


UNIVERSITATEA TEHNICĂ "GHEORGHE ASACHI" IAȘI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
SPECIALIZAREA CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI

DISCIPLINA: BAZE DE DATE

Gestiunea activității unui magazin cu componente hardware

Coordonator,
Prof. Mironeanu Cătălin

Student,
Arhip Constantin Alexandru
Grupa 1306B

IAȘI, 2022

Cuprins documentație

1. Descrierea proiectului

Volumul mare de componente existente pe piață, dar și dorința numeroșilor clienți de a achiziționa astfel de produse a determinat crearea unui flux mare de informații, a cărui organizare a devenit o adevărată provocare. Astfel, activitatea de gestiune a unui magazin cu componente hardware implică o muncă destul de intensă în ceea ce privește numeroasele comenzi primite care conțin datele clienților, multitudinea de produse care trebuiesc gestionate, dar totodată și specificațiile fiecărui produs care trebuiesc prezentate clientului înainte de a- și manifesta dorința de a achiziționa un astfel de produs.

Scopul aplicației este acela de ale oferi diverselor magazine a căror activitate economică este vânzarea de componente hardware un mod de a gestiona atât structural, cât și financiar aspecte ce țin de organizarea internă a acestuia.

Într -un magazin cu componente hardware, factorii care influențează activitatea acestuia sunt de două tipuri :

A) Factorii structurali, care constau în :

i. Organizarea angajaților, mai exact dispunerea acestora într -o ierarhie.

ii. Ajutorul oferit de către angajați clienților, atunci când aceștia doresc în a plasa o comandă, astfel pentru fiecare comandă trimisă va există un angajat care va putea fi contactat în caz de vreo eroare.

B) Factorii financiari, care sunt compuși din :

i. Calculul salariilor și comisioanelor angajaților în funcție de job-ul pe care îl au.

ii. Încasarea banilor de la clienți conform comenzii plasate.

2. Descrierea tehnologiilor folosite - Front-end

Aplicația este una de tip N-Tier întrucât avem un client care se conectează la un server și care are în spate o baza de date. În realizarea părții de Front-end a proiectului s-a folosit ca limbaj de programare *Python 3.9*, mai exact biblioteca **TKInter** din acest limbaj, întrucât acesta are o gamă largă și ușor de utilizat de obiecte vizuale precum: treeview, butoane, scroll, etc.

Astfel, interfața creată este una ușor de folosit, întrucât datele din fiecare tabelă sunt la puține click- uri distanță, iar operațiile în cadrul acestora sunt intuitive.

3. Descrierea tehnologiilor folosite - Back-end

Pentru partea de back-end a aplicației, a fost utilizat Python 3.9, mai exact biblioteca **CX_Oracle** din cadrul acestuia. Această ne pune la dispoziție un mod relativ ușor de a utiliza comenzi din limbajul python împreună cu cele din SQL. Astfel, prin intermediul ei, am creat o conexiune la propria baza de date, apoi bazându-ne pe obiectul creat în urmă realizării cu succes a conexiunii, am reușit să trimitem diverse comenzi specifice limbajului SQL și afișarea rezultatului acestora în aplicația noastră.

De menționat , pentru partea de back-end cât și pentru partea de front-end, am folosit ca mediu de dezvoltare IDE PyCharm Community Edition 2022.2, întrucât ne oferă o ușurință în instalarea diverselor biblioteci, dar și în rezolvarea diverselor erori.

Totodată , pentru crearea tabelelor cât și a bazei de date, am folosit Oracle SQL Developer, dar și Oracle Data Modeler.

