Python Connectors for SingleStore Databases

can you give python files to connect to cloud Singletore database wht are the pre-req ....what are the differenet options to connect singlestore database ......1) if we install singlestoredatabase on windows desktop 2) Cloud based Single store connect

I'll provide you with comprehensive information and Python files to connect to SingleStore databases. Here's everything you need:

**Pre-requisites**

1. **Python 3.7+** installed
2. **Required packages**:

bash

pip install pymysql sqlalchemy singlestoredb

**Option 1: Connect to Local SingleStore Database on Windows**

**1.1 Using PyMySQL**

python

*# connect\_singlestore\_local\_pymysql.py*

import pymysql

import pandas as pd

def connect\_local\_pymysql():

try:

connection = pymysql.connect(

host='localhost',

port=3306,

user='root',

password='your\_password',

database='your\_database'

)

print("Successfully connected to local SingleStore!")

*# Example query*

with connection.cursor() as cursor:

cursor.execute("SELECT 1 as test")

result = cursor.fetchone()

print(f"Test query result: {result}")

return connection

except Exception as e:

print(f"Error connecting to SingleStore: {e}")

return None

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

conn = connect\_local\_pymysql()

if conn:

conn.close()

**1.2 Using SQLAlchemy**

python

*# connect\_singlestore\_local\_sqlalchemy.py*

from sqlalchemy import create\_engine, text

import pandas as pd

def connect\_local\_sqlalchemy():

try:

*# Connection string format*

connection\_string = "mysql+pymysql://root:your\_password@localhost:3306/your\_database"

engine = create\_engine(connection\_string, pool\_pre\_ping=True)

*# Test connection*

with engine.connect() as conn:

result = conn.execute(text("SELECT 'Connected to SingleStore!' as status"))

print(result.fetchone()[0])

return engine

except Exception as e:

print(f"Error connecting to SingleStore: {e}")

return None

def execute\_query(engine, query):

"""Execute a query and return results as DataFrame"""

try:

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

return df

except Exception as e:

print(f"Error executing query: {e}")

return None

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

engine = connect\_local\_sqlalchemy()

if engine:

*# Example: Create a table*

create\_table\_query = """

CREATE TABLE IF NOT EXISTS sample\_data (

id INT PRIMARY KEY,

name VARCHAR(100),

value FLOAT,

created\_at TIMESTAMP DEFAULT NOW()

)

"""

with engine.connect() as conn:

conn.execute(text(create\_table\_query))

conn.commit()

print("Table created successfully!")

**Option 2: Connect to Cloud SingleStore Database**

**2.1 Using SingleStoreDB Python Connector (Recommended)**

python

*# connect\_singlestore\_cloud.py*

import singlestoredb as s2

import pandas as pd

from sqlalchemy import create\_engine, text

import os

def connect\_cloud\_singlestoredb():

"""

Connect to Cloud SingleStore using singlestoredb package

"""

try:

*# Get credentials from environment variables (recommended for security)*

host = os.getenv('SINGLESTORE\_HOST', 'your-host.singlestore.com')

port = int(os.getenv('SINGLESTORE\_PORT', '3306'))

user = os.getenv('SINGLESTORE\_USER', 'admin')

password = os.getenv('SINGLESTORE\_PASSWORD', 'your\_password')

database = os.getenv('SINGLESTORE\_DATABASE', 'your\_database')

connection = s2.connect(

host=host,

port=port,

user=user,

password=password,

database=database

)

print("Successfully connected to Cloud SingleStore!")

return connection

except Exception as e:

print(f"Error connecting to Cloud SingleStore: {e}")

return None

def connect\_cloud\_sqlalchemy():

"""

Connect using SQLAlchemy with SSL (recommended for cloud)

"""

try:

host = os.getenv('SINGLESTORE\_HOST', 'your-host.singlestore.com')

user = os.getenv('SINGLESTORE\_USER', 'admin')

password = os.getenv('SINGLESTORE\_PASSWORD', 'your\_password')

database = os.getenv('SINGLESTORE\_DATABASE', 'your\_database')

port = os.getenv('SINGLESTORE\_PORT', '3306')

*# Connection string with SSL*

connection\_string = (

f"mysql+pymysql://{user}:{password}@{host}:{port}/{database}"

"?ssl\_ca=/path/to/ca-certificate.crt"

"&ssl\_verify\_cert=true"

)

engine = create\_engine(connection\_string, pool\_pre\_ping=True)

*# Test connection*

with engine.connect() as conn:

result = conn.execute(text("SELECT 'Cloud SingleStore Connected!' as status"))

print(result.fetchone()[0])

return engine

except Exception as e:

print(f"Error connecting to Cloud SingleStore: {e}")

return None

def batch\_insert\_data(engine, data\_df, table\_name):

"""

Efficient batch insert using pandas

"""

try:

data\_df.to\_sql(

table\_name,

engine,

if\_exists='append',

index=False,

method='multi',

chunksize=1000

)

print(f"Successfully inserted {len(data\_df)} rows into {table\_name}")

except Exception as e:

print(f"Error inserting data: {e}")

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

*# Test cloud connection*

conn = connect\_cloud\_singlestoredb()

if conn:

conn.close()

*# Test SQLAlchemy connection*

engine = connect\_cloud\_sqlalchemy()

**2.2 Advanced Connection Manager with Error Handling**

python

*# singlestore\_manager.py*

import singlestoredb as s2

from sqlalchemy import create\_engine, text

import pandas as pd

import os

from contextlib import contextmanager

import time

from typing import Optional, Union

class SingleStoreManager:

def \_\_init\_\_(self, config: dict = None):

self.config = config or self.\_get\_default\_config()

self.engine = None

self.\_initialize\_engine()

def \_get\_default\_config(self) -> dict:

"""Get configuration from environment variables"""

return {

'host': os.getenv('SINGLESTORE\_HOST', 'localhost'),

'port': int(os.getenv('SINGLESTORE\_PORT', '3306')),

'user': os.getenv('SINGLESTORE\_USER', 'root'),

'password': os.getenv('SINGLESTORE\_PASSWORD', ''),

'database': os.getenv('SINGLESTORE\_DATABASE', 'test\_db'),

'use\_ssl': os.getenv('SINGLESTORE\_USE\_SSL', 'false').lower() == 'true'

}

def \_initialize\_engine(self):

"""Initialize SQLAlchemy engine with connection pooling"""

try:

connection\_string = self.\_build\_connection\_string()

self.engine = create\_engine(

connection\_string,

pool\_pre\_ping=True,

pool\_recycle=3600,

echo\_pool=True,

pool\_size=5,

max\_overflow=10

)

print("SingleStore engine initialized successfully")

except Exception as e:

print(f"Error initializing engine: {e}")

raise

def \_build\_connection\_string(self) -> str:

"""Build SQLAlchemy connection string"""

base\_string = f"mysql+pymysql://{self.config['user']}:{self.config['password']}@{self.config['host']}:{self.config['port']}/{self.config['database']}"

if self.config['use\_ssl']:

base\_string += "?ssl\_ca=/path/to/ca-certificate.crt&ssl\_verify\_cert=true"

return base\_string

@contextmanager

def get\_connection(self):

"""Context manager for database connections"""

connection = None

try:

connection = self.engine.connect()

yield connection

except Exception as e:

if connection:

connection.rollback()

print(f"Database error: {e}")

raise

finally:

if connection:

connection.close()

def execute\_query(self, query: str, params: dict = None) -> Optional[pd.DataFrame]:

"""Execute a query and return results as DataFrame"""

max\_retries = 3

for attempt in range(max\_retries):

try:

with self.get\_connection() as conn:

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

return df

except Exception as e:

if attempt == max\_retries - 1:

print(f"Failed to execute query after {max\_retries} attempts: {e}")

return None

time.sleep(2 \*\* attempt) *# Exponential backoff*

def execute\_command(self, command: str, params: dict = None) -> bool:

"""Execute a DDL or DML command"""

try:

with self.get\_connection() as conn:

if params:

conn.execute(text(command), params)

else:

conn.execute(text(command))

conn.commit()

return True

except Exception as e:

print(f"Error executing command: {e}")

return False

def batch\_insert(self, df: pd.DataFrame, table\_name: str, if\_exists: str = 'append') -> bool:

"""Batch insert DataFrame into table"""

try:

with self.get\_connection() as conn:

df.to\_sql(

table\_name,

conn,

if\_exists=if\_exists,

index=False,

method='multi',

chunksize=1000

)

print(f"Successfully inserted {len(df)} rows into {table\_name}")

return True

except Exception as e:

print(f"Error in batch insert: {e}")

return False

*# Usage example*

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

*# For cloud connection, set these environment variables:*

*# SINGLESTORE\_HOST, SINGLESTORE\_USER, SINGLESTORE\_PASSWORD, SINGLESTORE\_DATABASE*

db\_manager = SingleStoreManager()

*# Create a sample table*

create\_table\_sql = """

CREATE TABLE IF NOT EXISTS users (

id INT PRIMARY KEY,

username VARCHAR(50),

email VARCHAR(100),

created\_at TIMESTAMP DEFAULT NOW()

)

"""

if db\_manager.execute\_command(create\_table\_sql):

print("Table created successfully")

*# Insert sample data*

sample\_data = pd.DataFrame({

'id': [1, 2, 3],

'username': ['alice', 'bob', 'charlie'],

'email': ['alice@email.com', 'bob@email.com', 'charlie@email.com']

})

if db\_manager.batch\_insert(sample\_data, 'users'):

print("Data inserted successfully")

*# Query data*

users\_df = db\_manager.execute\_query("SELECT \* FROM users")

if users\_df is not None:

print(users\_df)

**Environment Configuration File**

Create a .env file for secure credential management:

bash

*# .env file*

SINGLESTORE\_HOST=your-host.singlestore.com

SINGLESTORE\_PORT=3306

SINGLESTORE\_USER=admin

SINGLESTORE\_PASSWORD=your\_secure\_password

SINGLESTORE\_DATABASE=your\_database

SINGLESTORE\_USE\_SSL=true

**Requirements File**

txt

# requirements.txt

pymysql==1.0.2

sqlalchemy==2.0.0

pandas==2.0.0

singlestoredb==1.0.0

python-dotenv==1.0.0

**Key Differences Between Local and Cloud Connections:**

1. **SSL/TLS**: Cloud connections typically require SSL
2. **Authentication**: Cloud uses username/password or API keys
3. **Network**: Cloud requires proper network configuration (whitelisted IPs)
4. **Connection Strings**: Different host formats and parameters

**Installation Commands:**

bash

*# Install all required packages*

pip install pymysql sqlalchemy pandas singlestoredb python-dotenv

*# Or using requirements file*

pip install -r requirements.txt

These Python files provide comprehensive connection options for both local and cloud SingleStore databases with proper error handling and best practices.

mysql -u "kemails2006" -h "svc-3482219c-a389-4079-b18b-d50662524e8a-shared-dml.aws-virginia-6.svc.singlestore.com" -P 3333 --default-auth=mysql\_native\_password --password='<password>' db\_kamlesh\_63c9e this is cloud singlestore can you python file to connect

Here's a Python file to connect to your Cloud SingleStore database using the connection details you provided:

**Method 1: Using PyMySQL (Direct Connection)**

python

*# connect\_singlestore\_cloud\_pymysql.py*

import pymysql

import pandas as pd

from typing import Optional, List, Tuple

import os

class SingleStoreCloudConnector:

def \_\_init\_\_(self):

self.connection = None

self.config = {

'host': 'svc-3482219c-a389-4079-b18b-d50662524e8a-shared-dml.aws-virginia-6.svc.singlestore.com',

'port': 3333,

'user': 'kemails2006',

'password': '<password>', *# Replace with actual password*

'database': 'db\_kamlesh\_63c9e',

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor

}

def connect(self) -> bool:

"""Establish connection to SingleStore Cloud"""

try:

self.connection = pymysql.connect(\*\*self.config)

print("✅ Successfully connected to SingleStore Cloud!")

return True

except Exception as e:

print(f"❌ Connection failed: {e}")

return False

def execute\_query(self, query: str, params: Optional[tuple] = None) -> Optional[List[dict]]:

"""Execute a SELECT query and return results"""

try:

with self.connection.cursor() as cursor:

cursor.execute(query, params)

result = cursor.fetchall()

return result

except Exception as e:

print(f"❌ Query execution failed: {e}")

return None

def execute\_command(self, command: str, params: Optional[tuple] = None) -> bool:

"""Execute INSERT, UPDATE, DELETE commands"""

try:

with self.connection.cursor() as cursor:

cursor.execute(command, params)

self.connection.commit()

print(f"✅ Command executed successfully. Rows affected: {cursor.rowcount}")

return True

except Exception as e:

print(f"❌ Command execution failed: {e}")

self.connection.rollback()

return False

def get\_dataframe(self, query: str, params: Optional[tuple] = None) -> Optional[pd.DataFrame]:

"""Execute query and return results as pandas DataFrame"""

try:

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

return df

except Exception as e:

print(f"❌ DataFrame creation failed: {e}")

return None

def close(self):

"""Close the database connection"""

if self.connection:

self.connection.close()

print("🔌 Connection closed.")

def test\_connection(self):

"""Test the database connection"""

try:

result = self.execute\_query("SELECT 'Connected to SingleStore Cloud!' as status, NOW() as timestamp")

if result:

print(f"🎯 Connection test successful: {result[0]}")

except Exception as e:

print(f"❌ Connection test failed: {e}")

*# Usage example*

def main():

*# Initialize connector*

db = SingleStoreCloudConnector()

try:

*# Connect to database*

if db.connect():

*# Test connection*

db.test\_connection()

*# Example: List tables in the database*

tables = db.execute\_query("""

SHOW TABLES

""")

if tables:

print("\n📊 Tables in database:")

for table in tables:

print(f" - {list(table.values())[0]}")

*# Example: Create a sample table if it doesn't exist*

create\_table\_sql = """

CREATE TABLE IF NOT EXISTS sample\_users (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(150) UNIQUE NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

"""

db.execute\_command(create\_table\_sql)

*# Example: Insert sample data*

insert\_sql = """

INSERT IGNORE INTO sample\_users (name, email)

VALUES (%s, %s)

"""

sample\_data = [

('John Doe', 'john.doe@email.com'),

('Jane Smith', 'jane.smith@email.com'),

('Bob Johnson', 'bob.johnson@email.com')

]

for data in sample\_data:

db.execute\_command(insert\_sql, data)

*# Example: Query data as DataFrame*

df = db.get\_dataframe("SELECT \* FROM sample\_users")

if df is not None:

print("\n📋 Sample Users Data:")

print(df)

*# Example: Execute custom query*

user\_count = db.execute\_query("SELECT COUNT(\*) as total\_users FROM sample\_users")

if user\_count:

print(f"\n👥 Total users: {user\_count[0]['total\_users']}")

finally:

*# Always close connection*

db.close()

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

main()

**Method 2: Using SQLAlchemy (Recommended for Production)**

python

*# connect\_singlestore\_sqlalchemy.py*

from sqlalchemy import create\_engine, text, MetaData, Table, Column, Integer, String, TIMESTAMP

from sqlalchemy.orm import sessionmaker

import pandas as pd

import os

class SingleStoreSQLAlchemy:

def \_\_init\_\_(self):

*# Build connection string*

self.connection\_string = (

"mysql+pymysql://kemails2006:<password>@"

"svc-3482219c-a389-4079-b18b-d50662524e8a-shared-dml.aws-virginia-6.svc.singlestore.com:"

"3333/db\_kamlesh\_63c9e"

)

*# Create engine with connection pooling*

self.engine = create\_engine(

self.connection\_string,

pool\_pre\_ping=True,

pool\_recycle=3600,

echo=False, *# Set to True for SQL query logging*

connect\_args={

'charset': 'utf8mb4'

}

)

*# Create session factory*

self.SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=self.engine)

def test\_connection(self):

"""Test database connection"""

try:

with self.engine.connect() as conn:

result = conn.execute(text("SELECT 'SQLAlchemy Connected!' as status, NOW() as timestamp"))

row = result.fetchone()

print(f"✅ {row['status']} at {row['timestamp']}")

return True

except Exception as e:

print(f"❌ Connection test failed: {e}")

return False

def execute\_raw\_sql(self, query: str, params: dict = None):

"""Execute raw SQL query"""

try:

with self.engine.connect() as conn:

if params:

result = conn.execute(text(query), params)

else:

result = conn.execute(text(query))

*# For SELECT queries, return results*

if query.strip().upper().startswith('SELECT'):

return result.fetchall()

else:

conn.commit()

return result.rowcount

except Exception as e:

print(f"❌ SQL execution failed: {e}")

return None

def get\_dataframe(self, query: str, params: dict = None) -> pd.DataFrame:

"""Get query results as pandas DataFrame"""

try:

df = pd.read\_sql(query, self.engine, params=params)

return df

except Exception as e:

print(f"❌ DataFrame creation failed: {e}")

return pd.DataFrame()

def batch\_insert\_dataframe(self, df: pd.DataFrame, table\_name: str, if\_exists: str = 'append'):

"""Insert DataFrame into database table"""

try:

df.to\_sql(

table\_name,

self.engine,

if\_exists=if\_exists,

index=False,

method='multi',

chunksize=1000

)

print(f"✅ Successfully inserted {len(df)} rows into {table\_name}")

except Exception as e:

print(f"❌ Batch insert failed: {e}")

*# Advanced usage with context manager*

from contextlib import contextmanager

@contextmanager

def get\_db\_session():

"""Context manager for database sessions"""

db = SingleStoreSQLAlchemy()

session = db.SessionLocal()

try:

yield session

session.commit()

except Exception as e:

session.rollback()

print(f"❌ Session error: {e}")

raise

finally:

session.close()

def main():

*# Initialize SQLAlchemy connection*

s2\_db = SingleStoreSQLAlchemy()

*# Test connection*

if s2\_db.test\_connection():

*# Get list of tables*

tables\_df = s2\_db.get\_dataframe("SHOW TABLES")

if not tables\_df.empty:

print("\n📊 Database Tables:")

print(tables\_df)

*# Example: Create and work with a products table*

create\_products\_table = """

CREATE TABLE IF NOT EXISTS products (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(200) NOT NULL,

price DECIMAL(10,2),

category VARCHAR(100),

in\_stock BOOLEAN DEFAULT TRUE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

"""

s2\_db.execute\_raw\_sql(create\_products\_table)

*# Insert sample products*

products\_data = [

{"name": "Laptop", "price": 999.99, "category": "Electronics", "in\_stock": True},

{"name": "Desk Chair", "price": 149.50, "category": "Furniture", "in\_stock": True},

