

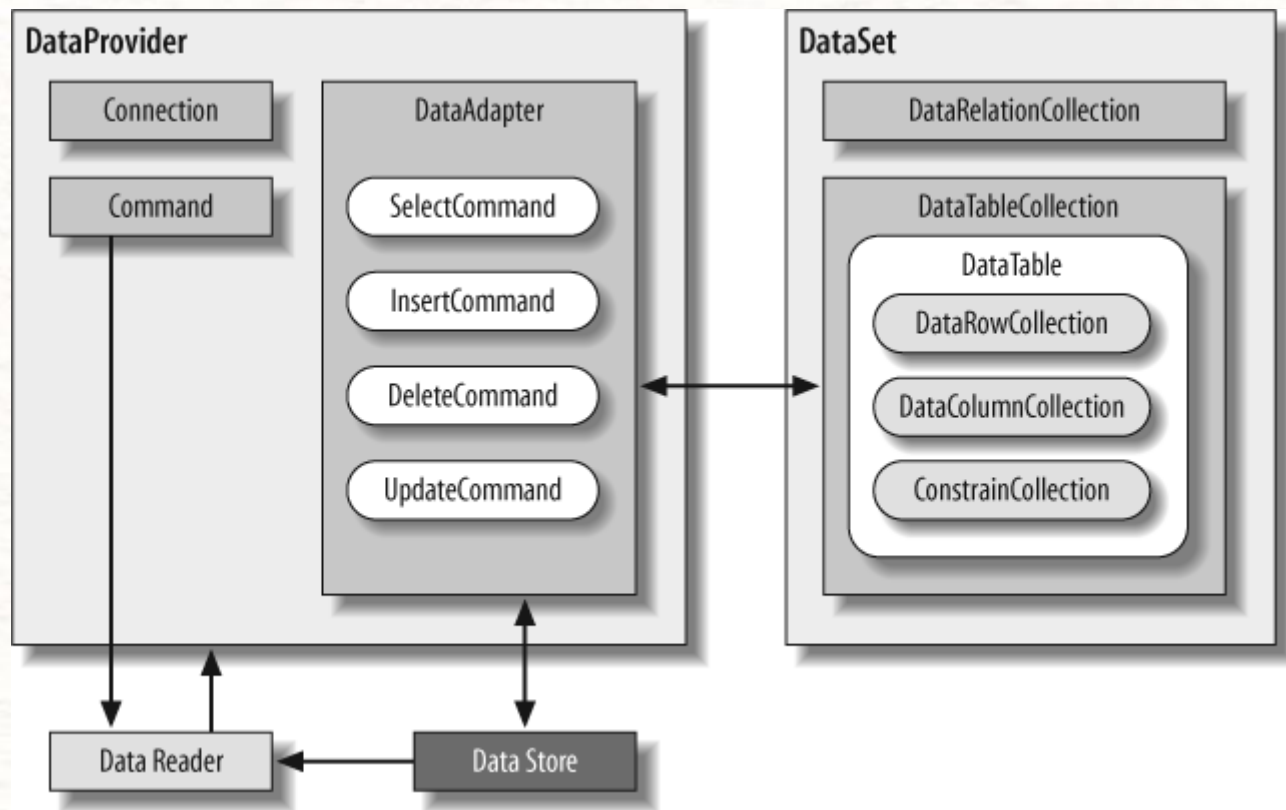
ADO.NET

Seminar 1

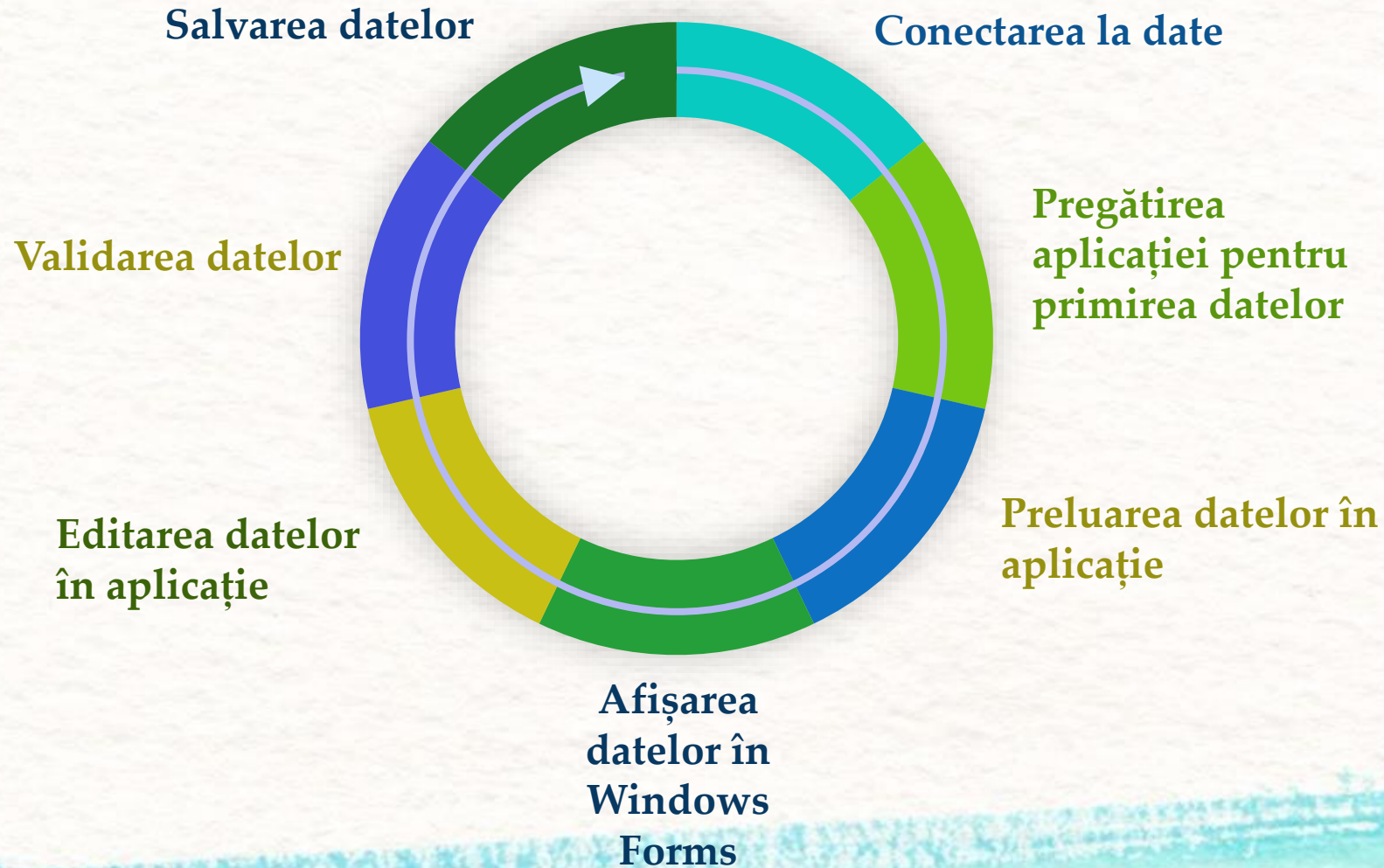
ADO.NET

- ADO.NET este un set de clase care expun servicii de acces a datelor pentru programatorii .NET
- ADO.NET:
 - Oferă un set bogat de componente pentru crearea aplicațiilor distribuite, care partajează date
 - Este o parte integrantă a .NET Framework, care oferă acces la date provenite din surse de date cum ar fi XML sau SQL Server
 - Include .NET Framework data providers pentru conectare la baza de date, execuție comenzi și returnare a rezultatelor
 - Namespace-ul **System.Data.SqlClient** este .NET Framework data provider pentru SQL Server

Diagrama arhitecturii ADO.NET



Ciclul datelor



Ciclul datelor

- **Conectarea la date:** stabilește o comunicare bidirecțională între aplicație și serverul de baze de date
- **Pregătirea aplicației pentru primirea datelor:** când se utilizează un model deconectat, anumite obiecte stochează datele temporar (*dataset-uri*, *entități*, obiecte *LINQ to SQL*)
- **Preluarea datelor în aplicație:** execuția interogărilor și a procedurilor stocate

Ciclul datelor

- **Afișarea datelor în Windows Forms:** se utilizează controale conectate la date (*data-bound*)
 - *DataGridView, TextBox, Label, ComboBox*
- **Editarea și validarea datelor în aplicație:** adăugarea/modificarea/ștergerea înregistrărilor și verificarea noilor valori (acestea din urmă trebuie să îndeplinească cerințele aplicației)
- **Salvarea datelor:** persistarea modificărilor în baza de date
 - *Update(DataSet, String)*
 - *ExecuteNonQuery()*

Modele de date

- DataSet-uri tipizate/netipizate (*typed/untyped*)