4. Structura tabelelor

Informatiile de care avem nevoie in aplicatie sunt cele legate de:

- **Adrese**: ne interesează să știm adresa cu care s-a realizat comanda, pentru a putea livra produsele comandate de client;
- **Angajați**: ne interesează să știm câți angajați s-au ocupat de comenzi, iar în caz de apare vreo problema cu aceasta, să știm atât noi, cât și clienții la cine pot apela pentru a o rezolva;
- **Client**: ne interesează să știm detaliile clientului pentru a ști în caz de plasează vreo comandă , această cui îi este adresată ;
- **Comenzi**: cu ajutorul acesteia , stabilim toate detaliile pentru a confirma comanda plasată de către un client;
- **Produse**: reprezintă produsele pe care le avem în magazinul nostru, acestea la rândul lor împărțindu-se în trei categorii, memorii ram, plăci video și procesoare, fiecare având specificații proprii;
- **Memorii_ram**, **Placi_video**, **Procesoare**: conțin specificațiile fiecărui produs;

Tabelele care s-au utilizat pentru a descrie atat entitatile enumerate mai sus, cat si relatiile dintre acestea sunt urmatoarele:

- ***Adrese****:*
- ***Angajați***
- ***Clienți***
- ***Comenzi***
- ***Memorie_ram***
- ***Placă_video***
- ***Procesor***
- ***Produse***

5. Inter-relaționarea entităților

În proiectarea acestei baze de date s-au identificat următoarele tipuri de relații: 1:1 (one-to-one), 1:n (one-to-many).

