Report Generator.py

"""Module for generating various comparison reports"""

import pandas as pd

import numpy as np

from typing import Dict, List, Tuple, Optional

import os

from datetime import datetime

from ydata\_profiling import ProfileReport

import datacompy

from openpyxl.styles import PatternFill, Font

from openpyxl.utils import get\_column\_letter

from config import EXCEL\_STYLES, REPORTS\_DIR

class ReportGenerator:

    @staticmethod

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

                                 column\_mapping: Dict[str, str], output\_path: str) -> None:

        """Generate regression report with multiple checks"""

        writer = pd.ExcelWriter(output\_path, engine='openpyxl')

        # Generate Aggregation Check

        ReportGenerator.\_generate\_aggregation\_check(source\_df, target\_df, column\_mapping, writer)

        # Generate Count Check

        ReportGenerator.\_generate\_count\_check(source\_df, target\_df, writer)

        # Generate Distinct Check

        ReportGenerator.\_generate\_distinct\_check(source\_df, target\_df, column\_mapping, writer)

        writer.save()

    @staticmethod

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

                                  column\_mapping: Dict[str, str], writer: pd.ExcelWriter) -> None:

        """Generate aggregation check for numeric columns"""

        agg\_data = []

        for source\_col, target\_col in column\_mapping.items():

            # Check if columns are numeric

            if pd.api.types.is\_numeric\_dtype(source\_df[source\_col]) and pd.api.types.is\_numeric\_dtype(target\_df[target\_col]):

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

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

                result = 'PASS' if np.isclose(source\_sum, target\_sum, rtol=1e-05) else 'FAIL'

                agg\_data.append({

                    'Source Column': source\_col,

                    'Target Column': target\_col,

                    'Source Sum': source\_sum,

                    'Target Sum': target\_sum,

                    'Result': result

                })

        if agg\_data:

            df\_agg = pd.DataFrame(agg\_data)

            df\_agg.to\_excel(writer, sheet\_name='AggregationCheck', index=False)

            # Apply conditional formatting

            worksheet = writer.sheets['AggregationCheck']

            for idx, row in enumerate(df\_agg['Result'], start=2):  # start=2 to skip header

                cell = worksheet[f'E{idx}']

                style = EXCEL\_STYLES['PASS'] if row == 'PASS' else EXCEL\_STYLES['FAIL']

                cell.fill = PatternFill(start\_color=style['fill']['fgColor'], end\_color=style['fill']['fgColor'], fill\_type='solid')

                cell.font = Font(color=style['font']['color'])

    @staticmethod

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

                            writer: pd.ExcelWriter) -> None:

        """Generate count check report"""

        count\_data = {

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

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

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

        }

        df\_count = pd.DataFrame([count\_data])

        df\_count.to\_excel(writer, sheet\_name='CountCheck', index=False)

        # Apply conditional formatting

        worksheet = writer.sheets['CountCheck']

        cell = worksheet['C2']  # Result column, first data row

        style = EXCEL\_STYLES['PASS'] if count\_data['Result'] == 'PASS' else EXCEL\_STYLES['FAIL']

        cell.fill = PatternFill(start\_color=style['fill']['fgColor'], end\_color=style['fill']['fgColor'], fill\_type='solid')

        cell.font = Font(color=style['font']['color'])

    @staticmethod

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

                               column\_mapping: Dict[str, str], writer: pd.ExcelWriter) -> None:

        """Generate distinct value check for non-numeric columns"""

        distinct\_data = []

        for source\_col, target\_col in column\_mapping.items():

            # Check if columns are non-numeric

            if not pd.api.types.is\_numeric\_dtype(source\_df[source\_col]):

                source\_distinct = source\_df[source\_col].nunique()

                target\_distinct = target\_df[target\_col].nunique()

                source\_values = set(source\_df[source\_col].dropna().unique())

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

                values\_match = source\_values == target\_values

                count\_match = source\_distinct == target\_distinct

                result = 'PASS' if values\_match and count\_match else 'FAIL'

                distinct\_data.append({

                    'Source Column': source\_col,

                    'Target Column': target\_col,

                    'Source Distinct Count': source\_distinct,

                    'Target Distinct Count': target\_distinct,

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

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

                    'Result': result

                })

        if distinct\_data:

            df\_distinct = pd.DataFrame(distinct\_data)

            df\_distinct.to\_excel(writer, sheet\_name='DistinctCheck', index=False)

