import os

import pandas as pd

import numpy as np

import pyodbc

import requests

import json

import zipfile

from io import BytesIO

from sqlalchemy import create\_engine

import logging

import dask.dataframe as dd

from .utils import check\_file\_size

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

def read\_csv\_dat(file, delimiter=',', chunksize=None):

    """

    Optimized CSV/DAT file reader for large files.

    Args:

        file: File object or path

        delimiter: Column separator

        chunksize: Number of rows to read at a time for large files

    Returns:

        pandas DataFrame

    """

    try:

        # Handle both string paths and file objects

        if hasattr(file, 'read'):

            # If it's already a file object

            file\_path = file.name if hasattr(file, 'name') else None

            if file\_path and not os.path.exists(file\_path):

                # Try to resolve relative path

                abs\_path = os.path.abspath(file\_path)

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

                    file\_path = abs\_path

        else:

            # If it's a string path

            file\_path = str(file)

            if not os.path.exists(file\_path):

                # Try to resolve relative path

                abs\_path = os.path.abspath(file\_path)

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

                    file\_path = abs\_path

                else:

                    # Try current working directory

                    cwd\_path = os.path.join(os.getcwd(), file\_path)

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

                        file\_path = cwd\_path

                    else:

                        raise FileNotFoundError(f"File not found: {file\_path}. Tried paths: {file\_path}, {abs\_path}, {cwd\_path}")

        logger.info(f"Reading file from path: {file\_path}")

        # Quick encoding check without reading entire file

        sample\_size = min(1024 \* 1024, os.path.getsize(file\_path))  # 1MB sample or file size

        # Read sample for encoding detection

        if hasattr(file, 'read'):

            raw\_sample = file.read(sample\_size)

            file.seek(0)

        else:

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

                raw\_sample = f.read(sample\_size)

        # Detect encoding

        try:

            import chardet

            detected = chardet.detect(raw\_sample)

            encoding = detected['encoding'] or 'utf-8'

        except:

            encoding = 'utf-8'

            logger.warning("Chardet detection failed, using default UTF-8 encoding")

        # Optimize reading for large files

        if check\_file\_size(file) > 3 \* 1024 \* 1024 \* 1024:  # 3GB

            logger.info("Large file detected, using optimized reading")

            # Get number of CPU cores while leaving some resources free

            n\_cores = max(1, os.cpu\_count() - 2)

            # Calculate optimal chunk size (aim for ~500MB per chunk)

            total\_size = os.path.getsize(file.name)

            chunk\_size = max(100000, min(1000000, total\_size // (500 \* 1024 \* 1024) + 1))

            # Initialize Dask DataFrame for out-of-memory processing

            import dask.dataframe as dd

            ddf = dd.read\_csv(

                file,

                delimiter=delimiter,

                encoding=encoding,

                on\_bad\_lines='skip',

                sample=100000,  # Sample rows for schema inference

                blocksize=chunk\_size \* 100,  # Larger blocks for fewer disk reads

                assume\_missing=True,  # Optimize for sparse data

                engine='c',  # Use faster C engine

                dtype\_backend='pandas',  # Use pandas dtypes for compatibility

                low\_memory=True

            )

            # Compute the final DataFrame using multiple cores

            return ddf.compute(scheduler='processes', num\_workers=n\_cores)

        else:

            # For smaller files, use pandas with optimized settings

            return pd.read\_csv(

                file,

                delimiter=delimiter,

                encoding=encoding,

                on\_bad\_lines='skip',

                engine='c',

                low\_memory=True,

                memory\_map=True,  # Memory mapping for better performance

                dtype\_backend='numpy'  # Use numpy backend for better performance

            )

    except Exception as e:

        logger.error(f"Error reading CSV/DAT file: {str(e)}")

        raise Exception(f"Failed to read file: {str(e)}")

def read\_sql(server, database, username=None, password=None, query=None):

    """

    Read data from SQL Server using a query.