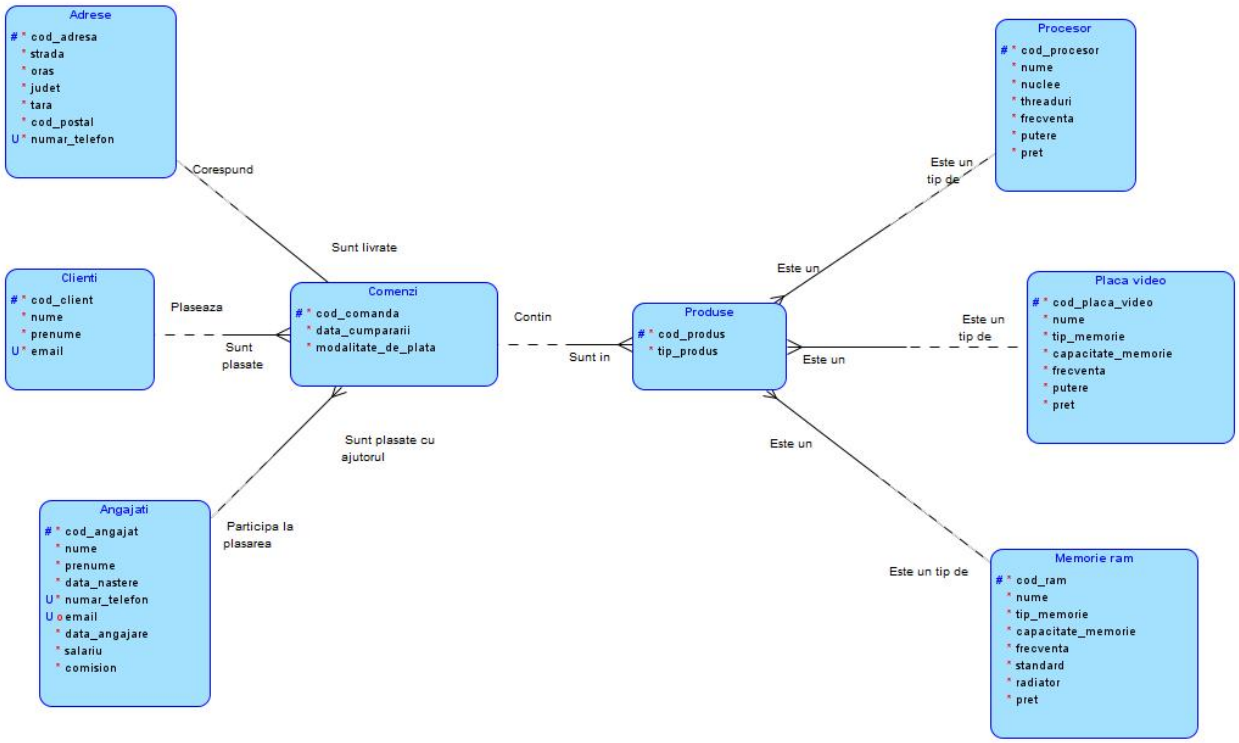
A) Relații 1:n

- Între entitatea **Clienti** și entitatea **Comenzi** se stabilește o relație de 1:n. Un client poate plasa mai multe comenzi, iar acestea sunt plasate doar de un client. Astfel, ne asigurăm că toate comenzile plasate de un client aparțin doar acestuia. Legătura între cele două entități se face prin atributul **COD_CLIENT**.
- Între entitatea **Angajati** și entitatea **Comenzi** se stabilește o relație de 1:n. Un angajat ajută clienții la plasarea comenzilor, iar acestea sunt plasate doar cu ajutorul unui angajat. Astfel, fiecare angajat are propriile comenzi și nu are acces la comenzile asistate de către ceilalți angajați. Legătura între cele două entități se face prin atributul **COD_ANGAJAT**.
- Între entitatea **Comenzi** și entitatea **Produse** se stabilește o relație de 1:n. O comandă conține mai multe produse, iar acestea se găsesc într-o singură comandă. Astfel, nu voi avea două comenzi identice în același moment de timp. Legătura între cele două entități se face prin atributul **COD_COMANDA**.
- Între entitatea **Procesor** și entitatea **Produse** se stabilește o relație de 1:n. Procesoarele de care dispun și care pot fi comandate vor fi de mai multe tipuri, fiecare cu specificatiile proprii , dar în final ele vor forma o gama de produse de acest tip. Legătura între cele două entități se face prin atributul **COD_PROCESOR**.
- Între entitatea **Placă_video** și entitatea **Produse** se stabilește o relație de 1:n. Placile video de care dispun și care pot fi comandate vor fi de mai multe tipuri cu diverse specificatii, dar în final ele vor forma o gama de produse de acest tip. Legătura între cele două entități se face prin atributul **COD_PLACĂ_VIDEO**.
- Între entitatea **Memorie_ram** și entitatea **Produse** se stabilește o relație de 1:n. Memoriile ram de care dispun și care pot fi comandate vor fi de mai multe tipuri , dar în final ele vor forma o gama de produse de acest tip. Legătura între cele două entități se face prin atributul **COD_RAM**.