{"name": "Coffee Mug", "price": 12.99, "category": "Kitchen", "in\_stock": False}

]

for product in products\_data:

s2\_db.execute\_raw\_sql(

"INSERT IGNORE INTO products (name, price, category, in\_stock) VALUES (:name, :price, :category, :in\_stock)",

product

)

*# Query products as DataFrame*

products\_df = s2\_db.get\_dataframe("SELECT \* FROM products ORDER BY created\_at DESC")

if not products\_df.empty:

print("\n🛍️ Products Data:")

print(products\_df)

*# Advanced analytics query*

analytics\_query = """

SELECT

category,

COUNT(\*) as product\_count,

AVG(price) as avg\_price,

SUM(CASE WHEN in\_stock THEN 1 ELSE 0 END) as in\_stock\_count

FROM products

GROUP BY category

ORDER BY product\_count DESC

"""

analytics\_df = s2\_db.get\_dataframe(analytics\_query)

if not analytics\_df.empty:

print("\n📈 Product Analytics:")

print(analytics\_df)

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

main()

**Method 3: Using Environment Variables (Secure Approach)**

Create a .env file:

bash

*# .env*

SINGLESTORE\_HOST=svc-3482219c-a389-4079-b18b-d50662524e8a-shared-dml.aws-virginia-6.svc.singlestore.com

SINGLESTORE\_PORT=3333

SINGLESTORE\_USER=kemails2006

SINGLESTORE\_PASSWORD=your\_actual\_password\_here

SINGLESTORE\_DATABASE=db\_kamlesh\_63c9e

python

*# connect\_singlestore\_env.py*

import pymysql

import pandas as pd

import os

from dotenv import load\_dotenv

*# Load environment variables*

load\_dotenv()

def connect\_with\_env():

"""Connect using environment variables"""

try:

connection = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

port=int(os.getenv('SINGLESTORE\_PORT', 3333)),

user=os.getenv('SINGLESTORE\_USER'),

password=os.getenv('SINGLESTORE\_PASSWORD'),

database=os.getenv('SINGLESTORE\_DATABASE'),

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor

)

print("✅ Connected using environment variables!")

return connection

except Exception as e:

print(f"❌ Connection failed: {e}")

return None

*# Quick test function*

def quick\_test():

"""Quick connection test"""

conn = connect\_with\_env()

if conn:

try:

with conn.cursor() as cursor:

cursor.execute("SELECT VERSION() as version, NOW() as current\_time")

result = cursor.fetchone()

print(f"SingleStore Version: {result['version']}")

print(f"Current Time: {result['current\_time']}")

finally:

conn.close()

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

quick\_test()

**Requirements File**

txt

# requirements.txt

pymysql==1.0.2

sqlalchemy==2.0.0

pandas==2.0.0

python-dotenv==1.0.0

**Installation & Usage**

bash

*# Install required packages*

pip install pymysql sqlalchemy pandas python-dotenv

*# Run the connection test*

python connect\_singlestore\_cloud\_pymysql.py

**Important Notes:**

1. **Replace**<password> with your actual password in the code
2. **Use environment variables** for production to keep passwords secure
3. **The connection uses port 3333** which is correct for SingleStore Cloud
4. **The host format** is typical for SingleStore's cloud service
5. **Consider using SSL** for production connections (add ssl={'ca': '/path/to/ca.pem'} to connection parameters)

Choose the method that best fits your needs. The SQLAlchemy approach is recommended for production applications due to better connection pooling and ORM capabilities.

sql alchemy could you give with .env conent

Here's a comprehensive SQLAlchemy setup with .env file configuration:

**.env File Content**

bash

*# .env*

*# SingleStore Cloud Database Configuration*

SINGLESTORE\_HOST=svc-3482219c-a389-4079-b18b-d50662524e8a-shared-dml.aws-virginia-6.svc.singlestore.com

SINGLESTORE\_PORT=3333

SINGLESTORE\_USER=kemails2006

SINGLESTORE\_PASSWORD=your\_actual\_password\_here

SINGLESTORE\_DATABASE=db\_kamlesh\_63c9e

*# Optional SSL Configuration*

SINGLESTORE\_SSL\_CA=/path/to/ca-certificate.pem

SINGLESTORE\_USE\_SSL=false

*# Connection Pool Settings*

SINGLESTORE\_POOL\_SIZE=5

SINGLESTORE\_MAX\_OVERFLOW=10

SINGLESTORE\_POOL\_RECYCLE=3600

**SQLAlchemy Connection Manager**

python

*# singlestore\_sqlalchemy\_env.py*

from sqlalchemy import create\_engine, text, MetaData, Table, Column, Integer, String, DateTime, Float, Boolean

from sqlalchemy.orm import sessionmaker, declarative\_base

from sqlalchemy.exc import SQLAlchemyError

from contextlib import contextmanager

import pandas as pd

import os

from dotenv import load\_dotenv

import logging

from typing import Optional, List, Dict, Any

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

*# SQLAlchemy Base*

Base = declarative\_base()

class SingleStoreDB:

def \_\_init\_\_(self):

self.engine = None

self.SessionLocal = None

self.metadata = MetaData()

self.\_setup\_engine()

def \_setup\_engine(self):

"""Setup SQLAlchemy engine with environment variables"""

try:

*# Get configuration from environment*

host = os.getenv('SINGLESTORE\_HOST')

port = os.getenv('SINGLESTORE\_PORT', 3333)

user = os.getenv('SINGLESTORE\_USER')

password = os.getenv('SINGLESTORE\_PASSWORD')

database = os.getenv('SINGLESTORE\_DATABASE')

*# Build connection URL*

connection\_url = f"mysql+pymysql://{user}:{password}@{host}:{port}/{database}"

*# SSL configuration*

connect\_args = {}

if os.getenv('SINGLESTORE\_USE\_SSL', 'false').lower() == 'true':

ssl\_ca = os.getenv('SINGLESTORE\_SSL\_CA')

if ssl\_ca:

connect\_args['ssl'] = {'ca': ssl\_ca}

*# Create engine with connection pooling*

self.engine = create\_engine(

connection\_url,

pool\_size=int(os.getenv('SINGLESTORE\_POOL\_SIZE', 5)),

max\_overflow=int(os.getenv('SINGLESTORE\_MAX\_OVERFLOW', 10)),

pool\_recycle=int(os.getenv('SINGLESTORE\_POOL\_RECYCLE', 3600)),

echo=False, *# Set to True for debug SQL logging*

connect\_args=connect\_args,

pool\_pre\_ping=True *# Verify connections before using them*

)

*# Create session factory*

self.SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=self.engine)

logger.info("✅ SQLAlchemy engine configured successfully")

except Exception as e:

logger.error(f"❌ Failed to setup engine: {e}")

raise

def test\_connection(self) -> bool:

"""Test database connection"""

try:

with self.engine.connect() as conn:

result = conn.execute(text("SELECT 1 as test\_value, NOW() as current\_time, VERSION() as db\_version"))

row = result.fetchone()

logger.info(f"✅ Connection test successful:")

logger.info(f" Test Value: {row['test\_value']}")

logger.info(f" Current Time: {row['current\_time']}")

logger.info(f" DB Version: {row['db\_version']}")

return True

except Exception as e:

logger.error(f"❌ Connection test failed: {e}")

return False

@contextmanager

def get\_session(self):

"""Context manager for database sessions"""

session = self.SessionLocal()

try:

yield session

session.commit()

except Exception as e:

session.rollback()

logger.error(f"❌ Session error: {e}")

raise

finally:

session.close()

def execute\_query(self, query: str, params: Dict = None) -> List[Dict[str, Any]]:

"""Execute a raw SQL query and return results"""

try:

with self.engine.connect() as conn:

if params:

result = conn.execute(text(query), params)

else:

result = conn.execute(text(query))

*# Return results as list of dictionaries*

return [dict(row.\_mapping) for row in result]

except SQLAlchemyError as e:

logger.error(f"❌ Query execution failed: {e}")

return []

def execute\_command(self, command: str, params: Dict = None) -> int:

"""Execute DML commands (INSERT, UPDATE, DELETE)"""

try:

with self.engine.connect() as conn:

if params:

result = conn.execute(text(command), params)

else:

result = conn.execute(text(command))

conn.commit()

logger.info(f"✅ Command executed. Rows affected: {result.rowcount}")

return result.rowcount

except SQLAlchemyError as e:

logger.error(f"❌ Command execution failed: {e}")

return 0

def get\_dataframe(self, query: str, params: Dict = None) -> pd.DataFrame:

"""Execute query and return results as pandas DataFrame"""

try:

df = pd.read\_sql(text(query), self.engine, params=params)

logger.info(f"✅ DataFrame created with {len(df)} rows")

return df

except Exception as e:

logger.error(f"❌ DataFrame creation failed: {e}")

return pd.DataFrame()

def batch\_insert\_dataframe(self, df: pd.DataFrame, table\_name: str, if\_exists: str = 'append', chunk\_size: int = 1000) -> bool:

"""Batch insert DataFrame into database table"""

try:

df.to\_sql(

table\_name,

self.engine,

if\_exists=if\_exists,

index=False,

method='multi',

chunksize=chunk\_size

)

logger.info(f"✅ Successfully inserted {len(df)} rows into {table\_name}")

return True

except Exception as e:

logger.error(f"❌ Batch insert failed: {e}")

return False

def get\_table\_info(self) -> pd.DataFrame:

"""Get information about all tables in the database"""

query = """

SELECT

table\_name,

table\_rows,

avg\_row\_length,

data\_length,

index\_length,

create\_time,

update\_time

FROM information\_schema.tables

WHERE table\_schema = :database

ORDER BY table\_name

"""

return self.get\_dataframe(query, {'database': os.getenv('SINGLESTORE\_DATABASE')})

def close(self):

"""Close database connections"""

if self.engine:

self.engine.dispose()

logger.info("🔌 Database connections closed")

*# Example ORM Models*

class User(Base):

\_\_tablename\_\_ = 'users'

id = Column(Integer, primary\_key=True, autoincrement=True)

username = Column(String(50), unique=True, nullable=False)

email = Column(String(100), unique=True, nullable=False)

full\_name = Column(String(100))

is\_active = Column(Boolean, default=True)

created\_at = Column(DateTime, server\_default=text('CURRENT\_TIMESTAMP'))

updated\_at = Column(DateTime, server\_default=text('CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP'))

class Product(Base):

\_\_tablename\_\_ = 'products'

id = Column(Integer, primary\_key=True, autoincrement=True)

name = Column(String(200), nullable=False)

description = Column(String(500))

price = Column(Float, nullable=False)

category = Column(String(100))

stock\_quantity = Column(Integer, default=0)

is\_available = Column(Boolean, default=True)

created\_at = Column(DateTime, server\_default=text('CURRENT\_TIMESTAMP'))

*# Usage Examples*

def main():

*# Initialize database connection*

db = SingleStoreDB()

try:

*# Test connection*

if not db.test\_connection():

return

*# Create tables using ORM*

Base.metadata.create\_all(bind=db.engine)

logger.info("✅ Tables created successfully")

*# Example 1: Using ORM with session*

with db.get\_session() as session:

*# Add sample user*

new\_user = User(

username='john\_doe',

email='john.doe@example.com',

full\_name='John Doe'

)

session.add(new\_user)

*# Session automatically commits when context manager exits*

*# Example 2: Using raw SQL queries*

*# Insert sample products*

products = [

{

'name': 'Laptop',

'description': 'High-performance laptop',

'price': 999.99,

'category': 'Electronics',

'stock\_quantity': 10

},

{

'name': 'Desk Chair',

'description': 'Ergonomic office chair',

'price': 249.99,

'category': 'Furniture',

'stock\_quantity': 15

},

{

'name': 'Coffee Mug',

'description': 'Ceramic coffee mug',

'price': 12.99,

'category': 'Kitchen',

'stock\_quantity': 50

}

]

for product in products:

db.execute\_command("""

INSERT INTO products (name, description, price, category, stock\_quantity)

VALUES (:name, :description, :price, :category, :stock\_quantity)

""", product)

*# Example 3: Using pandas DataFrame*

*# Query data as DataFrame*

users\_df = db.get\_dataframe("SELECT \* FROM users")

if not users\_df.empty:

print("\n👥 Users DataFrame:")

print(users\_df)

*# Example 4: Complex query with joins*

analytics\_query = """

SELECT

p.category,

COUNT(\*) as product\_count,

AVG(p.price) as avg\_price,

SUM(p.stock\_quantity) as total\_stock,

COUNT(u.id) as user\_count

FROM products p

CROSS JOIN users u

WHERE p.is\_available = TRUE

GROUP BY p.category

ORDER BY product\_count DESC

"""

analytics\_df = db.get\_dataframe(analytics\_query)

if not analytics\_df.empty:

print("\n📈 Analytics DataFrame:")

print(analytics\_df)

*# Example 5: Get table information*

tables\_info = db.get\_table\_info()

if not tables\_info.empty:

print("\n📊 Database Tables Info:")

print(tables\_info)

*# Example 6: Batch insert from DataFrame*

new\_products\_data = pd.DataFrame({

'name': ['Mouse', 'Keyboard', 'Monitor'],

'description': ['Wireless mouse', 'Mechanical keyboard', '4K Monitor'],

'price': [25.99, 89.99, 299.99],

'category': ['Electronics', 'Electronics', 'Electronics'],

'stock\_quantity': [30, 25, 8]

})

db.batch\_insert\_dataframe(new\_products\_data, 'products')

except Exception as e:

logger.error(f"❌ Main execution failed: {e}")

finally:

db.close()

*# Advanced usage with async support (if needed)*

class AsyncSingleStoreDB:

"""Async version for applications requiring async operations"""

def \_\_init\_\_(self):

self.engine = None

self.\_setup\_async\_engine()

def \_setup\_async\_engine(self):

"""Setup async SQLAlchemy engine"""

try:

from sqlalchemy.ext.asyncio import create\_async\_engine, AsyncSession

from sqlalchemy.orm import sessionmaker

host = os.getenv('SINGLESTORE\_HOST')

port = os.getenv('SINGLESTORE\_PORT', 3333)

user = os.getenv('SINGLESTORE\_USER')

password = os.getenv('SINGLESTORE\_PASSWORD')

database = os.getenv('SINGLESTORE\_DATABASE')

*# Async connection URL (using aiomysql)*

connection\_url = f"mysql+aiomysql://{user}:{password}@{host}:{port}/{database}"

self.engine = create\_async\_engine(

connection\_url,

echo=False,

pool\_pre\_ping=True

)

self.AsyncSessionLocal = sessionmaker(

self.engine, class\_=AsyncSession, expire\_on\_commit=False

)

logger.info("✅ Async SQLAlchemy engine configured successfully")

except ImportError:

logger.warning("⚠️ Async dependencies not installed. Install with: pip install aiomysql sqlalchemy[asyncio]")

except Exception as e:

logger.error(f"❌ Async engine setup failed: {e}")

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

main()

**Requirements File**

txt

# requirements.txt

sqlalchemy==2.0.0

pymysql==1.0.2

pandas==2.0.0

python-dotenv==1.0.0

**Installation & Usage**

bash

*# Install required packages*

pip install -r requirements.txt

*# Run the application*

python singlestore\_sqlalchemy\_env.py

**Key Features:**

1. **Environment Variable Configuration** - Secure credential management
2. **Connection Pooling** - Efficient database connections
3. **ORM Support** - SQLAlchemy declarative base for model definitions
4. **Context Managers** - Proper session handling
5. **Pandas Integration** - Easy DataFrame operations
6. **Comprehensive Logging** - Detailed operation tracking
7. **Error Handling** - Robust exception management
8. **Async Support** - Optional asynchronous operations

**Security Notes:**

* Never commit passwords to version control
* Use different .env files for development/production
* Consider using SSL certificates for production
* Regularly rotate database passwords

This setup provides a production-ready SQLAlchemy configuration for your SingleStore Cloud database with proper environment variable management.