    Args:

        server: SQL Server hostname

        database: Database name

        username: SQL Server username (optional for Windows auth)

        password: SQL Server password (optional for Windows auth)

        query: SQL query to execute

    Returns:

        pandas DataFrame

    """

    try:

        if not query:

            raise ValueError("Query cannot be empty")

        # Try different SQL Server drivers

        drivers = [

            'ODBC Driver 18 for SQL Server',

            'ODBC Driver 17 for SQL Server',

            'ODBC Driver 13 for SQL Server',

            'SQL Server',

            'SQL Server Native Client 11.0',

            'SQL Server Native Client 10.0'

        ]

        connection = None

        last\_error = None

        for driver in drivers:

            try:

                # Build connection string based on authentication method

                if username and password:

                    # SQL Server authentication

                    connection\_string = (

                        f"DRIVER={{{driver}}};"

                        f"SERVER={server};"

                        f"DATABASE={database};"

                        f"UID={username};"

                        f"PWD={password};"

                        "Trusted\_Connection=no;"

                    )

                else:

                    # Windows authentication

                    connection\_string = (

                        f"DRIVER={{{driver}}};"

                        f"SERVER={server};"

                        f"DATABASE={database};"

                        "Trusted\_Connection=yes;"

                        "TrustServerCertificate=yes;"

                    )

                connection = pyodbc.connect(connection\_string, timeout=30)

                logger.info(f"Successfully connected using driver: {driver}")

                break

            except Exception as e:

                last\_error = e

                logger.warning(f"Failed to connect using driver {driver}: {str(e)}")

                continue

        if connection is None:

            raise Exception(f"Failed to connect to SQL Server with any available driver. Last error: {str(last\_error)}")

        # Execute query in chunks

        chunks = []

        cursor = connection.cursor()

        # Execute the query

        cursor.execute(query)

        # Get column names

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

        # Fetch data in chunks

        while True:

            rows = cursor.fetchmany(500000)

            if not rows:

                break

            chunk\_df = pd.DataFrame.from\_records(rows, columns=columns)

            chunks.append(chunk\_df)

        cursor.close()

        connection.close()

        return pd.concat(chunks, ignore\_index=True) if chunks else pd.DataFrame()

    except Exception as e:

        logger.error(f"Error reading from SQL Server: {str(e)}")

        raise Exception(f"Failed to read from SQL Server: {str(e)}")

def read\_stored\_proc(server, database, username=None, password=None, proc\_name=None):

    """

    Execute a stored procedure and return results.

    Args:

        server: SQL Server hostname

        database: Database name

        username: SQL Server username (optional for Windows auth)

        password: SQL Server password (optional for Windows auth)

        proc\_name: Name of the stored procedure

    Returns:

        pandas DataFrame

    """

    try:

        if not proc\_name:

            raise ValueError("Stored procedure name cannot be empty")

        # Try different SQL Server drivers

        drivers = [

            'ODBC Driver 18 for SQL Server',

            'ODBC Driver 17 for SQL Server',

            'ODBC Driver 13 for SQL Server',

            'SQL Server',

            'SQL Server Native Client 11.0'

        ]

        connection = None

        last\_error = None

        for driver in drivers:

            try:

                # Build connection string based on authentication method

                if username and password:

                    # SQL Server authentication

                    connection\_string = (

                        f"DRIVER={{{driver}}};"

                        f"SERVER={server};"

                        f"DATABASE={database};"

                        f"UID={username};"

                        f"PWD={password};"

                        "Trusted\_Connection=no;"

                    )

                else:

                    # Windows authentication

                    connection\_string = (

                        f"DRIVER={{{driver}}};"

                        f"SERVER={server};"

                        f"DATABASE={database};"

                        "Trusted\_Connection=yes;"

                        "TrustServerCertificate=yes;"

                    )

                connection = pyodbc.connect(connection\_string, timeout=30)

                logger.info(f"Successfully connected using driver: {driver}")

                break

            except Exception as e:

                last\_error = e

                logger.warning(f"Failed to connect using driver {driver}: {str(e)}")

                continue

        if connection is None:

            raise Exception(f"Failed to connect to SQL Server with any available driver. Last error: {str(last\_error)}")

        try:

            cursor = connection.cursor()

            # Execute stored procedure

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

            # Fetch results

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

            results = []

            while True:

                rows = cursor.fetchmany(500000)  # Fetch in chunks

                if not rows:

                    break

                results.extend(rows)

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

        finally:

            cursor.close()

            connection.close()

    except Exception as e:

        logger.error(f"Error executing stored procedure: {str(e)}")

        raise Exception(f"Failed to execute stored procedure: {str(e)}")

def read\_teradata(server, database, username, password, query):

    """

    Read data from Teradata using a query.