            # Apply conditional formatting

            worksheet = writer.sheets['DistinctCheck']

            for idx, row in enumerate(df\_distinct['Result'], start=2):  # start=2 to skip header

                cell = worksheet[f'G{idx}']

                style = EXCEL\_STYLES['PASS'] if row == 'PASS' else EXCEL\_STYLES['FAIL']

                cell.fill = PatternFill(start\_color=style['fill']['fgColor'], end\_color=style['fill']['fgColor'], fill\_type='solid')

                cell.font = Font(color=style['font']['color'])

    @staticmethod

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

                                   column\_mapping: Dict[str, str], join\_keys: List[Tuple[str, str]],

                                   output\_path: str) -> None:

        """Generate side by side comparison report"""

        # Prepare DataFrames for comparison

        source\_cols = list(column\_mapping.keys())

        target\_cols = [column\_mapping[col] for col in source\_cols]

        # Merge DataFrames based on join keys

        if join\_keys:

            source\_join\_cols = [key[0] for key in join\_keys]

            target\_join\_cols = [key[1] for key in join\_keys]

            merged\_df = pd.merge(

                source\_df[source\_cols],

                target\_df[target\_cols],

                left\_on=source\_join\_cols,

                right\_on=target\_join\_cols,

                how='outer',

                indicator=True

            )

            # Find differences

            diff\_mask = merged\_df['\_merge'] != 'both'

            diff\_df = merged\_df[diff\_mask].copy()

            if not diff\_df.empty:

                # Rename columns to show source/target

                diff\_df.columns = [f'Source\_{col}' if col in source\_cols else f'Target\_{col}'

                                 for col in diff\_df.columns]

                # Save to Excel

                diff\_df.to\_excel(output\_path, index=False)

                return True

            else:

                # Create empty DataFrame with message

                pd.DataFrame({'Message': ['No differences found']}).to\_excel(output\_path, index=False)

                return False

        else:

            # If no join keys, compare row by row

            comparison\_df = pd.DataFrame()

            for s\_col, t\_col in column\_mapping.items():

                comparison\_df[f'Source\_{s\_col}'] = source\_df[s\_col]

                comparison\_df[f'Target\_{t\_col}'] = target\_df[t\_col]

            comparison\_df.to\_excel(output\_path, index=False)

            return True

    @staticmethod

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

                               timestamp: str) -> Dict[str, str]:

        """Generate Y-Data profiling reports"""

        reports = {}

        # Create reports directory

        os.makedirs(REPORTS\_DIR, exist\_ok=True)

        # Generate source profile

        source\_profile = ProfileReport(

            source\_df,

            title="Source Data Profile Report",

            minimal=False,

            correlations={"cramers": True},

            vars={"num": {"low\_categorical\_threshold": 0}}

        )

        source\_path = os.path.join(REPORTS\_DIR, f"SourceProfile\_{timestamp}.html")

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

        reports['source'] = source\_path

        # Generate target profile

        target\_profile = ProfileReport(

            target\_df,

            title="Target Data Profile Report",

            minimal=False,

            correlations={"cramers": True},

            vars={"num": {"low\_categorical\_threshold": 0}}

        )

        target\_path = os.path.join(REPORTS\_DIR, f"TargetProfile\_{timestamp}.html")

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

        reports['target'] = target\_path

        # Generate comparison profile

        comparison\_df = pd.DataFrame()

        for col in source\_df.columns:

            comparison\_df[f'Source\_{col}'] = source\_df[col]

        for col in target\_df.columns:

            comparison\_df[f'Target\_{col}'] = target\_df[col]

        comparison\_profile = ProfileReport(

            comparison\_df,

            title="Source vs Target Comparison Profile",

            minimal=False,

            correlations={"cramers": True},

            vars={"num": {"low\_categorical\_threshold": 0}}

        )

        comparison\_path = os.path.join(REPORTS\_DIR, f"ComparisonProfile\_{timestamp}.html")

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

        reports['comparison'] = comparison\_path

        return reports

    @staticmethod

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

                                column\_mapping: Dict[str, str], join\_keys: List[Tuple[str, str]],

                                output\_path: str) -> None:

        """Generate DataCompy comparison report"""

        # Prepare DataFrames with mapped columns

        source\_cols = list(column\_mapping.keys())

        target\_cols = [column\_mapping[col] for col in source\_cols]

        if join\_keys:

            source\_join\_cols = [key[0] for key in join\_keys]

            target\_join\_cols = [key[1] for key in join\_keys]

        else:

            # If no join keys, use index

            source\_df = source\_df.copy()

            target\_df = target\_df.copy()

            source\_df['\_index'] = range(len(source\_df))

            target\_df['\_index'] = range(len(target\_df))

            source\_join\_cols = ['\_index']

            target\_join\_cols = ['\_index']

        # Prepare DataFrames with mapped columns

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

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

        # Rename target columns to match source for proper comparison

        target\_compare\_df.columns = source\_cols

        # Create DataCompy comparison

        if join\_keys:

            comparison = datacompy.Compare(

                source\_compare\_df,

                target\_compare\_df,

                join\_columns=[key[0] for key in join\_keys],  # Use source column names

                df1\_name='Source',

                df2\_name='Target'

            )

        else:

            # If no join keys specified, use index-based comparison

            source\_compare\_df['\_index'] = range(len(source\_compare\_df))

            target\_compare\_df['\_index'] = range(len(target\_compare\_df))

            comparison = datacompy.Compare(

                source\_compare\_df,

                target\_compare\_df,

                join\_columns=['\_index'],

                df1\_name='Source',

                df2\_name='Target'

            )

        # Generate HTML report

        with open(output\_path, 'w') as f:

            f.write(f"""

            <html>

            <head>

                <title>DataCompy Comparison Report</title>

                <style>

                    body {{ font-family: Arial, sans-serif; margin: 20px; }}

                    .report {{ max-width: 1200px; margin: 0 auto; }}

                    .section {{ margin: 20px 0; padding: 20px; border: 1px solid #ddd; border-radius: 5px; }}

                    .match {{ color: green; }}

                    .mismatch {{ color: red; }}

                    table {{ border-collapse: collapse; width: 100%; }}

                    th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}

                    th {{ background-color: #f5f5f5; }}

                </style>

            </head>

            <body>

                <div class="report">

                    <h1>DataCompy Comparison Report</h1>

                    <div class="section">

                        <h2>Summary</h2>

                        <pre>{comparison.report()}</pre>

                    </div>

                    <div class="section">

                        <h2>Detailed Statistics</h2>

                        <h3>Matches</h3>

                        <div class="match">

                            <p>Number of rows match: {comparison.count\_matching\_rows()}</p>

                            <p>Number of columns match: {len(set(source\_cols).intersection(target\_cols))}</p>

                        </div>

                        <h3>Mismatches</h3>

                        <div class="mismatch">

                            <p>Rows only in Source: {len(comparison.df1\_unq\_rows)}</p>

                            <p>Rows only in Target: {len(comparison.df2\_unq\_rows)}</p>

                            <p>Source-only columns: {len(set(source\_cols) - set(target\_cols))}</p>

                            <p>Target-only columns: {len(set(target\_cols) - set(source\_cols))}</p>

                        </div>

                    </div>

                    <div class="section">

                        <h2>Column Statistics</h2>

                        {comparison.column\_stats.to\_html() if hasattr(comparison, 'column\_stats') else '<p>No column statistics available.</p>'}

                    </div>

                </div>

            </body>

            </html>

            """)

Data\_loader.py

"""Module for loading data from various sources"""

import pandas as pd

import dask.dataframe as dd

import pyodbc

import teradatasql

import requests

import zipfile

import io

from typing import Optional, Dict, Any

from config import CHUNK\_SIZE

class DataLoader:

    @staticmethod

    def read\_csv\_in\_chunks(file\_obj: Any, delimiter: str = ',', \*\*kwargs) -> pd.DataFrame:

        """Read large CSV files in chunks"""

        try:

            # Try different encodings

            encodings = ['utf-8', 'latin1', 'iso-8859-1', 'cp1252']

            for encoding in encodings:

                try:

                    # For uploaded files

                    if hasattr(file\_obj, 'read'):