(py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python singlestore\_sqlalchemy\_env.py INFO:\_\_main\_\_:✅ SQLAlchemy engine configured successfully ERROR:\_\_main\_\_:❌ Connection test failed: (pymysql.err.OperationalError) (1251, 'No SSL detected. Please make sure you enable SSL when connecting and our CA bundle found at: https://portal.singlestore.com/static/ca/singlestore\_bundle.pem\n(session ID: 461d49cd-d86c-4fd8-89a6-526e9b42baaf)') (Background on this error at: https://sqlalche.me/e/20/e3q8) INFO:\_\_main\_\_:🔌 Database connections closed

The error indicates that SSL is required for connecting to your SingleStore Cloud database. Here's the updated solution with SSL configuration:

**Updated .env File**

bash

*# .env*

*# SingleStore Cloud Database Configuration*

SINGLESTORE\_HOST=svc-3482219c-a389-4079-b18b-d50662524e8a-shared-dml.aws-virginia-6.svc.singlestore.com

SINGLESTORE\_PORT=3333

SINGLESTORE\_USER=kemails2006

SINGLESTORE\_PASSWORD=your\_actual\_password\_here

SINGLESTORE\_DATABASE=db\_kamlesh\_63c9e

*# SSL Configuration (REQUIRED for SingleStore Cloud)*

SINGLESTORE\_USE\_SSL=true

SINGLESTORE\_SSL\_VERIFY\_CERT=true

*# Connection Pool Settings*

SINGLESTORE\_POOL\_SIZE=5

SINGLESTORE\_MAX\_OVERFLOW=10

SINGLESTORE\_POOL\_RECYCLE=3600

**Updated Python File with SSL Support**

python

*# singlestore\_sqlalchemy\_ssl.py*

from sqlalchemy import create\_engine, text, MetaData, Table, Column, Integer, String, DateTime, Float, Boolean

from sqlalchemy.orm import sessionmaker, declarative\_base

from sqlalchemy.exc import SQLAlchemyError

from contextlib import contextmanager

import pandas as pd

import os

from dotenv import load\_dotenv

import logging

from typing import Optional, List, Dict, Any

import ssl

import pymysql

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

*# SQLAlchemy Base*

Base = declarative\_base()

class SingleStoreDB:

def \_\_init\_\_(self):

self.engine = None

self.SessionLocal = None

self.metadata = MetaData()

self.\_setup\_engine()

def \_get\_ssl\_context(self):

"""Create SSL context for SingleStore Cloud connection"""

try:

*# SingleStore requires SSL but often works with the system's default certs*

ssl\_context = ssl.create\_default\_context()

*# You can also download the SingleStore CA bundle if needed*

*# ssl\_context.load\_verify\_locations(cafile="path/to/singlestore\_bundle.pem")*

return ssl\_context

except Exception as e:

logger.warning(f"SSL context creation warning: {e}")

return None

def \_setup\_engine(self):

"""Setup SQLAlchemy engine with SSL configuration"""

try:

*# Get configuration from environment*

host = os.getenv('SINGLESTORE\_HOST')

port = os.getenv('SINGLESTORE\_PORT', 3333)

user = os.getenv('SINGLESTORE\_USER')

password = os.getenv('SINGLESTORE\_PASSWORD')

database = os.getenv('SINGLESTORE\_DATABASE')

use\_ssl = os.getenv('SINGLESTORE\_USE\_SSL', 'true').lower() == 'true'

*# Build connection URL*

connection\_url = f"mysql+pymysql://{user}:{password}@{host}:{port}/{database}"

*# SSL configuration for SingleStore Cloud*

connect\_args = {}

if use\_ssl:

ssl\_context = self.\_get\_ssl\_context()

if ssl\_context:

connect\_args['ssl'] = ssl\_context

else:

*# Fallback: use basic SSL without certificate verification*

connect\_args['ssl'] = {'ssl\_disabled': False}

*# Alternative SSL approach for pymysql*

connect\_args['ssl\_verify\_cert'] = os.getenv('SINGLESTORE\_SSL\_VERIFY\_CERT', 'true').lower() == 'true'

connect\_args['ssl\_verify\_identity'] = False

logger.info(f"🔐 SSL enabled: {use\_ssl}")

*# Create engine with connection pooling*

self.engine = create\_engine(

connection\_url,

pool\_size=int(os.getenv('SINGLESTORE\_POOL\_SIZE', 5)),

max\_overflow=int(os.getenv('SINGLESTORE\_MAX\_OVERFLOW', 10)),

pool\_recycle=int(os.getenv('SINGLESTORE\_POOL\_RECYCLE', 3600)),

echo=False,

connect\_args=connect\_args,

pool\_pre\_ping=True

)

*# Create session factory*

self.SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=self.engine)

logger.info("✅ SQLAlchemy engine configured successfully with SSL")

except Exception as e:

logger.error(f"❌ Failed to setup engine: {e}")

raise

def test\_connection(self) -> bool:

"""Test database connection"""

try:

with self.engine.connect() as conn:

result = conn.execute(text("SELECT 1 as test\_value, NOW() as current\_time, VERSION() as db\_version"))

row = result.fetchone()

logger.info(f"✅ Connection test successful:")

logger.info(f" Test Value: {row['test\_value']}")

logger.info(f" Current Time: {row['current\_time']}")

logger.info(f" DB Version: {row['db\_version']}")

return True

except Exception as e:

logger.error(f"❌ Connection test failed: {e}")

return False

*# ... [rest of the methods remain the same as previous example]*

@contextmanager

def get\_session(self):

"""Context manager for database sessions"""

session = self.SessionLocal()

try:

yield session

session.commit()

except Exception as e:

session.rollback()

logger.error(f"❌ Session error: {e}")

raise

finally:

session.close()

def execute\_query(self, query: str, params: Dict = None) -> List[Dict[str, Any]]:

"""Execute a raw SQL query and return results"""

try:

with self.engine.connect() as conn:

if params:

result = conn.execute(text(query), params)

else:

result = conn.execute(text(query))

return [dict(row.\_mapping) for row in result]

except SQLAlchemyError as e:

logger.error(f"❌ Query execution failed: {e}")

return []

def execute\_command(self, command: str, params: Dict = None) -> int:

"""Execute DML commands (INSERT, UPDATE, DELETE)"""

try:

with self.engine.connect() as conn:

if params:

result = conn.execute(text(command), params)

else:

result = conn.execute(text(command))

conn.commit()

logger.info(f"✅ Command executed. Rows affected: {result.rowcount}")

return result.rowcount

except SQLAlchemyError as e:

logger.error(f"❌ Command execution failed: {e}")

return 0

def get\_dataframe(self, query: str, params: Dict = None) -> pd.DataFrame:

"""Execute query and return results as pandas DataFrame"""

try:

df = pd.read\_sql(text(query), self.engine, params=params)

logger.info(f"✅ DataFrame created with {len(df)} rows")

return df

except Exception as e:

logger.error(f"❌ DataFrame creation failed: {e}")

return pd.DataFrame()

def close(self):

"""Close database connections"""

if self.engine:

self.engine.dispose()

logger.info("🔌 Database connections closed")

*# Alternative approach using direct PyMySQL with SSL*

def test\_direct\_pymysql\_connection():

"""Test direct PyMySQL connection with SSL"""

try:

connection = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

port=int(os.getenv('SINGLESTORE\_PORT', 3333)),

user=os.getenv('SINGLESTORE\_USER'),

password=os.getenv('SINGLESTORE\_PASSWORD'),

database=os.getenv('SINGLESTORE\_DATABASE'),

ssl={'ssl\_disabled': False}, *# Enable SSL*

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor

)

with connection.cursor() as cursor:

cursor.execute("SELECT 1 as test, NOW() as time")

result = cursor.fetchone()

print(f"✅ Direct PyMySQL connection successful: {result}")

connection.close()

return True

except Exception as e:

print(f"❌ Direct PyMySQL connection failed: {e}")

return False

*# Download SingleStore CA certificate (run this once)*

def download\_singlestore\_ca\_bundle():

"""Download SingleStore CA bundle if needed"""

import urllib.request

ca\_url = "https://portal.singlestore.com/static/ca/singlestore\_bundle.pem"

ca\_file = "singlestore\_bundle.pem"

if not os.path.exists(ca\_file):

try:

urllib.request.urlretrieve(ca\_url, ca\_file)

print(f"✅ Downloaded SingleStore CA bundle to {ca\_file}")

except Exception as e:

print(f"❌ Failed to download CA bundle: {e}")

return ca\_file

def main():

*# Option 1: Download CA bundle if needed*

*# ca\_file = download\_singlestore\_ca\_bundle()*

*# Option 2: Test direct connection first*

print("Testing direct PyMySQL connection...")

if test\_direct\_pymysql\_connection():

print("Direct connection successful! Proceeding with SQLAlchemy...")

*# Initialize database connection with SQLAlchemy*

db = SingleStoreDB()

try:

*# Test connection*

if db.test\_connection():

print("✅ SQLAlchemy connection successful!")

*# Example: List tables*

tables = db.execute\_query("SHOW TABLES")

if tables:

print("\n📊 Tables in database:")

for table in tables:

print(f" - {list(table.values())[0]}")

*# Example: Create a simple table*

create\_table\_sql = """

CREATE TABLE IF NOT EXISTS test\_connection (

id INT AUTO\_INCREMENT PRIMARY KEY,

message VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

"""

db.execute\_command(create\_table\_sql)

*# Insert test data*

db.execute\_command(

"INSERT INTO test\_connection (message) VALUES (%s)",

{"message": "Hello from SQLAlchemy with SSL!"}

)

*# Query test data*

results = db.execute\_query("SELECT \* FROM test\_connection ORDER BY created\_at DESC")

if results:

print("\n📝 Test data:")

for row in results:

print(f" ID: {row['id']}, Message: {row['message']}, Created: {row['created\_at']}")

except Exception as e:

logger.error(f"❌ Main execution failed: {e}")

finally:

db.close()

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

main()

**Alternative Simple SSL Solution**

If the above doesn't work, try this simplified version:

python

*# singlestore\_simple\_ssl.py*

from sqlalchemy import create\_engine, text

import os

from dotenv import load\_dotenv

import logging

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

def create\_engine\_with\_ssl():

"""Create SQLAlchemy engine with SSL for SingleStore"""

host = os.getenv('SINGLESTORE\_HOST')

port = os.getenv('SINGLESTORE\_PORT', 3333)

user = os.getenv('SINGLESTORE\_USER')

password = os.getenv('SINGLESTORE\_PASSWORD')

database = os.getenv('SINGLESTORE\_DATABASE')

*# Connection URL*

connection\_url = f"mysql+pymysql://{user}:{password}@{host}:{port}/{database}"

*# SSL configuration that works with SingleStore Cloud*

connect\_args = {

'ssl': {'ssl\_disabled': False}, *# Enable SSL*

'ssl\_verify\_cert': False, *# Disable certificate verification (if having CA issues)*

'ssl\_verify\_identity': False

}

engine = create\_engine(

connection\_url,

connect\_args=connect\_args,

pool\_pre\_ping=True,

echo=True *# Set to True to see SQL queries*

)

return engine

def main():

try:

engine = create\_engine\_with\_ssl()

*# Test connection*

with engine.connect() as conn:

result = conn.execute(text("SELECT 1 as test, NOW() as time, VERSION() as version"))

row = result.fetchone()

print(f"✅ Connection successful!")

print(f" Test: {row['test']}")

print(f" Time: {row['time']}")

print(f" Version: {row['version']}")

except Exception as e:

print(f"❌ Connection failed: {e}")

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

main()

**Updated Requirements**

txt

# requirements.txt

sqlalchemy==2.0.0

pymysql==1.0.2

pandas==2.0.0

python-dotenv==1.0.0

**Quick Fix**

If you need an immediate working solution, try this minimal version:

python

*# quick\_singlestore\_ssl.py*

from sqlalchemy import create\_engine, text

import os

from dotenv import load\_dotenv

load\_dotenv()

*# Simple connection with SSL*

engine = create\_engine(

f"mysql+pymysql://{os.getenv('SINGLESTORE\_USER')}:{os.getenv('SINGLESTORE\_PASSWORD')}@"

f"{os.getenv('SINGLESTORE\_HOST')}:{os.getenv('SINGLESTORE\_PORT')}/{os.getenv('SINGLESTORE\_DATABASE')}",

connect\_args={

'ssl': {'ssl\_disabled': False}

}

)

try:

with engine.connect() as conn:

result = conn.execute(text("SELECT 'Connected!' as status"))

print(result.fetchone())

print("✅ Success!")

except Exception as e:

print(f"❌ Error: {e}")

The key changes:

1. Added SSL configuration with 'ssl': {'ssl\_disabled': False}
2. Included SSL context creation
3. Added certificate verification options
4. Provided multiple approaches to handle SSL requirements

Run the simple version first to test the SSL connection, then use the comprehensive version for your application.

(py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python singlestore\_sqlalchemy\_env.py Testing direct PyMySQL connection... ❌ Direct PyMySQL connection failed: (1045, "Access denied for user 'kemails2006'@'10.4.168.138' (using password: YES)") INFO:\_\_main\_\_:🔐 SSL enabled: True INFO:\_\_main\_\_:✅ SQLAlchemy engine configured successfully with SSL ERROR:\_\_main\_\_:❌ Connection test failed: (pymysql.err.OperationalError) (1045, "Access denied for user 'kemails2006'@'10.4.168.138' (using password: YES)") (Background on this error at: https://sqlalche.me/e/20/e3q8) INFO:\_\_main\_\_:🔌 Database connections closed (py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python simple\_singleStore.py ❌ Connection failed: (pymysql.err.OperationalError) (1045, "Access denied for user 'kemails2006'@'10.4.168.138' (using password: YES)") (Background on this error at: https://sqlalche.me/e/20/e3q8)

The error indicates an **access denied** issue, which means the credentials are being rejected. Let's troubleshoot this step by step:

**Step 1: Verify Your Credentials**

First, let's create a diagnostic script to test different connection scenarios:

python

*# diagnose\_connection.py*

import pymysql

from sqlalchemy import create\_engine, text

import os

from dotenv import load\_dotenv

load\_dotenv()

def test\_connection\_details():

"""Test connection with detailed error reporting"""

config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False}

}

print("🔍 Testing connection with these details:")

print(f" Host: {config['host']}")

print(f" Port: {config['port']}")

print(f" User: {config['user']}")

print(f" Database: {config['database']}")

print(f" Password length: {len(config['password']) if config['password'] else 0} characters")

print(f" SSL: Enabled")

try:

connection = pymysql.connect(\*\*config)

print("✅ SUCCESS: Connected to SingleStore!")

connection.close()

return True

except pymysql.err.OperationalError as e:

print(f"❌ OperationalError: {e}")

print("\n💡 Troubleshooting tips:")

print("1. Check if your password is correct")

print("2. Verify your username is correct")

print("3. Ensure your IP is whitelisted in SingleStore portal")

print("4. Check if database exists")

return False

except pymysql.err.InternalError as e:

print(f"❌ InternalError: {e}")

return False

except Exception as e:

print(f"❌ Unexpected error: {e}")

return False

def test\_without\_database():

"""Test connection without specifying database"""

try:

connection = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

port=int(os.getenv('SINGLESTORE\_PORT', 3333)),

user=os.getenv('SINGLESTORE\_USER'),

password=os.getenv('SINGLESTORE\_PASSWORD'),

ssl={'ssl\_disabled': False}

)

print("✅ SUCCESS: Connected without database specified!")

connection.close()

return True

except Exception as e:

print(f"❌ Failed without database: {e}")

return False

def test\_mysql\_native\_password():

"""Test with mysql\_native\_password authentication"""

try:

connection = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

port=int(os.getenv('SINGLESTORE\_PORT', 3333)),

user=os.getenv('SINGLESTORE\_USER'),

password=os.getenv('SINGLESTORE\_PASSWORD'),

database=os.getenv('SINGLESTORE\_DATABASE'),

ssl={'ssl\_disabled': False},

auth\_plugin='mysql\_native\_password'

)

print("✅ SUCCESS: Connected with mysql\_native\_password!")

connection.close()

return True

except Exception as e:

print(f"❌ Failed with mysql\_native\_password: {e}")

return False

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

print("🚀 SingleStore Connection Diagnostic Tool")

print("=" \* 50)

*# Test 1: Basic connection*

print("\n1. Testing basic connection...")

test\_connection\_details()

*# Test 2: Connection without database*

print("\n2. Testing connection without database...")

test\_without\_database()

*# Test 3: With mysql\_native\_password*

print("\n3. Testing with mysql\_native\_password...")

test\_mysql\_native\_password()

**Step 2: Check Your .env File**

Make sure your .env file has the **exact** values:

bash

*# .env - DOUBLE CHECK THESE VALUES*

SINGLESTORE\_HOST=svc-3482219c-a389-4079-b18b-d50662524e8a-shared-dml.aws-virginia-6.svc.singlestore.com

SINGLESTORE\_PORT=3333

SINGLESTORE\_USER=kemails2006

SINGLESTORE\_PASSWORD=your\_actual\_password\_here

SINGLESTORE\_DATABASE=db\_kamlesh\_63c9e

SINGLESTORE\_USE\_SSL=true

**Step 3: Common Solutions**

**Solution A: IP Whitelisting**

The error shows your IP (10.4.168.138) is being denied. You need to whitelist your IP in the SingleStore portal:

python

*# check\_ip.py*

import requests

def get\_public\_ip():

"""Get your public IP address"""

try:

response = requests.get('https://httpbin.org/ip', timeout=5)

ip = response.json()['origin']

print(f"🌐 Your public IP address: {ip}")

print("💡 Add this IP to your SingleStore workspace whitelist")

return ip

except:

print("❌ Could not determine public IP")

return None

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

get\_public\_ip()

**Solution B: Password Verification**

python

*# password\_test.py*

import getpass

from dotenv import load\_dotenv

import os

load\_dotenv()

def verify\_password():

actual\_password = os.getenv('SINGLESTORE\_PASSWORD')

print(f"Password from .env: {'\*' \* len(actual\_password) if actual\_password else 'NOT SET'}")

*# Test manual entry*

manual\_password = getpass.getpass("Enter password manually: ")

if manual\_password == actual\_password:

print("✅ Passwords match!")

else:

print("❌ Passwords DO NOT match!")

print("💡 Check for:")

print(" - Extra spaces in .env file")

print(" - Special characters that need escaping")

print(" - Case sensitivity")

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

verify\_password()

**Step 4: Working Connection Script with Error Handling**

python

*# working\_singlestore.py*

import pymysql

from sqlalchemy import create\_engine, text

import os

from dotenv import load\_dotenv

load\_dotenv()

def create\_connection():

"""Create a connection with comprehensive error handling"""

*# Get credentials*

host = os.getenv('SINGLESTORE\_HOST')

user = os.getenv('SINGLESTORE\_USER')

password = os.getenv('SINGLESTORE\_PASSWORD')

database = os.getenv('SINGLESTORE\_DATABASE')

port = int(os.getenv('SINGLESTORE\_PORT', 3333))

print("Attempting connection with:")

print(f" Host: {host}")

print(f" User: {user}")

print(f" Database: {database}")

print(f" Port: {port}")

try:

*# Method 1: Direct PyMySQL*

print("\n1. Trying direct PyMySQL connection...")

connection = pymysql.connect(

host=host,

port=port,

user=user,

password=password,

database=database,

ssl={'ssl\_disabled': False},

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor

)

print("✅ PyMySQL connection successful!")

*# Test query*

with connection.cursor() as cursor:

cursor.execute("SELECT NOW() as current\_time, VERSION() as version")

result = cursor.fetchone()

print(f" Current Time: {result['current\_time']}")

print(f" Version: {result['version']}")

connection.close()

return True

except pymysql.err.OperationalError as e:

error\_code = e.args[0]

error\_message = e.args[1]

print(f"❌ Operational Error {error\_code}: {error\_message}")

if error\_code == 1045:

print("\n🔐 Access Denied Solutions:")

print(" A. Check your password in the .env file")

print(" B. Verify your username is correct")

print(" C. Whitelist your IP in SingleStore portal")

print(" D. Check if user has access to the database")

elif error\_code == 1044:

print("\n🗄️ Database Access Error:")

print(" The user doesn't have access to this database")

print(" Try connecting without specifying a database")

elif error\_code == 2003:

print("\n🌐 Network Error:")

print(" Cannot connect to the host")

print(" Check hostname and port")

return False

except Exception as e:

print(f"❌ Unexpected error: {e}")

return False

def test\_sqlalchemy\_connection():

"""Test SQLAlchemy connection"""

try:

connection\_string = (

f"mysql+pymysql://{os.getenv('SINGLESTORE\_USER')}:{os.getenv('SINGLESTORE\_PASSWORD')}"

f"@{os.getenv('SINGLESTORE\_HOST')}:{os.getenv('SINGLESTORE\_PORT')}"

f"/{os.getenv('SINGLESTORE\_DATABASE')}"

)

engine = create\_engine(

connection\_string,

connect\_args={'ssl': {'ssl\_disabled': False}},

echo=False

)

with engine.connect() as conn:

result = conn.execute(text("SELECT 'SQLAlchemy Connected!' as status"))

print(f"✅ {result.fetchone()['status']}")

return True

except Exception as e:

print(f"❌ SQLAlchemy connection failed: {e}")

return False

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

print("🚀 SingleStore Connection Test")

print("=" \* 40)

if create\_connection():

print("\n" + "=" \* 40)

print("Testing SQLAlchemy connection...")

test\_sqlalchemy\_connection()

**Step 5: Quick Fix Checklist**

Run this diagnostic:

bash

*# Install required package if needed*

pip install requests

*# Run diagnostics*

python diagnose\_connection.py

python check\_ip.py

python password\_test.py

python working\_singlestore.py

**Most Likely Issues:**

1. **Incorrect Password**: Double-check your password in the SingleStore portal
2. **IP Not Whitelisted**: Your IP 10.4.168.138 needs to be added to the whitelist
3. **User Permissions**: The user might not have access to the specific database

**To Whitelist Your IP:**

1. Go to SingleStore Portal
2. Navigate to your workspace
3. Find "IP Access Rules" or "Whitelist"
4. Add your IP address: 10.4.168.138
5. Save changes

Run the diagnostic scripts above and they should tell you exactly what's wrong. The most common issue is IP whitelisting for cloud databases.

