\_\_init\_\_.py

from .data\_reader import (

    read\_csv\_dat,

    read\_sql,

    read\_stored\_proc,

    read\_teradata,

    read\_api,

    read\_parquet,

    read\_zipped\_files

)

from .mapping\_utils import (

    auto\_map\_columns,

    validate\_join\_columns,

    apply\_column\_mapping,

    get\_excluded\_columns,

    validate\_mapping

)

from .report\_generator import (

    generate\_datacompy\_report,

    generate\_ydata\_profile,

    generate\_regression\_report,

    generate\_difference\_report,

    create\_consolidated\_report,

    create\_individual\_reports\_zip

)

from .utils import (

    setup\_logger,

    check\_file\_size,

    format\_file\_size,

    get\_file\_info,

    create\_download\_link,

    show\_progress\_bar,

    generate\_timestamp,

    validate\_dataframe,

    handle\_error,

    cleanup\_temp\_files,

    Timer

)

\_\_all\_\_ = [

    # Data Reader

    'read\_csv\_dat',

    'read\_sql',

    'read\_stored\_proc',

    'read\_teradata',

    'read\_api',

    'read\_parquet',

    'read\_zipped\_files',

    # Mapping Utils

    'auto\_map\_columns',

    'validate\_join\_columns',

    'apply\_column\_mapping',

    'get\_excluded\_columns',

    'validate\_mapping',

    # Report Generator

    'generate\_datacompy\_report',

    'generate\_ydata\_profile',

    'generate\_regression\_report',

    'generate\_difference\_report',

    'create\_consolidated\_report',

    'create\_individual\_reports\_zip',

    # Utils

    'setup\_logger',

    'check\_file\_size',

    'format\_file\_size',

    'get\_file\_info',

    'create\_download\_link',

    'show\_progress\_bar',

    'generate\_timestamp',

    'validate\_dataframe',

    'handle\_error',

    'cleanup\_temp\_files',

    'Timer'

]

Report\_Generator :

import pandas as pd

import numpy as np

import datacompy

import xlsxwriter

from datetime import datetime

import os

import zipfile

from io import BytesIO

import logging

from typing import Dict, List, Optional, Tuple

from .profile\_utils import (

    generate\_distribution\_plot,

    generate\_frequency\_plot,

    generate\_comparison\_plot,

    generate\_comparison\_rows,

    calculate\_column\_stats

)

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

def generate\_mismatch\_html(comparison):

    """Generate HTML for mismatches with proper error handling."""

    try:

        mismatches\_html = []

        # Handle source-only rows

        if hasattr(comparison, 'df1\_unq\_rows') and not comparison.df1\_unq\_rows.empty:

            mismatches\_html.append("<h3>Rows Only in Source</h3>")

            mismatches\_html.append(comparison.df1\_unq\_rows.head(5).to\_html())

        # Handle target-only rows

        if hasattr(comparison, 'df2\_unq\_rows') and not comparison.df2\_unq\_rows.empty:

            mismatches\_html.append("<h3>Rows Only in Target</h3>")

            mismatches\_html.append(comparison.df2\_unq\_rows.head(5).to\_html())

        if mismatches\_html:

            return "\n".join(mismatches\_html)

        return "<p>No mismatches found</p>"

    except Exception as e:

        logger.warning(f"Error generating mismatch HTML: {str(e)}")

        return f"<p>Error generating mismatch details: {str(e)}</p>"

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]) -> Tuple[BytesIO, BytesIO]:

    """Generate a DataCompy comparison report."""

    try:

        # Get excluded columns

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

        # Create mapping dictionary

        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

        }

        # Get all columns that need to be processed

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

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

        # Create unique column names for comparison

        source\_rename = {}

        target\_rename = {}

        final\_join\_columns = []

        # First, handle join columns to ensure they're unique

        for src\_col in join\_columns:

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

            unique\_join\_name = f"join\_{src\_col}"

            source\_rename[src\_col] = unique\_join\_name

            target\_rename[tgt\_col] = unique\_join\_name

            final\_join\_columns.append(unique\_join\_name)

        # Then handle comparison columns

        for src\_col, tgt\_col in valid\_columns.items():

            if src\_col not in join\_columns:  # Skip if it's already a join column

                unique\_col\_name = f"compare\_{src\_col}"

                source\_rename[src\_col] = unique\_col\_name

                target\_rename[tgt\_col] = unique\_col\_name

        # Create copies of DataFrames with only needed columns

        source\_compare = source\_df[list(set(source\_cols + join\_columns))].copy()

        target\_compare = target\_df[list(set(target\_cols + [join\_mappings[src] for src in join\_columns]))].copy()