                        # Reset file pointer

                        file\_obj.seek(0)

                        # Try reading with pandas first for smaller files

                        try:

                            return pd.read\_csv(file\_obj, delimiter=delimiter, encoding=encoding, \*\*kwargs)

                        except Exception:

                            # If that fails, use Dask for larger files

                            file\_obj.seek(0)

                            ddf = dd.read\_csv(file\_obj, blocksize=CHUNK\_SIZE, delimiter=delimiter, encoding=encoding, \*\*kwargs)

                            return ddf.compute()

                    # For file paths

                    else:

                        try:

                            return pd.read\_csv(file\_obj, delimiter=delimiter, encoding=encoding, \*\*kwargs)

                        except Exception:

                            ddf = dd.read\_csv(file\_obj, blocksize=CHUNK\_SIZE, delimiter=delimiter, encoding=encoding, \*\*kwargs)

                            return ddf.compute()

                except UnicodeDecodeError:

                    continue

                except Exception as e:

                    if encoding == encodings[-1]:  # If this was the last encoding to try

                        raise e

                    continue

            raise Exception("Unable to read file with any supported encoding")

        except Exception as e:

            raise Exception(f"Error reading CSV file: {str(e)}")

    @staticmethod

    def read\_dat\_file(file\_obj: Any, delimiter: str = '|', \*\*kwargs) -> pd.DataFrame:

        """Read DAT files"""

        try:

            return DataLoader.read\_csv\_in\_chunks(file\_obj, delimiter=delimiter, \*\*kwargs)

        except Exception as e:

            raise Exception(f"Error reading DAT file: {str(e)}")

    @staticmethod

    def read\_parquet(file\_path: str) -> pd.DataFrame:

        """Read Parquet files"""

        try:

            return dd.read\_parquet(file\_path).compute()

        except Exception as e:

            raise Exception(f"Error reading Parquet file: {str(e)}")

    @staticmethod

    def read\_sql\_server(conn\_params: Dict[str, Any], query: str) -> pd.DataFrame:

        """Read data from SQL Server"""

        try:

            conn\_str = (

                f"DRIVER={{ODBC Driver 17 for SQL Server}};"

                f"SERVER={conn\_params['server']};"

                f"DATABASE={conn\_params['database']};"

            )

            if conn\_params.get('use\_windows\_auth', True):

                conn\_str += "Trusted\_Connection=yes;"

            else:

                conn\_str += f"UID={conn\_params['username']};PWD={conn\_params['password']};"

            with pyodbc.connect(conn\_str) as conn:

                return pd.read\_sql(query, conn)

        except Exception as e:

            raise Exception(f"Error connecting to SQL Server: {str(e)}")

    @staticmethod

    def read\_teradata(conn\_params: Dict[str, Any], query: str) -> pd.DataFrame:

        """Read data from Teradata"""

        try:

            with teradatasql.connect(

                host=conn\_params['host'],

                user=conn\_params['username'],

                password=conn\_params['password']

            ) as conn:

                return pd.read\_sql(query, conn)

        except Exception as e:

            raise Exception(f"Error connecting to Teradata: {str(e)}")

    @staticmethod

    def read\_stored\_proc(conn\_params: Dict[str, Any], proc\_name: str, params: Optional[Dict] = None) -> pd.DataFrame:

        """Execute stored procedure and return results"""

        try:

            # Build the stored procedure execution string

            param\_str = ""

            if params:

                param\_str = ",".join([f"@{k}=?" for k in params.keys()])

            exec\_str = f"EXEC {proc\_name} {param\_str}"

            conn\_str = (

                f"DRIVER={{ODBC Driver 17 for SQL Server}};"

                f"SERVER={conn\_params['server']};"

                f"DATABASE={conn\_params['database']};"

            )

            if conn\_params.get('use\_windows\_auth', True):

                conn\_str += "Trusted\_Connection=yes;"

            else:

                conn\_str += f"UID={conn\_params['username']};PWD={conn\_params['password']};"

            with pyodbc.connect(conn\_str) as conn:

                if params:

                    return pd.read\_sql(exec\_str, conn, params=list(params.values()))

                else:

                    return pd.read\_sql(exec\_str, conn)

        except Exception as e:

            raise Exception(f"Error executing stored procedure: {str(e)}")