(py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python simple\_singleStore.py ❌ Connection failed: encoding with 'idna' codec failed (UnicodeError: label empty or too long) (py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python singlestore\_sqlalchemy\_env.py Testing direct PyMySQL connection... ✅ Direct PyMySQL connection successful: {'test': 1, 'time': datetime.datetime(2025, 10, 14, 23, 41)} Direct connection successful! Proceeding with SQLAlchemy... INFO:\_\_main\_\_:🔐 SSL enabled: True INFO:\_\_main\_\_:✅ SQLAlchemy engine configured successfully with SSL ERROR:\_\_main\_\_:❌ Connection test failed: encoding with 'idna' codec failed (UnicodeError: label empty or too long) INFO:\_\_main\_\_:🔌 Database connections closed

The error encoding with 'idna' codec failed (UnicodeError: label empty or too long) is a known issue with SQLAlchemy and long hostnames. Here are the solutions:

**Solution 1: Use PyMySQL Directly (Working Approach)**

Since PyMySQL works directly, let's create a wrapper that uses PyMySQL but provides SQLAlchemy-like functionality:

python

*# singlestore\_pymysql\_wrapper.py*

import pymysql

import pandas as pd

from typing import List, Dict, Any, Optional

from contextlib import contextmanager

import os

from dotenv import load\_dotenv

import logging

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

class SingleStoreDB:

def \_\_init\_\_(self):

self.connection\_config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor,

'autocommit': True

}

@contextmanager

def get\_connection(self):

"""Context manager for database connections"""

connection = None

try:

connection = pymysql.connect(\*\*self.connection\_config)

yield connection

except Exception as e:

logger.error(f"Connection error: {e}")

raise

finally:

if connection:

connection.close()

def execute\_query(self, query: str, params: Optional[tuple] = None) -> List[Dict[str, Any]]:

"""Execute a SELECT query and return results"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(query, params)

result = cursor.fetchall()

logger.info(f"Query executed successfully. Rows returned: {len(result)}")

return result

except Exception as e:

logger.error(f"Query execution failed: {e}")

return []

def execute\_command(self, command: str, params: Optional[tuple] = None) -> int:

"""Execute INSERT, UPDATE, DELETE commands"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(command, params)

rows\_affected = cursor.rowcount

logger.info(f"Command executed successfully. Rows affected: {rows\_affected}")

return rows\_affected

except Exception as e:

logger.error(f"Command execution failed: {e}")

return 0

def get\_dataframe(self, query: str, params: Optional[tuple] = None) -> pd.DataFrame:

"""Execute query and return results as pandas DataFrame"""

try:

with self.get\_connection() as conn:

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

logger.info(f"DataFrame created with {len(df)} rows")

return df

except Exception as e:

logger.error(f"DataFrame creation failed: {e}")

return pd.DataFrame()

def batch\_insert\_dataframe(self, df: pd.DataFrame, table\_name: str) -> bool:

"""Batch insert DataFrame into database table"""

try:

if df.empty:

logger.warning("DataFrame is empty, nothing to insert")

return True

*# Create placeholders for the insert query*

columns = ', '.join(df.columns)

placeholders = ', '.join(['%s'] \* len(df.columns))

insert\_query = f"INSERT INTO {table\_name} ({columns}) VALUES ({placeholders})"

with self.get\_connection() as conn:

with conn.cursor() as cursor:

*# Convert DataFrame to list of tuples*

data\_tuples = [tuple(row) for row in df.to\_numpy()]

cursor.executemany(insert\_query, data\_tuples)

logger.info(f"Successfully inserted {len(df)} rows into {table\_name}")

return True

except Exception as e:

logger.error(f"Batch insert failed: {e}")

return False

def test\_connection(self) -> bool:

"""Test the database connection"""

try:

result = self.execute\_query("SELECT 1 as test\_value, NOW() as current\_time, VERSION() as db\_version")

if result:

row = result[0]

logger.info(f"✅ Connection test successful:")

logger.info(f" Test Value: {row['test\_value']}")

logger.info(f" Current Time: {row['current\_time']}")

logger.info(f" DB Version: {row['db\_version']}")

return True

return False

except Exception as e:

logger.error(f"❌ Connection test failed: {e}")

return False

*# Usage examples*

def main():

db = SingleStoreDB()

*# Test connection*

if db.test\_connection():

print("✅ Database connection successful!")

*# List all tables*

tables = db.execute\_query("SHOW TABLES")

if tables:

print("\n📊 Tables in database:")

for table in tables:

print(f" - {list(table.values())[0]}")

*# Create a sample table*

create\_table\_sql = """

CREATE TABLE IF NOT EXISTS sample\_data (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(150) UNIQUE,

age INT,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

"""

db.execute\_command(create\_table\_sql)

print("✅ Sample table created/verified")

*# Insert sample data*

insert\_sql = """

INSERT IGNORE INTO sample\_data (name, email, age)

VALUES (%s, %s, %s)

"""

sample\_records = [

('John Doe', 'john.doe@example.com', 30),

('Jane Smith', 'jane.smith@example.com', 25),

('Bob Johnson', 'bob.johnson@example.com', 35)

]

for record in sample\_records:

db.execute\_command(insert\_sql, record)

print("✅ Sample data inserted")

*# Query data as DataFrame*

df = db.get\_dataframe("SELECT \* FROM sample\_data ORDER BY created\_at DESC")

if not df.empty:

print("\n📋 Sample Data:")

print(df)

*# Batch insert from DataFrame*

new\_data = pd.DataFrame({

'name': ['Alice Brown', 'Charlie Wilson'],

'email': ['alice@example.com', 'charlie@example.com'],

'age': [28, 42]

})

db.batch\_insert\_dataframe(new\_data, 'sample\_data')

*# Final query to show all data*

final\_df = db.get\_dataframe("SELECT \* FROM sample\_data")

if not final\_df.empty:

print(f"\n🎯 Total records in sample\_data: {len(final\_df)}")

print(final\_df)

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

main()

**Solution 2: Fix SQLAlchemy IDNA Issue**

If you prefer SQLAlchemy, here's the fix for the IDNA encoding issue:

python

*# singlestore\_sqlalchemy\_fixed.py*

from sqlalchemy import create\_engine, text

import pandas as pd

import os

from dotenv import load\_dotenv

import logging

from urllib.parse import quote\_plus

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

def create\_singlestore\_engine():

"""Create SQLAlchemy engine with IDNA fix"""

*# Get credentials*

host = os.getenv('SINGLESTORE\_HOST')

user = os.getenv('SINGLESTORE\_USER')

password = os.getenv('SINGLESTORE\_PASSWORD')

database = os.getenv('SINGLESTORE\_DATABASE')

port = os.getenv('SINGLESTORE\_PORT', 3333)

*# FIX: Use the IP address instead of hostname to avoid IDNA issue*

*# You can find the IP by pinging the hostname or use the original approach below*

*# Alternative FIX: URL encode the hostname*

encoded\_host = quote\_plus(host) if host else host

*# Connection string with encoded host*

connection\_string = f"mysql+pymysql://{user}:{password}@{encoded\_host}:{port}/{database}"

logger.info(f"Using encoded host: {encoded\_host}")

try:

engine = create\_engine(

connection\_string,

connect\_args={

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4'

},

pool\_pre\_ping=True,

echo=False *# Set to True for SQL logging*

)

return engine

except Exception as e:

logger.error(f"Engine creation failed: {e}")

return None

def test\_connection\_simple():

"""Simple test using PyMySQL approach"""

import pymysql

try:

connection = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

port=int(os.getenv('SINGLESTORE\_PORT', 3333)),

user=os.getenv('SINGLESTORE\_USER'),

password=os.getenv('SINGLESTORE\_PASSWORD'),

database=os.getenv('SINGLESTORE\_DATABASE'),

ssl={'ssl\_disabled': False},

charset='utf8mb4',

cursorclass=pymysql.cursors.DictCursor

)

with connection.cursor() as cursor:

cursor.execute("SELECT NOW() as time, VERSION() as version")

result = cursor.fetchone()

print(f"✅ Direct connection: {result}")

connection.close()

return True

except Exception as e:

print(f"❌ Direct connection failed: {e}")

return False

def main():

print("🚀 Testing SingleStore Connection")

print("=" \* 50)

*# Test direct connection first*

print("1. Testing direct PyMySQL connection...")

if test\_connection\_simple():

print("✅ PyMySQL connection works!")

else:

print("❌ PyMySQL connection failed")

return

*# Test SQLAlchemy with fix*

print("\n2. Testing SQLAlchemy with IDNA fix...")

engine = create\_singlestore\_engine()

if engine:

try:

with engine.connect() as conn:

result = conn.execute(text("SELECT 'SQLAlchemy Connected!' as status"))

row = result.fetchone()

print(f"✅ {row['status']}")

*# Show tables*

tables = conn.execute(text("SHOW TABLES"))

table\_list = [list(row)[0] for row in tables]

print(f"📊 Tables in database: {table\_list}")

except Exception as e:

print(f"❌ SQLAlchemy connection failed: {e}")

*# Fallback to PyMySQL wrapper*

print("\n🔄 Falling back to PyMySQL wrapper...")

from singlestore\_pymysql\_wrapper import SingleStoreDB

db = SingleStoreDB()

if db.test\_connection():

print("✅ PyMySQL wrapper connection successful!")

*# Use the wrapper for operations*

tables = db.execute\_query("SHOW TABLES")

if tables:

print("📊 Tables in database:")

for table in tables:

print(f" - {list(table.values())[0]}")

else:

print("❌ Could not create SQLAlchemy engine")

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

main()

**Solution 3: Simple Working Version**

python

*# simple\_working\_singlestore.py*

import pymysql

import pandas as pd

from dotenv import load\_dotenv

import os

load\_dotenv()

class SimpleSingleStore:

def \_\_init\_\_(self):

self.config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor

}

def query(self, sql, params=None):

"""Execute query and return results"""

conn = pymysql.connect(\*\*self.config)

try:

with conn.cursor() as cursor:

cursor.execute(sql, params)

result = cursor.fetchall()

return result

finally:

conn.close()

def execute(self, sql, params=None):

"""Execute command (INSERT, UPDATE, DELETE)"""

conn = pymysql.connect(\*\*self.config)

try:

with conn.cursor() as cursor:

cursor.execute(sql, params)

conn.commit()

return cursor.rowcount

except:

conn.rollback()

raise

finally:

conn.close()

def get\_df(self, sql, params=None):

"""Get results as pandas DataFrame"""

conn = pymysql.connect(\*\*self.config)

try:

return pd.read\_sql(sql, conn, params=params)

finally:

conn.close()

*# Usage*

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

db = SimpleSingleStore()

*# Test connection*

result = db.query("SELECT NOW() as time, VERSION() as version")

print("✅ Connection successful!")

print(f"Time: {result[0]['time']}")

print(f"Version: {result[0]['version']}")

*# Show tables*

tables = db.query("SHOW TABLES")

print(f"\n📊 Tables: {[list(t.values())[0] for t in tables]}")

*# Example: Create and query a table*

db.execute("""

CREATE TABLE IF NOT EXISTS test\_pymysql (

id INT AUTO\_INCREMENT PRIMARY KEY,

message VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

)

""")

db.execute("INSERT INTO test\_pymysql (message) VALUES (%s)", ("Hello from PyMySQL!",))

df = db.get\_df("SELECT \* FROM test\_pymysql")

print(f"\n📝 Test table data:")

print(df)

**Requirements**

txt

# requirements.txt

pymysql==1.0.2

pandas==2.0.0

python-dotenv==1.0.0

sqlalchemy==2.0.0

**Quick Start**

1. **Use the PyMySQL wrapper** (Solution 1) - it's the most reliable
2. Run the simple version to test:

bash

python simple\_working\_singlestore.py

The PyMySQL wrapper provides all the functionality you need without the SQLAlchemy IDNA issue. It's production-ready and handles connection management properly.

(py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python singlestore\_pymysql\_wrapper.py ERROR:\_\_main\_\_:Connection error: (1064, "You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'current\_time, VERSION() as db\_version' at line 1") ERROR:\_\_main\_\_:Query execution failed: (1064, "You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'current\_time, VERSION() as db\_version' at line 1")

The error indicates a SQL syntax issue. SingleStore has some differences from standard MySQL. Let me provide the corrected version:

**Fixed Version - SingleStore Compatible SQL**

python

*# singlestore\_pymysql\_wrapper\_fixed.py*

import pymysql

import pandas as pd

from typing import List, Dict, Any, Optional

from contextlib import contextmanager

import os

from dotenv import load\_dotenv

import logging

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

class SingleStoreDB:

def \_\_init\_\_(self):

self.connection\_config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor,

'autocommit': True

}

@contextmanager

def get\_connection(self):

"""Context manager for database connections"""

connection = None

try:

connection = pymysql.connect(\*\*self.connection\_config)

yield connection

except Exception as e:

logger.error(f"Connection error: {e}")

raise

finally:

if connection:

connection.close()

def execute\_query(self, query: str, params: Optional[tuple] = None) -> List[Dict[str, Any]]:

"""Execute a SELECT query and return results"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(query, params)

result = cursor.fetchall()

logger.info(f"Query executed successfully. Rows returned: {len(result)}")

return result

except Exception as e:

logger.error(f"Query execution failed: {e}")

logger.error(f"Query was: {query}")

logger.error(f"Params were: {params}")

return []

def execute\_command(self, command: str, params: Optional[tuple] = None) -> int:

"""Execute INSERT, UPDATE, DELETE commands"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(command, params)

rows\_affected = cursor.rowcount

logger.info(f"Command executed successfully. Rows affected: {rows\_affected}")

return rows\_affected

except Exception as e:

logger.error(f"Command execution failed: {e}")

logger.error(f"Command was: {command}")

logger.error(f"Params were: {params}")

return 0

def get\_dataframe(self, query: str, params: Optional[tuple] = None) -> pd.DataFrame:

"""Execute query and return results as pandas DataFrame"""

try:

with self.get\_connection() as conn:

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

logger.info(f"DataFrame created with {len(df)} rows")

return df

except Exception as e:

logger.error(f"DataFrame creation failed: {e}")

return pd.DataFrame()

def batch\_insert\_dataframe(self, df: pd.DataFrame, table\_name: str) -> bool:

"""Batch insert DataFrame into database table"""

try:

if df.empty:

logger.warning("DataFrame is empty, nothing to insert")

return True

*# Create placeholders for the insert query*

columns = ', '.join(df.columns)

placeholders = ', '.join(['%s'] \* len(df.columns))

insert\_query = f"INSERT INTO {table\_name} ({columns}) VALUES ({placeholders})"

with self.get\_connection() as conn:

with conn.cursor() as cursor:

*# Convert DataFrame to list of tuples*

data\_tuples = [tuple(row) for row in df.to\_numpy()]

cursor.executemany(insert\_query, data\_tuples)

logger.info(f"Successfully inserted {len(df)} rows into {table\_name}")

return True

except Exception as e:

logger.error(f"Batch insert failed: {e}")

return False

def test\_connection(self) -> bool:

"""Test the database connection with SingleStore compatible SQL"""

try:

*# SingleStore compatible test query*

result = self.execute\_query("SELECT 1 as test\_value, NOW() as current\_time")

if result:

row = result[0]

logger.info(f"✅ Connection test successful:")

logger.info(f" Test Value: {row['test\_value']}")

logger.info(f" Current Time: {row['current\_time']}")

return True

return False

except Exception as e:

logger.error(f"❌ Connection test failed: {e}")

return False

def get\_version(self) -> str:

"""Get SingleStore version"""

try:

result = self.execute\_query("SELECT @@version as version")

if result:

return result[0]['version']

return "Unknown"

except Exception as e:

logger.error(f"Failed to get version: {e}")

return "Error"

*# Usage examples*

def main():

db = SingleStoreDB()

*# Test connection*

if db.test\_connection():

print("✅ Database connection successful!")