        # Rename columns to ensure uniqueness

        source\_compare.rename(columns=source\_rename, inplace=True)

        target\_compare.rename(columns=target\_rename, inplace=True)

        # Update join columns for the comparison

        join\_columns = final\_join\_columns

        logger.info(f"Source columns after renaming: {source\_compare.columns.tolist()}")

        logger.info(f"Target columns after renaming: {target\_compare.columns.tolist()}")

        logger.info(f"Join columns for comparison: {join\_columns}")

        # Create comparison object with proper configuration

        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.warning(f"Initial comparison failed: {str(e)}, trying with string conversion")

            # Convert all columns to string type for comparison

            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 Excel report

        excel\_output = BytesIO()

        with pd.ExcelWriter(excel\_output, engine='xlsxwriter') as writer:

            # Write summary

            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'):

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

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

                elif isinstance(comparison.column\_stats, list):

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

                else:

                    pd.DataFrame({'Message': ['No column statistics available']}).to\_excel(

                        writer, sheet\_name='Column Stats', index=False)

            # Write sample mismatches with proper error handling

            try:

                mismatches\_df = pd.DataFrame()

                # Get source-only rows

                if hasattr(comparison, 'df1\_unq\_rows') and not comparison.df1\_unq\_rows.empty:

                    source\_mismatches = comparison.df1\_unq\_rows.head(5).copy()

                    source\_mismatches['Match Type'] = 'Source Only'

                    mismatches\_df = pd.concat([mismatches\_df, source\_mismatches])

                # Get target-only rows

                if hasattr(comparison, 'df2\_unq\_rows') and not comparison.df2\_unq\_rows.empty:

                    target\_mismatches = comparison.df2\_unq\_rows.head(5).copy()

                    target\_mismatches['Match Type'] = 'Target Only'

                    mismatches\_df = pd.concat([mismatches\_df, target\_mismatches])

                if not mismatches\_df.empty:

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

                else:

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

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

        # Generate HTML report

        html\_output = BytesIO()

        html\_report = f"""

        <!DOCTYPE html>

        <html lang="en">

        <head>

            <meta charset="UTF-8">

            <title>DataCompy Comparison Report</title>

            <style>

                body {{

                    font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, 'Helvetica Neue', Arial, sans-serif;

                    margin: 20px;

                    line-height: 1.6;

                    background-color: #f8f9fa;

                }}

                table {{

                    width: 100%;

                    border-collapse: collapse;

                    margin-bottom: 20px;

                    background-color: #ffffff;

                    box-shadow: 0 1px 3px rgba(0,0,0,0.1);

                }}

                th, td {{

                    border: 1px solid #dee2e6;

                    padding: 12px;

                    text-align: left;

                }}

                th {{

                    background-color: #f8f9fa;

                    font-weight: 600;

                }}

                .pass {{

                    color: #28a745;

                    font-weight: 600;

                }}

                .fail {{

                    color: #dc3545;

                    font-weight: 600;

                }}

                .section {{

                    background-color: #ffffff;

                    padding: 25px;

                    margin-bottom: 30px;

                    border-radius: 8px;

                    box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                }}

                h1, h2 {{

                    color: #333;

                    margin-top: 0;

                }}

                h1 {{

                    font-size: 24px;

                    margin-bottom: 30px;

                    padding: 25px;

                    background-color: #ffffff;

                    border-radius: 8px;

                    box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                }}

                h2 {{

                    font-size: 20px;

                    margin-bottom: 15px;

                }}

                p {{

                    margin: 10px 0;

                }}

            </style>

        </head>

        <body>

            <h1>DataCompy Comparison Report</h1>

            <div class="section">

                <h2>Summary</h2>

                <p>Source rows: {len(source\_compare)}</p>

                <p>Target rows: {len(target\_compare)}</p>

                <p>Rows in common: {comparison.intersect\_rows}</p>

                <p>Rows only in source: {len(comparison.df1\_unq\_rows) if hasattr(comparison, 'df1\_unq\_rows') else 0}</p>

                <p>Rows only in target: {len(comparison.df2\_unq\_rows) if hasattr(comparison, 'df2\_unq\_rows') else 0}</p>

                <p>Columns match: <span class="{'pass' if comparison.all\_columns\_match() else 'fail'}">{comparison.all\_columns\_match()}</span></p>