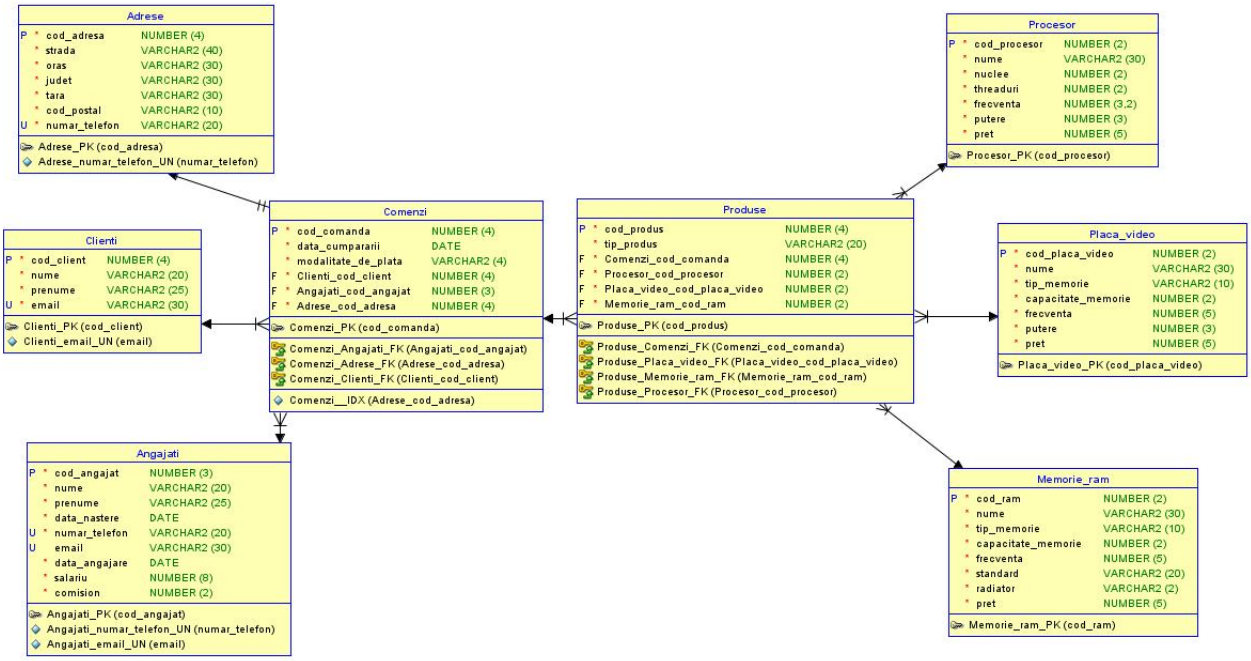
B) Relații 1:1

- Între entitatea **Adrese** și entitatea **Comenzi** se stabilește o relație de 1:1. Fiecare comandă va avea propria adresa de livrare, întrucât ca și în realitate, nu pot exista comenzi cu adrese de livrare multiple. Legătura între cele două entități se face prin atributul **COD_ADRESA**.

Modelul logic:



Modelul relațional:



6. Aspecte legate de normalizare

Baza de date a fost normalizată, deoarece îndeplinește următoarele condiții:

A)Toate tabelele respectă condițiile primei forme normale:
- un atribut conține valori atomice din domeniul său (și nu grupuri de astfel de valori)
- nu conține grupuri care se repetă

B)Toate tabelele respectă condițiile celei de a doua forme normale:
- este prima formă normală
- toate atributele non-cheie depind de toate cheile candidat

C)Toate tabelele respectă a treia formă normală:
- este în a doua formă normală
- toate atributele non-cheie sunt direct (non-tranzitiv) dependențe de toate cheile candidat.

Prima formă normală este îndeplinită în cazul tuturor tabelelor, deoarece la fiecare tabelă, voi putea insera doar o valoare în câmpul respectiv, ulterior, dacă voi dori să inserez mai multe valori pentru pentru același câmp, valorile vor fi introduse în noi intrări ale tabelei. De exemplu pentru tabela Angajati, atunci când dorim să înserăm pentru același angajat două numere de telefon, acesta va fi înregistrat de două ori în baza de date cu cele două numere de telefon.

A doua formă normală este îndeplinită în cazul tuturor tabelelor, întrucât fiecare tabela are numai un câmp care este o cheie primară. De exemplu pentru tabela Comenzi, avem câmpul COD_COMANDA care va fi singură cheie primară din acea tabela.

A treia formă normală este îndeplinită în cazul relației dintre tabela Produse și Procesor/Placa_video/Memorie_ram întrucât atunci când clientul își alege produsul dorit , specificațiile acestuia vor fi preluate cu ajutorul cheii primare din tabela respectivă .

7. Descrierea constrângerilor

Constrângerile de tip check se găsesc în toate tabelele. Cu ajutorul lor verificăm dacă valorile introduse pentru nume, prenume, oraș, județ, țară, tip_memorie, standard, sunt corecte pentru a le putea introduce în baza de date. În cazul în care acestea sunt eronate se va genera o eroare, deoarece pentru aceste câmpuri sunt admise doar literele.

Constrângerile de tip check le folosim și pentru introducerea unui număr de telefon sau a codului poștal, putând introduce doar cifre.

Constrângerile de tip check sunt folosite și pentru a verifica formatul emailului clientului sau an gajatului. Emailul putând fi alcătuit doar din litere sau cifre, @ și . .