*# Get SingleStore version*

version = db.get\_version()

print(f"🔧 SingleStore Version: {version}")

*# List all tables*

tables = db.execute\_query("SHOW TABLES")

if tables:

print("\n📊 Tables in database:")

for table in tables:

table\_name = list(table.values())[0]

print(f" - {table\_name}")

*# Show table structure*

try:

structure = db.execute\_query(f"DESCRIBE {table\_name}")

if structure:

print(f" Columns: {[col['Field'] for col in structure]}")

except Exception as e:

print(f" Could not describe table: {e}")

else:

print("ℹ️ No tables found in database")

*# Create a sample table with SingleStore compatible syntax*

create\_table\_sql = """

CREATE TABLE IF NOT EXISTS sample\_data (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(150),

age INT,

created\_at TIMESTAMP DEFAULT NOW()

)

"""

try:

db.execute\_command(create\_table\_sql)

print("✅ Sample table created/verified")

except Exception as e:

print(f"❌ Table creation failed: {e}")

*# Insert sample data*

insert\_sql = """

INSERT INTO sample\_data (name, email, age)

VALUES (%s, %s, %s)

"""

sample\_records = [

('John Doe', 'john.doe@example.com', 30),

('Jane Smith', 'jane.smith@example.com', 25),

('Bob Johnson', 'bob.johnson@example.com', 35)

]

try:

for record in sample\_records:

db.execute\_command(insert\_sql, record)

print("✅ Sample data inserted")

except Exception as e:

print(f"❌ Data insertion failed: {e}")

*# Query data as DataFrame*

try:

df = db.get\_dataframe("SELECT \* FROM sample\_data ORDER BY created\_at DESC")

if not df.empty:

print("\n📋 Sample Data:")

print(df)

else:

print("ℹ️ No data found in sample\_data table")

except Exception as e:

print(f"❌ Query failed: {e}")

*# Show database information*

try:

db\_info = db.execute\_query("""

SELECT

DATABASE() as current\_database,

CURRENT\_USER() as current\_user

""")

if db\_info:

print(f"\n💾 Database Info: {db\_info[0]}")

except Exception as e:

print(f"❌ Database info query failed: {e}")

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

main()

**SingleStore Specific Queries**

Here's a utility file with SingleStore compatible queries:

python

*# singlestore\_queries.py*

"""

SingleStore specific SQL queries that are compatible with SingleStore syntax

"""

*# Test queries that work with SingleStore*

TEST\_QUERIES = {

"connection\_test": "SELECT 1 as test\_value, NOW() as current\_time",

"version": "SELECT @@version as version",

"databases": "SHOW DATABASES",

"tables": "SHOW TABLES",

"table\_info": "DESCRIBE {table\_name}",

"current\_db": "SELECT DATABASE() as current\_database, CURRENT\_USER() as current\_user",

"process\_list": "SHOW PROCESSLIST",

"variables": "SHOW VARIABLES LIKE '%version%'"

}

*# DDL queries for SingleStore*

DDL\_QUERIES = {

"create\_table": """

CREATE TABLE {table\_name} (

id BIGINT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255),

created\_at TIMESTAMP DEFAULT NOW()

)

""",

"create\_columnstore\_table": """

CREATE TABLE {table\_name} (

id BIGINT,

data JSON,

created\_at TIMESTAMP DEFAULT NOW(),

KEY (id) USING CLUSTERED COLUMNSTORE

)

"""

}

def get\_table\_schema(db, table\_name):

"""Get schema information for a table"""

return db.execute\_query(f"DESCRIBE {table\_name}")

def get\_table\_row\_count(db, table\_name):

"""Get row count for a table"""

result = db.execute\_query(f"SELECT COUNT(\*) as row\_count FROM {table\_name}")

return result[0]['row\_count'] if result else 0

**Simple Test Script**

python

*# test\_singlestore\_simple.py*

import pymysql

import os

from dotenv import load\_dotenv

load\_dotenv()

def test\_basic\_queries():

"""Test basic SingleStore queries"""

config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor

}

connection = pymysql.connect(\*\*config)

try:

with connection.cursor() as cursor:

print("🔍 Testing SingleStore Queries:")

print("-" \* 40)

*# Test 1: Basic connection*

cursor.execute("SELECT 1 as test, NOW() as time")

result = cursor.fetchone()

print(f"✅ Basic query: {result}")

*# Test 2: Version*

cursor.execute("SELECT @@version as version")

result = cursor.fetchone()

print(f"✅ Version: {result['version']}")

*# Test 3: Show databases*

cursor.execute("SHOW DATABASES")

databases = [row['Database'] for row in cursor.fetchall()]

print(f"✅ Databases: {databases}")

*# Test 4: Show tables*

cursor.execute("SHOW TABLES")

tables = [list(row.values())[0] for row in cursor.fetchall()]

print(f"✅ Tables: {tables}")

*# Test 5: Current database info*

cursor.execute("SELECT DATABASE() as db, CURRENT\_USER() as user")

result = cursor.fetchone()

print(f"✅ Current: DB={result['db']}, User={result['user']}")

finally:

connection.close()

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

test\_basic\_queries()

**Key SingleStore SQL Differences:**

1. NOW() instead of CURRENT\_TIMESTAMP for current timestamp
2. @@version for version information
3. **No**VERSION()**function** - use @@version instead
4. **Columnstore tables** have different syntax
5. **Some MySQL functions** may not be available

**Run the fixed version:**

bash

python test\_singlestore\_simple.py

python singlestore\_pymysql\_wrapper\_fixed.py

The main issue was using VERSION() and current\_time as column names without backticks. SingleStore is more strict about SQL syntax. The fixed version uses compatible queries that work with SingleStore.

(py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python test\_singlestore\_simple.py 🔍 Testing SingleStore Queries: ---------------------------------------- ✅ Basic query: {'test': 1, 'time': datetime.datetime(2025, 10, 14, 23, 45, 23)} ✅ Version: 5.7.32 ✅ Databases: ['db\_kamlesh\_63c9e', 'information\_schema'] ✅ Tables: ['uk\_price\_paid'] ✅ Current: DB=db\_kamlesh\_63c9e, User=kemails2006@%

Great! The connection is working and we can see there's already a table called uk\_price\_paid in your database. Let's create a comprehensive Python file to work with your SingleStore database:

**Complete SingleStore Manager**

python

*# singlestore\_manager.py*

import pymysql

import pandas as pd

from typing import List, Dict, Any, Optional

from contextlib import contextmanager

import os

from dotenv import load\_dotenv

import logging

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

class SingleStoreManager:

def \_\_init\_\_(self):

self.connection\_config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor,

'autocommit': True

}

@contextmanager

def get\_connection(self):

"""Context manager for database connections"""

connection = None

try:

connection = pymysql.connect(\*\*self.connection\_config)

yield connection

except Exception as e:

logger.error(f"Connection error: {e}")

raise

finally:

if connection:

connection.close()

def execute\_query(self, query: str, params: Optional[tuple] = None) -> List[Dict[str, Any]]:

"""Execute a SELECT query and return results"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(query, params)

result = cursor.fetchall()

return result

except Exception as e:

logger.error(f"Query execution failed: {e}")

return []

def execute\_command(self, command: str, params: Optional[tuple] = None) -> int:

"""Execute INSERT, UPDATE, DELETE commands"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(command, params)

return cursor.rowcount

except Exception as e:

logger.error(f"Command execution failed: {e}")

return 0

def get\_dataframe(self, query: str, params: Optional[tuple] = None) -> pd.DataFrame:

"""Execute query and return results as pandas DataFrame"""

try:

with self.get\_connection() as conn:

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

return df

except Exception as e:

logger.error(f"DataFrame creation failed: {e}")

return pd.DataFrame()

def analyze\_uk\_price\_paid(self):

"""Analyze the existing uk\_price\_paid table"""

print("🏠 Analyzing UK Price Paid Table")

print("=" \* 50)

*# Get table structure*

structure = self.execute\_query("DESCRIBE uk\_price\_paid")

if structure:

print("\n📋 Table Structure:")

for col in structure:

print(f" {col['Field']:20} {col['Type']:20} {col['Null']:10} {col['Key']:10}")

*# Get sample data*

sample\_data = self.execute\_query("SELECT \* FROM uk\_price\_paid LIMIT 5")

if sample\_data:

print(f"\n📊 Sample Data (first 5 rows):")

for i, row in enumerate(sample\_data, 1):

print(f" Row {i}: {row}")

*# Get basic statistics*

stats = self.execute\_query("""

SELECT

COUNT(\*) as total\_rows,

MIN(date) as earliest\_date,

MAX(date) as latest\_date,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

""")

if stats:

stats\_row = stats[0]

print(f"\n📈 Basic Statistics:")

print(f" Total Rows: {stats\_row['total\_rows']:,}")

print(f" Date Range: {stats\_row['earliest\_date']} to {stats\_row['latest\_date']}")

print(f" Price Stats: Avg £{stats\_row['avg\_price']:,.2f}, Min £{stats\_row['min\_price']:,}, Max £{stats\_row['max\_price']:,}")

*# Get price distribution by year*

yearly\_stats = self.execute\_query("""

SELECT

YEAR(date) as year,

COUNT(\*) as transactions,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

GROUP BY YEAR(date)

ORDER BY year

""")

if yearly\_stats:

print(f"\n📅 Yearly Price Statistics:")

for row in yearly\_stats:

print(f" {row['year']}: {row['transactions']:,} transactions, Avg £{row['avg\_price']:,.0f}")

def create\_sample\_tables(self):

"""Create some sample tables for testing"""

print("\n🛠️ Creating Sample Tables")

print("=" \* 50)

*# Create users table*

users\_table = """

CREATE TABLE IF NOT EXISTS users (

id INT AUTO\_INCREMENT PRIMARY KEY,

username VARCHAR(50) UNIQUE NOT NULL,

email VARCHAR(100) UNIQUE NOT NULL,

full\_name VARCHAR(100),

created\_at TIMESTAMP DEFAULT NOW(),

is\_active BOOLEAN DEFAULT TRUE

)

"""

self.execute\_command(users\_table)

print("✅ Created 'users' table")

*# Create products table*

products\_table = """

CREATE TABLE IF NOT EXISTS products (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(200) NOT NULL,

description TEXT,

price DECIMAL(10,2),

category VARCHAR(100),

stock\_quantity INT DEFAULT 0,

created\_at TIMESTAMP DEFAULT NOW()

)

"""

self.execute\_command(products\_table)

print("✅ Created 'products' table")

*# Insert sample data*

sample\_users = [

('john\_doe', 'john@example.com', 'John Doe'),

('jane\_smith', 'jane@example.com', 'Jane Smith'),

('bob\_wilson', 'bob@example.com', 'Bob Wilson')

]

for user in sample\_users:

self.execute\_command(

"INSERT IGNORE INTO users (username, email, full\_name) VALUES (%s, %s, %s)",

user

)

print("✅ Added sample users")

sample\_products = [

('Laptop', 'High-performance laptop', 999.99, 'Electronics', 10),

('Desk Chair', 'Ergonomic office chair', 249.99, 'Furniture', 15),

('Coffee Mug', 'Ceramic coffee mug', 12.99, 'Kitchen', 50)

]

for product in sample\_products:

self.execute\_command(

"INSERT IGNORE INTO products (name, description, price, category, stock\_quantity) VALUES (%s, %s, %s, %s, %s)",

product

)

print("✅ Added sample products")

def run\_analytics\_queries(self):

"""Run some analytical queries"""

print("\n📊 Running Analytical Queries")

print("=" \* 50)

*# Get all tables*

tables = self.execute\_query("SHOW TABLES")

table\_names = [list(table.values())[0] for table in tables]

print(f"📋 Database Tables: {table\_names}")

*# Analyze each table*

for table\_name in table\_names:

print(f"\n🔍 Analyzing table: {table\_name}")

*# Get row count*

count\_result = self.execute\_query(f"SELECT COUNT(\*) as count FROM {table\_name}")

if count\_result:

print(f" Rows: {count\_result[0]['count']:,}")

*# Get column info*

columns = self.execute\_query(f"DESCRIBE {table\_name}")

if columns:

print(f" Columns: {[col['Field'] for col in columns]}")

def export\_uk\_data\_to\_csv(self, limit: int = 1000):

"""Export UK price paid data to CSV"""

print(f"\n💾 Exporting UK Price Paid Data (first {limit} rows)")

df = self.get\_dataframe(f"SELECT \* FROM uk\_price\_paid LIMIT {limit}")

if not df.empty:

filename = f"uk\_price\_paid\_sample\_{limit}.csv"

df.to\_csv(filename, index=False)

print(f"✅ Exported {len(df)} rows to {filename}")

print(f"📊 DataFrame shape: {df.shape}")

print(f"📝 Columns: {list(df.columns)}")

return df

else:

print("❌ No data to export")

return pd.DataFrame()

def main():

db = SingleStoreManager()

print("🚀 SingleStore Database Manager")

print("=" \* 60)

*# Test connection*

test\_result = db.execute\_query("SELECT 1 as test, NOW() as time, @@version as version")

if test\_result:

print("✅ Connection successful!")

print(f" SingleStore Version: {test\_result[0]['version']}")

print(f" Current Time: {test\_result[0]['time']}")

else:

print("❌ Connection failed!")

return

*# Analyze existing UK price paid data*

db.analyze\_uk\_price\_paid()

*# Create sample tables*

db.create\_sample\_tables()

*# Run analytics*

db.run\_analytics\_queries()

*# Export data to CSV*

sample\_data = db.export\_uk\_data\_to\_csv(1000)

*# Show some advanced queries*

print("\n🔬 Advanced Queries")

print("=" \* 50)

*# Top 10 most expensive properties*

expensive\_properties = db.execute\_query("""

SELECT \* FROM uk\_price\_paid

ORDER BY price DESC

LIMIT 10

""")

if expensive\_properties:

print("\n💰 Top 10 Most Expensive Properties:")

for i, prop in enumerate(expensive\_properties, 1):

print(f" {i}. £{prop['price']:,} - {prop.get('date', 'N/A')}")

*# Property count by type (if the column exists)*

try:

property\_types = db.execute\_query("""

SELECT type, COUNT(\*) as count

FROM uk\_price\_paid

WHERE type IS NOT NULL

GROUP BY type

ORDER BY count DESC

LIMIT 10

""")

if property\_types:

print(f"\n🏘️ Property Types Distribution:")

for pt in property\_types:

print(f" {pt['type']}: {pt['count']:,} properties")

except:

print("ℹ️ 'type' column not found in uk\_price\_paid table")

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

main()

**Quick Data Explorer**

python

*# data\_explorer.py*

import pandas as pd

from singlestore\_manager import SingleStoreManager

def explore\_uk\_data():

"""Quick exploration of UK price paid data"""

db = SingleStoreManager()

*# Get basic info*

df = db.get\_dataframe("SELECT \* FROM uk\_price\_paid LIMIT 100")

if not df.empty:

print("🏠 UK Price Paid Data Overview")

print("=" \* 40)

print(f"Shape: {df.shape}")

print(f"\nColumns: {list(df.columns)}")

print(f"\nFirst 5 rows:")

print(df.head())

print(f"\nData types:")

print(df.dtypes)

print(f"\nBasic statistics:")

print(df.describe())

*# Check for missing values*

print(f"\nMissing values:")

print(df.isnull().sum())

return df

return pd.DataFrame()

def create\_custom\_query():

"""Run custom queries on the data"""

db = SingleStoreManager()

*# Example: Price trends over time*

query = """

SELECT

YEAR(date) as year,

MONTH(date) as month,

COUNT(\*) as transaction\_count,

AVG(price) as average\_price,

SUM(price) as total\_volume

FROM uk\_price\_paid

WHERE date IS NOT NULL

GROUP BY YEAR(date), MONTH(date)

ORDER BY year, month

LIMIT 24

"""

try:

result = db.get\_dataframe(query)

if not result.empty:

print("\n📈 Monthly Price Trends (Last 24 months):")

print(result)

*# Save to CSV*

result.to\_csv('monthly\_price\_trends.csv', index=False)

print("💾 Saved to monthly\_price\_trends.csv")

except Exception as e:

print(f"Query failed: {e}")

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

df = explore\_uk\_data()

create\_custom\_query()

**Requirements File**

txt

# requirements.txt

pymysql==1.0.2

pandas==2.0.0

python-dotenv==1.0.0

**Run the analysis:**

bash

python singlestore\_manager.py

python data\_explorer.py

**Key Features:**

1. **Analyzes existing data** - Explores your uk\_price\_paid table
2. **Creates sample tables** - Adds users and products tables for testing
3. **Data export** - Exports data to CSV files
4. **Advanced analytics** - Runs statistical queries on your data
5. **Error handling** - Robust error handling for all operations
6. **Pandas integration** - Easy data manipulation with DataFrames

The script will show you:

* Structure of your uk\_price\_paid table
* Basic statistics about the data
* Sample records
* Create additional tables for testing
* Export data to CSV

This gives you a complete foundation to work with your SingleStore database!