                <p>All rows match: <span class="{'pass' if comparison.matches() else 'fail'}">{comparison.matches()}</span></p>

            </div>

            <div class="section">

                <h2>Column Statistics</h2>

                {

                    pd.DataFrame(comparison.column\_stats).to\_html()

                    if hasattr(comparison, 'column\_stats') and

                       (isinstance(comparison.column\_stats, pd.DataFrame) or

                        isinstance(comparison.column\_stats, list))

                    else '<p>No column statistics available</p>'

                }

            </div>

            <div class="section">

                <h2>Sample Mismatches</h2>

                {generate\_mismatch\_html(comparison)}

            </div>

        </body>

        </html>

        """

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

        html\_output.seek(0)

        excel\_output.seek(0)

        return excel\_output, html\_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 column-wise Y-Data Profiling reports."""

    try:

        # Create mapping dictionary

        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

        }

        # Initialize output buffers

        source\_output = BytesIO()

        target\_output = BytesIO()

        comparison\_output = BytesIO()

        # Start HTML reports with headers and CSS

        html\_header = """

            <!DOCTYPE html>

            <html lang="en">

            <head>

                <meta charset="UTF-8">

                <title>{title}</title>

                <style>

                    body {{

                        font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif;

                        margin: 20px;

                        line-height: 1.6;

                        background-color: #f8f9fa;

                    }}

                    .column-profile {{

                        margin-bottom: 40px;

                        padding: 20px;

                        border: 1px solid #ddd;

                        border-radius: 8px;

                        background-color: #ffffff;

                        box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                    }}

                    .stats-table {{

                        width: 100%;

                        border-collapse: collapse;

                        margin-top: 15px;

                        background-color: #ffffff;

                    }}

                    .stats-table th, .stats-table td {{

                        border: 1px solid #dee2e6;

                        padding: 12px;

                        text-align: left;

                    }}

                    .stats-table th {{

                        background-color: #f8f9fa;

                        font-weight: 600;

                    }}

                    .visualization {{

                        margin-top: 25px;

                        padding: 15px;

                        background-color: #ffffff;

                        border-radius: 6px;

                    }}

                    .chart {{

                        width: 100%;

                        max-width: 800px;

                        margin: 20px auto;

                    }}

                    .header {{

                        background-color: #ffffff;

                        padding: 25px;

                        margin-bottom: 30px;

                        border-radius: 8px;

                        box-shadow: 0 2px 4px rgba(0,0,0,0.1);

                    }}

                    .summary {{

                        margin: 20px 0;

                        display: flex;

                        flex-wrap: wrap;

                        gap: 15px;

                    }}

                    .metric {{

                        display: inline-block;

                        padding: 15px 20px;

                        background: #ffffff;

                        border: 1px solid #dee2e6;

                        border-radius: 6px;

                        box-shadow: 0 1px 3px rgba(0,0,0,0.1);

                    }}

                    .diff {{

                        background-color: #fff3f3;

                        border-color: #ffcdd2;

                    }}

                    .match {{

                        background-color: #f1f8f1;

                        border-color: #c8e6c9;

                    }}

                    h1, h2, h3 {{

                        color: #333;

                        margin-top: 0;

                    }}

                    h1 {{

                        font-size: 24px;

                        margin-bottom: 10px;

                    }}

                    h2 {{

                        font-size: 20px;

                        margin-bottom: 15px;

                    }}

                    h3 {{

                        font-size: 18px;

                        margin-bottom: 10px;

                    }}

                </style>

            </head>

            <body>

            <div class="header">

                <h1>{title}</h1>

                <p>Generated on: {timestamp}</p>

            </div>

        """

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

        source\_html = [html\_header.format(title="Source Data Profile", timestamp=timestamp)]

        target\_html = [html\_header.format(title="Target Data Profile", timestamp=timestamp)]

        comparison\_html = [html\_header.format(title="Column Comparison Profile", timestamp=timestamp)]

        # Process each column pair

        for src\_col, tgt\_col in valid\_columns.items():

            logger.info(f"Processing column pair: {src\_col} -> {tgt\_col}")

            # Get column data

            source\_data = source\_df[src\_col]

            target\_data = target\_df[tgt\_col]

            # Calculate statistics

            source\_stats = calculate\_column\_stats(source\_data)

            target\_stats = calculate\_column\_stats(target\_data)

            # Generate HTML for source column

            source\_html.append(f"""