 - Un **DataSet tipizat** este o clasă care derivă din clasa *DataSet* și moștenește toate metodele, evenimentele și proprietățile ei și oferă metode, evenimente și proprietăți puternic tipizate
- Model conceptual bazat pe *Entity Data Model* care poate fi utilizat de *Entity Framework* sau *WCF Data Services*
 - **Entity Framework** este un object-relational mapper care permite lucrul cu date provenite dintr-un sistem relațional folosind obiecte specifice domeniului
 - **WCF Data Services** oferă posibilitatea de a crea/utiliza servicii de date pe Web sau intranet
- Clase *LINQ to SQL*
 - Suportă interogări pe un model obiect care corespunde structurii bazei de date relaționale fără a folosi un model conceptual intermediar
 - Transformă interogările language-integrated din modelul obiect în Transact-SQL și le transmite bazei de date pentru execuție

Modele de date – Exemplu de clasă LINQ to SQL

```
[Table(Name="Cadouri")]
public class Cadou
{
    [Column(Name="cod_cadou",IsPrimaryKey=true,IsDbGenerated=true)]
    public int CodC;

    [Column(Name="descriere")]
    public string Descriere;

    [Column(Name="posesor")]
    public string Posesor;

    [Column(Name="pret")]
    public float Pret;
}
```


DataSet

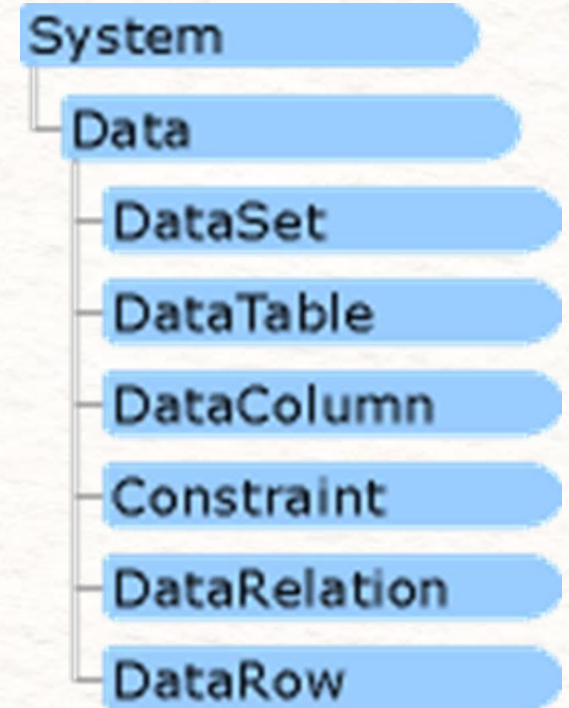
- **DataSet**-urile sunt obiecte care conțin data tables în care se pot stoca temporar date spre a fi utilizate în aplicație
- Oferă un cache local, în memorie, al datelor
- Funcționează chiar dacă aplicația este deconectată de la baza de date
- Structura unui *DataSet* este similară cu cea a unei baze de date relaționale deoarece conține:
 - Tabele
 - Înregistrări
 - Coloane
 - Constrângeri
 - Relații

