use PracticalExam

go

create table CoffeShops(

id int primary key identity(1, 1),

name varchar(100),

address varchar(100)

)

create table Categories(

id int primary key identity(1, 1),

name varchar(100),

description varchar(100)

)

create table CoffeProducts(

id int primary key identity(1, 1),

name varchar(100),

categoryId int references Categories(id)

)

create table Customers(

id int primary key identity(1, 1),

name varchar(100)

)

create table FavouriteCoffeProducts(

customerId int foreign key references Customers(id),

coffeProductId int foreign key references CoffeProducts(id),

primary key(customerId, coffeProductId)

)

create table ShopProducts(

shopId int foreign key references CoffeShops(id),

coffeProductId int foreign key references CoffeProducts(id),

price int

primary key (shopId, coffeProductId)

)

create table Orders(

id int primary key identity(1, 1),

customerId int foreign key references Customers(id),

shopId int,

coffeProductId int,

orderDatetime datetime,

foreign key (shopId, coffeProductId) references ShopProducts(shopId, coffeProductId)

)

insert into Categories (name, description) VALUES

('n1', 'd1'),

('n2', 'd1'),

('n3', 'd2'),

('n4', 'd2')

select \* from Categories

select \* from CoffeProducts

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Data.SqlClient;

namespace PracticalExam

{

public partial class Form1 : Form

{

SqlDataAdapter coffeProductsDataAdapter;

DataSet dataSet;

public Form1()

{

InitializeComponent();

}

private void Form1\_Load(object sender, EventArgs e)

{

SqlConnection dbConnection = new SqlConnection();

dbConnection.ConnectionString = "Data Source = DESKTOP-E94DLUN\\SQLEXPRESS;" +

"Initial Catalog = PracticalExam; Integrated Security = SSPI";

coffeProductsDataAdapter = new SqlDataAdapter("SELECT \* FROM CoffeProducts", dbConnection);

SqlDataAdapter categoriesDataAdapter = new SqlDataAdapter("SELECT \* FROM Categories", dbConnection);

dataSet = new DataSet();

SqlCommandBuilder builder = new SqlCommandBuilder(coffeProductsDataAdapter);

coffeProductsDataAdapter.Fill(dataSet, "CoffeProducts");

categoriesDataAdapter.Fill(dataSet, "Categories");

DataRelation dataRelation = new DataRelation("FK\_CoffeProductsCategories",

dataSet.Tables["Categories"].Columns["id"],

dataSet.Tables["CoffeProducts"].Columns["categoryId"]);

dataSet.Relations.Add(dataRelation);

BindingSource bsCategories = new BindingSource()

{

DataSource = dataSet,

DataMember = "Categories"

};

BindingSource bsCoffeProducts = new BindingSource

{

DataSource = bsCategories,

DataMember = "FK\_CoffeProductsCategories"

};

dgvCategories.DataSource = bsCategories ;

dgvProducts.DataSource = bsCoffeProducts;

dgvCategories.Columns.Remove("id");

dgvProducts.Columns.Remove("id");

dgvProducts.Columns.Remove("categoryId");

}

private void updateDatabase\_Click(object sender, EventArgs e)

{

try

{

coffeProductsDataAdapter.Update(dataSet, "CoffeProducts");

MessageBox.Show("DB updated", "Success");

}

catch (SqlException exc)

{

(from dataRow in dataSet.Tables["CoffeProducts"].AsEnumerable()

where dataRow.HasErrors

select dataRow)

.ToList().ForEach(row => dataSet.Tables["CoffeProducts"].Rows.Remove(row));

MessageBox.Show(exc.Message, "Could not update table");

}

}

}

}

use PracticalExam

go

/\*

Dirty read is a concurrency problem which occurs on the

read uncommitted isolation level.

A dirty read means that a transaction reads uncommitted data.

This means that this transaction will read the value written

by the other transaction before it rollbacks, thus reading wrong

data. This happens because under read uncommitted

there are no shared locks acquired meaning that the exclusive

lock acquired by the update does not stop the reading.

The solution is to run the first tran under read committed

isolation level (or higher)

so that shared locks are acquired when trying to read.

This means that the exclusive lock acquired by the update

will prevent the second select to read the uncommitted data

because the second select will wait until the exclusive lock is

dropped because it tries to acquire a shared lock

T1 is executed first then T2

waitfor delay command helps to simulate the order of execution

so that the update is executed between the selects

and the second select to be executed after the update but before

the rollback so that the dirty read can occur

\*/

--T1

set transaction isolation level read uncommitted

-- FIX: set transaction isolation level read committed

begin tran

select \* from Categories where id=1

waitfor delay '00:00:05'

select \* from Categories where id=1

commit tran

--T2

begin tran

update Categories set description='dirty read' where id=1

waitfor delay '00:00:05'

rollback