                <div class="column-profile">

                    <h2>Column: {src\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Type:</strong> {source\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Unique Values:</strong> {source\_stats['Unique Values']}

                        </div>

                        <div class="metric">

                            <strong>Missing %:</strong> {source\_stats['Missing %']:.2f}%

                        </div>

                    </div>

                    <table class="stats-table">

                        <tr><th>Metric</th><th>Value</th></tr>

                        {

                            ''.join([f"<tr><td>{k}</td><td>{v if isinstance(v, str) else f'{v:.2f}' if isinstance(v, float) else v}</td></tr>"

                                    for k, v in source\_stats.items()])

                        }

                    </table>

                    <div class="visualization">

                        <h3>Value Distribution</h3>

                        {generate\_distribution\_plot(source\_data) if pd.api.types.is\_numeric\_dtype(source\_data) else generate\_frequency\_plot(source\_data)}

                    </div>

                </div>

            """)

            # Generate HTML for target column

            target\_html.append(f"""

                <div class="column-profile">

                    <h2>Column: {tgt\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Type:</strong> {target\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Unique Values:</strong> {target\_stats['Unique Values']}

                        </div>

                        <div class="metric">

                            <strong>Missing %:</strong> {target\_stats['Missing %']:.2f}%

                        </div>

                    </div>

                    <table class="stats-table">

                        <tr><th>Metric</th><th>Value</th></tr>

                        {

                            ''.join([f"<tr><td>{k}</td><td>{v if isinstance(v, str) else f'{v:.2f}' if isinstance(v, float) else v}</td></tr>"

                                    for k, v in target\_stats.items()])

                        }

                    </table>

                    <div class="visualization">

                        <h3>Value Distribution</h3>

                        {generate\_distribution\_plot(target\_data) if pd.api.types.is\_numeric\_dtype(target\_data) else generate\_frequency\_plot(target\_data)}

                    </div>

                </div>

            """)

            # Generate comparison HTML

            comparison\_html.append(f"""

                <div class="comparison-profile">

                    <h2>Column Comparison: {src\_col} ↔ {tgt\_col}</h2>

                    <div class="summary">

                        <div class="metric">

                            <strong>Source Type:</strong> {source\_data.dtype}

                        </div>

                        <div class="metric">

                            <strong>Target Type:</strong> {target\_data.dtype}

                        </div>

                    </div>

                    <table class="comparison-table">

                        <tr>

                            <th>Metric</th>

                            <th>Source</th>

                            <th>Target</th>

                            <th>Difference</th>

                        </tr>

                        {generate\_comparison\_rows(source\_stats, target\_stats)}

                    </table>

                    <div class="visualization">

                        <h3>Distribution Comparison</h3>

                        {generate\_comparison\_plot(source\_data, target\_data)}

                    </div>

                </div>

            """)

        # Close HTML documents properly

        html\_footer = """

            </body>

        </html>

        """

        # Write to output buffers with proper HTML structure

        source\_output.write(''.join(source\_html + [html\_footer]).encode('utf-8'))

        target\_output.write(''.join(target\_html + [html\_footer]).encode('utf-8'))

        comparison\_output.write(''.join(comparison\_html + [html\_footer]).encode('utf-8'))

        # Reset buffer positions

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

                             datacompy\_html: 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."""

    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\_excel.getvalue())

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

                       datacompy\_html.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 = f"""

            Consolidated Data Comparison Report

            This ZIP file contains the following reports:

            1. datacompy\_report - Detailed comparison of datasets

            2. ydata\_profile - Column-wise data profiling

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

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

            Generated on: {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)}")

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."""

    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)

    if not agg\_df.empty:

        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)

    else:

        pd.DataFrame({'Message': ['No numeric columns to compare']}).to\_excel(

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

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)

    if not distinct\_df.empty:

        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)

    else:

        pd.DataFrame({'Message': ['No non-numeric columns to compare']}).to\_excel(

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

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

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

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

    """Generate a detailed difference report comparing source and target datasets."""

    try:

        output = BytesIO()

        # Process mappings and join columns

        try:

            # Initialize combined mapping dictionary

            all\_mappings = {}

            # Initialize mappings

            column\_mappings = {}

            # Process mapping\_df

            try:

                # Convert to DataFrame if needed

                if isinstance(mapping\_df, list):

                    if not mapping\_df:

                        raise ValueError("Empty mapping provided")

                    # Convert list to DataFrame

                    if isinstance(mapping\_df[0], dict):