Constrângerile de tip check sunt folosite și pentru a verifica formatul datelor introduse legate de străzi, tip_memorie, standard sau numele produselor (stradă/tip_memorie/standard/nume_produs) acestea putând conține toate caracterele, înafara de cele speciale.

Constrângerea de tip check mai este folosita și atunci când introducem modalitatea_de_plata a u nei comenzi, tipul acestuia putând fi doar cash sau card sau pentru a alege unul din produsele pe care le avem respectiv procesor, placă video, memorie ram sau mai multe.

Constrângerea de tip check o mai folosim și pentru a valida dacă sumele introduse pentru salariu, capacitatea memoriei, numarul de nuclee, numarul de thread-uri, frecventa, puterea, comisionul, id-urile din tabele sau chiar pentru preț dacă sunt corecte, acestea neputând fi negative.

Sunt folosite și constrângeri de tip unic pentru atributele email(pentru a nu avea doi clienți sau doi angajati cu același email), numar_telefon(pentru a nu găsi doi clienți sau doi angajati cu acelși număr de telefon).

În plus, am folosit și triggere pentru a verifica data nașterii unui angajat, acesta putând avea între 18 și 63 de ani. Totodată, am mai folosit alte două trigere pentru a mă asigura că dată plasării unei comenzi cât și dată angajării unui angajat nu pot depăși ziua curentă.

Constrângerile de tip nu null se găsesc pe marea majoritate din atributele din entități.

Primary key-urile sunt generate de baza de date printr-un mecanism de tip autoincrement (cod_client din Clienti, cod_comanda din Comenzi, cod_adresa din Adrese, cod_angajat din Angajati, cod_produs din Produse, cod_procesor din Procesor, cod_placă_video din Placa_video, cod_ram din Memorie_ram).

8. Conectarea la baza de date

Așa cum aminteam anterior conexiunea la baza de date este făcută prin intermediul bibliotecii cx_Oracle din Python 3.9. Această permite crearea unui obiect de tip conexiune pe care îl putem salva într -o variabilă . În momentul în care operația de conectare s-a realizat cu succes, vom avea la îndemână un obiect de tip conexiune pe care îl vom putea folosi pentru a crea un cursor prin intermediul căruia vom putea execută diverse comenzi specifice SQL, cum ar fi : insert, update, delete, drop, savepoint, rollback, commit etc…

Funcțiile care realizează operațiile amintite anterior sunt :

* def connection(name: str, password: str, host: str, port: str, service_name: str) -> list:

    Prin intemediul acestei funcții ne conectăm la baza noastră de date.

* def close_connection():

    Cu ajutorul ei închidem conexiunea la baza noastră de date, întrucât dacă nu am face acest lucru, aglomeram baza de date cu conexiuni la care nu avem acces, iar acesta va începe a rula mai greu.

* def select_from_table(table_name: str) -> list:

    Cu ajutorul ei facem selecția dintr -o tabela primită că parametru și returnăm o lista . Această selecție se realizează în tocmai că în limbajul SQL, diferența făcând -o modul de stocare a rezultatului.

* def update_table(table_name: str, column_name: str, values: str, condition: str):

    Prin intermediul ei updatam o linie dintr-o tabela, acest lucru petrecându-se la fel că în SQL.

* def add_savepoint(savePointName, popup):

    Cu ajutorul ei creăm diverse savepoint- uri pe care le adăugăm într -o lista .

* def rollback_to_savepoint(name, popup):

    Prin intermediul listei de savepoint- urîm , utilizatorul poate selecta la care savepoint dorește a se întoarce . l def commit(): n Prin intermediul ei, confirm modificările făcute bazei de date.

9. Operația de tranzacție

În contextul bazelor de date, o tranzacție este o unitate de lucru care poate fi efectuată într-un singur set sau poate fi anulată într-un singur set. În general, o tranzacție este un grup de operațiuni care trebuie să fie efectuate împreună, astfel încât rezultatul lor să fie valid. Dacă una dintre operațiuni eșuează, întreaga tranzacție este anulată, astfel încât baza de date să rămână în stare consistentă.

