app.py

import streamlit as st

import pandas as pd

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

import os

from datetime import datetime

import datacompy

from ydata\_profiling import ProfileReport

import streamlit.components.v1 as components

# Import our modules

from data\_reader import DataReader

from mapping\_manager import MappingManager

from report\_generator import ReportGenerator

from db\_connector import DatabaseConnector

from api\_fetcher import APIFetcher

from utils import log\_error, format\_timestamp

# Set page config

st.set\_page\_config(

    page\_title="Data Comparison Framework",

    page\_icon="📊",

    layout="wide"

)

# Define source types

SOURCE\_TYPES = [

    "CSV File",

    "DAT File",

    "SQL Server",

    "Stored Procedure",

    "Teradata",

    "API",

    "Parquet File",

    "Zipped Flat Files"

]

def main():

    """Main application function"""

    # Add header with modern styling

    st.markdown(

        """

        <div style='text-align: center; background-color: #f0f2f6; padding: 1rem; border-radius: 10px;'>

            <h1 style='color: #1f77b4;'>Data Comparison Framework</h1>

            <p style='color: #666;'>Compare data from multiple sources with advanced mapping and reporting capabilities</p>

        </div>

        """,

        unsafe\_allow\_html=True

    )

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

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

    if 'report\_paths' not in st.session\_state:

        st.session\_state.report\_paths = {}

    # Create two columns for source and target selection

    col1, col2 = st.columns(2)

    with col1:

        st.subheader("Source Configuration")

        source\_type = st.selectbox("Select Source Type", SOURCE\_TYPES, key="source\_type")

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

    with col2:

        st.subheader("Target Configuration")

        target\_type = st.selectbox("Select Target Type", SOURCE\_TYPES, key="target\_type")

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

    # If both source and target data are loaded

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

        st.session\_state.source\_df = source\_data

        st.session\_state.target\_df = target\_data

        # Initialize or update column mapping

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

            st.session\_state.column\_mapping = MappingManager.auto\_map\_columns(source\_data, target\_data)

        # Show column mapping interface

        st.subheader("Column Mapping")

        show\_column\_mapping\_interface(source\_data, target\_data)

        # Add join key selection based on mapped columns

        st.subheader("Join Key Selection")

        # Get mapped columns (where source and target are mapped)

        mapped\_columns = {source\_col: target\_col

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

                        if source\_col in source\_data.columns and target\_col in target\_data.columns}

        if not mapped\_columns:

            st.warning("No mapped columns available for join keys. Please map columns first.")

            join\_keys = None

        else:

            # Show mapped columns available for joining

            st.write("Mapped columns available for joining:")

            mapped\_preview = []

            for source\_col, target\_col in mapped\_columns.items():

                st.write(f"- Source: {source\_col} → Target: {target\_col}")

                # Get sample values

                source\_sample = source\_data[source\_col].head(3).tolist()

                target\_sample = target\_data[target\_col].head(3).tolist()

                mapped\_preview.append({

                    'Source Column': source\_col,

                    'Target Column': target\_col,

                    'Source Sample': str(source\_sample),

                    'Target Sample': str(target\_sample)

                })

            # Show preview of mapped columns

            if mapped\_preview:

                st.write("Preview of mapped columns:")

                st.dataframe(pd.DataFrame(mapped\_preview))

            # Select join keys from mapped columns

            selected\_keys = st.multiselect(

                "Select Join Key(s)",

                options=list(mapped\_columns.keys()),

                help="Select one or more mapped columns to use as join keys for comparison. These columns should uniquely identify records."

            )

            if selected\_keys:

                # Create the join keys mapping (source to target columns)

                join\_keys = [(source\_col, mapped\_columns[source\_col]) for source\_col in selected\_keys]

                st.write("Selected join keys:")

                for source\_col, target\_col in join\_keys:

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

            else:

                st.warning("No join keys selected. Comparison will be done row by row.")

                join\_keys = None

        # Store join keys in session state

        st.session\_state.join\_keys = join\_keys

        # Compare button

        if st.button("Compare Data", type="primary"):

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

                perform\_comparison()

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

    """Handle different types of data sources"""

    try:

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

            return handle\_file\_upload(source\_type, prefix)

        elif source\_type in ["SQL Server", "Teradata", "Stored Procedure"]:

            return handle\_database\_connection(source\_type, prefix)

        elif source\_type == "API":

            return handle\_api\_connection(prefix)

    except Exception as e:

        log\_error(f"Error handling {source\_type}: {str(e)}")

        st.error(f"Error processing {source\_type}: {str(e)}")

    return None

def handle\_file\_upload(file\_type: str, prefix: str) -> Optional[pd.DataFrame]:

    """Handle file upload for different file types"""

    uploaded\_file = st.file\_uploader(f"Upload {file\_type}", key=f"{prefix}\_file")

    if uploaded\_file:

        delimiter = st.text\_input(f"Delimiter (for {file\_type})",

                                value=',' if file\_type == "CSV File" else '|',

                                key=f"{prefix}\_delimiter")

        try:

            if file\_type == "CSV File":

                return DataReader.load\_csv(uploaded\_file, delimiter=delimiter)

            elif file\_type == "DAT File":

                return DataReader.load\_dat(uploaded\_file, delimiter=delimiter)

            elif file\_type == "Parquet File":

                return DataReader.load\_parquet(uploaded\_file)

            elif file\_type == "Zipped Flat Files":

                return DataReader.load\_zipped\_flat\_files(uploaded\_file, separator=delimiter)

        except Exception as e:

            st.error(f"Error reading {file\_type}: {str(e)}")

    return None

def handle\_database\_connection(db\_type: str, prefix: str) -> Optional[pd.DataFrame]:

    """Handle database connections"""

    # Add SQL Server specific UI elements

    if db\_type == "SQL Server":

        st.markdown(

            """

            <div style='background-color: #f8f9fa; padding: 1rem; border-radius: 0.5rem; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

                <h4 style='margin: 0; color: #212529;'>SQL Server Connection</h4>

                <p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

                    Column names will be automatically standardized for better mapping.

