

Arquivo: database.py

```
from PyQt6.QtWidgets import *
import sqlite3
import logging
import pandas as pd
import re

class DatabaseATASManager:
    def __init__(self, db_path):
        self.db_path = str(db_path)
        self.connection = None

    def set_database_path(self, db_path):
        """Permite alterar dinamicamente o caminho do banco de dados."""
        self.db_path = db_path
        logging.info(f"Database path set to: {self.db_path}")

    def connect_to_database(self):
        if self.connection is None:
            self.connection = sqlite3.connect(self.db_path)
            logging.info(f"Conexão com o banco de dados aberta em {self.db_path}")
        return self.connection

    def close_connection(self):
        if self.connection:
            logging.info("Fechando conexão com o banco de dados...")
            self.connection.close()
            self.connection = None
            logging.info(f"Conexão com o banco de dados fechada em {self.db_path}")

    def is_closed(self):
        return self.connection is None

    def __enter__(self):
        self.connect_to_database()
        return self.connection

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close_connection()

    def save_dataframe(self, df, table_name):
        conn = self.connect_to_database()
        try:
            # Identifica itens duplicados na tabela e exclui-os
            cursor = conn.cursor()
            duplicados = df['item'].tolist()
            placeholders = ', '.join('?' for _ in duplicados)
            delete_query = f"DELETE FROM {table_name} WHERE item IN ({placeholders})"
            cursor.execute(delete_query, duplicados)

            # Insere o novo DataFrame após excluir os duplicados
        except Exception as e:
            logging.error(f"Error saving DataFrame to {table_name}: {e}")
            raise
```

```

        df.to_sql(table_name, conn, if_exists='append', index=False)
        logging.info(f"DataFrame salvo na tabela {table_name}.")
```

```

    except sqlite3.IntegrityError as e:
        valor_duplicado = df.loc[df.duplicated(subset=['item'], keep=False), 'item']
        mensagem_erro = f"Erro ao salvar o DataFrame: Valor duplicado(s) encontrado(s) na
        coluna 'item': {valor_duplicado.to_list()}."
        logging.error(mensagem_erro)
        QMessageBox.warning(None, "Erro de Duplicação", mensagem_erro)
```

```

    except sqlite3.Error as e:
        logging.error(f"Erro ao salvar DataFrame: {e}")
```

```

    finally:
        self.close_connection()
```

```

def delete_record(self, table_name, column, value):
    conn = self.connect_to_database()
    try:
        cursor = conn.cursor()
        cursor.execute(f"DELETE FROM {table_name} WHERE {column} = ?", (value,))
        conn.commit()
        logging.info(f"Registro deletado da tabela {table_name} onde {column} = {value}.")
    except sqlite3.Error as e:
        logging.error(f"Erro ao deletar registro: {e}")
    finally:
        self.close_connection()
```

```

def execute_query(self, query, params=None):
    conn = self.connect_to_database()
    try:
        cursor = conn.cursor()
        if params:
            cursor.execute(query, params)
        else:
            cursor.execute(query)
        conn.commit()
        return cursor.fetchall()
    except sqlite3.Error as e:
        logging.error(f"Erro ao executar consulta: {query}, Erro: {e}")
        return None
    finally:
        self.close_connection()
```

```

def execute_update(self, query, params=None):
    with self.connect_to_database() as conn:
        try:
            cursor = conn.cursor()
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            conn.commit()
        except sqlite3.Error as e:
            logging.error(f"Erro ao executar consulta: {query}, Erro: {e}")
```

```

        except sqlite3.Error as e:
            logging.error(f"Error executing update: {query}, Error: {e}")
            return False
        return True

def consultar_registro(self, tabela, campo, valor):
    """
    Consulta um registro na tabela especificada com base no campo e valor fornecidos.

    :param tabela: Nome da tabela no banco de dados.
    :param campo: Nome do campo a ser filtrado (ex: 'cnpj').
    :param valor: Valor a ser buscado no campo.
    :return: Dicionário com os dados do registro, ou None se não encontrado.
    """

    query = f"SELECT * FROM {tabela} WHERE {campo} = ?"
    print(f"Executando consulta: {query} com valor: {valor}")

    resultado = self.execute_query(query, (valor,))

    # Verifica se o resultado da consulta está vazio
    if resultado:
        print(f"Resultado da consulta encontrado: {resultado}")
        try:
            # Obtenção dos nomes das colunas para transformar o resultado em dicionário
            with self.connect_to_database() as conn:
                cursor = conn.cursor()
                cursor.execute(query, (valor,))
                colunas = [desc[0] for desc in cursor.description]

            print(f"Colunas da tabela: {colunas}")
            registro_dict = dict(zip(colunas, resultado[0]))
            print(f"Registro encontrado: {registro_dict}")
            return registro_dict

        except sqlite3.Error as e:
            print(f"Erro ao transformar o resultado em dicionário: {e}")
            logging.error(f"Erro ao transformar o resultado em dicionário: {e}")
            return None

    print(f"Nenhum registro encontrado para {campo} = {valor} na tabela {tabela}.")
    logging.info(f"Nenhum registro encontrado para {campo} = {valor} na tabela {tabela}.")
    return None

@staticmethod
def verify_and_create_columns(conn, table_name, required_columns):
    cursor = conn.cursor()
    cursor.execute(f"PRAGMA table_info({table_name})")
    existing_columns = {row[1]: row[2] for row in cursor.fetchall()}  # Storing column names and types

    # Criar uma lista das colunas na ordem correta e criar as colunas que faltam
    for column, column_type in required_columns.items():