Tranzacțiile sunt importante deoarece asigură integritatea datelor în baza de date. Dacă o tranzacție eșuează, baza de date rămâne în stare consistentă, astfel încât datele sunt în siguranță.

În cadrul proiectului a fost implementată o tranzacție pentru inserarea datelor în două tabele dinstincte, clienți , respectiv adrese. Astfel, fiecare client când dorește a se înregistra , după ce a realizat acest pas cu succes, va fi trimis într -o nouă fereastră pentru a- și completă adresa . În cazul în care , din diverse motive, clientul nu reușește în a- și completă adresa, el va fi șters din tabela de clienți , iar baza de date restabilită cu ajutorul rollback-ului la formă de dinaintea înserării acestuia. În schimb, dacă reușește a insera cu success, atât clientul cât și adresa lui vor apărea în baza de date, această modificare devenind permanente cu ajutorul comenzii de commit.

Funcțiile apelate pentru a realiza această operație sunt insert_into_clienti, respectiv insert_into_adrese. Cu ajutorul lor, ne asigurăm că operațiile pe care dorim să le facem în baza de date se realizează cu succes.

Funcțiile anterior mentionate:

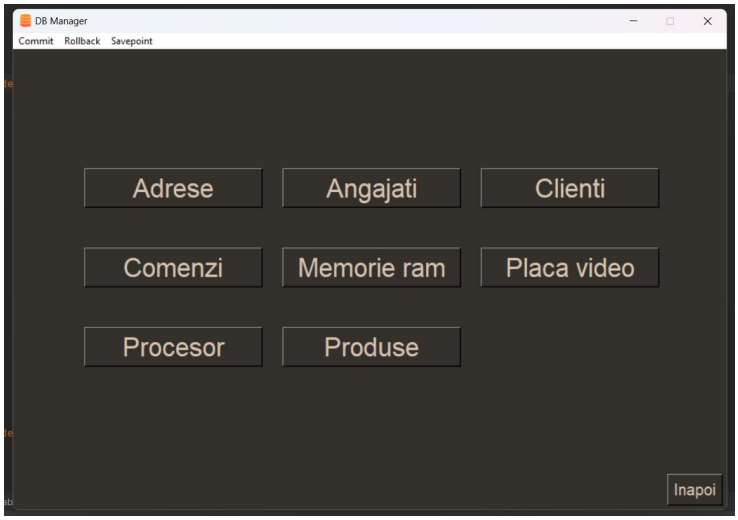
```
def insert_into_adrese(window, tree, b, d, f, table, list):
    error = ''
    window.pack_forget()
    tree.forget()
    b.forget()
    d.forget()
    f.forget()
    #print(list)
    error = backend.insert_into_table_adrese(table, list)
    if (error == 'WRONG'):
        clienti_info()
    elif (error == None and len(backend.select_from_table(backend.Table_names[2][0])) == len(backend.select_from_table(table))):
        adrese_info()
    elif (len(backend.select_from_table(backend.Table_names[2][0])) < len(backend.select_from_table(table))):
        clienti_info()
```