                </p>

            </div>

            """,

            unsafe\_allow\_html=True

        )

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

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

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

        if db\_type in ["SQL Server", "Stored Procedure"]:

            use\_windows\_auth = st.checkbox("Use Windows Authentication", value=True, key=f"{prefix}\_use\_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:

            use\_windows\_auth = False

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

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

        if db\_type == "Stored Procedure":

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

            params = st.text\_area("Parameters (JSON format)", key=f"{prefix}\_params")

        else:

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

        if st.button("Connect", key=f"{prefix}\_connect"):

            try:

                # Build connection parameters

                conn\_params = {

                    'host': host,

                    'database': database,

                    'use\_windows\_auth': use\_windows\_auth if db\_type in ["SQL Server", "Stored Procedure"] else False

                }

                # Add username/password only if not using Windows Auth

                if not (db\_type in ["SQL Server", "Stored Procedure"] and use\_windows\_auth):

                    conn\_params.update({

                        'username': username,

                        'password': password

                    })

                if db\_type == "SQL Server":

                    df = DatabaseConnector.get\_sqlserver\_data(conn\_params, query)

                    if df is not None:

                        st.success("✅ SQL Server connection successful. Column names have been standardized for better mapping.")

                    return df

                elif db\_type == "Teradata":

                    return DatabaseConnector.get\_teradata\_data(conn\_params, query)

                elif db\_type == "Stored Procedure":

                    df = DatabaseConnector.get\_data\_from\_stored\_proc(

                        conn\_params, proc\_name, eval(params) if params else None)

                    if df is not None:

                        st.success("✅ Stored Procedure executed successfully. Column names have been standardized for better mapping.")

                    return df

            except Exception as e:

                st.error(f"Database connection error: {str(e)}")

    return None

def handle\_api\_connection(prefix: str) -> Optional[pd.DataFrame]:

    """Handle API connections"""

    with st.expander("API Connection Details"):

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

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

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

        params = st.text\_area("Parameters (JSON format)", key=f"{prefix}\_params")

        if st.button("Connect", key=f"{prefix}\_connect"):

            try:

                return APIFetcher.fetch\_api\_data(

                    api\_url=api\_url,

                    method=method,

                    headers=eval(headers) if headers else None,

                    params=eval(params) if params else None

                )

            except Exception as e:

                st.error(f"API connection error: {str(e)}")

    return None

def show\_column\_mapping\_interface(source\_df: pd.DataFrame, target\_df: pd.DataFrame):

    """Show interface for column mapping"""

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

    # Add auto-map button

    if st.button("Auto-Map Columns", key="auto\_map\_btn"):

        st.session\_state.column\_mapping = MappingManager.auto\_map\_columns(source\_df, target\_df)

        st.success("Columns automatically mapped!")

    # Show current mapping status

    if st.session\_state.column\_mapping:

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

        # Create a DataFrame to display the mapping

        mapping\_data = []

        for source\_col in source\_df.columns:

            source\_sample = str(source\_df[source\_col].head(2).tolist())

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

            target\_sample = str(target\_df[target\_col].head(2).tolist()) if target\_col else ''

            mapping\_data.append({

                'Source Column': source\_col,

                'Source Sample': source\_sample,

                'Target Column': target\_col,

                'Target Sample': target\_sample,

                'Mapped': '✓' if target\_col else '✗'

            })

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

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

    # Create three columns for the mapping interface

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

    with col1:

        st.markdown("\*\*Source Column\*\*")

    with col2:

        st.markdown("\*\*Target Column\*\*")

    with col3:

        st.markdown("\*\*Exclude\*\*")

    # Show mapping interface for each source column

    for source\_col in source\_df.columns:

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

        with col1:

            st.text(source\_col)

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

        with col2:

            # Dropdown for target column selection

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

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

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

            target\_col = st.selectbox(

                f"Map to",

                options=target\_options,

                index=target\_index,

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

            )

            if target\_col:

                if target\_col != current\_mapping:

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

                    # Show sample of selected target column

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

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

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

        with col3:

            # Checkbox for excluding column from comparison

            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)

def perform\_comparison():

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

    try:

        st.markdown("### Comparison Results")

        # Create reports directory if it doesn't exist

        os.makedirs("reports", exist\_ok=True)

        timestamp = format\_timestamp()

        # Generate reports

        join\_keys = st.session\_state.get('join\_keys', None)

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

            try:

                from ydata\_profiling import ProfileReport

                import streamlit.components.v1 as components

                # Generate Y-Data HTML profiling reports

                st.write("Generating Y-Data Profiling Reports...")

                # Source profiling

                source\_profile = ProfileReport(

                    st.session\_state.source\_df,

                    title="Source Data Profile Report",

                    minimal=True

                )

                source\_html = f"reports/SourceProfile\_{timestamp}.html"

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

                # Target profiling

                target\_profile = ProfileReport(

                    st.session\_state.target\_df,

                    title="Target Data Profile Report",

                    minimal=True

                )

                target\_html = f"reports/TargetProfile\_{timestamp}.html"

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

                # Generate other reports

                st.write("Generating Comparison Reports...")