DataSet

- **Proprietăți:**
 - **Tables** – returnează colecția de tabele incluse în *DataSet*
 - **Relations** – returnează colecția de relații care conectează tabelele și permit navigarea dinspre tabelele părinte spre tabelele copil
- **Metode:**
 - **Clear()** – șterge toate datele stocate în tabelele din *DataSet*
 - **HasChanges()** – returnează o valoare care indică dacă *DataSet*-ul conține modificări, inclusiv înregistrări noi, modificate sau șterse

DataSet

- Clasa **DataSet** include:
 - DataTableCollection
 - DataRelationCollection
- Clasa **DataTable** include:
 - DataRowCollection
 - DataColumnCollection
 - ChildRelations
 - ParentRelations
- Clasa **DataRow** include proprietatea **RowState** (valori posibile: *Deleted, Modified, Added* și *Unchanged*)



Clasa Console

- Clasa **Console** reprezintă stream-urile de intrare, de ieșire și de eroare pentru aplicațiile de tip consolă
- **Proprietăți:**
 - **WindowLeft** – returnează sau setează cea mai din stânga poziție a ferestrei consolei relativ la buffer-ul ecranului
 - **WindowTop** – returnează sau setează cea mai de sus poziție a ferestrei consolei relativ la buffer-ul ecranului
 - **WindowHeight** – returnează sau setează înălțimea ferestrei consolei
 - **WindowWidth** – returnează sau setează lățimea ferestrei consolei
 - **Title** – returnează sau setează titlul afișat în bara de titlu a consolei
 - **BackgroundColor** – returnează sau setează culoarea de fundal a consolei

Clasa Console

- **Metode:**

- **Write(...)**

- **Write(String)** – scrie valoarea de tip string specificată în stream-ul standard de ieșire

- **WriteLine(...)**

- **WriteLine(String)** – scrie valoarea de tip string specificată urmată de current line terminator în stream-ul standard de ieșire

- **Read()** – citește următorul caracter din stream-ul standard de intrare

- **ReadLine()** – citește următoarea linie de caractere din stream-ul standard de intrare

- **ReadKey()** – obține următorul caracter sau tastă function apăsată de către utilizator

- **Clear()** – șterge buffer-ul consolei și informația afișată în fereastra consolei

SqlConnection

- Reprezintă o conexiune deschisă la baza de date
- Nu poate fi moștenită
- Dacă iese din domeniul de vizibilitate, nu este închisă (conexiunea trebuie închisă în mod explicit)
- **Proprietăți:**
 - **ConnectionString** – returnează sau setează string-ul folosit pentru a deschide o bază de date SQL Server
 - **ConnectionTimeout** – returnează timpul de așteptare pentru stabilirea unei conexiuni (la expirare, încercarea de stabilire a conexiunii se termină și este generată o eroare)
- **Metode**
 - **Open()** – deschide o conexiune la baza de date cu setările specificate în *ConnectionString*
 - **Close()** – închide conexiunea la baza de date
- Dacă este generat un *SqlException*, *SqlConnection* rămâne deschisă dacă nivelul de severitate al erorii este ≤ 19

SqlCommand

- Reprezintă o instrucțiune sau procedură stocată Transact-SQL care se dorește a fi executată pe o bază de date SQL Server
- **Proprietăți:**
 - **CommandText** – returnează sau setează instrucțiunea Transact-SQL, numele tabelului sau a procedurii stocate ce urmează să fie executată la nivelul sursei de date
 - **CommandTimeout** – returnează sau setează timpul de așteptare pentru execuția unei comenzi (la expirare, încercarea de execuție a comenzii se termină și este generată o eroare)
- **Metode:**
 - **ExecuteNonQuery()** – execută o instrucțiune Transact-SQL folosind o conexiune și returnează numărul de înregistrări afectate
 - **ExecuteScalar()** – execută interogarea și returnează valoarea primei coloane a primei înregistrări din result-set-ul returnat de către interogare (coloanele și înregistrările adiționale sunt ignorate)
 - **ExecuteReader()** – construiește un *SqlDataReader*

SqlDataReader

- **SqlDataReader** citește un stream forward-only de înregistrări dintr-o bază de date SQL Server
- Pentru a crea un *SqlDataReader*, se va apela metoda *ExecuteReader* a unui obiect *SqlCommand* în locul folosirii directe a unui constructor
- În timpul utilizării unui *SqlDataReader*, *SqlConnection*-ul asociat este ocupat și nicio altă operație nu poate fi aplicată pe *SqlConnection*, în afară de închiderea sa
- Singurele proprietăți care pot fi accesate după ce *SqlDataReader*-ul a fost închis sunt *IsClosed* și *RecordsAffected*
- Modificările efectuate într-un result-set de către alt proces sau fir de execuție în timp ce datele sunt citite pot fi vizibile pentru utilizatorul *SqlDataReader*-ului (totuși, comportamentul exact este dependent de sincronizare)

SqlDataAdapter

- Este o punte între *DataSet* și SQL Server pentru obținerea și salvarea datelor
- Atunci când un *SqlDataAdapter* umple un *DataSet*, creează tabelele și coloanele necesare pentru datele returnate, dacă acestea nu există deja
- *SqlDataAdapter* este folosit împreună cu *SqlConnection* și *SqlCommand* pentru a îmbunătăți performanța în cazul conectării la o bază de date SQL Server
- **Proprietăți:**
 - **InsertCommand** – returnează sau setează instrucțiunea Transact-SQL sau procedura stocată folosită pentru a adăuga noi înregistrări în sursa de date
 - **UpdateCommand** – returnează sau setează instrucțiunea Transact-SQL sau procedura stocată folosită pentru a actualiza înregistrări în sursa de date
 - **DeleteCommand** – returnează sau setează instrucțiunea Transact-SQL sau procedura stocată folosită pentru a șterge înregistrări din setul de date

SqlDataAdapter

- **Metode:**

Fill(DataSet, String) – adaugă sau reîncarcă înregistrările în obiectul *DataSet*, astfel încât acestea să corespundă celor din sursa de date folosind numele *DataSet*-ului și al *DataTable*-ului