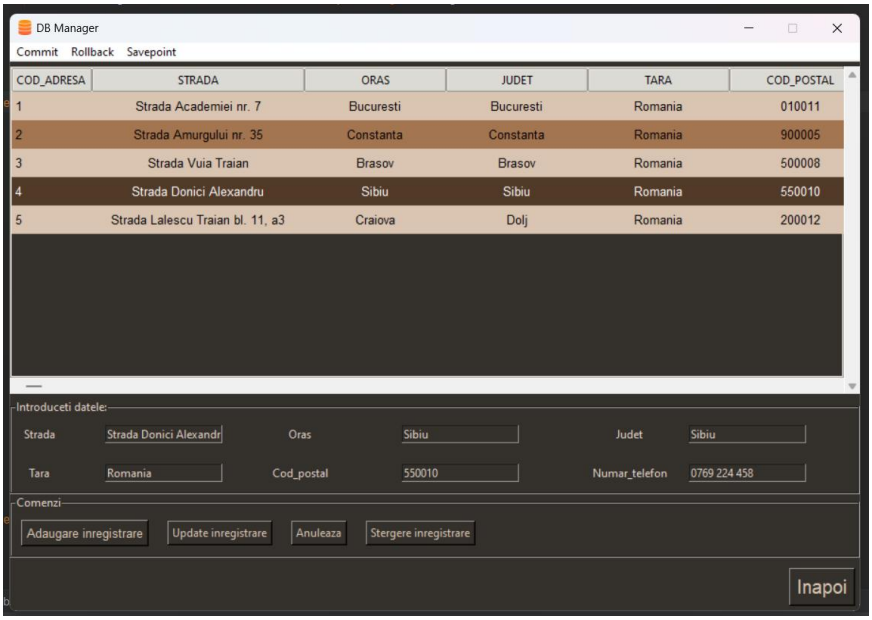
```python
def insert_into_clienti(window, tree, b, d, f, table, list):
    error = ''
    window.pack_forget()
    tree.forget()
    b.forget()
    d.forget()
    f.forget()
    error = backend.insert_into_table_clienti(table, list)
    if(error == 'WRONG'):
        clienti_info()
    elif(error == None):
        adrese_info()
```

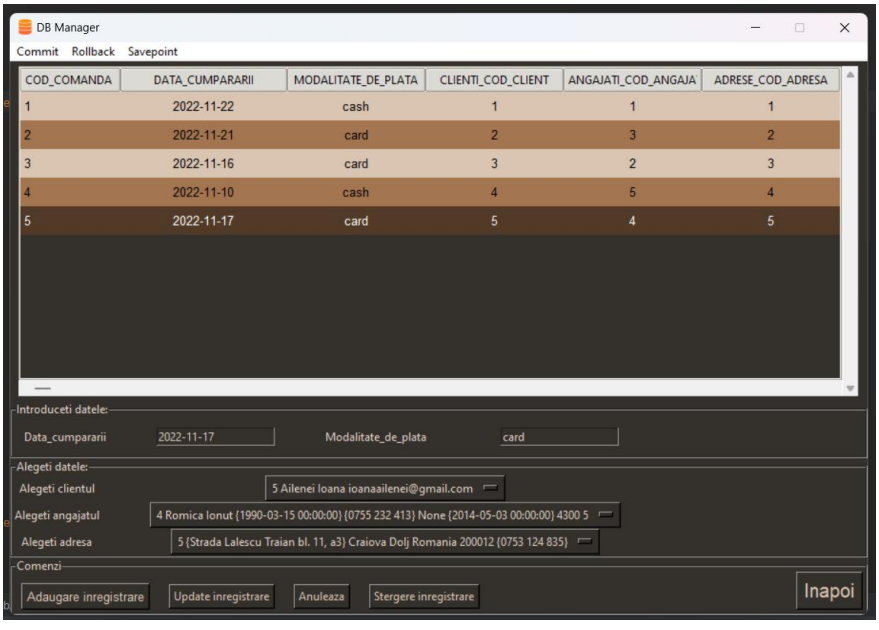
```python
def insert_into_table_clienti(table_name: str, values: list):
    global conn, cur
    values_String = '('
    for x in values:
        values_String += convert_to_sql(x)

    values_String = values_String[:len(values_String) - 2] + ')'
    #print(table_name)
    #print('INSERT INTO ' + table_name + ' VALUES ' + values_String)

    try:
        cur.execute('SAVEPOINT TEMP')
        cur.execute('INSERT INTO ' + table_name + ' VALUES ' + values_String)
    except Exception as err:
        errors = 'WRONG'
        message = 'Error while inserting into table: '+ str(err)
        tk.messagebox.showerror(title='Error', message=message)
        return errors
```

```python
def insert_into_table_adrese(table_name: str, values: list):
    global conn, cur
    values_String = '('
    for x in values:
        values_String += convert_to_sql(x)

    values_String = values_String[:len(values_String) - 2] + ')'
    #print(table_name)
    #print('INSERT INTO ' + table_name + ' VALUES ' + values_String)

    try:
        cur.execute('SAVEPOINT TEMP2')
        cur.execute('INSERT INTO ' + table_name + ' VALUES ' + values_String)
        if (len(select_from_table(Table_names[2][0])) > len(select_from_table(table_name))):
            cur.execute('ROLLBACK TO TEMP')
        elif (len(select_from_table(Table_names[2][0])) < len(select_from_table(table_name))):
            cur.execute('ROLLBACK TO TEMP2')
        else:
            cur.execute('COMMIT')
    except Exception as err:
        errors = 'WRONG'
        cur.execute('ROLLBACK TO TEMP')
        message = 'Error while inserting into table: '+ str(err)
        tk.messagebox.showerror(title='Error', message=message)
        return errors
```