                        mapping\_df = pd.DataFrame(mapping\_df)

                    else:

                        # Assume list of tuples/lists [source, target]

                        mapping\_df = pd.DataFrame(mapping\_df, columns=['Source Column', 'Target Column'])

                # Standardize column names

                rename\_dict = {}

                for col in mapping\_df.columns:

                    col\_lower = col.lower().replace('\_', ' ')

                    if col\_lower in ['source', 'source column']:

                        rename\_dict[col] = 'Source Column'

                    elif col\_lower in ['target', 'target column']:

                        rename\_dict[col] = 'Target Column'

                    elif col\_lower in ['exclude', 'exclude from comparison']:

                        rename\_dict[col] = 'Exclude from Comparison'

                if rename\_dict:

                    mapping\_df = mapping\_df.rename(columns=rename\_dict)

                # Validate required columns

                if 'Source Column' not in mapping\_df.columns or 'Target Column' not in mapping\_df.columns:

                    raise ValueError("Mapping must contain source and target column information")

                # Create mapping dictionary

                column\_mappings = dict(zip(mapping\_df['Source Column'], mapping\_df['Target Column']))

                # Handle excluded columns

                excluded\_columns = []

                if 'Exclude from Comparison' in mapping\_df.columns:

                    excluded\_mask = mapping\_df['Exclude from Comparison'].fillna(False)

                    excluded\_columns = mapping\_df[excluded\_mask]['Source Column'].tolist()

                # Remove excluded columns from mappings

                for col in excluded\_columns:

                    column\_mappings.pop(col, None)

            except Exception as e:

                logger.error(f"Error processing mapping\_df: {str(e)}")

                raise ValueError(f"Invalid mapping format: {str(e)}")

            # Validate and process join columns

            if not isinstance(join\_mappings, dict):

                raise ValueError("join\_mappings must be a dictionary")

            # Ensure all join columns have mappings

            missing\_joins = [col for col in join\_columns if col not in join\_mappings]

            if missing\_joins:

                raise ValueError(f"Missing mappings for join columns: {', '.join(missing\_joins)}")

            # Combine all mappings

            all\_mappings = {\*\*column\_mappings, \*\*join\_mappings}

            # Create valid columns mapping (excluding join columns)

            valid\_columns = {

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

                if src not in join\_columns

            }

            # Validate mappings

            if not valid\_columns:

                raise ValueError("No valid non-join columns found for comparison")

            # Validate source DataFrame columns

            source\_cols = set(join\_columns) | set(valid\_columns.keys())

            missing\_source = source\_cols - set(source\_df.columns)

            if missing\_source:

                raise ValueError(f"Source columns not found: {', '.join(sorted(missing\_source))}")

            # Validate target DataFrame columns

            target\_cols = set(join\_mappings.values()) | set(valid\_columns.values())

            missing\_target = target\_cols - set(target\_df.columns)

            if missing\_target:

                raise ValueError(f"Target columns not found: {', '.join(sorted(missing\_target))}")

        except Exception as e:

            logger.error(f"Error processing column mappings: {str(e)}")

            raise Exception(f"Failed to process column mappings: {str(e)}")

        # Create copies of DataFrames with only needed columns

        source\_cols = list(source\_required)

        target\_cols = list(target\_required)

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

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

        # Rename target columns to match source for comparison

        reverse\_mapping = {v: k for k, v in valid\_columns.items()}

        reverse\_mapping.update({v: k for k, v in join\_mappings.items()})

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

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

            workbook = writer.book

            # Create formats

            header\_format = workbook.add\_format({

                'bold': True,

                'bg\_color': '#D3D3D3',

                'border': 1

            })

            diff\_format = workbook.add\_format({

                'bg\_color': '#FFB6C6',  # Light pink

                'border': 1

            })

            normal\_format = workbook.add\_format({

                'border': 1

            })

            # Merge datasets safely

            try:

                # Ensure join columns exist in both dataframes

                common\_join\_cols = [col for col in join\_columns if col in source\_compare.columns and col in target\_compare.columns]

                if not common\_join\_cols:

                    raise ValueError("No valid join columns found in both datasets")

                merged\_df = source\_compare.merge(

                    target\_compare,

                    on=common\_join\_cols,

                    how='outer',

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

                    indicator=True

                )

                # Convert \_merge column to string type

                merged\_df['\_merge'] = merged\_df['\_merge'].astype(str)