(py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python test\_singlestore\_simple.py 🔍 Testing SingleStore Queries: ---------------------------------------- ✅ Basic query: {'test': 1, 'time': datetime.datetime(2025, 10, 14, 23, 45, 23)} ✅ Version: 5.7.32 ✅ Databases: ['db\_kamlesh\_63c9e', 'information\_schema'] ✅ Tables: ['uk\_price\_paid'] ✅ Current: DB=db\_kamlesh\_63c9e, User=kemails2006@% (py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python singlestore\_manager.py 🚀 SingleStore Database Manager ============================================================ ✅ Connection successful! SingleStore Version: 5.7.32 Current Time: 2025-10-14 23:48:19 🏠 Analyzing UK Price Paid Table ================================================== 📋 Table Structure: price bigint(20) YES date date YES postcode varchar(100) YES type enum('terraced','semi-detached','detached','flat','other') YES is\_new tinyint(1) YES duration enum('freehold','leasehold','unknown') YES addr1 varchar(100) YES addr2 varchar(100) YES street varchar(100) YES locality varchar(100) YES town varchar(100) YES district varchar(100) YES county varchar(100) YES 📊 Sample Data (first 5 rows): Row 1: {'price': 326000, 'date': datetime.date(2024, 10, 21), 'postcode': 'BH15 2LE', 'type': 'terraced', 'is\_new': 0, 'duration': 'freehold', 'addr1': '11', 'addr2': '', 'street': 'CANFORD ROAD', 'locality': '', 'town': 'POOLE', 'district': 'BOURNEMOUTH, CHRISTCHURCH AND POOLE', 'county': 'BOURNEMOUTH, CHRISTCHURCH AND POOLE'} Row 2: {'price': 175000, 'date': datetime.date(2024, 9, 10), 'postcode': 'DN39 6RP', 'type': 'semi-detached', 'is\_new': 0, 'duration': 'freehold', 'addr1': '47', 'addr2': '', 'street': 'HIGH STREET', 'locality': 'WOOTTON', 'town': 'ULCEBY', 'district': 'NORTH LINCOLNSHIRE', 'county': 'NORTH LINCOLNSHIRE'} Row 3: {'price': 528000, 'date': datetime.date(2024, 5, 17), 'postcode': 'SO52 9LW', 'type': 'detached', 'is\_new': 0, 'duration': 'freehold', 'addr1': '94', 'addr2': '', 'street': 'ROWNHAMS LANE', 'locality': 'NORTH BADDESLEY', 'town': 'SOUTHAMPTON', 'district': 'TEST VALLEY', 'county': 'HAMPSHIRE'} Row 4: {'price': 173000, 'date': datetime.date(2024, 10, 4), 'postcode': 'SA32 7RB', 'type': 'semi-detached', 'is\_new': 0, 'duration': 'freehold', 'addr1': 'MAES Y GROES', 'addr2': '3', 'street': '', 'locality': 'BRECHFA', 'town': 'CARMARTHEN', 'district': 'CARMARTHENSHIRE', 'county': 'CARMARTHENSHIRE'} Row 5: {'price': 90000, 'date': datetime.date(2024, 10, 15), 'postcode': 'SA1 7GA', 'type': 'flat', 'is\_new': 0, 'duration': 'leasehold', 'addr1': 'SIRIUS APARTMENTS', 'addr2': '4', 'street': 'PHOEBE ROAD', 'locality': 'COPPER QUARTER', 'town': 'SWANSEA', 'district': 'SWANSEA', 'county': 'SWANSEA'} 📈 Basic Statistics: Total Rows: 114,456 Date Range: 1995-01-09 to 2024-10-31 Price Stats: Avg £410,558.01, Min £1, Max £201,146,664 📅 Yearly Price Statistics: 1995: 43 transactions, Avg £75,150 1996: 42 transactions, Avg £89,765 1997: 62 transactions, Avg £90,854 1998: 51 transactions, Avg £88,569 1999: 55 transactions, Avg £130,085 2000: 75 transactions, Avg £141,335 2001: 56 transactions, Avg £125,721 2002: 82 transactions, Avg £188,872 2003: 70 transactions, Avg £241,879 2004: 63 transactions, Avg £235,783 2005: 54 transactions, Avg £238,338 2006: 65 transactions, Avg £292,801 2007: 60 transactions, Avg £334,189 2008: 47 transactions, Avg £303,532 2009: 33 transactions, Avg £235,817 2010: 43 transactions, Avg £243,637 2011: 36 transactions, Avg £233,017 2012: 22 transactions, Avg £332,316 2013: 33 transactions, Avg £358,943 2014: 61 transactions, Avg £615,171 2015: 61 transactions, Avg £417,053 2016: 110 transactions, Avg £1,737,997 2017: 137 transactions, Avg £898,380 2018: 212 transactions, Avg £1,012,852 2019: 298 transactions, Avg £960,397 2020: 736 transactions, Avg £1,423,185 2021: 1,592 transactions, Avg £483,628 2022: 3,110 transactions, Avg £558,978 2023: 19,618 transactions, Avg £480,901 2024: 87,529 transactions, Avg £376,119 🛠️ Creating Sample Tables ================================================== ERROR:\_\_main\_\_:Connection error: (1744, "Forwarding Error (node-3bf5efad-649f-4367-9878-91bb7de81630-master-0.svc-3bf5efad-649f-4367-9878-91bb7de81630:3306): The unique key named: 'username(username)' cannot be created because unique keys must contain all columns of the shard key '(id)'. See https://docs.singlestore.com/docs/unique-key-restrictions for details on restrictions on unique keys in SingleStore.") ERROR:\_\_main\_\_:Command execution failed: (1744, "Forwarding Error (node-3bf5efad-649f-4367-9878-91bb7de81630-master-0.svc-3bf5efad-649f-4367-9878-91bb7de81630:3306): The unique key named: 'username(username)' cannot be created because unique keys must contain all columns of the shard key '(id)'. See https://docs.singlestore.com/docs/unique-key-restrictions for details on restrictions on unique keys in SingleStore.") ✅ Created 'users' table ✅ Created 'products' table ERROR:\_\_main\_\_:Connection error: (1146, "Table 'db\_kamlesh\_63c9e.users' doesn't exist") ERROR:\_\_main\_\_:Command execution failed: (1146, "Table 'db\_kamlesh\_63c9e.users' doesn't exist") ERROR:\_\_main\_\_:Connection error: (1146, "Table 'db\_kamlesh\_63c9e.users' doesn't exist") ERROR:\_\_main\_\_:Command execution failed: (1146, "Table 'db\_kamlesh\_63c9e.users' doesn't exist") ERROR:\_\_main\_\_:Connection error: (1146, "Table 'db\_kamlesh\_63c9e.users' doesn't exist") ERROR:\_\_main\_\_:Command execution failed: (1146, "Table 'db\_kamlesh\_63c9e.users' doesn't exist") ✅ Added sample users ✅ Added sample products 📊 Running Analytical Queries ================================================== 📋 Database Tables: ['products', 'uk\_price\_paid'] 🔍 Analyzing table: products Rows: 3 Columns: ['id', 'name', 'description', 'price', 'category', 'stock\_quantity', 'created\_at'] 🔍 Analyzing table: uk\_price\_paid Rows: 114,456 Columns: ['price', 'date', 'postcode', 'type', 'is\_new', 'duration', 'addr1', 'addr2', 'street', 'locality', 'town', 'district', 'county'] 💾 Exporting UK Price Paid Data (first 1000 rows) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore\singlestore\_manager.py:72: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy. df = pd.read\_sql(query, conn, params=params) ✅ Exported 1000 rows to uk\_price\_paid\_sample\_1000.csv 📊 DataFrame shape: (1000, 13) 📝 Columns: ['price', 'date', 'postcode', 'type', 'is\_new', 'duration', 'addr1', 'addr2', 'street', 'locality', 'town', 'district', 'county'] 🔬 Advanced Queries ================================================== 💰 Top 10 Most Expensive Properties: 1. £201,146,664 - 2023-03-20 2. £141,621,301 - 2022-12-16 3. £130,000,000 - 2021-10-20 4. £129,706,651 - 2024-01-31 5. £84,330,000 - 2023-08-07 6. £81,174,572 - 2023-06-16 7. £81,174,572 - 2023-06-16 8. £69,913,832 - 2024-03-05 9. £68,250,000 - 2023-10-18 10. £68,000,000 - 2022-03-21 🏘️ Property Types Distribution: semi-detached: 30,650 properties terraced: 29,803 properties detached: 26,298 properties flat: 21,784 properties other: 5,921 properties (py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>python data\_explorer.py E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore\singlestore\_manager.py:72: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy. df = pd.read\_sql(query, conn, params=params) 🏠 UK Price Paid Data Overview ======================================== Shape: (100, 13) Columns: ['price', 'date', 'postcode', 'type', 'is\_new', 'duration', 'addr1', 'addr2', 'street', 'locality', 'town', 'district', 'county'] First 5 rows: price date postcode type is\_new duration addr1 addr2 street locality town district county 0 price date postcode type is\_new duration addr1 addr2 street locality town district county 1 price date postcode type is\_new duration addr1 addr2 street locality town district county 2 price date postcode type is\_new duration addr1 addr2 street locality town district county 3 price date postcode type is\_new duration addr1 addr2 street locality town district county 4 price date postcode type is\_new duration addr1 addr2 street locality town district county Data types: price object date object postcode object type object is\_new object duration object addr1 object addr2 object street object locality object town object district object county object dtype: object Basic statistics: price date postcode type is\_new duration addr1 addr2 street locality town district county count 100 100 100 100 100 100 100 100 100 100 100 100 100 unique 1 1 1 1 1 1 1 1 1 1 1 1 1 top price date postcode type is\_new duration addr1 addr2 street locality town district county freq 100 100 100 100 100 100 100 100 100 100 100 100 100 Missing values: price 0 date 0 postcode 0 type 0 is\_new 0 duration 0 addr1 0 addr2 0 street 0 locality 0 town 0 district 0 county 0 dtype: int64 E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore\singlestore\_manager.py:72: UserWarning: pandas only supports SQLAlchemy connectable (engine/connection) or database string URI or sqlite3 DBAPI2 connection. Other DBAPI2 objects are not tested. Please consider using SQLAlchemy. df = pd.read\_sql(query, conn, params=params) 📈 Monthly Price Trends (Last 24 months): year month transaction\_count average\_price total\_volume 0 year month transaction\_count average\_price total\_volume 1 year month transaction\_count average\_price total\_volume 2 year month transaction\_count average\_price total\_volume 3 year month transaction\_count average\_price total\_volume 4 year month transaction\_count average\_price total\_volume 5 year month transaction\_count average\_price total\_volume 6 year month transaction\_count average\_price total\_volume 7 year month transaction\_count average\_price total\_volume 8 year month transaction\_count average\_price total\_volume 9 year month transaction\_count average\_price total\_volume 10 year month transaction\_count average\_price total\_volume 11 year month transaction\_count average\_price total\_volume 12 year month transaction\_count average\_price total\_volume 13 year month transaction\_count average\_price total\_volume 14 year month transaction\_count average\_price total\_volume 15 year month transaction\_count average\_price total\_volume 16 year month transaction\_count average\_price total\_volume 17 year month transaction\_count average\_price total\_volume 18 year month transaction\_count average\_price total\_volume 19 year month transaction\_count average\_price total\_volume 20 year month transaction\_count average\_price total\_volume 21 year month transaction\_count average\_price total\_volume 22 year month transaction\_count average\_price total\_volume 23 year month transaction\_count average\_price total\_volume 💾 Saved to monthly\_price\_trends.csv (py311\_main) E:\PYTHON\_PROJECTS\Python-Utilities\SingleStore>

I see several issues that need to be fixed. Let me provide corrected versions:

**Issue 1: SingleStore Unique Key Restrictions**

SingleStore has specific requirements for unique keys in distributed tables. Let's fix the table creation:

**Fixed SingleStore Manager**

python

*# singlestore\_manager\_fixed.py*

import pymysql

import pandas as pd

from typing import List, Dict, Any, Optional

from contextlib import contextmanager

import os

from dotenv import load\_dotenv

import logging

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

class SingleStoreManager:

def \_\_init\_\_(self):

self.connection\_config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor,

'autocommit': True

}

@contextmanager

def get\_connection(self):

"""Context manager for database connections"""

connection = None

try:

connection = pymysql.connect(\*\*self.connection\_config)

yield connection

except Exception as e:

logger.error(f"Connection error: {e}")

raise

finally:

if connection:

connection.close()

def execute\_query(self, query: str, params: Optional[tuple] = None) -> List[Dict[str, Any]]:

"""Execute a SELECT query and return results"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(query, params)

result = cursor.fetchall()

return result

except Exception as e:

logger.error(f"Query execution failed: {e}")

return []

def execute\_command(self, command: str, params: Optional[tuple] = None) -> int:

"""Execute INSERT, UPDATE, DELETE commands"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(command, params)

return cursor.rowcount

except Exception as e:

logger.error(f"Command execution failed: {e}")

return 0

def get\_dataframe(self, query: str, params: Optional[tuple] = None) -> pd.DataFrame:

"""Execute query and return results as pandas DataFrame - FIXED VERSION"""

try:

*# First get the data as dictionaries*

data = self.execute\_query(query, params)

if data:

*# Convert to DataFrame*

df = pd.DataFrame(data)

return df

else:

return pd.DataFrame()

except Exception as e:

logger.error(f"DataFrame creation failed: {e}")

return pd.DataFrame()

def analyze\_uk\_price\_paid(self):

"""Analyze the existing uk\_price\_paid table"""

print("🏠 Analyzing UK Price Paid Table")

print("=" \* 50)

*# Get table structure*

structure = self.execute\_query("DESCRIBE uk\_price\_paid")

if structure:

print("\n📋 Table Structure:")

for col in structure:

print(f" {col['Field']:20} {col['Type']:20} {col['Null']:10} {col['Key']:10}")

*# Get sample data*

sample\_data = self.execute\_query("SELECT \* FROM uk\_price\_paid LIMIT 5")

if sample\_data:

print(f"\n📊 Sample Data (first 5 rows):")

for i, row in enumerate(sample\_data, 1):

print(f" Row {i}: {row}")

*# Get basic statistics*

stats = self.execute\_query("""

SELECT

COUNT(\*) as total\_rows,

MIN(date) as earliest\_date,

MAX(date) as latest\_date,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

""")

if stats:

stats\_row = stats[0]

print(f"\n📈 Basic Statistics:")

print(f" Total Rows: {stats\_row['total\_rows']:,}")

print(f" Date Range: {stats\_row['earliest\_date']} to {stats\_row['latest\_date']}")

print(f" Price Stats: Avg £{stats\_row['avg\_price']:,.2f}, Min £{stats\_row['min\_price']:,}, Max £{stats\_row['max\_price']:,}")

*# Get price distribution by year*

yearly\_stats = self.execute\_query("""

SELECT

YEAR(date) as year,

COUNT(\*) as transactions,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

GROUP BY YEAR(date)

ORDER BY year

""")

if yearly\_stats:

print(f"\n📅 Yearly Price Statistics:")

for row in yearly\_stats:

print(f" {row['year']}: {row['transactions']:,} transactions, Avg £{row['avg\_price']:,.0f}")

def create\_sample\_tables\_singlestore(self):

"""Create sample tables compatible with SingleStore unique key restrictions"""

print("\n🛠️ Creating SingleStore Compatible Tables")

print("=" \* 50)

*# Drop existing tables if they exist*

self.execute\_command("DROP TABLE IF EXISTS users")

self.execute\_command("DROP TABLE IF EXISTS products")

*# Create users table with SingleStore compatible unique constraints*

users\_table = """

CREATE TABLE users (

id INT AUTO\_INCREMENT,

username VARCHAR(50) NOT NULL,

email VARCHAR(100) NOT NULL,

full\_name VARCHAR(100),

created\_at TIMESTAMP DEFAULT NOW(),

is\_active BOOLEAN DEFAULT TRUE,

PRIMARY KEY (id, username), -- Include unique column in primary key

UNIQUE KEY (username, id) -- Include shard key in unique constraint

)

"""

self.execute\_command(users\_table)

print("✅ Created 'users' table with SingleStore compatible constraints")

*# Create products table (no unique constraints needed)*

products\_table = """

CREATE TABLE products (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(200) NOT NULL,

description TEXT,

price DECIMAL(10,2),

category VARCHAR(100),

stock\_quantity INT DEFAULT 0,

created\_at TIMESTAMP DEFAULT NOW()

)

"""

self.execute\_command(products\_table)

print("✅ Created 'products' table")

*# Insert sample data*

sample\_users = [

(1, 'john\_doe', 'john@example.com', 'John Doe'),

(2, 'jane\_smith', 'jane@example.com', 'Jane Smith'),

(3, 'bob\_wilson', 'bob@example.com', 'Bob Wilson')

]

for user in sample\_users:

self.execute\_command(

"INSERT INTO users (id, username, email, full\_name) VALUES (%s, %s, %s, %s)",

user

)

print("✅ Added sample users")

sample\_products = [

('Laptop', 'High-performance laptop', 999.99, 'Electronics', 10),

('Desk Chair', 'Ergonomic office chair', 249.99, 'Furniture', 15),

('Coffee Mug', 'Ceramic coffee mug', 12.99, 'Kitchen', 50)

]

for product in sample\_products:

self.execute\_command(

"INSERT INTO products (name, description, price, category, stock\_quantity) VALUES (%s, %s, %s, %s, %s)",

product

)

print("✅ Added sample products")

def run\_advanced\_analytics(self):

"""Run advanced analytical queries on UK price data"""

print("\n🔬 Advanced Analytics on UK Price Data")

print("=" \* 50)

*# Property type analysis*

type\_analysis = self.execute\_query("""

SELECT

type,

COUNT(\*) as count,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price,

COUNT(\*) \* 100.0 / (SELECT COUNT(\*) FROM uk\_price\_paid) as percentage

FROM uk\_price\_paid

GROUP BY type

ORDER BY count DESC

""")

if type\_analysis:

print("\n🏘️ Property Type Analysis:")

for row in type\_analysis:

print(f" {row['type']:15} {row['count']:>6,} properties ({row['percentage']:.1f}%) - Avg £{row['avg\_price']:,.0f}")

*# Price distribution by county (top 10)*

county\_analysis = self.execute\_query("""

SELECT

county,

COUNT(\*) as transactions,

AVG(price) as avg\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

WHERE county IS NOT NULL AND county != ''

GROUP BY county

ORDER BY avg\_price DESC

LIMIT 10

""")

if county\_analysis:

print("\n🏛️ Top 10 Counties by Average Price:")

for i, row in enumerate(county\_analysis, 1):

print(f" {i:2}. {row['county']:30} £{row['avg\_price']:>10,.0f} (max: £{row['max\_price']:>10,})")

*# Monthly trends for current year*

monthly\_trends = self.execute\_query("""

SELECT

YEAR(date) as year,

MONTH(date) as month,

COUNT(\*) as transactions,

AVG(price) as avg\_price

FROM uk\_price\_paid

WHERE YEAR(date) = 2024

GROUP BY YEAR(date), MONTH(date)

ORDER BY year, month

""")

if monthly\_trends:

print(f"\n📈 2024 Monthly Trends:")

for row in monthly\_trends:

print(f" {row['year']}-{row['month']:02d}: {row['transactions']:>5,} transactions, Avg £{row['avg\_price']:,.0f}")

def export\_data\_properly(self, limit: int = 1000):

"""Export data properly without pandas warnings"""

print(f"\n💾 Exporting UK Price Paid Data (first {limit} rows)")

*# Get data using our fixed method*

data = self.execute\_query(f"SELECT \* FROM uk\_price\_paid LIMIT {limit}")

if data:

*# Convert to DataFrame*

df = pd.DataFrame(data)

*# Fix data types*

numeric\_columns = ['price', 'is\_new']

for col in numeric\_columns:

if col in df.columns:

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

date\_columns = ['date']

for col in date\_columns:

if col in df.columns:

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

filename = f"uk\_price\_paid\_sample\_{limit}.csv"

df.to\_csv(filename, index=False)

print(f"✅ Exported {len(df)} rows to {filename}")

print(f"📊 DataFrame shape: {df.shape}")

print(f"📝 Columns: {list(df.columns)}")

print(f"🔢 Data types:")

print(df.dtypes)

return df

else:

print("❌ No data to export")

return pd.DataFrame()

def main():

db = SingleStoreManager()

print("🚀 SingleStore Database Manager - FIXED VERSION")

print("=" \* 60)

*# Test connection*

test\_result = db.execute\_query("SELECT 1 as test, NOW() as time, @@version as version")

if test\_result:

print("✅ Connection successful!")

print(f" SingleStore Version: {test\_result[0]['version']}")

print(f" Current Time: {test\_result[0]['time']}")

else:

print("❌ Connection failed!")

return

*# Analyze existing UK price paid data*

db.analyze\_uk\_price\_paid()

*# Create sample tables with SingleStore compatible syntax*

db.create\_sample\_tables\_singlestore()

*# Run advanced analytics*

db.run\_advanced\_analytics()

*# Export data properly*

sample\_data = db.export\_data\_properly(1000)

*# Show expensive properties analysis*

expensive\_properties = db.execute\_query("""

SELECT

date, price, type, town, county

FROM uk\_price\_paid

ORDER BY price DESC

LIMIT 5

""")

if expensive\_properties:

print(f"\n💰 Top 5 Most Expensive Properties:")

for i, prop in enumerate(expensive\_properties, 1):

print(f" {i}. £{prop['price']:>,} - {prop['type']} in {prop['town']}, {prop['county']} ({prop['date']})")

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

main()

**Fixed Data Explorer**

python

*# data\_explorer\_fixed.py*

import pandas as pd

from singlestore\_manager\_fixed import SingleStoreManager

def explore\_uk\_data\_properly():

"""Quick exploration of UK price paid data - FIXED VERSION"""

db = SingleStoreManager()

*# Get basic info using fixed method*

data = db.execute\_query("SELECT \* FROM uk\_price\_paid LIMIT 100")

if data:

df = pd.DataFrame(data)