                # Generate difference report

                diff\_df, has\_differences = ReportGenerator.generate\_diff\_report(

                    st.session\_state.source\_df,

                    st.session\_state.target\_df,

                    st.session\_state.column\_mapping,

                    st.session\_state.excluded\_columns,

                    join\_keys=join\_keys

                )

                # Generate profiling report

                profile\_df = ReportGenerator.generate\_profiling\_report(

                    st.session\_state.source\_df,

                    st.session\_state.target\_df,

                    st.session\_state.column\_mapping,

                    join\_keys=join\_keys

                )

                # Generate regression report (independent of join keys)

                regression\_path = f"reports/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 report

                side\_by\_side\_path = f"reports/DifferenceReport\_{timestamp}.xlsx"

                side\_by\_side\_df, \_ = ReportGenerator.generate\_side\_by\_side\_report(

                    st.session\_state.source\_df,

                    st.session\_state.target\_df,

                    st.session\_state.column\_mapping,

                    side\_by\_side\_path,

                    join\_keys=join\_keys

                )

                # Display Y-Data Profiling Reports in expandable sections

                st.subheader("Y-Data Profiling Reports")

                # Generate comparison profile

                comparison\_profile = ProfileReport(

                    pd.concat([

                        st.session\_state.source\_df.add\_prefix('Source\_'),

                        st.session\_state.target\_df.add\_prefix('Target\_')

                    ], axis=1),

                    title="Source vs Target Comparison Profile",

                    minimal=True

                )

                comparison\_html = f"reports/ComparisonProfile\_{timestamp}.html"

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

                # Generate DataCompy comparison report

                try:

                    # Prepare join keys for DataCompy

                    join\_columns = []

                    source\_df = st.session\_state.source\_df.copy()

                    target\_df = st.session\_state.target\_df.copy()

                    if join\_keys:

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

                    else:

                        # If no join keys specified, use index

                        source\_df['\_index'] = source\_df.index

                        target\_df['\_index'] = target\_df.index

                        join\_columns = ['\_index']

                    # Create DataCompy comparison

                    comparison = datacompy.Compare(

                        source\_df,

                        target\_df,

                        join\_columns=join\_columns,

                        df1\_name='Source',

                        df2\_name='Target'

                    )

                    # Get column statistics safely

                    column\_stats\_html = '<p>No column statistics available.</p>'

                    try:

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

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

                            not comparison.column\_stats.empty):

                            column\_stats\_html = comparison.column\_stats.to\_html()

                    except Exception as e:

                        st.warning(f"Could not generate column statistics: {str(e)}")

                    # Generate HTML report

                    datacompy\_html = f"reports/DataCompyReport\_{timestamp}.html"

                    with open(datacompy\_html, 'w') as f:

                        f.write(f"""

                        <html>

                        <head>

                            <title>DataCompy Comparison Report</title>

                            <style>

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

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

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

                                .match {{ color: green; }}

                                .mismatch {{ color: red; }}

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

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

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

                            </style>

                        </head>

                        <body>

                            <div class="report">

                                <h1>DataCompy Comparison Report</h1>

                                <div class="section">

                                    <h2>Summary</h2>

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

                                </div>

                                <div class="section">

                                    <h2>Detailed Statistics</h2>

                                    <h3>Matches</h3>

                                    <div class="match">

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

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

                                    </div>

                                    <h3>Mismatches</h3>

                                    <div class="mismatch">

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

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

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

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

                                    </div>

                                </div>

                                <div class="section">

                                    <h2>Column Statistics</h2>

                                    {column\_stats\_html}

                                </div>

                            </div>

                        </body>

                        </html>

                        """)

                except Exception as e:

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

                    datacompy\_html = None

                # Store DataCompy report path

                st.session\_state.report\_paths['datacompy\_html'] = datacompy\_html

                # Store report paths in session state to prevent page reset

                if 'report\_paths' not in st.session\_state:

                    st.session\_state.report\_paths = {}

                st.session\_state.report\_paths.update({

                    'source\_html': source\_html,

                    'target\_html': target\_html,

                    'comparison\_html': comparison\_html

                })

                tab1, tab2, tab3, tab4 = st.tabs(["Source Profile", "Target Profile", "Comparison Profile", "DataCompy Report"])

                with tab1:

                    with open(source\_html, 'r', encoding='utf-8') as f:

                        components.html(f.read(), height=600, scrolling=True)

                    # Download button for source profile

                    with open(source\_html, 'rb') as f:

                        st.download\_button(

                            "Download Source Profile Report",

                            f,

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

                            mime="text/html",

                            key="download\_source"

                        )

                with tab2:

                    with open(target\_html, 'r', encoding='utf-8') as f:

                        components.html(f.read(), height=600, scrolling=True)

                    # Download button for target profile

                    with open(target\_html, 'rb') as f:

                        st.download\_button(

                            "Download Target Profile Report",

                            f,

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

                            mime="text/html",

                            key="download\_target"

                        )

                with tab3:

                    with open(comparison\_html, 'r', encoding='utf-8') as f:

                        components.html(f.read(), height=600, scrolling=True)

                    # Download button for comparison profile

                    with open(comparison\_html, 'rb') as f:

                        st.download\_button(

                            "Download Comparison Profile Report",

                            f,

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

                            mime="text/html",

                            key="download\_comparison"

                        )

                with tab4:

                    if datacompy\_html and os.path.exists(datacompy\_html):

                        with open(datacompy\_html, 'r', encoding='utf-8') as f:

                            components.html(f.read(), height=600, scrolling=True)

                        col1, col2 = st.columns([3, 1])

                        with col1:

                            # Download button for DataCompy report

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

                                st.download\_button(

                                    "📥 Download DataCompy Report",

                                    f,

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

                                    mime="text/html",

                                    key="download\_datacompy",

                                    use\_container\_width=True

                                )

                        # Show additional DataCompy insights

                        with st.expander("📊 DataCompy Insights", expanded=False):

                            try:

                                st.write("### Match Statistics")

                                col1, col2, col3 = st.columns(3)

                                matching\_rows = comparison.count\_matching\_rows()

                                source\_only\_rows = len(comparison.df1\_unq\_rows)

                                target\_only\_rows = len(comparison.df2\_unq\_rows)

                                with col1:

                                    st.metric(

                                        "Matching Rows",

                                        matching\_rows,

                                        delta=f"{matching\_rows / len(source\_df) \* 100:.1f}%"

                                    )

                                with col2:

                                    st.metric(

                                        "Source Only Rows",

                                        source\_only\_rows,

                                        delta=f"{source\_only\_rows / len(source\_df) \* 100:.1f}%"

                                    )

                                with col3:

                                    st.metric(

                                        "Target Only Rows",

                                        target\_only\_rows,

                                        delta=f"{target\_only\_rows / len(target\_df) \* 100:.1f}%"

                                    )

                                st.write("### Column Analysis")

                                # Get column sets

                                source\_cols = set(source\_df.columns)

                                target\_cols = set(target\_df.columns)

                                common\_cols = source\_cols.intersection(target\_cols)

                                source\_only = source\_cols - target\_cols

                                target\_only = target\_cols - source\_cols

                                # Display column sets

                                col1, col2 = st.columns(2)

                                with col1:

                                    st.write("Source-only Columns:")

                                    if source\_only:

                                        for col in sorted(source\_only):

                                            st.info(f"- {col}")

                                    else:

                                        st.success("No source-only columns")

                                with col2:

                                    st.write("Target-only Columns:")

                                    if target\_only:

                                        for col in sorted(target\_only):

                                            st.info(f"- {col}")

                                    else:

                                        st.success("No target-only columns")

                                st.write("### Common Column Analysis")

                                if common\_cols:

                                    for col in sorted(common\_cols):

                                        # Calculate match rate for common columns

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

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

                                        common\_values = source\_values.intersection(target\_values)

                                        all\_values = source\_values.union(target\_values)

                                        match\_rate = len(common\_values) / len(all\_values) if all\_values else 1.0

                                        # Display progress bar with details

                                        st.progress(

                                            match\_rate,

                                            text=f"{col}: {match\_rate\*100:.1f}% unique values match"

                                        )

                                        with st.expander(f"Details for {col}", expanded=False):

                                            st.write(f"- Unique values in Source: {len(source\_values)}")

                                            st.write(f"- Unique values in Target: {len(target\_values)}")

                                            st.write(f"- Common unique values: {len(common\_values)}")

                                            if len(source\_values - target\_values) > 0:

                                                st.write("- Values only in Source:", list(source\_values - target\_values)[:5])

                                            if len(target\_values - source\_values) > 0:

                                                st.write("- Values only in Target:", list(target\_values - source\_values)[:5])

                                else:

                                    st.warning("No common columns found between source and target datasets")

                            except Exception as e:

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

                    else:

                        st.error("❌ DataCompy report generation failed. Please check the data and try again.")

                        with col2:

                            st.write("Target-only Columns:")

                            if target\_only:

                                for col in sorted(target\_only):

                                    st.info(f"- {col}")

                            else:

                                st.success("No target-only columns")

                        st.write("### Common Column Analysis")

                        if common\_cols:

                            for col in sorted(common\_cols):

                                # Calculate match rate for common columns

                                source\_values = set(st.session\_state.source\_df[col].dropna())

                                target\_values = set(st.session\_state.target\_df[col].dropna())

                                common\_values = source\_values.intersection(target\_values)

                                all\_values = source\_values.union(target\_values)

                                match\_rate = len(common\_values) / len(all\_values) if all\_values else 1.0

                                # Display progress bar with details

                                st.progress(

                                    match\_rate,

                                    text=f"{col}: {match\_rate\*100:.1f}% unique values match"

                                )

                                with st.expander(f"Details for {col}", expanded=False):

                                    st.write(f"- Unique values in Source: {len(source\_values)}")

                                    st.write(f"- Unique values in Target: {len(target\_values)}")

                                    st.write(f"- Common unique values: {len(common\_values)}")

                                    if len(source\_values - target\_values) > 0:

                                        st.write("- Values only in Source:", list(source\_values - target\_values)[:5])

                                    if len(target\_values - source\_values) > 0:

                                        st.write("- Values only in Target:", list(target\_values - source\_values)[:5])

                        else:

                            st.warning("No common columns found between source and target datasets")

            except ImportError:

                st.warning("Y-Data Profiling package not installed. Please install ydata-profiling package.")