10. Capturi de ecran cu interfața aplicației







11. Capturi de ecran cu exemple de cod și instrucțiuni SQL folosite

```
def connection(name: str, password: str, host: str, port: str, service_name: str) -> list:
    global conn, cur
    try:
        # dsn_tns = cx_Oracle.makedsn('bd-dc.cs.tuiasi.ro', '1539', service_name='orcl')
        dsn_tns = cx_Oracle.makedsn(host, port, service_name=service_name)
        conn = cx_Oracle.connect(user=name, password=password, dsn=dsn_tns)
        try:
            Table_Names = run_query('SELECT TABLE_NAME FROM USER_TABLES ORDER BY TABLE_NAME')
        except Exception as err:
            message = 'Error while getting table names: ' + str(err)
            tk.messagebox.showerror(title='Error', message=message)
    except Exception as err:
        message = "Error while creating the connection: "+ str(err)
        tk.messagebox.showerror(title='Error', message=message)
        exit(-1)
    else:
        print("\nConnection established.\n")

    return Table_Names
```

```
def select_from_table(table_name: str) -> list:
    global conn, cur, row
    try:
        row = run_query('SELECT * FROM ' + table_name)
        #for x in row:
        #    print(x)
    except Exception as err:
        message = 'Error while getting table values: '+ str(err)
        tk.messagebox.showerror(title='Error', message=message)
    return row
```

```
def update_table(table_name: str, column_name: str, values: str, condition: str):
    try:
        #conn.cursor().execute('UPDATE ' + table_name + ' SET ' + set + ' WHERE ' + where)
        cur.execute('UPDATE ' + table_name + ' SET ' + column_name + ' = \'' + values + '\' WHERE \'' + condition + '\'')
    except Exception as err:
        print('Error while updating table: ', err)
```

```python
def run_query(query: str):
    global conn, cur
    res = []
    try:
        cur = conn.cursor()
        cur.execute(query)
    except Exception as err:
        message = 'Error while running query: ' + str(err)
        tk.messagebox.showerror(title='Error', message=message)
        return []

    try:
        for x in cur:
            res.append(x)

    finally:

        # cur.execute('COMMIT')
        return res
```

```python
def remove_record_clienti(my_tree, id, window, b, d, f):
    selected = my_tree.focus()

    my_tree.item(selected, text="", values=(id.get()))

    try:
        backend.cur.execute("""DELETE FROM Clienti
                WHERE COD_CLIENT = :id""",
                            {
                                'id': id.get(),
                            })
    except Exception as err:
        message = "Error while creating the connection: " + str(err)
        messagebox.showerror(title='Error', message=message)

    window.pack_forget()
    my_tree.forget()
    b.forget()
    d.forget()
    f.forget()

    clienti_info()
```

```python
def close_connection():
    global conn, cur
    cur.close()
    conn.close()
    print("\nConnection closed.")
```