\_unq\_rows') else pd.DataFrame()

                # Create sample mismatches DataFrame

                if not df1\_unq.empty or not df2\_unq.empty:

                    mismatches = pd.concat([

                        df1\_unq.assign(Source='Source Only').head(5),

                        df2\_unq.assign(Source='Target Only').head(5)

                    ], ignore\_index=True)

                    mismatches.to\_excel(writer, sheet\_name='Sample Mismatches', index=True)

                else:

                    pd.DataFrame({'Message': ['No mismatches found']}).to\_excel(

                        writer, sheet\_name='Sample Mismatches', index=False)

            except Exception as e:

                logger.warning(f"Error generating sample mismatches: {str(e)}")

                pd.DataFrame({'Message': [f'Error generating sample mismatches: {str(e)}']}).to\_excel(

                    writer, sheet\_name='Sample Mismatches', index=False)

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error generating DataCompy report: {str(e)}")

        raise Exception(f"Failed to generate DataCompy report: {str(e)}")

def generate\_ydata\_profile(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                         mapping\_df: pd.DataFrame) -> Tuple[BytesIO, BytesIO, BytesIO]:

    """

    Generate Y-Data Profiling reports including individual profiles and comparison.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        mapping\_df: DataFrame containing column mapping information

    Returns:

        Tuple of (source\_profile, target\_profile, comparison\_profile) as BytesIO objects

    """

    try:

        # Create mapping dictionary from mapping\_df

        column\_mapping = dict(zip(

            mapping\_df['Source Column'],

            mapping\_df['Target Column']

        ))

        # Filter out unmapped and excluded columns

        excluded\_columns = mapping\_df[mapping\_df['Exclude from Comparison']]['Source Column'].tolist()

        valid\_columns = {

            src: tgt for src, tgt in column\_mapping.items()

            if tgt and not pd.isna(tgt) and src not in excluded\_columns

        }

        # Prepare DataFrames for comparison

        source\_cols = list(valid\_columns.keys())

        target\_cols = [valid\_columns[src] for src in source\_cols]

        source\_compare = source\_df[source\_cols].copy()

        target\_compare = target\_df[target\_cols].copy()

        # Rename target columns to match source columns for comparison

        target\_compare.columns = source\_cols

        # Convert problematic data types to string

        for col in source\_compare.columns:

            if source\_compare[col].dtype.name not in ['int64', 'float64', 'bool', 'datetime64[ns]', 'object']:

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

            if target\_compare[col].dtype.name not in ['int64', 'float64', 'bool', 'datetime64[ns]', 'object']:

                target\_compare[col] = target\_compare[col].astype(str)

        # Handle null values

        source\_compare = source\_compare.fillna(pd.NA)

        target\_compare = target\_compare.fillna(pd.NA)

        try:

            # Generate profiles with minimal configuration

            source\_profile = ProfileReport(

                source\_compare,

                title="Source Data Profile",

                minimal=True,

                explorative=True

            )

            target\_profile = ProfileReport(

                target\_compare,

                title="Target Data Profile",

                minimal=True,

                explorative=True

            )

            # Generate comparison profile

            comparison\_profile = source\_profile.compare(target\_profile)

            # Save to BytesIO objects

            source\_output = BytesIO()

            target\_output = BytesIO()

            comparison\_output = BytesIO()

            source\_profile.to\_file(source\_output)

            target\_profile.to\_file(target\_output)

            comparison\_profile.to\_file(comparison\_output)

            source\_output.seek(0)

            target\_output.seek(0)

            comparison\_output.seek(0)

            return source\_output, target\_output, comparison\_output

        except Exception as e:

            logger.error(f"Error in profile generation: {str(e)}")

            # Fallback to basic HTML reports

            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\_compare.describe().to\_html()}</body></html>

            """

            target\_report = f"""

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

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

            <h2>Target Data Summary</h2>{target\_compare.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)

            return source\_output, target\_output, comparison\_output

    except Exception as e:

        logger.error(f"Error generating Y-Data profile: {str(e)}")

        raise Exception(f"Failed to generate Y-Data profile: {str(e)}")

def generate\_regression\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                            mapping\_df: pd.DataFrame, dtype\_mapping: Dict[str, str]) -> BytesIO:

    """

    Generate Excel-based regression report with multiple tabs.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        mapping\_df: DataFrame containing column mapping information

        dtype\_mapping: Dictionary mapping columns to their desired data types

    Returns:

        BytesIO object containing the report

    """

    try:

        output = BytesIO()

        with pd.ExcelWriter(output, engine='xlsxwriter') as writer:

            workbook = writer.book

            # Create formats for PASS/FAIL cells

            pass\_format = workbook.add\_format({'bg\_color': '#90EE90'})  # Light green

            fail\_format = workbook.add\_format({'bg\_color': '#FFB6C6'})  # Light pink

            # Generate AggregationCheck tab

            \_generate\_aggregation\_check(source\_df, target\_df, mapping\_df, writer,

                                     pass\_format, fail\_format)

            # Generate CountCheck tab

            \_generate\_count\_check(source\_df, target\_df, writer, pass\_format, fail\_format)

            # Generate DistinctCheck tab

            \_generate\_distinct\_check(source\_df, target\_df, mapping\_df, writer,

                                  pass\_format, fail\_format)

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error generating regression report: {str(e)}")

        raise Exception(f"Failed to generate regression report: {str(e)}")

def \_generate\_aggregation\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                              mapping\_df: pd.DataFrame, writer: pd.ExcelWriter,

                              pass\_format: xlsxwriter.format.Format,

                              fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the AggregationCheck tab in the regression report."""

    # Get numeric columns

    numeric\_cols = source\_df.select\_dtypes(include=[np.number]).columns

    results = []

    for col in numeric\_cols:

        if col in mapping\_df['Source Column'].values:

            target\_col = mapping\_df[mapping\_df['Source Column'] == col]['Target Column'].iloc[0]

            source\_sum = source\_df[col].sum()

            target\_sum = target\_df[target\_col].sum()

            match = np.isclose(source\_sum, target\_sum, rtol=1e-05)

            results.append({

                'Source Column': col,

                'Target Column': target\_col,

                'Source Sum': source\_sum,

                'Target Sum': target\_sum,

                'Result': 'PASS' if match else 'FAIL'

            })

    # Create DataFrame and write to Excel

    agg\_df = pd.DataFrame(results)

    agg\_df.to\_excel(writer, sheet\_name='AggregationCheck', index=False)

    # Apply conditional formatting

    worksheet = writer.sheets['AggregationCheck']

    result\_col = agg\_df.columns.get\_loc('Result')

    for row in range(len(agg\_df)):

        if agg\_df.iloc[row]['Result'] == 'PASS':

            worksheet.write(row + 1, result\_col, 'PASS', pass\_format)

        else:

            worksheet.write(row + 1, result\_col, 'FAIL', fail\_format)

def \_generate\_count\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                         writer: pd.ExcelWriter,

                         pass\_format: xlsxwriter.format.Format,

                         fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the CountCheck tab in the regression report."""

    count\_data = {

        'Source File Name': source\_df.name if hasattr(source\_df, 'name') else 'Source',

        'Target File Name': target\_df.name if hasattr(target\_df, 'name') else 'Target',

        'Source Count': len(source\_df),

        'Target Count': len(target\_df),

        'Result': 'PASS' if len(source\_df) == len(target\_df) else 'FAIL'

    }

    count\_df = pd.DataFrame([count\_data])

    count\_df.to\_excel(writer, sheet\_name='CountCheck', index=False)

    # Apply conditional formatting

    worksheet = writer.sheets['CountCheck']

    result\_col = count\_df.columns.get\_loc('Result')

    if count\_data['Result'] == 'PASS':

        worksheet.write(1, result\_col, 'PASS', pass\_format)

    else:

        worksheet.write(1, result\_col, 'FAIL', fail\_format)

def \_generate\_distinct\_check(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                           mapping\_df: pd.DataFrame, writer: pd.ExcelWriter,

                           pass\_format: xlsxwriter.format.Format,

                           fail\_format: xlsxwriter.format.Format) -> None:

    """Generate the DistinctCheck tab in the regression report."""