Update(DataSet, String) – modifică valorile din baza de date, executând instrucțiunile INSERT, UPDATE sau DELETE pentru fiecare înregistrare adăugată, modificată sau ștearsă din *DataSet*, folosind numele specificat al *DataTable*-ului

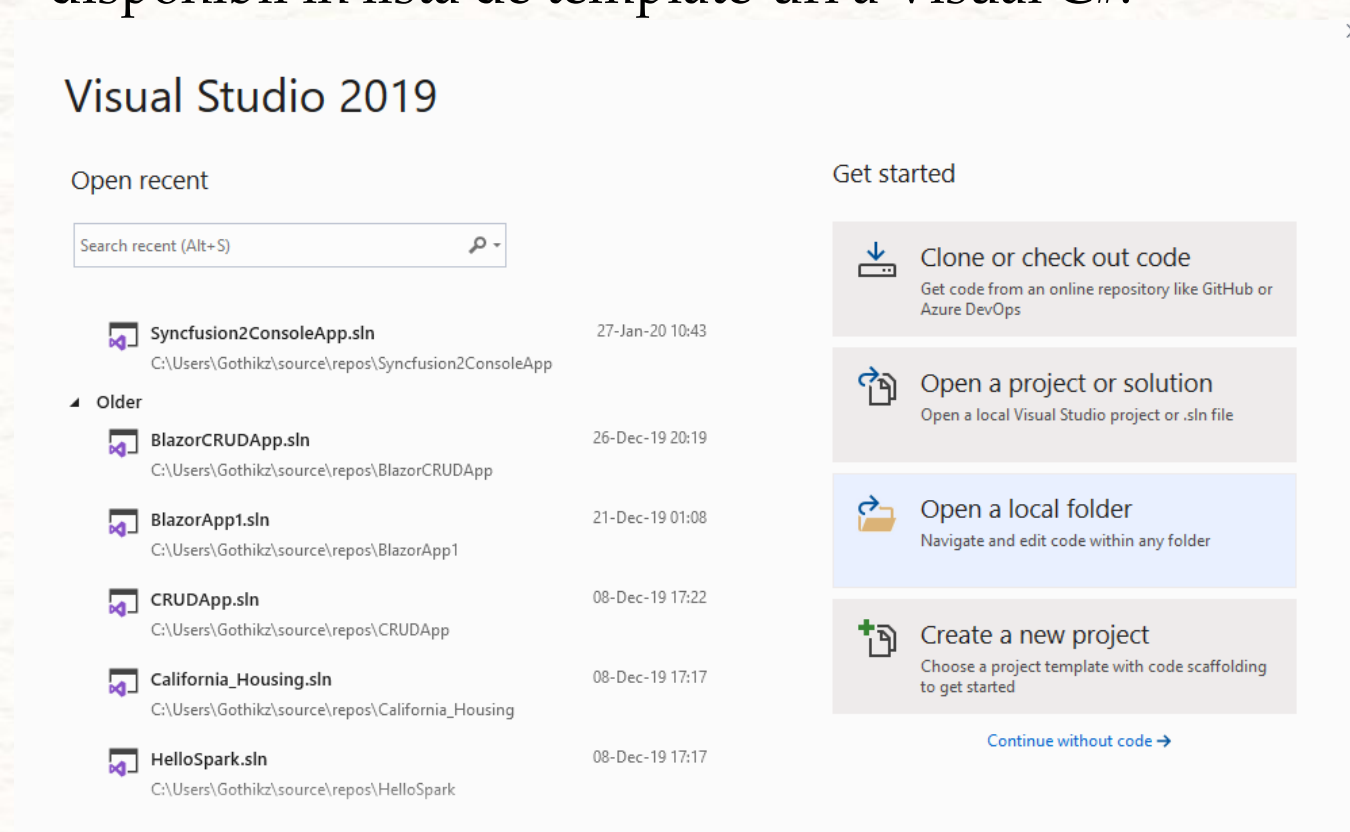
Exemplu – C# Console App

- În SQL Server, vom crea o nouă bază de date numită “SGBDIR”
- După ce baza de date a fost creată, vom crea un tabel nou:

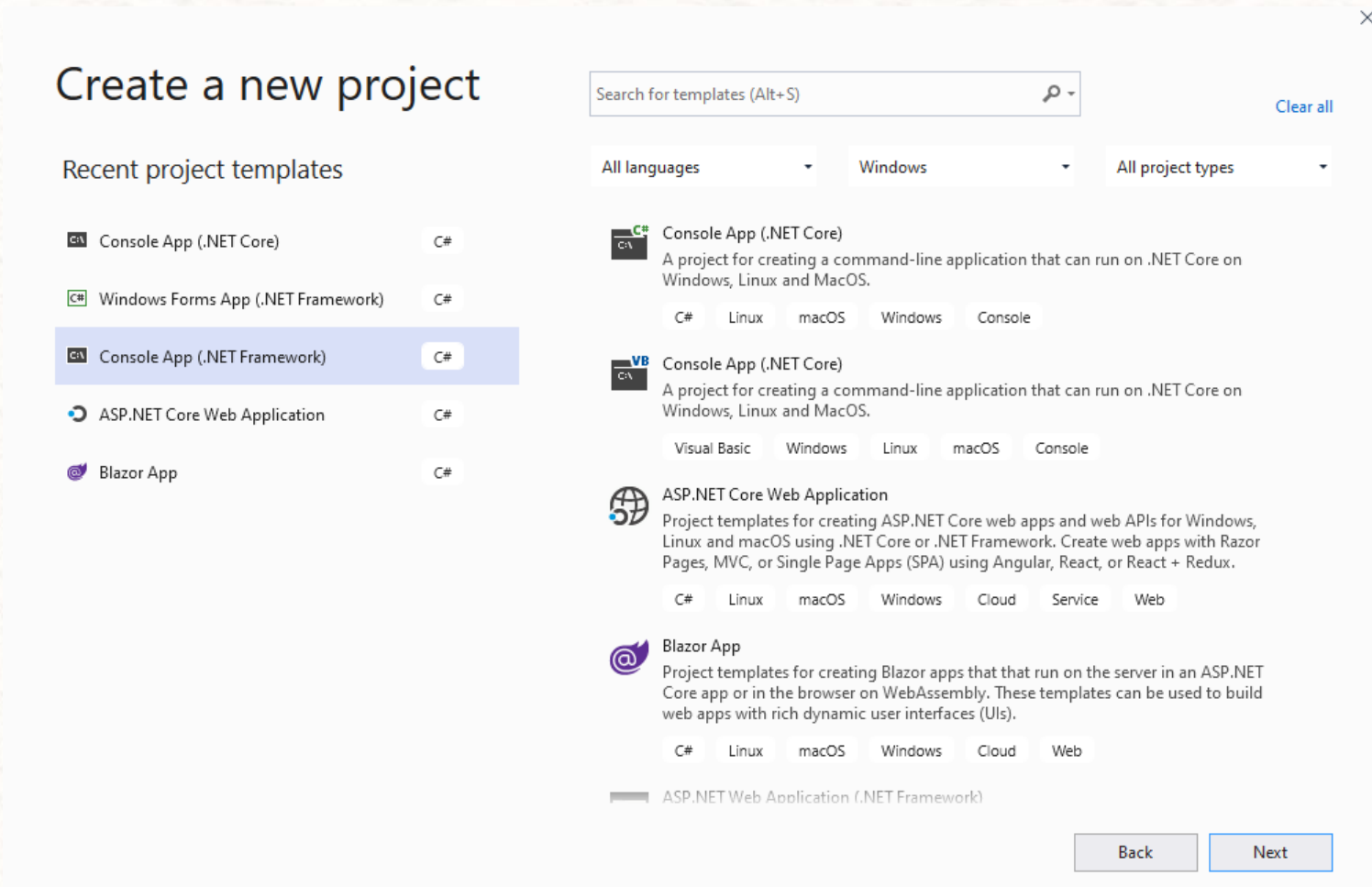
```
CREATE TABLE Cadouri  
(  
    cod_cadou INT PRIMARY KEY IDENTITY,  
    descriere VARCHAR(100),  
    posesor VARCHAR(100),  
    pret REAL  
);
```