                # Continue with other reports...

        # Store report paths in session state

        st.session\_state.report\_paths.update({

            'side\_by\_side\_path': side\_by\_side\_path,

            'profiling\_path': f"reports/ProfilingReport\_{timestamp}.xlsx",

            'regression\_path': regression\_path

        })

        # Create zip file containing all reports

        st.divider()

        zip\_path = f"reports/FinalComparison\_{timestamp}.zip"

        try:

            import zipfile

            with zipfile.ZipFile(zip\_path, 'w') as zipf:

                # Add all reports to zip

                for report\_name, report\_path in st.session\_state.report\_paths.items():

                    if report\_path and os.path.exists(report\_path):

                        zipf.write(report\_path, os.path.basename(report\_path))

            # Add download button for zip file

            st.markdown("### 📦 Download Complete Comparison Package")

            st.caption("Contains all reports: Regression, Side by Side, DataCompy, and Profile Reports")

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

                st.download\_button(

                    "📥 Download All Reports (ZIP)",

                    f,

                    file\_name=f"FinalComparison\_{timestamp}.zip",

                    mime="application/zip",

                    key="download\_all",

                    use\_container\_width=True

                )

        except Exception as e:

            st.error(f"Error creating zip file: {str(e)}")

        st.divider()

        # Display results and download buttons

        col1, col2 = st.columns(2)

        with col1:

            st.markdown("#### Data Comparison Results")

            st.dataframe(diff\_df)

            if os.path.exists(side\_by\_side\_path):

                with open(side\_by\_side\_path, 'rb') as f:

                    st.download\_button(

                        "Download Difference Report",

                        f,

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

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

                        key="download\_difference"

                    )

        with col2:

            st.markdown("#### Y-Data Profiling Results")

            if not profile\_df.empty:

                # Format the dataframe for better display

                st.write("Column-wise Statistical Comparison:")

                # Create tabs for different views

                metrics\_tab, details\_tab = st.tabs(["Main Metrics", "Detailed Statistics"])

                with metrics\_tab:

                    main\_metrics = profile\_df[['Column', 'Source\_Count', 'Target\_Count', 'Match\_Percentage']]

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

                with details\_tab:

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

                # Save profiling report

                profiling\_path = st.session\_state.report\_paths['profiling\_path']

                profile\_df.to\_excel(profiling\_path, index=False)

                # Download buttons with unique keys

                col3, col4 = st.columns(2)

                with col3:

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

                        st.download\_button(

                            "Download Profiling Report",

                            f,

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

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

                            key="download\_profiling"

                        )

                with col4:

                    if os.path.exists(regression\_path):

                        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",

                                key="download\_regression"

                            )

            else:

                st.error("Unable to generate profiling report. Please check the data and try again.")

    except Exception as e:

        log\_error(f"Error performing comparison: {str(e)}")

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

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

    main()

Db\_connector

import pandas as pd

from sqlalchemy import create\_engine, text

from typing import Dict, Union, Optional, Any

import pyodbc

import teradatasql

from utils import log\_error, validate\_connection\_params, clean\_df\_columns

class DatabaseConnector:

    """Class to handle database connections and data retrieval"""

    @staticmethod

    def get\_sqlserver\_data(connection\_params: Dict[str, str], query: str) -> Optional[pd.DataFrame]:

        """

        Retrieve data from SQL Server using provided connection parameters and query.

        Args:

            connection\_params: Dictionary containing connection parameters

            query: SQL query to execute

        Returns:

            Optional[pd.DataFrame]: DataFrame containing query results, None if error occurs

        """

        try:

            if not validate\_connection\_params(connection\_params):

                log\_error("Invalid connection parameters for SQL Server")

                return None

            try:

                # Get server name from either 'server' or 'host' parameter

                server = connection\_params.get('server') or connection\_params.get('host')

                if not server:

                    raise ValueError("Server/Host is required")

                database = connection\_params.get('database')

                if not database:

                    raise ValueError("Database name is required")

                # Build connection string based on authentication type

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

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        "Trusted\_Connection=yes"

                    )

                else:

                    username = connection\_params.get('username')

                    password = connection\_params.get('password')

                    if not username or not password:

                        raise ValueError("Username and password are required for SQL authentication")

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        f"Uid={username};"

                        f"Pwd={password}"

                    )

                # Create engine using the pyodbc connection string

                engine = create\_engine(

                    f"mssql+pyodbc:///?odbc\_connect={conn\_str}",

                    fast\_executemany=True

                )

            with engine.connect() as connection:

                df = pd.read\_sql(text(query), connection)

                df = clean\_df\_columns(df)  # Clean column names for consistency

                return df

        except Exception as e:

            log\_error(f"Error retrieving data from SQL Server: {str(e)}")

            return None

    @staticmethod

    def get\_teradata\_data(connection\_params: Dict[str, str], query: str) -> Optional[pd.DataFrame]:

        """

        Retrieve data from Teradata using provided connection parameters and query.

        Args:

            connection\_params: Dictionary containing connection parameters

            query: SQL query to execute

        Returns:

            Optional[pd.DataFrame]: DataFrame containing query results, None if error occurs

        """

        try:

            if not validate\_connection\_params(connection\_params):

                log\_error("Invalid connection parameters for Teradata")

                return None

            # Create Teradata connection

            conn = teradatasql.connect(

                host=connection\_params['host'],

                user=connection\_params['username'],

                password=connection\_params['password'],

                database=connection\_params['database']

            )

            # Execute query and return results as DataFrame

            df = pd.read\_sql(query, conn)

            conn.close()

            return df

        except Exception as e:

            log\_error(f"Error retrieving data from Teradata: {str(e)}")

            return None

    @staticmethod

    def get\_data\_from\_stored\_proc(connection\_params: Dict[str, str],

                                 proc\_name: str,

                                 params: Optional[Dict[str, Any]] = None) -> Optional[pd.DataFrame]:

        """

        Execute a stored procedure and retrieve its results.