    # Get non-numeric columns

    non\_numeric\_cols = source\_df.select\_dtypes(exclude=[np.number]).columns

    results = []

    for col in non\_numeric\_cols:

        if col in mapping\_df['Source Column'].values:

            target\_col = mapping\_df[mapping\_df['Source Column'] == col]['Target Column'].iloc[0]

            source\_distinct = set(source\_df[col].dropna().unique())

            target\_distinct = set(target\_df[target\_col].dropna().unique())

            source\_count = len(source\_distinct)

            target\_count = len(target\_distinct)

            count\_match = source\_count == target\_count

            values\_match = source\_distinct == target\_distinct

            results.append({

                'Source Column': col,

                'Target Column': target\_col,

                'Source Distinct Count': source\_count,

                'Target Distinct Count': target\_count,

                'Count Match': 'PASS' if count\_match else 'FAIL',

                'Values Match': 'PASS' if values\_match else 'FAIL',

                'Source Distinct Values': ', '.join(map(str, sorted(source\_distinct))),

                'Target Distinct Values': ', '.join(map(str, sorted(target\_distinct)))

            })

    # Create DataFrame and write to Excel

    distinct\_df = pd.DataFrame(results)

    distinct\_df.to\_excel(writer, sheet\_name='DistinctCheck', index=False)

    # Apply conditional formatting

    worksheet = writer.sheets['DistinctCheck']

    count\_match\_col = distinct\_df.columns.get\_loc('Count Match')

    values\_match\_col = distinct\_df.columns.get\_loc('Values Match')

    for row in range(len(distinct\_df)):

        if distinct\_df.iloc[row]['Count Match'] == 'PASS':

            worksheet.write(row + 1, count\_match\_col, 'PASS', pass\_format)

        else:

            worksheet.write(row + 1, count\_match\_col, 'FAIL', fail\_format)

        if distinct\_df.iloc[row]['Values Match'] == 'PASS':

            worksheet.write(row + 1, values\_match\_col, 'PASS', pass\_format)

        else:

            worksheet.write(row + 1, values\_match\_col, 'FAIL', fail\_format)

def generate\_difference\_report(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                             join\_columns: List[str], mapping\_df: pd.DataFrame,

                             join\_mappings: Dict[str, str]) -> BytesIO:

    """

    Generate side-by-side difference report.

    Args:

        source\_df: Source DataFrame

        target\_df: Target DataFrame

        join\_columns: List of source columns to join on

        mapping\_df: DataFrame containing column mapping information

        join\_mappings: Dictionary mapping source join columns to target join columns

    Returns:

        BytesIO object containing the report

    """

    try:

        output = BytesIO()

        # Create mapping dictionary from mapping\_df

        column\_mapping = dict(zip(

            mapping\_df['Source Column'],

            mapping\_df['Target Column']

        ))

        # Filter out unmapped and excluded columns

        excluded\_columns = mapping\_df[mapping\_df['Exclude from Comparison']]['Source Column'].tolist()

        valid\_columns = {

            src: tgt for src, tgt in column\_mapping.items()

            if tgt and not pd.isna(tgt) and src not in excluded\_columns

        }

        # Prepare DataFrames for comparison

        source\_cols = list(valid\_columns.keys())

        target\_cols = [valid\_columns[src] for src in source\_cols]

        source\_compare = source\_df[source\_cols].copy()

        target\_compare = target\_df[target\_cols].copy()

        # Get the target join column names

        target\_join\_columns = [join\_mappings[src] for src in join\_columns]

        # Merge datasets

        merged = pd.merge(

            source\_compare, target\_compare,

            left\_on=join\_columns,

            right\_on=target\_join\_columns,

            how='outer',

            suffixes=('\_source', '\_target'),

            indicator=True

        )

        # Find differences

        differences = merged[merged['\_merge'] != 'both']

        with pd.ExcelWriter(output, engine='xlsxwriter') as writer:

            if len(differences) > 0:

                differences.to\_excel(writer, sheet\_name='Differences', index=False)

            else:

                pd.DataFrame({'Message': ['No differences found']}).to\_excel(

                    writer, sheet\_name='Differences', index=False)

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error generating difference report: {str(e)}")

        raise Exception(f"Failed to generate difference report: {str(e)}")

def create\_individual\_reports\_zip(datacompy\_report: BytesIO,

                                ydata\_source\_report: BytesIO,

                                ydata\_target\_report: BytesIO,

                                ydata\_comparison\_report: BytesIO,

                                regression\_report: BytesIO,

                                difference\_report: BytesIO) -> BytesIO:

    """

    Create a ZIP file containing individual reports in separate folders.