Exemplu – C# Console App

- În Visual Studio, vom crea un nou proiect folosind template-ul Console App disponibil în lista de template-uri a Visual C#:



Exemplu – C# Console App



Exemplu – C# Console App

×

Configure your new project

Console App (.NET Framework) C# Windows Console

Project name

SqlConsoleApp

Location

C:\Users\userpc\source\repos

...

Solution name i

SqlConsoleApp

☒ Place solution and project in the same directory

Framework

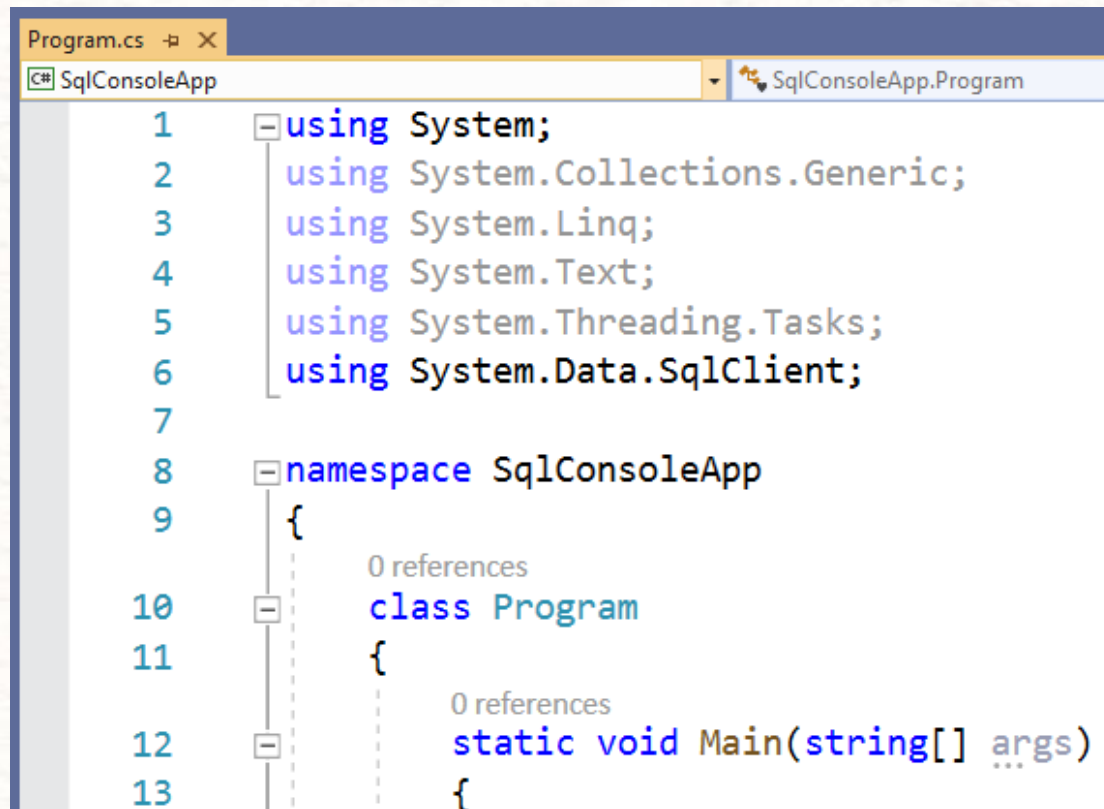
.NET Framework 4.7.2

Back

Create

Exemplu – C# Console App

- După ce proiectul a fost creat, vom include namespace-ul System.Data.SqlClient, care este .NET Data Provider pentru SQL Server:



```
Program.cs
C# SqlConsoleApp
1  using System;
2      using System.Collections.Generic;
3      using System.Linq;
4      using System.Text;
5      using System.Threading.Tasks;
6      using System.Data.SqlClient;
7
8  namespace SqlConsoleApp
9  {
10     0 references
11     class Program
12     {
13         0 references
14         static void Main(string[] args)
15         {
```