        Args:

            connection\_params: Dictionary containing connection parameters

            proc\_name: Name of the stored procedure to execute

            params: Optional dictionary of parameters for the stored procedure

        Returns:

            Optional[pd.DataFrame]: DataFrame containing procedure results, None if error occurs

        """

        try:

            if not validate\_connection\_params(connection\_params):

                log\_error("Invalid connection parameters for stored procedure execution")

                return None

            try:

                # Get server name from either 'server' or 'host' parameter

                server = connection\_params.get('server') or connection\_params.get('host')

                if not server:

                    raise ValueError("Server/Host is required")

                database = connection\_params.get('database')

                if not database:

                    raise ValueError("Database name is required")

                # Build connection string based on authentication type

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

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        "Trusted\_Connection=yes"

                    )

                else:

                    username = connection\_params.get('username')

                    password = connection\_params.get('password')

                    if not username or not password:

                        raise ValueError("Username and password are required for SQL authentication")

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        f"Uid={username};"

                        f"Pwd={password}"

                    )

                # Connect and execute stored procedure

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

                cursor = conn.cursor()

                if params:

                    # Build parameter string

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

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

                    cursor.execute(exec\_str, list(params.values()))

                else:

                    cursor.execute(f"EXEC {proc\_name}")

                # Fetch results

                columns = [column[0] for column in cursor.description]

                results = cursor.fetchall()

                # Convert to DataFrame and clean column names

                df = pd.DataFrame.from\_records(results, columns=columns)

                df = clean\_df\_columns(df)  # Clean column names for consistency

                return df

        except Exception as e:

            log\_error(f"Error executing stored procedure: {str(e)}")

            return None

    @staticmethod

    def test\_connection(connection\_params: Dict[str, str], db\_type: str) -> bool:

        """

        Test database connection without executing any queries.

        Args:

            connection\_params: Dictionary containing connection parameters

            db\_type: Type of database ('sqlserver' or 'teradata')

        Returns:

            bool: True if connection successful, False otherwise

        """

        try:

            if not validate\_connection\_params(connection\_params):

                return False

            if db\_type.lower() == 'sqlserver':

                try:

                    # Get server name from either 'server' or 'host' parameter

                    server = connection\_params.get('server') or connection\_params.get('host')

                    if not server:

                        raise ValueError("Server/Host is required")

                    database = connection\_params.get('database')

                    if not database:

                        raise ValueError("Database name is required")

                    # Build connection string based on authentication type

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

                        conn\_str = (

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

                            f"Server={server};"

                            f"Database={database};"

                            "Trusted\_Connection=yes"

                        )

                    else:

                        username = connection\_params.get('username')

                        password = connection\_params.get('password')

                        if not username or not password:

                            raise ValueError("Username and password are required for SQL authentication")

                        conn\_str = (

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

                            f"Server={server};"

                            f"Database={database};"

                            f"Uid={username};"

                            f"Pwd={password}"

                        )

                    # Test connection

                    conn = pyodbc.connect(conn\_str)

                    conn.close()

                    return True

                except Exception as e:

                    log\_error(f"SQL Server connection test failed: {str(e)}")

                    return False

            elif db\_type.lower() == 'teradata':

                conn = teradatasql.connect(

                    host=connection\_params['host'],

                    user=connection\_params['username'],

                    password=connection\_params['password'],

                    database=connection\_params['database']

                )

                conn.close()

                return True

            else:

                log\_error(f"Unsupported database type: {db\_type}")

                return False

        except Exception as e:

            log\_error(f"Error testing connection: {str(e)}")

            return False

    @staticmethod

    def get\_table\_schema(connection\_params: Dict[str, str],

                        table\_name: str,

                        db\_type: str) -> Optional[pd.DataFrame]:

        """

        Retrieve schema information for a specified table.

        Args:

            connection\_params: Dictionary containing connection parameters

            table\_name: Name of the table

            db\_type: Type of database ('sqlserver' or 'teradata')

        Returns:

            Optional[pd.DataFrame]: DataFrame containing schema information, None if error occurs

        """

        try:

            if db\_type.lower() == 'sqlserver':

                query = f"""

                SELECT

                    COLUMN\_NAME,

                    DATA\_TYPE,

                    CHARACTER\_MAXIMUM\_LENGTH,

                    IS\_NULLABLE

                FROM INFORMATION\_SCHEMA.COLUMNS

                WHERE TABLE\_NAME = '{table\_name}'

                ORDER BY ORDINAL\_POSITION

                """

                return DatabaseConnector.get\_sqlserver\_data(connection\_params, query)

            elif db\_type.lower() == 'teradata':

                query = f"""

                SELECT

                    ColumnName,

                    ColumnType,

                    CharacterLength,

                    Nullable

                FROM DBC.Columns

                WHERE TableName = '{table\_name}'

                ORDER BY ColumnId

                """

                return DatabaseConnector.get\_teradata\_data(connection\_params, query)

            else:

                log\_error(f"Unsupported database type: {db\_type}")

                return None

        except Exception as e:

            log\_error(f"Error retrieving table schema: {str(e)}")

            return None

utils.py

import logging

import os

from datetime import datetime

# Configure logging

logging.basicConfig(

    level=logging.INFO,

    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

    handlers=[

        logging.StreamHandler(),

        logging.FileHandler('comparison\_framework.log')

    ]

)

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

def log\_error(error\_message: str) -> None:

    """

    Log error messages to both console and file.

    Args:

        error\_message (str): The error message to log

    """

    logger.error(error\_message)

def check\_file\_size(file) -> bool:

    """

    Check if file size exceeds 3GB threshold.