    @staticmethod

    def read\_api(url: str, method: str = 'GET', headers: Optional[Dict] = None, params: Optional[Dict] = None) -> pd.DataFrame:

        """Read data from API endpoint"""

        try:

            response = requests.request(method, url, headers=headers, params=params)

            response.raise\_for\_status()

            return pd.DataFrame(response.json())

        except Exception as e:

            raise Exception(f"Error fetching API data: {str(e)}")

    @staticmethod

    def read\_zipped\_flat\_files(file\_obj: io.BytesIO, delimiter: str = '|') -> pd.DataFrame:

        """Read flat files inside zipped folder"""

        try:

            with zipfile.ZipFile(file\_obj) as z:

                dfs = []

                for filename in z.namelist():

                    if filename.endswith('.csv') or filename.endswith('.dat') or filename.endswith('.txt'):

                        with z.open(filename) as f:

                            df = pd.read\_csv(f, delimiter=delimiter)

                            dfs.append(df)

                if dfs:

                    return pd.concat(dfs, ignore\_index=True)

                else:

                    raise Exception("No suitable flat files found in zip archive")

        except Exception as e:

            raise Exception(f"Error reading zipped flat files: {str(e)}")

app.py

"""Main Streamlit application for data comparison framework"""

import streamlit as st

import pandas as pd

import os

from datetime import datetime

from typing import Dict, List, Tuple, Optional

import io

import tempfile

from config import SOURCE\_TYPES, TYPE\_MAPPING, DEFAULT\_DELIMITERS, REPORTS\_DIR

from data\_loader import DataLoader

from report\_generator import ReportGenerator

def init\_session\_state():

    """Initialize session state variables"""

    if 'source\_df' not in st.session\_state:

        st.session\_state.source\_df = None

    if 'target\_df' not in st.session\_state:

        st.session\_state.target\_df = None

    if 'column\_mapping' not in st.session\_state:

        st.session\_state.column\_mapping = {}

    if 'join\_columns' not in st.session\_state:

        st.session\_state.join\_columns = []

    if 'excluded\_columns' not in st.session\_state:

        st.session\_state.excluded\_columns = []

def show\_data\_source\_config(prefix: str) -> None:

    """Show configuration options for data source/target"""

    st.subheader(f"{'Source' if prefix == 'source' else 'Target'} Configuration")

    source\_type = st.selectbox(

        f"Select {'Source' if prefix == 'source' else 'Target'} Type",

        options=SOURCE\_TYPES,

        key=f"{prefix}\_type"

    )

    try:

        if source\_type in ['CSV File', 'DAT File', 'Parquet File', 'Zipped Flat Files']:

            file = st.file\_uploader(

                f"Upload {source\_type}",

                type=['csv', 'dat', 'parquet', 'zip'],

                key=f"{prefix}\_file"

            )

            if file:

                if source\_type in ['CSV File', 'DAT File', 'Zipped Flat Files']:

                    delimiter = st.text\_input(

                        "Delimiter",

                        value=DEFAULT\_DELIMITERS.get(source\_type, ','),

                        key=f"{prefix}\_delimiter"

                    )

                    if st.button(f"Load {source\_type}", key=f"load\_{prefix}"):

                        with st.spinner("Loading data..."):

                            try:

                                # Reset file pointer

                                file.seek(0)

                                if source\_type == 'Zipped Flat Files':

                                    df = DataLoader.read\_zipped\_flat\_files(io.BytesIO(file.read()), delimiter)

                                elif source\_type == 'CSV File':

                                    df = DataLoader.read\_csv\_in\_chunks(file, delimiter=delimiter)

                                else:  # DAT File

                                    df = DataLoader.read\_dat\_file(file, delimiter=delimiter)

                                if df is not None and len(df) > 0:

                                    st.session\_state[f"{prefix}\_df"] = df

                                    st.success(f"✅ Successfully loaded {len(df)} rows and {len(df.columns)} columns")

                                else:

                                    st.error("❌ No data was loaded from the file")

                            except Exception as e:

                                st.error(f"❌ Error loading file: {str(e)}")

                elif source\_type == 'Parquet File':

                    if st.button(f"Load {source\_type}", key=f"load\_{prefix}"):

                        with st.spinner("Loading data..."):