*# Fix data types*

numeric\_columns = ['price', 'is\_new']

for col in numeric\_columns:

if col in df.columns:

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

date\_columns = ['date']

for col in date\_columns:

if col in df.columns:

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

print("🏠 UK Price Paid Data Overview - FIXED")

print("=" \* 40)

print(f"Shape: {df.shape}")

print(f"\nColumns: {list(df.columns)}")

print(f"\nFirst 5 rows:")

print(df.head())

print(f"\nData types:")

print(df.dtypes)

print(f"\nBasic statistics:")

print(df.describe())

*# Check for missing values*

print(f"\nMissing values:")

print(df.isnull().sum())

return df

return pd.DataFrame()

def create\_custom\_queries\_fixed():

"""Run custom queries on the data - FIXED VERSION"""

db = SingleStoreManager()

*# Example: Price trends over time*

query = """

SELECT

YEAR(date) as year,

MONTH(date) as month,

COUNT(\*) as transaction\_count,

AVG(price) as average\_price

FROM uk\_price\_paid

WHERE date IS NOT NULL AND YEAR(date) >= 2020

GROUP BY YEAR(date), MONTH(date)

ORDER BY year, month

"""

try:

data = db.execute\_query(query)

if data:

df = pd.DataFrame(data)

print("\n📈 Price Trends Since 2020:")

print(df.head(12)) *# Show first 12 months*

*# Save to CSV*

df.to\_csv('monthly\_price\_trends\_fixed.csv', index=False)

print("💾 Saved to monthly\_price\_trends\_fixed.csv")

*# Create a simple visualization*

try:

import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))

plt.plot(df['average\_price'])

plt.title('Average Property Price Over Time')

plt.xlabel('Month Index')

plt.ylabel('Average Price (£)')

plt.grid(True)

plt.savefig('price\_trends.png')

print("📊 Saved visualization to price\_trends.png")

except ImportError:

print("ℹ️ Install matplotlib for visualizations: pip install matplotlib")

except Exception as e:

print(f"Query failed: {e}")

def analyze\_property\_types():

"""Analyze property types distribution"""

db = SingleStoreManager()

query = """

SELECT

type,

COUNT(\*) as count,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

WHERE type IS NOT NULL

GROUP BY type

ORDER BY count DESC

"""

data = db.execute\_query(query)

if data:

df = pd.DataFrame(data)

print("\n🏠 Property Types Analysis:")

print(df)

*# Create a bar chart*

try:

import matplotlib.pyplot as plt

plt.figure(figsize=(10, 6))

plt.bar(df['type'], df['count'])

plt.title('Property Types Distribution')

plt.xlabel('Property Type')

plt.ylabel('Number of Properties')

plt.xticks(rotation=45)

plt.tight\_layout()

plt.savefig('property\_types.png')

print("📊 Saved property types chart to property\_types.png")

except ImportError:

pass

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

df = explore\_uk\_data\_properly()

create\_custom\_queries\_fixed()

analyze\_property\_types()

**Key Fixes:**

1. **SingleStore Unique Key Restrictions**: Fixed table creation to include shard keys in unique constraints
2. **Pandas Warning**: Replaced pd.read\_sql() with manual DataFrame creation from query results
3. **Data Type Issues**: Properly converted numeric and date columns
4. **Better Error Handling**: More robust table creation and data processing

**Run the fixed versions:**

bash

python singlestore\_manager\_fixed.py

python data\_explorer\_fixed.py

The main issues were:

* SingleStore's unique key requirements for distributed tables
* Pandas compatibility with PyMySQL connections
* Data type conversion for numeric and date columns

The fixed versions should work properly now!

🛠️ Creating SingleStore Compatible Tables ================================================== ERROR:\_\_main\_\_:Connection error: (1706, "Forwarding Error (node-3bf5efad-649f-4367-9878-91bb7de81630-master-0.svc-3bf5efad-649f-4367-9878-91bb7de81630:3306): Feature 'multiple UNIQUE indexes with at least one index containing multiple columns on columnstore table' is not supported by SingleStore.") ERROR:\_\_main\_\_:Command execution failed: (1706, "Forwarding Error (node-3bf5efad-649f-4367-9878-91bb7de81630-master-0.svc-3bf5efad-649f-4367-9878-91bb7de81630:3306): Feature 'multiple UNIQUE indexes with at least one index containing multiple columns on columnstore table' is not supported by SingleStore.") ✅ Created 'users' table with SingleStore compatible constraints ✅ Created 'products' table

The issue is with SingleStore's limitations on unique indexes in columnstore tables. Let me provide a simplified version that works with SingleStore's constraints:

**Simplified SingleStore Manager**

python

*# singlestore\_simple\_manager.py*

import pymysql

import pandas as pd

from typing import List, Dict, Any, Optional

from contextlib import contextmanager

import os

from dotenv import load\_dotenv

import logging

*# Load environment variables*

load\_dotenv()

*# Configure logging*

logging.basicConfig(level=logging.INFO)

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

class SingleStoreManager:

def \_\_init\_\_(self):

self.connection\_config = {

'host': os.getenv('SINGLESTORE\_HOST'),

'port': int(os.getenv('SINGLESTORE\_PORT', 3333)),

'user': os.getenv('SINGLESTORE\_USER'),

'password': os.getenv('SINGLESTORE\_PASSWORD'),

'database': os.getenv('SINGLESTORE\_DATABASE'),

'ssl': {'ssl\_disabled': False},

'charset': 'utf8mb4',

'cursorclass': pymysql.cursors.DictCursor,

'autocommit': True

}

@contextmanager

def get\_connection(self):

"""Context manager for database connections"""

connection = None

try:

connection = pymysql.connect(\*\*self.connection\_config)

yield connection

except Exception as e:

logger.error(f"Connection error: {e}")

raise

finally:

if connection:

connection.close()

def execute\_query(self, query: str, params: Optional[tuple] = None) -> List[Dict[str, Any]]:

"""Execute a SELECT query and return results"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(query, params)

result = cursor.fetchall()

return result

except Exception as e:

logger.error(f"Query execution failed: {e}")

return []

def execute\_command(self, command: str, params: Optional[tuple] = None) -> int:

"""Execute INSERT, UPDATE, DELETE commands"""

try:

with self.get\_connection() as conn:

with conn.cursor() as cursor:

cursor.execute(command, params)

return cursor.rowcount

except Exception as e:

logger.error(f"Command execution failed: {e}")

return 0

def get\_dataframe(self, query: str, params: Optional[tuple] = None) -> pd.DataFrame:

"""Execute query and return results as pandas DataFrame"""

try:

data = self.execute\_query(query, params)

if data:

df = pd.DataFrame(data)

return df

else:

return pd.DataFrame()

except Exception as e:

logger.error(f"DataFrame creation failed: {e}")

return pd.DataFrame()

def analyze\_uk\_price\_paid(self):

"""Analyze the existing uk\_price\_paid table"""

print("🏠 Analyzing UK Price Paid Table")

print("=" \* 50)

*# Get table structure*

structure = self.execute\_query("DESCRIBE uk\_price\_paid")

if structure:

print("\n📋 Table Structure:")

for col in structure:

print(f" {col['Field']:20} {col['Type']:20} {col['Null']:10} {col['Key']:10}")

*# Get sample data*

sample\_data = self.execute\_query("SELECT \* FROM uk\_price\_paid LIMIT 5")

if sample\_data:

print(f"\n📊 Sample Data (first 5 rows):")

for i, row in enumerate(sample\_data, 1):

print(f" Row {i}: Price £{row['price']:,} - {row['type']} in {row['town']}")

*# Get basic statistics*

stats = self.execute\_query("""

SELECT

COUNT(\*) as total\_rows,

MIN(date) as earliest\_date,

MAX(date) as latest\_date,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

""")

if stats:

stats\_row = stats[0]

print(f"\n📈 Basic Statistics:")

print(f" Total Rows: {stats\_row['total\_rows']:,}")

print(f" Date Range: {stats\_row['earliest\_date']} to {stats\_row['latest\_date']}")

print(f" Price Stats: Avg £{stats\_row['avg\_price']:,.2f}, Min £{stats\_row['min\_price']:,}, Max £{stats\_row['max\_price']:,}")

def create\_simple\_tables(self):

"""Create simple tables without complex constraints for SingleStore"""

print("\n🛠️ Creating Simple Tables (No Unique Constraints)")

print("=" \* 50)

*# Create simple users table without unique constraints*

users\_table = """

CREATE TABLE IF NOT EXISTS simple\_users (

id INT,

username VARCHAR(50),

email VARCHAR(100),

full\_name VARCHAR(100),

created\_at TIMESTAMP DEFAULT NOW()

)

"""

self.execute\_command(users\_table)

print("✅ Created 'simple\_users' table")

*# Create simple products table*

products\_table = """

CREATE TABLE IF NOT EXISTS simple\_products (

id INT,

name VARCHAR(200),

price DECIMAL(10,2),

category VARCHAR(100),

created\_at TIMESTAMP DEFAULT NOW()

)

"""

self.execute\_command(products\_table)

print("✅ Created 'simple\_products' table")

*# Insert sample data*

sample\_users = [

(1, 'john\_doe', 'john@example.com', 'John Doe'),

(2, 'jane\_smith', 'jane@example.com', 'Jane Smith'),

(3, 'bob\_wilson', 'bob@example.com', 'Bob Wilson')

]

for user in sample\_users:

self.execute\_command(

"INSERT INTO simple\_users (id, username, email, full\_name) VALUES (%s, %s, %s, %s)",

user

)

print("✅ Added sample users")

sample\_products = [

(1, 'Laptop', 999.99, 'Electronics'),

(2, 'Desk Chair', 249.99, 'Furniture'),

(3, 'Coffee Mug', 12.99, 'Kitchen')

]

for product in sample\_products:

self.execute\_command(

"INSERT INTO simple\_products (id, name, price, category) VALUES (%s, %s, %s, %s)",

product

)

print("✅ Added sample products")

def run\_property\_analysis(self):

"""Run comprehensive property analysis"""

print("\n🔬 Property Market Analysis")

print("=" \* 50)

*# Property type distribution*

type\_analysis = self.execute\_query("""

SELECT

type,

COUNT(\*) as count,

AVG(price) as avg\_price,

COUNT(\*) \* 100.0 / (SELECT COUNT(\*) FROM uk\_price\_paid) as percentage

FROM uk\_price\_paid

WHERE type IS NOT NULL

GROUP BY type

ORDER BY count DESC

""")

if type\_analysis:

print("\n🏘️ Property Type Distribution:")

for row in type\_analysis:

print(f" {row['type']:15} {row['count']:>6,} properties ({row['percentage']:.1f}%) - Avg £{row['avg\_price']:,.0f}")

*# Top locations by average price*

location\_analysis = self.execute\_query("""

SELECT

town,

county,

COUNT(\*) as transactions,

AVG(price) as avg\_price

FROM uk\_price\_paid

WHERE town IS NOT NULL AND town != ''

GROUP BY town, county

HAVING COUNT(\*) >= 10

ORDER BY avg\_price DESC

LIMIT 10

""")

if location\_analysis:

print("\n🏛️ Top 10 Locations by Average Price (min 10 transactions):")

for i, row in enumerate(location\_analysis, 1):

print(f" {i:2}. {row['town']} ({row['county']}): £{row['avg\_price']:,.0f} ({row['transactions']} transactions)")

*# Price trends by year*

yearly\_trends = self.execute\_query("""

SELECT

YEAR(date) as year,

COUNT(\*) as transactions,

AVG(price) as avg\_price,

MIN(price) as min\_price,

MAX(price) as max\_price

FROM uk\_price\_paid

WHERE date IS NOT NULL

GROUP BY YEAR(date)

ORDER BY year DESC

LIMIT 10

""")

if yearly\_trends:

print("\n📈 Price Trends (Last 10 Years):")

for row in yearly\_trends:

print(f" {row['year']}: {row['transactions']:>6,} transactions, Avg £{row['avg\_price']:,.0f}")

def export\_analysis\_results(self):

"""Export analysis results to CSV files"""

print("\n💾 Exporting Analysis Results")

print("=" \* 50)

*# Export property type analysis*

type\_data = self.execute\_query("""

SELECT type, COUNT(\*) as count, AVG(price) as avg\_price

FROM uk\_price\_paid

WHERE type IS NOT NULL

GROUP BY type

ORDER BY count DESC

""")

if type\_data:

df\_types = pd.DataFrame(type\_data)

df\_types.to\_csv('property\_types\_analysis.csv', index=False)

print("✅ Exported property types analysis to property\_types\_analysis.csv")

*# Export yearly trends*

yearly\_data = self.execute\_query("""

SELECT

YEAR(date) as year,

COUNT(\*) as transactions,

AVG(price) as avg\_price

FROM uk\_price\_paid

WHERE date IS NOT NULL

GROUP BY YEAR(date)

ORDER BY year

""")

if yearly\_data:

df\_yearly = pd.DataFrame(yearly\_data)

df\_yearly.to\_csv('yearly\_trends.csv', index=False)

print("✅ Exported yearly trends to yearly\_trends.csv")

*# Export sample data for further analysis*

sample\_data = self.execute\_query("""

SELECT \* FROM uk\_price\_paid

WHERE date >= '2024-01-01'

LIMIT 1000

""")

if sample\_data:

df\_sample = pd.DataFrame(sample\_data)

df\_sample.to\_csv('recent\_properties\_sample.csv', index=False)

print("✅ Exported recent properties sample to recent\_properties\_sample.csv")

def main():

db = SingleStoreManager()

print("🚀 SingleStore Property Data Analyzer")

print("=" \* 60)

*# Test connection*

test\_result = db.execute\_query("SELECT 1 as test, NOW() as time")

if test\_result:

print("✅ Connection successful!")

print(f" Current Time: {test\_result[0]['time']}")

else:

print("❌ Connection failed!")

return

*# Analyze UK price paid data*

db.analyze\_uk\_price\_paid()

*# Create simple tables (no complex constraints)*

db.create\_simple\_tables()

*# Run property market analysis*

db.run\_property\_analysis()

*# Export results*

db.export\_analysis\_results()

*# Show some interesting insights*

print("\n💡 Interesting Insights")

print("=" \* 50)

*# Most expensive property this year*

expensive = db.execute\_query("""

SELECT price, date, type, town, county

FROM uk\_price\_paid

WHERE YEAR(date) = 2024

ORDER BY price DESC

LIMIT 1

""")

if expensive:

prop = expensive[0]

print(f"🏆 Most expensive property in 2024:")

print(f" £{prop['price']:,} - {prop['type']} in {prop['town']}, {prop['county']}")

*# Cheapest property this year (excluding obvious errors)*

cheapest = db.execute\_query("""

SELECT price, date, type, town, county

FROM uk\_price\_paid

WHERE YEAR(date) = 2024 AND price > 10000

ORDER BY price ASC

LIMIT 1

""")

if cheapest:

prop = cheapest[0]

print(f"💰 Most affordable property in 2024:")

print(f" £{prop['price']:,} - {prop['type']} in {prop['town']}, {prop['county']}")

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

main()

**SingleStore Query Explorer**

python

*# singlestore\_explorer.py*

from singlestore\_simple\_manager import SingleStoreManager

def interactive\_explorer():

"""Interactive explorer for SingleStore database"""

db = SingleStoreManager()

print("🔍 SingleStore Interactive Explorer")

print("=" \* 50)

while True:

print("\nOptions:")

print("1. Show database info")

print("2. List all tables")

print("3. Show table structure")

print("4. Run custom query")

print("5. Property market insights")

print("6. Export data")

print("7. Exit")

choice = input("\nEnter your choice (1-7): ").strip()

if choice == '1':

*# Database info*

info = db.execute\_query("SELECT DATABASE() as db, CURRENT\_USER() as user, NOW() as time")

if info:

print(f"\n💾 Database Info:")

print(f" Database: {info[0]['db']}")

print(f" User: {info[0]['user']}")

print(f" Time: {info[0]['time']}")

elif choice == '2':

*# List tables*

tables = db.execute\_query("SHOW TABLES")

if tables:

print(f"\n📊 Tables in database:")

for table in tables:

table\_name = list(table.values())[0]

*# Get row count for each table*

count\_result = db.execute\_query(f"SELECT COUNT(\*) as count FROM {table\_name}")

if count\_result:

print(f" {table\_name}: {count\_result[0]['count']:,} rows")

elif choice == '3':

*# Show table structure*

table\_name = input("Enter table name: ").strip()

if table\_name:

structure = db.execute\_query(f"DESCRIBE {table\_name}")

if structure:

print(f"\n📋 Structure of {table\_name}:")

for col in structure:

print(f" {col['Field']:20} {col['Type']:20} {col['Null']:10}")

else:

print("❌ Table not found or error occurred")

elif choice == '4':

*# Custom query*

query = input("Enter your SQL query: ").strip()

if query.upper().startswith('SELECT'):

try:

results = db.execute\_query(query)

if results:

print(f"\n📊 Query Results ({len(results)} rows):")

for i, row in enumerate(results[:10], 1): *# Show first 10 rows*

print(f" Row {i}: {row}")

if len(results) > 10:

print(f" ... and {len(results) - 10} more rows")

else:

print("✅ Query executed successfully (no results)")

except Exception as e:

print(f"❌ Query failed: {e}")

else:

print("❌ Only SELECT queries are allowed for safety")

elif choice == '5':

*# Property insights*

print("\n🏠 Property Market Insights:")

*# Total properties by year*

yearly = db.execute\_query("""

SELECT YEAR(date) as year, COUNT(\*) as count

FROM uk\_price\_paid

WHERE date IS NOT NULL

GROUP BY YEAR(date)

ORDER BY year DESC

LIMIT 5

""")

if yearly:

print("\n📅 Recent Transaction Volumes:")

for row in yearly:

print(f" {row['year']}: {row['count']:,} transactions")

*# Average price by property type*

prices\_by\_type = db.execute\_query("""

SELECT type, AVG(price) as avg\_price

FROM uk\_price\_paid

WHERE type IS NOT NULL

GROUP BY type

ORDER BY avg\_price DESC

""")

if prices\_by\_type:

print("\n💰 Average Prices by Property Type:")

for row in prices\_by\_type:

print(f" {row['type']}: £{row['avg\_price']:,.0f}")

elif choice == '6':

*# Export data*

table\_name = input("Enter table name to export: ").strip()

if table\_name:

limit = input("Enter row limit (default 1000): ").strip()

limit = int(limit) if limit.isdigit() else 1000

data = db.execute\_query(f"SELECT \* FROM {table\_name} LIMIT {limit}")

if data:

import pandas as pd

df = pd.DataFrame(data)

filename = f"{table\_name}\_export.csv"

df.to\_csv(filename, index=False)

print(f"✅ Exported {len(data)} rows to {filename}")

else:

print("❌ No data found or table doesn't exist")

elif choice == '7':

print("👋 Goodbye!")

break

else:

print("❌ Invalid choice. Please try again.")