    Args:

        datacompy\_report: DataCompy report as BytesIO

        ydata\_source\_report: Source data profile as BytesIO

        ydata\_target\_report: Target data profile as BytesIO

        ydata\_comparison\_report: Profile comparison report as BytesIO

        regression\_report: Regression report as BytesIO

        difference\_report: Difference report as BytesIO

    Returns:

        BytesIO object containing the ZIP file with individual reports

    """

    try:

        timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')

        output = BytesIO()

        with zipfile.ZipFile(output, 'w', zipfile.ZIP\_DEFLATED) as zf:

            # DataCompy Report

            zf.writestr(f'datacompy/datacompy\_report\_{timestamp}.xlsx',

                       datacompy\_report.getvalue())

            # Y-Data Profiles

            zf.writestr(f'ydata\_profile/source\_profile\_{timestamp}.html',

                       ydata\_source\_report.getvalue())

            zf.writestr(f'ydata\_profile/target\_profile\_{timestamp}.html',

                       ydata\_target\_report.getvalue())

            zf.writestr(f'ydata\_profile/comparison\_profile\_{timestamp}.html',

                       ydata\_comparison\_report.getvalue())

            # Regression Report

            zf.writestr(f'regression/regression\_report\_{timestamp}.xlsx',

                       regression\_report.getvalue())

            # Difference Report

            zf.writestr(f'differences/difference\_report\_{timestamp}.xlsx',

                       difference\_report.getvalue())

            # Add a README file

            readme\_content = """

Data Comparison Reports

1. datacompy/ - Contains detailed comparison report

2. ydata\_profile/ - Contains comprehensive data profiling

3. regression/ - Contains aggregation, count, and distinct value checks

4. differences/ - Contains side-by-side difference report

Generated on: {timestamp}

            """.format(timestamp=timestamp)

            zf.writestr('README.txt', readme\_content.strip())

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error creating individual reports zip: {str(e)}")

        raise Exception(f"Failed to create individual reports zip: {str(e)}")

def create\_consolidated\_report(datacompy\_report: BytesIO,

                             ydata\_source\_report: BytesIO,

                             ydata\_target\_report: BytesIO,

                             ydata\_comparison\_report: BytesIO,

                             regression\_report: BytesIO,

                             difference\_report: BytesIO) -> BytesIO:

    """

    Combine all reports into a single consolidated report.

    Args:

        datacompy\_report: DataCompy report as BytesIO

        ydata\_source\_report: Source data profile as BytesIO

        ydata\_target\_report: Target data profile as BytesIO

        ydata\_comparison\_report: Profile comparison report as BytesIO

        regression\_report: Regression report as BytesIO

        difference\_report: Difference report as BytesIO

    Returns:

        BytesIO object containing the consolidated ZIP file

    """

    try:

        timestamp = datetime.now().strftime('%Y%m%d\_%H%M%S')

        output = BytesIO()

        with zipfile.ZipFile(output, 'w', zipfile.ZIP\_DEFLATED) as zf:

            # Add all reports to a single consolidated file

            zf.writestr(f'reports/datacompy\_report\_{timestamp}.xlsx',

                       datacompy\_report.getvalue())

            zf.writestr(f'reports/ydata\_source\_profile\_{timestamp}.html',

                       ydata\_source\_report.getvalue())

            zf.writestr(f'reports/ydata\_target\_profile\_{timestamp}.html',

                       ydata\_target\_report.getvalue())

            zf.writestr(f'reports/ydata\_comparison\_profile\_{timestamp}.html',

                       ydata\_comparison\_report.getvalue())

            zf.writestr(f'reports/regression\_report\_{timestamp}.xlsx',

                       regression\_report.getvalue())

            zf.writestr(f'reports/difference\_report\_{timestamp}.xlsx',

                       difference\_report.getvalue())

            # Add a summary README

            readme\_content = """

Consolidated Data Comparison Report

This ZIP file contains the following reports:

1. datacompy\_report - Detailed comparison of datasets

2. ydata\_profile - Comprehensive data profiling

3. regression\_report - Aggregation, count, and distinct value checks

4. difference\_report - Side-by-side differences

Generated on: {timestamp}

            """.format(timestamp=timestamp)

            zf.writestr('README.txt', readme\_content.strip())

        output.seek(0)

        return output

    except Exception as e:

        logger.error(f"Error creating consolidated report: {str(e)}")

        raise Exception(f"Failed to create consolidated report: {str(e)}")

app.py

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\_report = 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\_report,

                            regression\_report,

                            difference\_report

                        )

                        progress\_bar.progress(100)

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

                        # Provide download links

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

                        col1, col2 = st.columns(2)

                        with col1:

                            st.download\_button(

                                label="📥 Download Consolidated Report",

                                data=create\_consolidated\_report(

                                    datacompy\_report,

                                    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\_report,

                                    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()