                            with tempfile.NamedTemporaryFile(delete=False) as tmp:

                                tmp.write(file.read())

                                df = DataLoader.read\_parquet(tmp.name)

                            os.unlink(tmp.name)

                if 'df' in locals():

                    st.session\_state[f"{prefix}\_df"] = df

                    st.success(f"✅ Successfully loaded {len(df)} rows and {len(df.columns)} columns")

        elif source\_type in ['SQL Server', 'Teradata']:

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

                if source\_type == 'SQL Server':

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

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

                    use\_windows\_auth = st.checkbox("Use Windows Authentication", key=f"{prefix}\_windows\_auth")

                    if not use\_windows\_auth:

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

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

                else:  # Teradata

                    host = st.text\_input("Host", key=f"{prefix}\_host")

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

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

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

                if st.button("Execute Query", key=f"execute\_{prefix}"):

                    with st.spinner("Executing query..."):

                        try:

                            if source\_type == 'SQL Server':

                                conn\_params = {

                                    'server': server,

                                    'database': database,

                                    'use\_windows\_auth': use\_windows\_auth

                                }

                                if not use\_windows\_auth:

                                    conn\_params.update({'username': username, 'password': password})

                                df = DataLoader.read\_sql\_server(conn\_params, query)

                            else:  # Teradata

                                conn\_params = {

                                    'host': host,

                                    'username': username,

                                    'password': password

                                }

                                df = DataLoader.read\_teradata(conn\_params, query)

                            st.session\_state[f"{prefix}\_df"] = df

                            st.success(f"✅ Successfully loaded {len(df)} rows and {len(df.columns)} columns")

                        except Exception as e:

                            st.error(f"❌ Error: {str(e)}")

        elif source\_type == 'Stored Procedure':

            with st.expander("Stored Procedure Details"):

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

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

                use\_windows\_auth = st.checkbox("Use Windows Authentication", key=f"{prefix}\_sp\_windows\_auth")

                if not use\_windows\_auth:

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

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

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

                params = st.text\_area(

                    "Parameters (as Python dict, e.g., {'param1': 'value1'})",

                    key=f"{prefix}\_sp\_params"

                )

                if st.button("Execute Stored Procedure", key=f"execute\_sp\_{prefix}"):

                    with st.spinner("Executing stored procedure..."):

                        try:

                            conn\_params = {

                                'server': server,

                                'database': database,

                                'use\_windows\_auth': use\_windows\_auth

                            }

                            if not use\_windows\_auth:

                                conn\_params.update({'username': username, 'password': password})

                            df = DataLoader.read\_stored\_proc(

                                conn\_params,

                                proc\_name,

                                eval(params) if params else None

                            )

                            st.session\_state[f"{prefix}\_df"] = df

                            st.success(f"✅ Successfully loaded {len(df)} rows and {len(df.columns)} columns")

                        except Exception as e:

                            st.error(f"❌ Error: {str(e)}")

        elif source\_type == 'API':

            with st.expander("API Details"):

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

                method = st.selectbox(

                    "HTTP Method",

                    options=['GET', 'POST', 'PUT', 'DELETE'],

                    key=f"{prefix}\_api\_method"

                )

                headers = st.text\_area(

                    "Headers (as Python dict)",

                    key=f"{prefix}\_api\_headers"

                )

                params = st.text\_area(

                    "Parameters (as Python dict)",

                    key=f"{prefix}\_api\_params"

                )

                if st.button("Fetch API Data", key=f"fetch\_api\_{prefix}"):

                    with st.spinner("Fetching API data..."):

                        try:

                            df = DataLoader.read\_api(

                                url,

                                method=method,

                                headers=eval(headers) if headers else None,

                                params=eval(params) if params else None

                            )

                            st.session\_state[f"{prefix}\_df"] = df

                            st.success(f"✅ Successfully loaded {len(df)} rows and {len(df.columns)} columns")

                        except Exception as e:

                            st.error(f"❌ Error: {str(e)}")

    except Exception as e:

        st.error(f"❌ Error: {str(e)}")

def show\_column\_mapping\_interface():