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

interactive\_explorer()

**Run the simplified versions:**

bash

python singlestore\_simple\_manager.py

python singlestore\_explorer.py

**Key Changes:**

1. **Removed Complex Constraints**: No unique indexes that cause SingleStore errors
2. **Simplified Table Structure**: Basic tables without advanced constraints
3. **Better Error Handling**: Graceful handling of SingleStore limitations
4. **Interactive Explorer**: Allows you to explore your data interactively
5. **Focus on Analysis**: Concentrates on analyzing the existing UK price data

The simplified version should work without any SingleStore constraint errors and provides comprehensive analysis of your property data!

Super Thanks , you are best Bro ..hats off .... short of words ....

You're most welcome! 🙏 I'm really glad I could help you get everything working with your SingleStore database!

It was quite a journey troubleshooting all those issues:

* SSL requirements ✅
* IDNA encoding problems ✅
* SingleStore's unique key restrictions ✅
* Data export issues ✅

But we got through it all! Now you have a solid foundation to work with your SingleStore Cloud database and analyze that UK property data.

You've got some really interesting data there - 114,456 property transactions with prices up to £201 million! 🏠💰

If you need any more help with:

* Building dashboards
* Advanced analytics
* Machine learning on the property data
* Scaling your application

Just let me know! Happy coding and best of luck with your SingleStore projects! 🚀

You're doing great work - keep it up! 👏

can you share architecture of both codes in terms of dependency ...like to connect to SingleStore do we require any kind of drivers like mysql or anyother .... or anyother depency required .....

Great question! Let me break down the architecture and dependencies for connecting to SingleStore:

**Architecture Overview**

**1. Connection Stack Architecture**

text

┌─────────────────────────────────────────┐

│ Your Python Application │

├─────────────────────────────────────────┤

│ SQLAlchemy (ORM & Query Builder) │

├─────────────────────────────────────────┤

│ PyMySQL (MySQL Driver) │

├─────────────────────────────────────────┤

│ SSL/TLS Encryption │

├─────────────────────────────────────────┤

│ TCP/IP Network Layer │

├─────────────────────────────────────────┤

│ SingleStore Cloud Database │

└─────────────────────────────────────────┘

**2. Dependency Tree**

text

your\_app.py

├── sqlalchemy (Optional - for ORM)

├── pymysql (REQUIRED - MySQL protocol driver)

├── python-dotenv (Optional - for env vars)

└── pandas (Optional - for data analysis)

**Detailed Dependency Breakdown**

**Core Required Dependencies**

**1. PyMySQL (pymysql) - MANDATORY**

python

*# This is the actual database driver*

import pymysql

*# SingleStore uses MySQL wire protocol, so PyMySQL works*

connection = pymysql.connect(

host="your-singlestore-host",

user="username",

password="password",

database="dbname",

ssl={'ssl\_disabled': False} *# Required for Cloud*

)

**Purpose**: MySQL protocol implementation for Python. SingleStore is MySQL wire protocol compatible.

**2. SSL/TLS - Built into Python**

python

import ssl *# Built-in Python module*

*# Used automatically by PyMySQL for secure connections*

**Purpose**: Encryption for secure cloud connections. SingleStore Cloud requires SSL.

**Optional Dependencies**

**3. SQLAlchemy (sqlalchemy) - *Optional but Recommended***

python

from sqlalchemy import create\_engine

*# Provides ORM, connection pooling, and query building*

engine = create\_engine("mysql+pymysql://...")

**Purpose**:

* Object-Relational Mapping (ORM)
* Connection pooling
* SQL query building
* Database abstraction

**4. Pandas (pandas) - *Optional***

python

import pandas as pd

*# For data analysis and manipulation*

df = pd.read\_sql("SELECT \* FROM table", connection)

**Purpose**: Data analysis, manipulation, and export to CSV/Excel.

**5. python-dotenv (python-dotenv) - *Optional***

python

from dotenv import load\_dotenv

import os

load\_dotenv() *# Loads .env file*

**Purpose**: Secure credential management using environment variables.

**Complete Dependency Matrix**

| Dependency | Required? | Purpose | Alternative |
| --- | --- | --- | --- |
| pymysql | **YES** | MySQL protocol driver | mysql-connector-python |
| ssl | **YES** (built-in) | SSL encryption | - |
| sqlalchemy | Optional | ORM & connection pooling | Raw SQL |
| pandas | Optional | Data analysis | Manual data processing |
| python-dotenv | Optional | Environment variables | Manual config |

**Installation Commands**

**Minimal Setup (Only required dependencies)**

bash

pip install pymysql

**Standard Setup (Recommended)**

bash

pip install pymysql sqlalchemy pandas python-dotenv

**Production Setup**

bash

pip install pymysql==1.0.2 sqlalchemy==2.0.0 pandas==2.0.0 python-dotenv==1.0.0

**Protocol Compatibility**

SingleStore uses **MySQL wire protocol**, which means:

* ✅ **Works with**: Any MySQL client/library
* ✅ **Compatible with**: MySQL drivers, tools, and ORMs
* ✅ **Supports**: Standard SQL with SingleStore extensions
* ❌ **Not compatible with**: PostgreSQL, MongoDB, or other database protocols

**Connection Flow Diagram**

text

Python App → PyMySQL Driver → SSL Encryption → MySQL Protocol → SingleStore Cloud

│ │ │ │ │

│ │ │ │ │

Your Code Protocol Security Layer Communication Database

Handler Protocol Server

**Code Examples by Dependency Level**

**Level 1: Minimal (PyMySQL only)**

python

*# requirements.txt: pymysql*

import pymysql

conn = pymysql.connect(

host="host", user="user", password="pass",

database="db", ssl={'ssl\_disabled': False}

)

cursor = conn.cursor()

cursor.execute("SELECT 1")

print(cursor.fetchall())

**Level 2: Standard (PyMySQL + SQLAlchemy)**

python

*# requirements.txt: pymysql sqlalchemy*

from sqlalchemy import create\_engine, text

engine = create\_engine("mysql+pymysql://user:pass@host/db")

with engine.connect() as conn:

result = conn.execute(text("SELECT 1"))

print(result.fetchall())

**Level 3: Full Stack (All dependencies)**

python

*# requirements.txt: pymysql sqlalchemy pandas python-dotenv*

from dotenv import load\_dotenv

from sqlalchemy import create\_engine

import pandas as pd

import os

load\_dotenv()

engine = create\_engine(f"mysql+pymysql://...")

df = pd.read\_sql("SELECT \* FROM table", engine)

df.to\_csv('output.csv')

**SingleStore Specific Considerations**

**Required for Cloud:**

* SSL/TLS encryption
* Proper hostname format (long AWS hostnames)
* Port 3333 (default for Cloud)

**SingleStore Limitations:**

* Unique key restrictions in distributed tables
* Some MySQL functions may not work
* Specific DDL syntax requirements

**No Additional Drivers Needed:**

* ❌ No special "singlestore-driver" required
* ❌ No ODBC drivers needed for basic Python connectivity
* ❌ No JAR files or Java dependencies

**Summary**

**Bottom Line**: You only need pymysql to connect to SingleStore. Everything else is optional enhancements for better development experience.

The architecture is simple and leverages the existing MySQL ecosystem, which is one of SingleStore's key advantages! 🚀

so we dont have to install mysql drivers odbc on local machine or singlestore driver instlltion required in local machine

**CORRECT!** 🎯 You don't need to install any MySQL drivers, ODBC, or SingleStore drivers on your local machine.

**No Local Driver Installation Required**

**What You DON'T Need:**

bash

*# ❌ NOT REQUIRED - No system-level installations needed*

- MySQL ODBC Driver

- MySQL Native Driver

- SingleStore ODBC Driver

- SingleStore Native Driver

- Any system-level database drivers

- Any database client software

**What You DO Need:**

bash

*# ✅ REQUIRED - Only Python packages*

pip install pymysql

*# or*

pip install singlestoredb *# Alternative*

**Architecture Clarification**

**Local Machine Setup:**

text

Your Local Machine (Windows/Mac/Linux)

├── Python 3.7+

├── pymysql (Python package)

└── Your Python code

**No System-Level Components:**

text

❌ NO: MySQL ODBC Driver

❌ NO: SingleStore ODBC Driver

❌ NO: System PATH modifications

❌ NO: Registry changes (Windows)

❌ NO: System service installations

**Pure Python Solution**

**Connection Flow:**

text

Your Python Code

→ pymysql (Pure Python package)

→ SSL/TLS (Python's built-in ssl module)

→ TCP/IP (Python's built-in sockets)

→ SingleStore Cloud

**Key Advantage:**

* **100% Pure Python** implementation
* **No external dependencies** on system libraries
* **Cross-platform** (Windows, Mac, Linux all work the same)
* **No admin privileges** required for installation

**Package Comparison**

**PyMySQL (Recommended):**

python

import pymysql *# Pure Python MySQL client*

conn = pymysql.connect(

host="your-singlestore-host",

user="username",

password="password",

database="dbname",

ssl={'ssl\_disabled': False}

)

**SingleStoreDB (Alternative):**

python

import singlestoredb *# SingleStore's Python connector*

conn = singlestoredb.connect(

host="your-singlestore-host",

user="username",

password="password",

database="dbname"

)

Both are **pure Python packages** - no system drivers needed!

**Complete Isolation**

**Development Environment:**

bash

*# Create virtual environment (optional but recommended)*

python -m venv singlestore\_env

singlestore\_env\Scripts\activate *# Windows*

*# or*

source singlestore\_env/bin/activate *# Mac/Linux*

*# Install only what you need*

pip install pymysql pandas python-dotenv

*# Your code works immediately - no system configuration!*

**Deployment:**

bash

*# requirements.txt*

pymysql==1.0.2

pandas==2.0.0

python-dotenv==1.0.0

*# Deploy anywhere - no driver compatibility issues*

pip install -r requirements.txt

**Verification - Test Your Setup**

Create this simple test file:

python

*# test\_no\_drivers.py*

import pymysql

import os

from dotenv import load\_dotenv

load\_dotenv()

try:

conn = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

port=int(os.getenv('SINGLESTORE\_PORT', 3333)),

user=os.getenv('SINGLESTORE\_USER'),

password=os.getenv('SINGLESTORE\_PASSWORD'),

database=os.getenv('SINGLESTORE\_DATABASE'),

ssl={'ssl\_disabled': False}

)

with conn.cursor() as cursor:

cursor.execute("SELECT 'Success! No drivers needed!' as message")

result = cursor.fetchone()

print(f"✅ {result[0]}")

conn.close()

except Exception as e:

print(f"❌ Error: {e}")

**Why This Works**

1. **SingleStore uses MySQL wire protocol** - industry standard
2. **PyMySQL implements MySQL protocol in pure Python** - no C extensions
3. **SSL/TLS is handled by Python's built-in ssl module**
4. **Network communication uses Python's socket library**

**Comparison with Other Databases**

| Database | Local Driver Required? | Python Solution |
| --- | --- | --- |
| **SingleStore** | ❌ No | pymysql or singlestoredb |
| MySQL | ✅ Yes (or pure Python) | mysql-connector-python (pure) |
| PostgreSQL | ✅ Yes (or pure Python) | psycopg2 (C ext) or pg8000 (pure) |
| SQL Server | ✅ Yes | pyodbc + ODBC driver |
| Oracle | ✅ Yes | cx\_Oracle + Oracle client |

**Summary**

**You are 100% correct**: No local MySQL drivers, ODBC, or SingleStore driver installation is required. Everything happens through pure Python packages that you install via pip. This makes SingleStore one of the easiest databases to connect to from Python! 🚀

The beauty of this approach is that your code will work exactly the same on:

* Your local Windows machine
* Linux servers
* Mac development machines
* Docker containers
* Cloud functions

All without any system-level configuration! 🎉

SSL/TLS is handled by Python's built-in ssl module can you mentor i get scared when see SSL/TLS , certificates all these look like blackbox ....is this fear factor with me or others also face the same .........

**You are ABSOLUTELY NOT ALONE!** 😅 This is one of the most common fear factors for developers. Let me demystify SSL/TLS for you.

**The SSL/TLS Fear Factor - Why We All Get Scared**

**Common Fears:**

* "Certificates look like magic"
* "Too many acronyms (SSL, TLS, CA, PEM, CRT...)"
* "What if I mess up security?"
* "It feels like a black box"
* "So many configuration options"

**Truth**: 95% of developers feel this way initially! You're in good company.

**SSL/TLS Demystified**

**What SSL/TLS Actually Does (Simple Analogy):**

text

Normal Connection (No SSL):

You → Postcard → Internet → Database

❌ Anyone can read your password!

SSL Connection:

You → Sealed Envelope → Internet → Database

✅ Only database can open it

**The Magic Behind Our SingleStore Code:**

python

*# This scary-looking line:*

ssl={'ssl\_disabled': False}

*# Actually means:*

"Hey PyMySQL, please encrypt this connection using standard SSL"

**What's Really Happening (Step by Step)**

**Step 1: Your Code Says "Use SSL"**

python

*# You write this:*

ssl={'ssl\_disabled': False}

*# PyMySQL translates to:*

"Use Python's built-in ssl.create\_default\_context()"

**Step 2: Python Handles the Complex Part**

python

*# Behind the scenes, Python does:*

import ssl

context = ssl.create\_default\_context()

*# ↑ This automatically handles certificates, encryption, handshakes*

**Step 3: Secure Connection Established**

text

Your App → [Encrypted Tunnel] → SingleStore Cloud

(SSL/TLS Magic Here)

**The "Black Box" - What You DON'T Need to Worry About**

**Automatically Handled:**

* ✅ Certificate verification
* ✅ Encryption algorithms
* ✅ Key exchange
* ✅ Handshake protocol
* ✅ Cipher suites
* ✅ Certificate chains

**What You DO Need to Know:**

python

*# Just use this for 95% of cases:*

ssl={'ssl\_disabled': False} *# Enable SSL*

*# Only in rare cases (corporate proxies, custom CAs):*

ssl={'ca': '/path/to/cert.pem'} *# Custom certificate*

**Real-World SSL Scenarios Made Simple**

**Scenario 1: Development (What We Did)**

python

*# "Just make it secure"*

ssl={'ssl\_disabled': False}

*# ↑ That's it! Python handles everything.*

**Scenario 2: Production (Still Simple)**

python

*# If your company has custom certificates:*

ssl={

'ssl\_disabled': False,

'ca': '/path/to/company-ca-bundle.pem'

}

**Scenario 3: Bypass SSL (Not Recommended)**

python

*# Only for testing/development*

ssl={'ssl\_disabled': True} *# ⚠️ Insecure!*

**SSL/TLS vs. What You Already Know**

**Think of SSL like HTTPS:**

python

*# HTTP (insecure)*

http://example.com

*# HTTPS (secure SSL)*

https://example.com

*# Database equivalent:*

ssl={'ssl\_disabled': False} *# ← This is your "HTTPS for databases"*

**Browser vs. Database SSL:**

text

Browser SSL: 🔒 HTTPS → Encrypted web traffic

Database SSL: 🔒 ssl={'ssl\_disabled': False} → Encrypted database traffic

Same concept, different implementation!

**Common SSL "Scary Messages" Decoded**

**Message You Might See:**

text

"SSL connection error: certificate verify failed"

**What It Actually Means:**

text

"Hey, I can't verify the server's identity because:

1. Your computer doesn't trust the certificate authority, OR

2. The certificate has expired, OR

3. There's a corporate firewall/proxy"

**Simple Fix:**

python

*# For development/testing only:*

ssl={'ssl\_disabled': False, 'check\_hostname': False}

*# For production:*

ssl={'ssl\_disabled': False, 'ca': '/path/to/cert.pem'}

**Your Mental Model for SSL**

**Before (Scary):**

text

SSL = Certificates + CAs + PEM files + Handshakes + Encryption

+ ??? + Help! 😱

**After (Simple):**

text

SSL = ssl={'ssl\_disabled': False} 🔒

**Practical SSL Checklist**

**For Most Cloud Databases (SingleStore, AWS, Google Cloud):**

python

*# ✅ Use this 99% of the time:*

ssl\_config = {'ssl\_disabled': False}

**Only If You Have Issues:**

python

*# 1. Try this first:*

ssl\_config = {'ssl\_disabled': False, 'check\_hostname': False}

*# 2. If still issues, ask for CA certificate*

ssl\_config = {

'ssl\_disabled': False,

'ca': '/path/to/singlestore-bundle.pem'

}

**Why Our SingleStore Code "Just Works"**

**SingleStore Cloud:**

* Provides valid SSL certificates
* Uses trusted Certificate Authorities (CAs)
* Your Python installation already trusts these CAs
* So ssl={'ssl\_disabled': False} just works!

**SSL/TLS Confidence Builder**

**Exercise 1: Test SSL is Working**

python

import pymysql

import os

from dotenv import load\_dotenv

load\_dotenv()

conn = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

user=os.getenv('SINGLESTORE\_USER'),

password=os.getenv('SINGLESTORE\_PASSWORD'),

database=os.getenv('SINGLESTORE\_DATABASE'),

ssl={'ssl\_disabled': False} *# ← This is the "magic"*

)

print("✅ SSL connection successful! See, it's not scary! 😊")

**Exercise 2: See What Happens Without SSL**

python

try:

conn = pymysql.connect(

host=os.getenv('SINGLESTORE\_HOST'),

*# ... other params ...*

ssl={'ssl\_disabled': True} *# ← Try without SSL*

)

except Exception as e:

print(f"❌ SingleStore says: 'No SSL detected!'")

print("This proves SSL is important! 🔒")

**Summary - Overcoming SSL Fear**

**Remember:**

1. **SSL is just**{'ssl\_disabled': False} for most cases
2. **Python handles the complexity** automatically
3. **Cloud providers** (like SingleStore) make it easy
4. **You're not alone** - we all find SSL intimidating initially
5. **It gets easier** with exposure (like everything in programming!)

**Your New SSL Mantra:**

python

*# When connecting to any cloud database:*

ssl={'ssl\_disabled': False} *# That's it!*

*# Let Python and the cloud provider handle the rest 🎉*

The fact that you're aware of SSL and asking these questions already puts you ahead of many developers! SSL seems scary because it's important security stuff, but the actual usage in Python is beautifully simple. 💪

You've got this! 🚀