    Args:

        file: File object or path

    Returns:

        bool: True if file size is within limits, False otherwise

    """

    try:

        if hasattr(file, 'seek'):

            file.seek(0, os.SEEK\_END)

            size = file.tell()

            file.seek(0)  # Reset file pointer

        else:

            size = os.path.getsize(file)

        # 3GB in bytes

        GB\_3 = 3 \* 1024 \* 1024 \* 1024

        return size <= GB\_3

    except Exception as e:

        log\_error(f"Error checking file size: {str(e)}")

        return False

def format\_timestamp() -> str:

    """

    Get formatted timestamp for file naming.

    Returns:

        str: Formatted timestamp string

    """

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

def validate\_connection\_params(params: dict) -> bool:

    """

    Validate database connection parameters.

    Args:

        params (dict): Dictionary containing connection parameters

    Returns:

        bool: True if parameters are valid, False otherwise

    """

    try:

        if not isinstance(params, dict):

            log\_error("Connection parameters must be a dictionary")

            return False

        # Check if using Windows Authentication

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

            # For Windows Auth, we need either host or server, and database

            if not params.get('database'):

                log\_error("Database name is required")

                return False

            if not (params.get('host') or params.get('server')):

                log\_error("Server/Host is required")

                return False

            return True

        else:

            # For SQL Auth, we need host/server, database, username, and password

            required\_params = ['database', 'username', 'password']

            if not (params.get('host') or params.get('server')):

                log\_error("Server/Host is required")

                return False

            for param in required\_params:

                if not params.get(param):

                    log\_error(f"{param} is required for SQL Authentication")

                    return False

            return True

    except Exception as e:

        log\_error(f"Error validating connection parameters: {str(e)}")

        return False

def sanitize\_filename(filename: str) -> str:

    """

    Sanitize filename by removing invalid characters.

    Args:

        filename (str): Original filename

    Returns:

        str: Sanitized filename

    """

    # Replace invalid characters with underscore

    invalid\_chars = '<>:"/\\|?\*'

    for char in invalid\_chars:

        filename = filename.replace(char, '\_')

    return filename

def get\_file\_extension(filename: str) -> str:

    """

    Get file extension from filename.

    Args:

        filename (str): Name of the file

    Returns:

        str: File extension without dot

    """

    return os.path.splitext(filename)[1][1:].lower()

def clean\_df\_columns(df) -> 'pd.DataFrame':

    """

    Returns a copy of the DataFrame with whitespace trimmed from all column names.

    Args:

        df (pd.DataFrame): Input DataFrame.

    Returns:

        pd.DataFrame: DataFrame with cleaned column names.

    """

    try:

        df\_copy = df.copy()

        df\_copy.columns = df\_copy.columns.str.strip()

        return df\_copy

    except Exception as e:

        log\_error(f"Error cleaning DataFrame columns: {str(e)}")

        return df  # Return original DataFrame if cleaning fails

import pandas as pd

from sqlalchemy import create\_engine, text

from typing import Dict, Union, Optional, Any

import pyodbc

import teradatasql

from utils import log\_error, validate\_connection\_params, clean\_df\_columns

class DatabaseConnector:

    """Class to handle database connections and data retrieval"""

    @staticmethod

    def get\_sqlserver\_data(connection\_params: Dict[str, str], query: str) -> Optional[pd.DataFrame]:

        """

        Retrieve data from SQL Server using provided connection parameters and query.

        Args:

            connection\_params: Dictionary containing connection parameters

            query: SQL query to execute

        Returns:

            Optional[pd.DataFrame]: DataFrame containing query results, None if error occurs

        """

        try:

            if not validate\_connection\_params(connection\_params):

                log\_error("Invalid connection parameters for SQL Server")

                return None

            try:

                # Get server name from either 'server' or 'host' parameter

                server = connection\_params.get('server') or connection\_params.get('host')

                if not server:

                    raise ValueError("Server/Host is required")

                database = connection\_params.get('database')

                if not database:

                    raise ValueError("Database name is required")

                # Build connection string based on authentication type

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

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        "Trusted\_Connection=yes"

                    )

                else:

                    username = connection\_params.get('username')

                    password = connection\_params.get('password')

                    if not username or not password:

                        raise ValueError("Username and password are required for SQL authentication")

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        f"Uid={username};"

                        f"Pwd={password}"

                    )

                # Create engine using the pyodbc connection string

                engine = create\_engine(

                    f"mssql+pyodbc:///?odbc\_connect={conn\_str}",

                    fast\_executemany=True

                )

                # Connect and execute query

                with engine.connect() as connection:

                    df = pd.read\_sql(text(query), connection)

                    df = clean\_df\_columns(df)  # Clean column names for consistency

                    return df

        except Exception as e:

            log\_error(f"Error retrieving data from SQL Server: {str(e)}")

            return None

    @staticmethod

    def get\_teradata\_data(connection\_params: Dict[str, str], query: str) -> Optional[pd.DataFrame]:

        """

        Retrieve data from Teradata using provided connection parameters and query.

        Args:

            connection\_params: Dictionary containing connection parameters

            query: SQL query to execute

        Returns:

            Optional[pd.DataFrame]: DataFrame containing query results, None if error occurs

        """

        try:

            if not validate\_connection\_params(connection\_params):

                log\_error("Invalid connection parameters for Teradata")

                return None

            # Create Teradata connection

            conn = teradatasql.connect(

                host=connection\_params['host'],

                user=connection\_params['username'],

                password=connection\_params['password'],

                database=connection\_params['database']

            )

            # Execute query and return results as DataFrame

            df = pd.read\_sql(query, conn)

            conn.close()

            return df

        except Exception as e:

            log\_error(f"Error retrieving data from Teradata: {str(e)}")

            return None

    @staticmethod

    def get\_data\_from\_stored\_proc(connection\_params: Dict[str, str],

                                 proc\_name: str,

                                 params: Optional[Dict[str, Any]] = None) -> Optional[pd.DataFrame]:

        """

        Execute a stored procedure and retrieve its results.