Exemplu – C# Console App

- Următoarea secvență de cod (inclusă în fișierul Program.cs) deschide o conexiune la baza de date “SGBDIR” pentru a adăuga, actualiza și șterge înregistrări din tabelul “Cadouri” folosind *SqlCommand* și *SqlDataReader*:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Data.SqlClient;

namespace SqlConsoleApp

{class Program

    {static void Main(string[] args)
```


Exemplu – C# Console App

```
{//Setarea titlului consolei

    Console.Title = "SqlConsoleApp";

    string connectionString =
        "Server=ACERASPIRE;Database=SGBDIR;Integrated Security=true;";

    try
    {

        using(SqlConnection connection=new SqlConnection(connectionString))
        {

            //Deschiderea conexiunii și verificarea stării

            connection.Open();

            Console.WriteLine("Starea conexiunii: {0}", connection.State);
```

Exemplu – C# Console App

```
//Adăugare
```

```
string desc1 = "lumanare";
```

```
string posesor1 = "Ion";
```

```
float pret1 = 2.5F;
```

```
string desc2 = "bicicleta";
```

```
string posesor2 = "Ioana";
```

```
float pret2 = 12.5F;
```

```
SqlCommand insertCommand = new SqlCommand("INSERT INTO Cadouri  
(descriere,posesor,pret) VALUES (@desc1,@posesor1,@pret1),  
(@desc2,@posesor2,@pret2);", connection);
```


Exemplu – C# Console App

```
insertCommand.Parameters.AddWithValue("@desc1", desc1);
insertCommand.Parameters.AddWithValue("@posesor1", posesor1);
insertCommand.Parameters.AddWithValue("@pret1", pret1);
insertCommand.Parameters.AddWithValue("@desc2", desc2);
insertCommand.Parameters.AddWithValue("@posesor2", posesor2);
insertCommand.Parameters.AddWithValue("@pret2", pret2);
int insertRowCount = insertCommand.ExecuteNonQuery();
if(insertRowCount==1)
    Console.WriteLine("A fost adaugata o inregistrare");
else
    Console.WriteLine("Au fost adaugate {0} inregistrari", insertRowCount);
```

Exemplu – C# Console App

```
//Returnare înregistrări
```

```
SqlCommand selectCommand = new SqlCommand("SELECT descriere, posesor, pret FROM  
Cadouri;", connection);
```

```
SqlDataReader reader = selectCommand.ExecuteReader();
```

```
if(reader.HasRows)
```

```
{
```

```
    Console.WriteLine("Instructiunea SELECT a returnat urmatorul result set:");
```

```
    while (reader.Read())
```

```
    {
```

```
        Console.WriteLine("{0}\t{1}\t{2}", reader.GetString(0), reader.GetString(1),  
reader.GetFloat(2));
```

```
    }
```


Exemplu – C# Console App

```
}  
else  
    Console.WriteLine("Nicio inregistrare returnata");  
    reader.Close();  
  
    //Actualizare  
  
    string descriereModificata = "Semn de carte";  
  
    SqlCommand updateCommand = new SqlCommand("UPDATE Cadouri SET  
descriere=@descriereModificata WHERE posesor=@posesor;", connection);  
  
    updateCommand.Parameters.AddWithValue("@descriereModificata",  
descriereModificata);  
  
    updateCommand.Parameters.AddWithValue("@posesor", posesor1);
```

Exemplu – C# Console App

```
int updateRowCount = updateCommand.ExecuteNonQuery();  
if(updateRowCount == 1)  
    Console.WriteLine("Actualizarea a afectat o inregistrare");  
else  
    Console.WriteLine("Actualizarea a afectat {0} inregistrari",  
        updateRowCount);  
  
//Ștergere  
SqlCommand deleteCommand = new SqlCommand("DELETE FROM Cadouri WHERE  
posesor=@posesor;", connection);
```


Exemplu – C# Console App

```
deleteCommand.Parameters.AddWithValue("@posesor", posesor2);  
int deleteRowCount = deleteCommand.ExecuteNonQuery();  
if(deleteRowCount == 1)  
    Console.WriteLine("Stergerea a afectat o inregistrare");  
else  
    Console.WriteLine("Stergerea a afectat {0} inregistrari",  
deleteRowCount);  
  
//Returnare înregistrări din nou  
reader = selectCommand.ExecuteReader();
```

Exemplu – C# Console App

```
if (reader.HasRows)
{
    Console.WriteLine("Dupa actualizare si stergere, instructiunea SELECT a returnat urmatorul result set:");
    while (reader.Read())
    {
        Console.WriteLine("{0}\t{1}\t{2}", reader.GetString(0), reader.GetString(1), reader.GetFloat(2));
    }
}
else
    Console.WriteLine("Nicio inregistrare returnata");
reader.Close();
}}
```


Exemplu – C# Console App