```

```

if column not in existing_columns:
    # Assume a default type if not specified, e.g., TEXT
    cursor.execute(f"ALTER TABLE {table_name} ADD COLUMN {column} {column_type}")
    logging.info(f"Column {column} added to {table_name} with type {column_type}")
else:
    # Check if the type matches, if not, you might handle or log this situation
    if existing_columns[column] != column_type:
        logging.warning(f"Type mismatch for {column}: expected {column_type}, found
{existing_columns[column]}")

    conn.commit()
    logging.info(f"All required columns are verified/added in {table_name}")

def get_tables_with_keyword(self, keyword):
    """Retorna uma lista de tabelas que contêm o 'keyword' no nome."""
    try:
        with sqlite3.connect(self.db_path) as conn:
            cursor = conn.cursor()
            cursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND name LIKE
?", ('%' + keyword + '%'))
            tables = [row[0] for row in cursor.fetchall()]
        return tables
    except Exception as e:
        QMessageBox.warning(None, "Erro", f"Erro ao obter tabelas do banco de dados: {e}")
        return []

def load_table_to_dataframe(self, table_name):
    """Carrega a tabela especificada em um DataFrame."""
    try:
        with sqlite3.connect(self.db_path) as conn:
            df = pd.read_sql_query(f"SELECT * FROM [{table_name}]", conn)
        return df
    except Exception as e:
        QMessageBox.warning(None, "Erro", f"Erro ao carregar a tabela '{table_name}': {e}")
        return None

# Dentro da classe DatabaseATASManager, em database.py

def substituir_dados_controle_atas(self, df):
    """
    Substitui completamente os dados da tabela 'controle_atas' por um novo DataFrame.
    Esta operação é atômica: ou tudo funciona, ou nada é alterado.
    """
    try:
        with self.connect_to_database() as conn:
            cursor = conn.cursor()

            # 1. Apaga a tabela antiga
            cursor.execute("DROP TABLE IF EXISTS controle_atas")
            logging.info("Tabela 'controle_atas' antiga removida.")

            # 2. Recria a tabela com a estrutura correta
            cursor.execute(""""

```

```

CREATE TABLE controle_atas (
    grupo TEXT,
    item TEXT PRIMARY KEY,
    catalogo TEXT,
    descricao TEXT,
    descricao_detalhada TEXT,
    unidade TEXT,
    quantidade TEXT,
    valor_estimado TEXT,
    valor_homologado_item_unitario TEXT,
    percentual_desconto TEXT,
    valor_estimado_total_do_item TEXT,
    valor_homologado_total_item TEXT,
    marca_fabricante TEXT,
    modelo_versao TEXT,
    situacao TEXT,
    uasg TEXT,
    orgao_responsavel TEXT,
    num_pregao TEXT,
    ano_pregao TEXT,
    srp TEXT,
    objeto TEXT,
    melhor_lance TEXT,
    valor_negociado TEXT,
    ordenador_despesa TEXT,
    empresa TEXT,
    cnpj TEXT
)
"""

logging.info("Tabela 'controle_atas' recriada com sucesso.")

# 3. Insere os novos dados do DataFrame na tabela recém-criada
df.to_sql("controle_atas", conn, if_exists='append', index=False)
logging.info(f"{len(df)} registros inseridos em 'controle_atas'.")

conn.commit()
return True, "Dados carregados com sucesso!"

except Exception as e:
    logging.error(f"Falha ao substituir dados da tabela 'controle_atas': {e}")
    # Se houvesse uma transação, aqui ocorreria um rollback automático ao sair do 'with'.
    return False, f"Erro ao carregar dados: {e}"

def criar_tabela_itens_pregao(self, numeroCompra, anoCompra, unidadeOrgaoCodigoUnidade):
    table_name = f"resultAPI_{numeroCompra}_{anoCompra}_{unidadeOrgaoCodigoUnidade}"
    column_order = [
        'grupo', 'item PRIMARY KEY', 'catalogo', 'descricao', 'descricao_detalhada',
        'unidade', 'quantidade', 'valor_estimado',
        'valor_homologado_item_unitario', 'percentual_desconto',
        'valor_estimado_total_do_item',
        'valor_homologado_total_item', 'marca_fabricante', 'modelo_versao', 'situacao',
        'uasg', 'orgao_responsavel', 'num_pregao', 'ano_pregao', 'srp', 'objeto',
        'melhor_lance', 'valor_negociado', 'ordenador_despesa', 'empresa',
    ]

```

```
'cnpj', 'endereco', 'cep', 'municipio', 'telefone', 'email', 'responsavel_legal'  
]  
  
with self.connect_to_database() as conn:  
    cursor = conn.cursor()  
    columns_definition = ", ".join(column_order)  
    create_table_query = f"CREATE TABLE IF NOT EXISTS {table_name}"  
({columns_definition})"  
    cursor.execute(create_table_query)  
    conn.commit()
```