        Args:

            connection\_params: Dictionary containing connection parameters

            proc\_name: Name of the stored procedure to execute

            params: Optional dictionary of parameters for the stored procedure

        Returns:

            Optional[pd.DataFrame]: DataFrame containing procedure results, None if error occurs

        """

        try:

            if not validate\_connection\_params(connection\_params):

                log\_error("Invalid connection parameters for stored procedure execution")

                return None

            try:

                # Get server name from either 'server' or 'host' parameter

                server = connection\_params.get('server') or connection\_params.get('host')

                if not server:

                    raise ValueError("Server/Host is required")

                database = connection\_params.get('database')

                if not database:

                    raise ValueError("Database name is required")

                # Build connection string based on authentication type

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

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        "Trusted\_Connection=yes"

                    )

                else:

                    username = connection\_params.get('username')

                    password = connection\_params.get('password')

                    if not username or not password:

                        raise ValueError("Username and password are required for SQL authentication")

                    conn\_str = (

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

                        f"Server={server};"

                        f"Database={database};"

                        f"Uid={username};"

                        f"Pwd={password}"

                    )

                # Connect and execute stored procedure

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

                    cursor = conn.cursor()

                    if params:

                        # Build parameter string

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

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

                        cursor.execute(exec\_str, list(params.values()))

                    else:

                        cursor.execute(f"EXEC {proc\_name}")

                    # Fetch results

                    columns = [column[0] for column in cursor.description]

                    results = cursor.fetchall()

                    # Convert to DataFrame and clean column names

                    df = pd.DataFrame.from\_records(results, columns=columns)

                    df = clean\_df\_columns(df)  # Clean column names for consistency

                    return df

        except Exception as e:

            log\_error(f"Error executing stored procedure: {str(e)}")

            return None

    @staticmethod

    def test\_connection(connection\_params: Dict[str, str], db\_type: str) -> bool:

        """

        Test database connection without executing any queries.

        Args:

            connection\_params: Dictionary containing connection parameters

            db\_type: Type of database ('sqlserver' or 'teradata')

        Returns:

            bool: True if connection successful, False otherwise

        """

        try:

            if not validate\_connection\_params(connection\_params):

                return False

            if db\_type.lower() == 'sqlserver':

                try:

                    # Get server name from either 'server' or 'host' parameter

                    server = connection\_params.get('server') or connection\_params.get('host')

                    if not server:

                        raise ValueError("Server/Host is required")

                    database = connection\_params.get('database')

                    if not database:

                        raise ValueError("Database name is required")

                    # Build connection string based on authentication type

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

                        conn\_str = (

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

                            f"Server={server};"

                            f"Database={database};"

                            "Trusted\_Connection=yes"

                        )

                    else:

                        username = connection\_params.get('username')

                        password = connection\_params.get('password')

                        if not username or not password:

                            raise ValueError("Username and password are required for SQL authentication")

                        conn\_str = (

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

                            f"Server={server};"

                            f"Database={database};"

                            f"Uid={username};"

                            f"Pwd={password}"

                        )

                    # Test connection

                    conn = pyodbc.connect(conn\_str)

                    conn.close()

                    return True

                except Exception as e:

                    log\_error(f"SQL Server connection test failed: {str(e)}")

                    return False

            elif db\_type.lower() == 'teradata':

                conn = teradatasql.connect(

                    host=connection\_params['host'],

                    user=connection\_params['username'],

                    password=connection\_params['password'],

                    database=connection\_params['database']

                )

                conn.close()

                return True

            else:

                log\_error(f"Unsupported database type: {db\_type}")

                return False

        except Exception as e:

            log\_error(f"Error testing connection: {str(e)}")

            return False

    @staticmethod

    def get\_table\_schema(connection\_params: Dict[str, str],

                        table\_name: str,

                        db\_type: str) -> Optional[pd.DataFrame]:

        """

        Retrieve schema information for a specified table.

        Args:

            connection\_params: Dictionary containing connection parameters

            table\_name: Name of the table

            db\_type: Type of database ('sqlserver' or 'teradata')

        Returns:

            Optional[pd.DataFrame]: DataFrame containing schema information, None if error occurs

        """

        try:

            if db\_type.lower() == 'sqlserver':

                query = f"""

                SELECT

                    COLUMN\_NAME,

                    DATA\_TYPE,

                    CHARACTER\_MAXIMUM\_LENGTH,

                    IS\_NULLABLE

                FROM INFORMATION\_SCHEMA.COLUMNS

                WHERE TABLE\_NAME = '{table\_name}'

                ORDER BY ORDINAL\_POSITION

                """

                return DatabaseConnector.get\_sqlserver\_data(connection\_params, query)

            elif db\_type.lower() == 'teradata':

                query = f"""

                SELECT

                    ColumnName,

                    ColumnType,

                    CharacterLength,

                    Nullable

                FROM DBC.Columns

                WHERE TableName = '{table\_name}'

                ORDER BY ColumnId

                """

                return DatabaseConnector.get\_teradata\_data(connection\_params, query)

            else:

                log\_error(f"Unsupported database type: {db\_type}")

                return None

        except Exception as e:

            log\_error(f"Error retrieving table schema: {str(e)}")

            return None