```
catch (Exception e)
{
    //Schimbă culoarea textului din consolă în roșu și afișează mesajul erorii
    Console.ForegroundColor = ConsoleColor.Red;
    Console.WriteLine("Mesajul erorii: \n{0}", e.Message);
    Console.ReadKey();
}
Console.ReadKey();
}
}
```

Exemplu – C# Console App

- După execuția aplicației, obținem următorul rezultat:

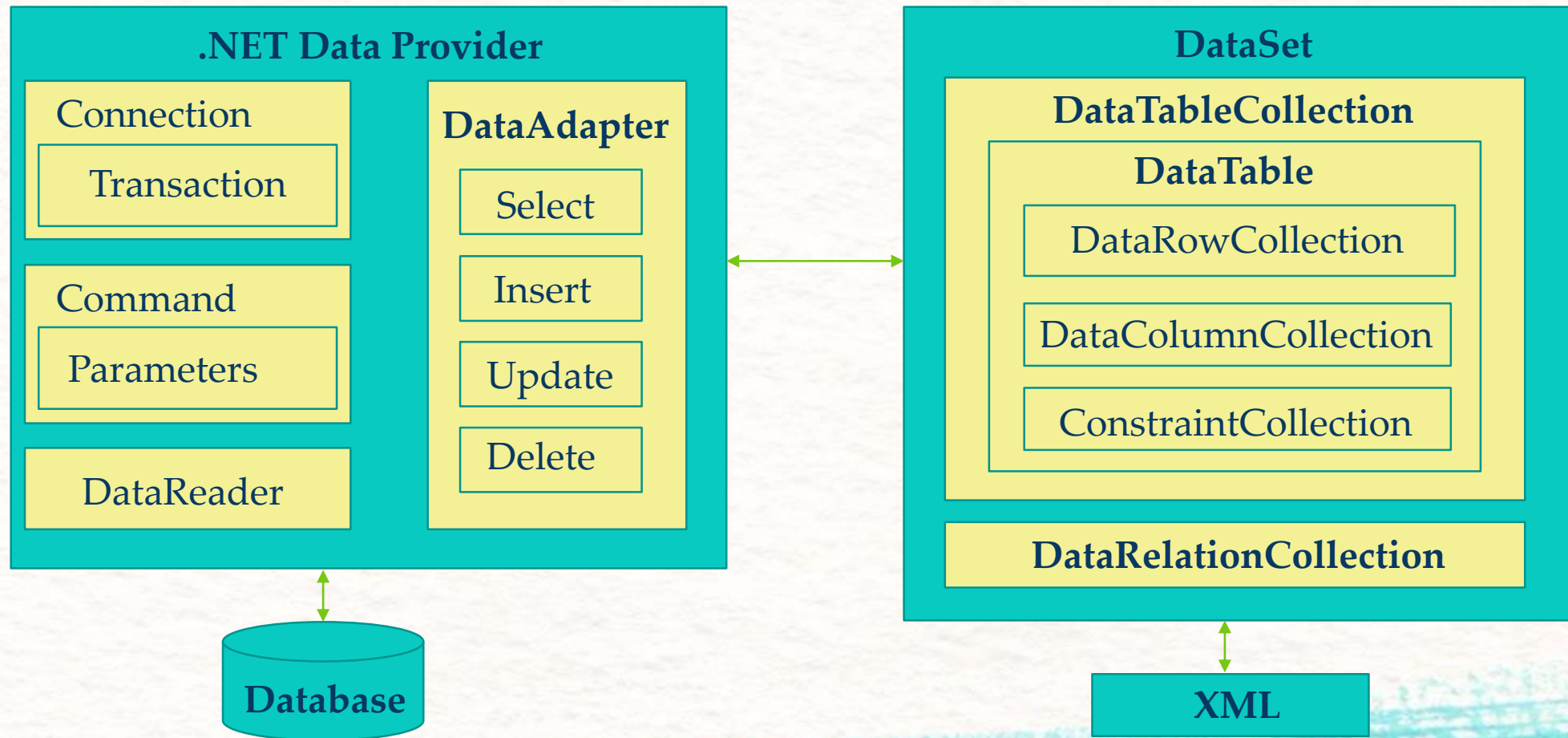
 SqlConsoleApp