    """Show interface for column mapping"""

    st.subheader("Column Mapping Configuration")

    if not isinstance(st.session\_state.source\_df, pd.DataFrame):

        st.error("❌ Source data not loaded")

        return

    if not isinstance(st.session\_state.target\_df, pd.DataFrame):

        st.error("❌ Target data not loaded")

        return

    # Create tabs for auto and manual mapping

    auto\_tab, manual\_tab = st.tabs(["Automatic Mapping", "Manual Mapping"])

    with auto\_tab:

        if st.button("🔄 Auto-Map Columns", use\_container\_width=True):

            # Simple auto-mapping based on column names

            mapping = {}

            source\_cols = {col.lower().strip(): col for col in st.session\_state.source\_df.columns}

            target\_cols = {col.lower().strip(): col for col in st.session\_state.target\_df.columns}

            for s\_norm, s\_orig in source\_cols.items():

                if s\_norm in target\_cols:

                    mapping[s\_orig] = target\_cols[s\_norm]

            st.session\_state.column\_mapping = mapping

            if mapping:

                st.success(f"✅ Successfully mapped {len(mapping)} columns!")

            else:

                st.warning("⚠️ No automatic matches found. Please map columns manually.")

    with manual\_tab:

        st.markdown("### Manual Column Mapping")

        # Show mapping interface for each source column

        for source\_col in st.session\_state.source\_df.columns:

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

            with col1:

                st.markdown(f"\*\*Source:\*\* {source\_col}")

                st.caption(f"Sample: {str(st.session\_state.source\_df[source\_col].head(2).tolist())}")

            with col2:

                # Get current mapping

                current\_mapping = st.session\_state.column\_mapping.get(source\_col, '')

                # Create dropdown with target columns

                target\_options = [''] + list(st.session\_state.target\_df.columns)

                selected\_target = st.selectbox(

                    "Map to target column",

                    options=target\_options,

                    index=target\_options.index(current\_mapping) if current\_mapping in target\_options else 0,

                    key=f"mapping\_{source\_col}"

                )

                if selected\_target:

                    if selected\_target != current\_mapping:

                        st.session\_state.column\_mapping[source\_col] = selected\_target

                        st.caption(f"Target sample: {str(st.session\_state.target\_df[selected\_target].head(2).tolist())}")

                elif source\_col in st.session\_state.column\_mapping:

                    del st.session\_state.column\_mapping[source\_col]

            with col3:

                excluded = st.checkbox(

                    "Exclude",

                    key=f"exclude\_{source\_col}",

                    value=source\_col in st.session\_state.excluded\_columns

                )

                if excluded and source\_col not in st.session\_state.excluded\_columns:

                    st.session\_state.excluded\_columns.append(source\_col)

                elif not excluded and source\_col in st.session\_state.excluded\_columns:

                    st.session\_state.excluded\_columns.remove(source\_col)

    # Show current mapping summary

    if st.session\_state.column\_mapping:

        st.success(f"✅ Currently mapped: {len(st.session\_state.column\_mapping)} columns")

        with st.expander("View Current Mappings"):

            for source\_col, target\_col in st.session\_state.column\_mapping.items():

                st.write(f"{source\_col} → {target\_col}")

def show\_join\_column\_selection():

    """Show interface for selecting join columns"""

    st.subheader("Join Column Selection")

    if not st.session\_state.column\_mapping:

        st.warning("⚠️ Please map columns first")

        return

    # Get mapped columns

    mapped\_columns = list(st.session\_state.column\_mapping.items())

    # Allow multiple join column selection

    selected\_joins = []

    for i in range(len(mapped\_columns)):

        col1, col2 = st.columns(2)

        with col1:

            source\_col = st.selectbox(

                f"Source Join Column {i+1}",

                options=[''] + [col[0] for col in mapped\_columns],

                key=f"source\_join\_{i}"

            )

        with col2:

            if source\_col:

                target\_col = st.session\_state.column\_mapping.get(source\_col, '')

                st.write(f"Target Join Column: {target\_col}")

                if source\_col and target\_col:

                    selected\_joins.append((source\_col, target\_col))

    st.session\_state.join\_columns = selected\_joins

    if selected\_joins:

        st.success(f"✅ Selected {len(selected\_joins)} join column pairs")

def perform\_comparison():