                # Calculate differences for matching rows

                matching\_rows = merged\_df[merged\_df['\_merge'] == 'both'].copy()

                source\_cols = [c for c in matching\_rows.columns if c.endswith('\_source') and not c.startswith('join\_')]

                target\_cols = [c.replace('\_source', '\_target') for c in source\_cols]

                # Compare values and count differences

                differences\_mask = False

                for src\_col, tgt\_col in zip(source\_cols, target\_cols):

                    differences\_mask |= matching\_rows[src\_col].ne(matching\_rows[tgt\_col])

                rows\_with\_differences = len(matching\_rows[differences\_mask])

                # Generate summary

                summary\_data = {

                    'Metric': [

                        'Total Rows in Source',

                        'Total Rows in Target',

                        'Rows Only in Source',

                        'Rows Only in Target',

                        'Common Rows',

                        'Rows with Differences'

                    ],

                    'Count': [

                        len(source\_compare),

                        len(target\_compare),

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

                        len(merged\_df[merged\_df['\_merge'] == 'right\_only']),

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

                        rows\_with\_differences

                    ]

                }

            except Exception as e:

                logger.error(f"Error during merge operation: {str(e)}")

                raise Exception(f"Failed to merge datasets: {str(e)}")

            # Write Summary sheet

            summary\_df = pd.DataFrame(summary\_data)

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

            # Format Summary sheet

            summary\_sheet = writer.sheets['Summary']

            for col\_num, value in enumerate(summary\_df.columns.values):

                summary\_sheet.write(0, col\_num, value, header\_format)

            # Write Source Only sheet

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

            if not source\_only.empty:

                source\_cols = [c for c in source\_only.columns if not c.endswith('\_target')]

                source\_only[source\_cols].drop('\_merge', axis=1).to\_excel(

                    writer, sheet\_name='Source Only', index=False)

            # Write Target Only sheet

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

            if not target\_only.empty:

                target\_cols = [c for c in target\_only.columns if not c.endswith('\_source')]

                target\_only[target\_cols].drop('\_merge', axis=1).to\_excel(

                    writer, sheet\_name='Target Only', index=False)

            # Write Differences sheet

            differences = merged\_df[merged\_df['\_merge'] == 'both'].copy()

            if not differences.empty:

                # Identify columns with differences

                source\_cols = [c for c in differences.columns if c.endswith('\_source') and

                             not c.startswith('join\_')]

                target\_cols = [c.replace('\_source', '\_target') for c in source\_cols]

                # Create side-by-side comparison

                comparison\_cols = []

                for src, tgt in zip(source\_cols, target\_cols):

                    base\_col = src.replace('\_source', '')

                    differences[f'{base\_col}\_diff'] = differences[src] != differences[tgt]

                    comparison\_cols.extend([src, tgt, f'{base\_col}\_diff'])

                # Add join columns at the start

                final\_cols = join\_columns + comparison\_cols

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

                # Format Differences sheet

                diff\_sheet = writer.sheets['Differences']

                for col\_num, value in enumerate(final\_cols):

                    diff\_sheet.write(0, col\_num, value, header\_format)

                    if value.endswith('\_diff'):

                        # Highlight cells where differences exist

                        for row\_num in range(1, len(differences) + 1):

                            if differences.iloc[row\_num-1][value]:

                                diff\_sheet.write(row\_num, col\_num, 'True', diff\_format)

                            else:

                                diff\_sheet.write(row\_num, col\_num, 'False', normal\_format)

        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\_excel: BytesIO,

                                datacompy\_html: 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."""

    try:

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

        output = BytesIO()

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

            # DataCompy Reports

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

                       datacompy\_excel.getvalue())

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

                       datacompy\_html.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 = f"""

            Data Comparison Reports

            1. datacompy/ - Contains detailed comparison report

            2. ydata\_profile/ - Contains column-wise data profiling

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

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

            Generated on: {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)}")

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")

        # Initialize edited\_mappings list

        edited\_mappings = []

        # Create initial mapping DataFrame with editable source and target columns

        mapping\_df = pd.DataFrame({

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

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

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

        })

        st.write("Edit column mappings below. You can modify both source and target columns.")