    Args:

        server: Teradata server hostname

        database: Database name

        username: Teradata username

        password: Teradata password

        query: SQL query to execute

    Returns:

        pandas DataFrame

    """

    try:

        connection\_string = f"teradatasql://{username}:{password}@{server}/{database}"

        engine = create\_engine(connection\_string)

        # Execute query in chunks for large datasets

        chunks = []

        for chunk in pd.read\_sql(query, engine, chunksize=500000):

            chunks.append(chunk)

        return pd.concat(chunks, ignore\_index=True) if chunks else pd.DataFrame()

    except Exception as e:

        logger.error(f"Error reading from Teradata: {str(e)}")

        raise Exception(f"Failed to read from Teradata: {str(e)}")

def read\_api(url, method="GET", headers=None, body=None):

    """

    Read data from an API endpoint.

    Args:

        url: API endpoint URL

        method: HTTP method (GET or POST)

        headers: Request headers as dictionary

        body: Request body as dictionary

    Returns:

        pandas DataFrame

    """

    try:

        headers = json.loads(headers) if headers else {}

        body = json.loads(body) if body else {}

        if method.upper() == "GET":

            response = requests.get(url, headers=headers, params=body)

        else:

            response = requests.post(url, headers=headers, json=body)

        response.raise\_for\_status()

        data = response.json()

        # Handle different JSON structures

        if isinstance(data, list):

            return pd.DataFrame(data)

        elif isinstance(data, dict):

            # Try to find the data array in the response

            for key, value in data.items():

                if isinstance(value, list):

                    return pd.DataFrame(value)

            return pd.DataFrame([data])

        else:

            raise ValueError("Unexpected API response format")

    except Exception as e:

        logger.error(f"Error reading from API: {str(e)}")

        raise Exception(f"Failed to read from API: {str(e)}")

def read\_excel(file):

    """

    Read Excel file (xls/xlsx) into a pandas DataFrame.

    Args:

        file: File object or path

    Returns:

        pandas DataFrame

    """

    try:

        # Try to read with default settings first

        try:

            return pd.read\_excel(file)

        except Exception as first\_error:

            logger.warning(f"First attempt to read Excel failed: {str(first\_error)}")

            # If first attempt fails, try with more flexible settings

            file.seek(0)  # Reset file pointer

            try:

                return pd.read\_excel(

                    file,

                    engine='openpyxl',  # Try alternative engine

                    on\_bad\_lines='skip'  # Skip problematic rows

                )

            except Exception as second\_error:

                logger.error(f"Both attempts to read Excel failed. First error: {str(first\_error)}, Second error: {str(second\_error)}")

                raise Exception(f"Failed to read Excel file. Please check the file format. Error: {str(second\_error)}")

    except Exception as e:

        logger.error(f"Error reading Excel file: {str(e)}")

        raise Exception(f"Failed to read Excel file: {str(e)}")

def read\_parquet(file):

    """

    Read a Parquet file into a pandas DataFrame.

    Args:

        file: File object or path

    Returns:

        pandas DataFrame

    """

    try:

        return pd.read\_parquet(file)

    except Exception as e:

        logger.error(f"Error reading Parquet file: {str(e)}")

        raise Exception(f"Failed to read Parquet file: {str(e)}")

def read\_zipped\_files(zip\_file, delimiter=','):

    """

    Read flat files from a ZIP archive.

    Args:

        zip\_file: ZIP file object or path

        delimiter: Column separator for flat files

    Returns:

        pandas DataFrame

    """

    try:

        with zipfile.ZipFile(zip\_file) as z:

            # Get all file names in the ZIP

            flat\_files = [f for f in z.namelist() if f.endswith(('.csv', '.dat', '.txt'))]

            if not flat\_files:

                raise ValueError("No supported files found in ZIP archive")

            # Read and combine all files

            dfs = []

            for file in flat\_files:

                with z.open(file) as f:

                    # Convert to BytesIO for pandas to read

                    buffer = BytesIO(f.read())

                    df = read\_csv\_dat(buffer, delimiter=delimiter)

                    dfs.append(df)

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

    except Exception as e:

        logger.error(f"Error reading from ZIP archive: {str(e)}")

        raise Exception(f"Failed to read from ZIP archive: {str(e)}")