    """Perform the comparison and generate reports"""

    st.subheader("Comparison Results")

    if not st.session\_state.column\_mapping:

        st.error("❌ Please map columns first")

        return

    if not st.session\_state.join\_columns:

        st.error("❌ Please select at least one join column")

        return

    try:

        # Create reports directory

        os.makedirs(REPORTS\_DIR, exist\_ok=True)

        # Generate timestamp for report files

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

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

            # Generate DataCompy report

            datacompy\_path = os.path.join(REPORTS\_DIR, f"DataCompyReport\_{timestamp}.html")

            ReportGenerator.generate\_datacompy\_report(

                st.session\_state.source\_df,

                st.session\_state.target\_df,

                st.session\_state.column\_mapping,

                st.session\_state.join\_columns,

                datacompy\_path

            )

            # Generate Y-Data Profile reports

            profile\_reports = ReportGenerator.generate\_profile\_reports(

                st.session\_state.source\_df,

                st.session\_state.target\_df,

                timestamp

            )

            # Generate regression report

            regression\_path = os.path.join(REPORTS\_DIR, f"RegressionReport\_{timestamp}.xlsx")

            ReportGenerator.generate\_regression\_report(

                st.session\_state.source\_df,

                st.session\_state.target\_df,

                st.session\_state.column\_mapping,

                regression\_path

            )

            # Generate side by side comparison

            diff\_path = os.path.join(REPORTS\_DIR, f"DifferenceReport\_{timestamp}.xlsx")

            has\_differences = ReportGenerator.generate\_side\_by\_side\_report(

                st.session\_state.source\_df,

                st.session\_state.target\_df,

                st.session\_state.column\_mapping,

                st.session\_state.join\_columns,

                diff\_path

            )

            # Show download links

            st.success("✅ Reports generated successfully!")

            st.markdown("### Download Reports")

            col1, col2 = st.columns(2)

            with col1:

                with open(datacompy\_path, 'rb') as f:

                    st.download\_button(

                        "📊 Download DataCompy Report",

                        f,

                        file\_name=f"DataCompyReport\_{timestamp}.html",

                        mime="text/html"

                    )

                with open(profile\_reports['source'], 'rb') as f:

                    st.download\_button(

                        "📈 Download Source Profile",

                        f,

                        file\_name=f"SourceProfile\_{timestamp}.html",

                        mime="text/html"

                    )

            with col2:

                with open(regression\_path, 'rb') as f:

                    st.download\_button(

                        "📑 Download Regression Report",

                        f,

                        file\_name=f"RegressionReport\_{timestamp}.xlsx",

                        mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"

                    )

                with open(profile\_reports['target'], 'rb') as f:

                    st.download\_button(

                        "📉 Download Target Profile",

                        f,

                        file\_name=f"TargetProfile\_{timestamp}.html",

                        mime="text/html"

                    )

            # Show comparison profile and difference report

            st.markdown("### Additional Reports")

            col3, col4 = st.columns(2)

            with col3:

                with open(profile\_reports['comparison'], 'rb') as f:

                    st.download\_button(

                        "🔄 Download Comparison Profile",

                        f,

                        file\_name=f"ComparisonProfile\_{timestamp}.html",

                        mime="text/html"

                    )

            with col4:

                if has\_differences:

                    with open(diff\_path, 'rb') as f:

                        st.download\_button(

                            "❗ Download Difference Report",

                            f,

                            file\_name=f"DifferenceReport\_{timestamp}.xlsx",

                            mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"

                        )

                else:

                    st.info("✅ No differences found between source and target")

    except Exception as e:

        st.error(f"❌ Error generating reports: {str(e)}")

def main():

    """Main application"""

    st.title("Data Comparison Framework")

    # Initialize session state

    init\_session\_state()

    # Create tabs for different sections

    source\_tab, target\_tab, mapping\_tab, compare\_tab = st.tabs([

        "Source Configuration",

        "Target Configuration",

        "Column Mapping",

        "Compare & Reports"

    ])

    with source\_tab:

        show\_data\_source\_config("source")

    with target\_tab:

        show\_data\_source\_config("target")

    with mapping\_tab:

        show\_column\_mapping\_interface()

        show\_join\_column\_selection()

    with compare\_tab:

        if st.button("🔍 Compare Data", use\_container\_width=True):

            perform\_comparison()

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

    main()