"""Module for generating comparison reports"""

import pandas as pd

import numpy as np

from typing import Dict, List, Optional

import os

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

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

                           output\_path: str) -> None:

    """Generate pandas-based comparison report"""

    try:

        # Get source and target columns

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

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

        # Select relevant columns and copy DataFrames

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

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

        # Rename target columns to match source

        target\_compare\_df.columns = source\_cols

        # Convert all columns to string and handle nulls

        for col in source\_cols:

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

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

        # Perform merge

        if isinstance(join\_keys, (list, tuple)) and join\_keys:

            # Use provided join keys if they exist in source columns

            valid\_keys = [k for k in join\_keys if k in source\_cols]

            if valid\_keys:

                merged\_df = pd.merge(

                    source\_compare\_df,

                    target\_compare\_df,

                    on=valid\_keys,

                    how='outer',

                    indicator=True

                )

            else:

                # Fall back to index-based comparison if no valid join keys

                merged\_df = pd.merge(

                    source\_compare\_df,

                    target\_compare\_df,

                    left\_index=True,

                    right\_index=True,

                    how='outer',

                    indicator=True

                )

        else:

            # Use index-based comparison if no join keys provided

            merged\_df = pd.merge(

                source\_compare\_df,

                target\_compare\_df,

                left\_index=True,

                right\_index=True,

                how='outer',

                indicator=True

            )

        # Calculate statistics

        stats = {

            'total\_rows\_source': len(source\_compare\_df),

            'total\_rows\_target': len(target\_compare\_df),

            'matching\_rows': len(merged\_df[merged\_df['\_merge'] == 'both']),

            'source\_only\_rows': len(merged\_df[merged\_df['\_merge'] == 'left\_only']),

            'target\_only\_rows': len(merged\_df[merged\_df['\_merge'] == 'right\_only'])

        }

        # Calculate match percentage

        total\_rows = len(source\_compare\_df)

        match\_percentage = (stats['matching\_rows'] / total\_rows \* 100) if total\_rows > 0 else 0

        # Generate summary

        summary = f"""

        Comparison Summary:

        - Total Rows in Source: {stats['total\_rows\_source']}

        - Total Rows in Target: {stats['total\_rows\_target']}

        - Matching Rows: {stats['matching\_rows']}

        - Source Only Rows: {stats['source\_only\_rows']}

        - Target Only Rows: {stats['target\_only\_rows']}

        - Match Percentage: {match\_percentage:.2f}%

        """

        # Generate column statistics

        col\_stats = []

        for col in source\_cols:

            col\_stats.append({

                'Column': col,

                'Source\_Count': len(source\_compare\_df[col].dropna()),

                'Source\_Unique': source\_compare\_df[col].nunique(),

                'Target\_Count': len(target\_compare\_df[col].dropna()),

                'Target\_Unique': target\_compare\_df[col].nunique(),

                'Match\_Status': 'Match' if source\_compare\_df[col].equals(target\_compare\_df[col]) else 'Mismatch'

            })

        col\_stats\_df = pd.DataFrame(col\_stats)

        col\_stats\_html = col\_stats\_df.to\_html(classes='table table-striped', index=False)

        # Generate HTML for mismatched records

        source\_only = merged\_df[merged\_df['\_merge'] == 'left\_only']

        target\_only = merged\_df[merged\_df['\_merge'] == 'right\_only']

        source\_only\_html = source\_only.to\_html(classes='table table-striped', index=False) if not source\_only.empty else '<p>No records unique to source</p>'

        target\_only\_html = target\_only.to\_html(classes='table table-striped', index=False) if not target\_only.empty else '<p>No records unique to target</p>'

        # Write HTML report

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

            f.write(f"""

            <!DOCTYPE html>

            <html lang="en">

            <head>

                <title>Data 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>Data Comparison Report</h1>

                    <div class="section">

                        <h2>Summary</h2>

                        <pre>{summary}</pre>

                    </div>

                    <div class="section">

                        <h2>Column Statistics</h2>

                        {col\_stats\_html}

                    </div>

                    <div class="section">

                        <h2>Mismatched Records</h2>

                        <h3>Records only in Source:</h3>

                        {source\_only\_html}

                        <h3>Records only in Target:</h3>

                        {target\_only\_html}

                    </div>

                </div>

            </body>

            </html>

            """)

    except Exception as e:

        # Generate error report

        error\_html = f"""

        <!DOCTYPE html>

        <html lang="en">

        <head><title>Comparison Error Report</title></head>

        <body>

            <h1>Error Generating Comparison Report</h1>

            <p>An error occurred: {str(e)}</p>

            <p>Please check that:</p>

            <ul>

                <li>All required columns exist in both datasets</li>

                <li>Join keys are valid column names</li>

                <li>Data types are compatible for comparison</li>

            </ul>

        </body>

        </html>

        """

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

            f.write(error\_html)

        raise Exception(f"Error generating comparison report: {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\_new import generate\_comparison\_report

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

    # Initialize join\_columns in session state if not present

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

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

    # Get valid columns that exist in both dataframes

    valid\_columns = [

        col for col in st.session\_state.column\_mapping.keys()

        if col in st.session\_state.source\_df.columns and

        st.session\_state.column\_mapping[col] in st.session\_state.target\_df.columns

    ]

    # Show multiselect for join columns

    selected\_columns = st.multiselect(

        "Select columns to use as join keys",

        options=valid\_columns,

        default=st.session\_state.join\_columns,

        help="Select one or more columns to use as join keys for comparison"

    )

    # Update session state

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

    if selected\_columns:

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

        # Show selected mappings

        st.write("Selected Join Keys:")

        for col in selected\_columns:

            st.write(f"- Source: {col} → Target: {st.session\_state.column\_mapping[col]}")

    else:

        st.info("ℹ️ No join columns selected. Index-based comparison will be used.")

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 comparison report using new generator

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

            # Use join columns directly as a list of strings

            join\_keys = list(st.session\_state.join\_columns) if st.session\_state.join\_columns else []

            # Import and use the new report generator

            from report\_generator\_new import generate\_comparison\_report

            generate\_comparison\_report(

                st.session\_state.source\_df,

                st.session\_state.target\_df,

                st.session\_state.column\_mapping,

                join\_keys,

                comparison\_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(comparison\_path, 'rb') as f:

                    st.download\_button(

                        "📊 Download Comparison Report",

                        f,

                        file\_name=f"ComparisonReport\_{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()