        # Add button to add new mapping row

        if st.button("Add New Mapping"):

            new\_row = pd.DataFrame({

                'Source Column': [''],

                'Target Column': [''],

                'Exclude from Comparison': [False]

            })

            mapping\_df = pd.concat([mapping\_df, new\_row], ignore\_index=True)

        # Create a selection widget for each mapping row

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

            col1, col2, col3, col4 = st.columns([2, 2, 2, 1])

            with col1:

                # Dropdown for source column selection

                selected\_source = st.selectbox(

                    f"Source column",

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

                    index=list(source\_data.columns).index(row['Source Column']) + 1 if row['Source Column'] in source\_data.columns else 0,

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

                    help="Select the source column"

                )

            with col2:

                st.markdown("→")  # Arrow indicating mapping direction

            with col3:

                # Dropdown for target column selection

                selected\_target = st.selectbox(

                    f"Target column",

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

                    index=list(target\_data.columns).index(row['Target Column']) + 1 if row['Target Column'] in target\_data.columns else 0,

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

                    help="Select the corresponding target column"

                )

            with col4:

                # Checkbox for exclusion

                exclude = st.checkbox(

                    "Exclude",

                    value=row['Exclude from Comparison'],

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

                    help="Check to exclude this column from comparison"

                )

            # Add mapping if either both columns are selected or neither is selected

            if (selected\_source and selected\_target) or (not selected\_source and not selected\_target):

                edited\_mappings.append({

                    'Source Column': selected\_source,

                    'Target Column': selected\_target,

                    'Exclude from Comparison': exclude

                })

        # Convert edited mappings to DataFrame

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

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

        valid\_mappings = {

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

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

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

        }

        # Show mapping summary

        if valid\_mappings:

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

        else:

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

        # Join columns selection

        st.subheader("Join Columns")

        # Create list of available join columns from valid mappings

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

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

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

        if available\_columns:

            selected\_join\_columns = st.multiselect(

                "Select Join Columns",

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

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

            )

        else:

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

            selected\_join\_columns = []

        # Data type mapping

        st.subheader("Data Type Mapping")

        dtype\_mapping = {}

        for col in source\_data.columns:

            # Get current column type

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

            # Map numpy/pandas types to our type mapping

            if 'int' in current\_type:

                base\_type = 'int'

            elif 'float' in current\_type:

                base\_type = 'Float'

            elif 'datetime' in current\_type:

                base\_type = 'datetime'

            elif 'bool' in current\_type:

                base\_type = 'Boolean'

            elif 'object' in current\_type:

                # Try to infer type from data

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

                if len(sample) > 0:

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

                        base\_type = 'int'

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

                        base\_type = 'Float'

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

                        base\_type = 'Boolean'

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

                        base\_type = 'datetime'

                    else:

                        base\_type = 'varchar'

                else:

                    base\_type = 'varchar'

            else:

                base\_type = 'varchar'

            # Get mapped type from TYPE\_MAPPING

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

            # Create options list from TYPE\_MAPPING values

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

            # Create the selectbox with proper type selection

            dtype\_mapping[col] = st.selectbox(

                f"Type for {col}",

                options=type\_options,

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

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

            )

        # Validate mappings and show comparison button

        if valid\_mappings:

            # Validate data type compatibility

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

                source\_data, target\_data,

                valid\_mappings,

                dtype\_mapping

            )

            if not is\_valid:

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

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

            # Compare button

            compare\_button = st.button(

                "Compare",

                type="primary",

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

                help="Generate comparison reports for the mapped columns"

            )

            if compare\_button:

                try:

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

                        progress\_text = st.empty()

                        progress\_bar = st.progress(0)

                        # Get the actual join columns and their mappings

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

                        join\_cols = []

                        join\_mappings = {}

                        for col\_pair in selected\_join\_columns:

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

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

                            join\_cols.append(src\_col)

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

                        progress\_bar.progress(10)

                        # Apply data type conversions

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

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

                            # Skip if column is not in valid mappings

                            if col not in valid\_mappings:

                                continue

                            target\_col = valid\_mappings[col]

                            try:

                                if dtype == 'int64':

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

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

                                elif dtype == 'float64':

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

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

                                elif dtype == 'datetime64[ns]':

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

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

                                elif dtype == 'bool':

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

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

                                else:  # string type

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

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

                            except Exception as e:

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

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

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

                        progress\_bar.progress(20)

                        # Generate reports with progress updates

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

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

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

                        )

                        progress\_bar.progress(40)

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

                        try:

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

                                source\_data, target\_data, edited\_mapping

                            )

                        except Exception as e:

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

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

                            # Create basic HTML reports as fallback

                            source\_output = BytesIO()

                            target\_output = BytesIO()

                            comparison\_output = BytesIO()

                            source\_report = f"""

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

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

                            """

                            target\_report = f"""

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

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

                            """

                            comparison\_report = f"""

                            <html>

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

                            <body>

                            <h1>Data Comparison Report</h1>

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

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

                            </body></html>

                            """

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

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

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

                            source\_output.seek(0)

                            target\_output.seek(0)

                            comparison\_output.seek(0)

                            ydata\_source\_report = source\_output

                            ydata\_target\_report = target\_output

                            ydata\_comparison\_report = comparison\_output

                        progress\_bar.progress(60)

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

                        regression\_report = generate\_regression\_report(

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

                        )

                        progress\_bar.progress(80)

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

                        difference\_report = generate\_difference\_report(

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

                        )

                        progress\_bar.progress(90)

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

                        consolidated\_report = create\_consolidated\_report(

                            datacompy\_excel,

                            datacompy\_html,

                            ydata\_source\_report,

                            ydata\_target\_report,

                            ydata\_comparison\_report,

                            regression\_report,

                            difference\_report

                        )

                        progress\_bar.progress(100)

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

                        # Display reports in UI

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

                        # Create tabs for different reports

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

                        with report\_tabs[0]:

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

                            with ydata\_subtabs[0]:

                                st.markdown("""

                                    <style>

                                        iframe {

                                            width: 100%;

                                            min-width: 1200px !important;

                                            height: 2000px !important;

                                            border: none;

                                        }

                                        .stMarkdown {

                                            max-width: 100% !important;

                                        }

                                        .element-container {

                                            width: 100% !important;

                                            max-width: none !important;

                                        }

                                    </style>

                                """, unsafe\_allow\_html=True)

                                st.components.v1.html(

                                    ydata\_source\_report.getvalue().decode(),

                                    height=2000,

                                    scrolling=True,

                                    width=1200

                                )

                            with ydata\_subtabs[1]:

                                st.components.v1.html(

                                    ydata\_target\_report.getvalue().decode(),

                                    height=2000,

                                    scrolling=True,

                                    width=1200

                                )

                            with ydata\_subtabs[2]:

                                st.components.v1.html(

                                    ydata\_comparison\_report.getvalue().decode(),

                                    height=2000,

                                    scrolling=True,

                                    width=1200

                                )

                        with report\_tabs[1]:

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

                            with datacompy\_subtabs[0]:

                                try:

                                    # Show Excel report in tables

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

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

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

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

                                        st.markdown("---")

                                except Exception as e:

                                    st.error(f"Error displaying Excel report: {str(e)}")

                            with datacompy\_subtabs[1]:

                                # Show HTML report - use the already generated report

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

                        with report\_tabs[2]:

                            # Provide download links in columns

                            col1, col2 = st.columns(2)

                            with col1:

                                st.download\_button(

                                label="📥 Download Consolidated Report",

                                data=create\_consolidated\_report(

                                    datacompy\_excel,

                                    datacompy\_html,

                                    ydata\_source\_report,

                                    ydata\_target\_report,

                                    ydata\_comparison\_report,

                                    regression\_report,

                                    difference\_report

                                ),

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

                                mime="application/zip",

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

                            )

                            with col2:

                                st.download\_button(

                                label="📊 Download Individual Reports",

                                data=create\_individual\_reports\_zip(

                                    datacompy\_excel,

                                    datacompy\_html,

                                    ydata\_source\_report,

                                    ydata\_target\_report,

                                    ydata\_comparison\_report,

                                    regression\_report,

                                    difference\_report

                                ),

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

                                mime="application/zip",

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

                            )

                except Exception as e:

                    # Clear progress indicators

                    if 'progress\_text' in locals():

                        progress\_text.empty()

                    if 'progress\_bar' in locals():

                        progress\_bar.empty()

                    # Show detailed error message

                    error\_msg = str(e)

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

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

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

                    elif "SQL Server" in error\_msg:

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

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

                    elif "data type conversion" in error\_msg:

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

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

                    else:

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

                    # Log the full error

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

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

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

    try:

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

            file\_types = {

                "CSV file": ["csv"],

                "DAT file": ["dat"],

                "Parquet file": ["parquet"],

                "Excel file": ["xls", "xlsx"]

            }

            uploaded\_file = st.file\_uploader(

                f"Upload {data\_type}",

                type=file\_types[data\_type],

                key=f"{prefix}\_file"

            )

            if uploaded\_file:

                if data\_type == "Parquet file":

                    return read\_parquet(uploaded\_file)

                elif data\_type == "Excel file":

                    return read\_excel(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()