```
Starea conexiunii: Open
Au fost adaugate 2 inregistrari
Instrucțiunea SELECT a returnat următorul result set:
lumanare      Ion      2.5
bicicleta     Ioana   12.5
Actualizarea a afectat o inregistrare
Stergerea a afectat o inregistrare
Dupa actualizare si stergere, instrucțiunea SELECT a returnat următorul result set:
Semn de carte Ion      2.5
```


ADO.NET

Seminar 2

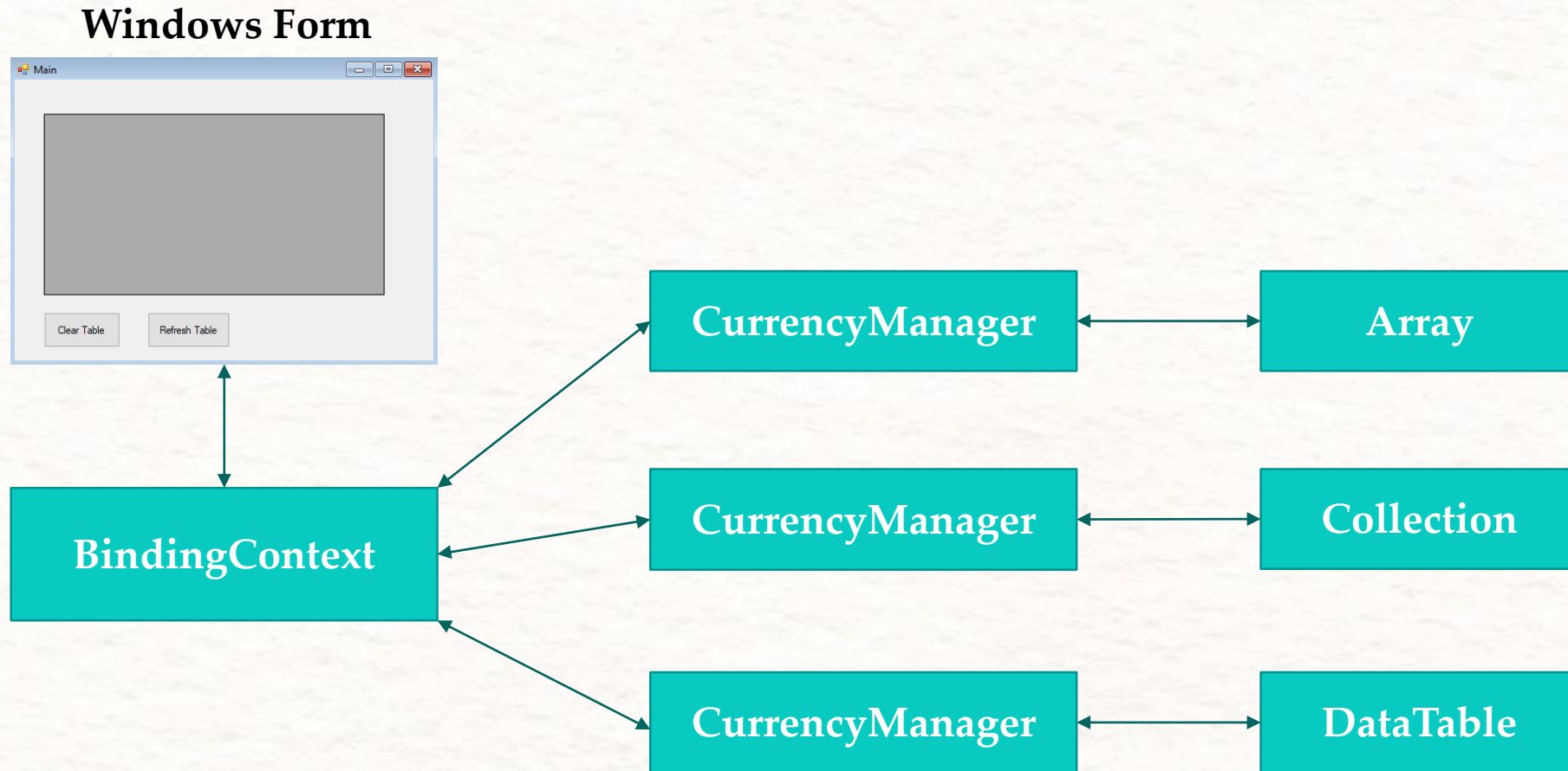
Arhitectura ADO.NET



Data Binding – Windows Forms

- În Windows Forms, **data binding**-ul oferă modalități de a afișa și de a modifica informații care provin din surse de date în controalele din form
- **Data binding**-ul este o modalitate automată de a seta orice proprietate accesibilă la runtime a oricărui control din form
- În Windows Forms, **data binding**-ul permite accesarea datelor din diferite surse de date (aproape orice structură care conține date poate fi o sursă de date: Array, Collection, DataTable, etc.)
- Există două tipuri de data binding:
 - **Simple data binding**: legarea unui control (TextBox, Label, etc.) la un singur data element, cum ar fi o valoare dintr-o coloană a unui DataTable dintr-un DataSet
 - **Complex data binding**: legarea unui control (DataGridView, ListBox, ComboBox, etc.) la mai multe data elements, în general la mai multe înregistrări dintr-o bază de date

Data Binding – Windows Forms



Consumatori de date în .NET

- **BindingContext** gestionează colecția de obiecte *BindingManagerBase* pentru orice obiect care derivă din clasa *Control*
- Fiecare Windows Form are cel puțin un obiect **BindingContext** care gestionează colecția de obiecte *BindingManagerBase* pentru form
- **BindingManagerBase** este o clasă abstractă, deci tipul returnat al proprietății *Item[Object]* poate fi **CurrencyManager** sau **PropertyManager**
- Dacă sursa de date este un obiect care poate returna doar o singură proprietate (în locul unei liste de obiecte), tipul va fi **PropertyManager**
- Dacă sursa de date este un obiect care implementează interfața *IList* sau *IBindingList*, tipul va fi **CurrencyManager**
- Pentru fiecare sursă de date asociată cu un Windows Form, există un singur obiect **CurrencyManager** sau **PropertyManager**

Consumatori de date în .NET

- **PropertyManager**

- Menține legătura dintre proprietatea unui obiect și proprietatea data-bound a unui control
- Derivă din clasa *BindingManagerBase*

- **CurrencyManager**

- Menține controalele legate la date sincronizate între ele (administrează o listă de obiecte Binding)
- Derivă din clasa *BindingManagerBase*
- **Proprietăți:**
 - **Current** – returnează elementul curent din listă
 - **Position** – returnează sau setează poziția curentă a tuturor controalelor conectate la același *CurrencyManager*

Controale data-bound

- **BindingSource**

- Conectează controalele din form la un *DataTable* din *DataSet*
- Simplifică legarea controalelor din form la date oferind currency management, change notification și alte servicii
- Sursele de date care sunt legate la o componentă **BindingSource** pot fi parcurse și administrate cu *BindingNavigator*

- **BindingNavigator**

- Folosit pentru parcurgerea înregistrărilor din tabel
- Interfața utilizator a unui **BindingNavigator** este compusă dintr-o serie de butoane *ToolStrip*, text boxes și static text elements pentru cele mai comune acțiuni asupra datelor, cum ar fi adăugarea, ștergerea și parcurgerea înregistrărilor

Surse de date în .NET

- O listă trebuie să implementeze interfața *IList* pentru a putea îndeplini funcția de sursă de date
- ADO.NET furnizează structuri de date potrivite pentru binding:
- **DataColumn**
 - Este componenta fundamentală în construirea structurii unui *DataTable* (structura tabelului este construită prin adăugarea unui obiect sau a mai multor obiecte *DataRow* unei *DataRowCollection*)
 - Are o proprietate numită *DataType* care determină tipul de date
- **DataTable**
 - Este un obiect central în biblioteca ADO.NET și poate fi folosit de obiectele *DataSet* și *DataView*
 - Reprezintă un tabel cu date stocate în memorie și conține o colecție de obiecte *DataRow*, o colecție de obiecte *DataRow* și o colecție de obiecte *Constraint*

Surse de date în .NET

- **DataView**

- Este un view personalizat pentru sortare, filtrare, căutare, editare și navigare a unui *DataTable* și permite data binding
- Permite crearea mai multor view-uri diferite a datelor stocate într-un *DataTable*
- Oferă vizualizarea dinamică a unui singur set de date, asupra căruia se pot aplica diferite criterii de sortare și filtrare (asemănător unui view dintr-o bază de date)
- Nu poate fi tratat ca un tabel și nu poate conține date din mai multe tabele
- Nu poate exclude coloane care există în tabelul sursă, nici nu poate adăuga coloane care nu există în tabelul sursă (cum ar fi coloanele calculate)
- Nu stochează date, ci doar reprezintă un view conectat al *DataTable*-ului corespunzător
- Poate fi personalizat pentru a afișa doar o parte a datelor din *DataTable* (acest lucru permite ca două controale legate la același *DataTable* să afișeze două versiuni diferite a datelor)

Surse de date în .NET

- **DataSet**

- Reprezintă un cache în memorie al datelor
- Este compus din tabele, relații și constrângeri

- **DataManager**

- Reprezintă un view personalizat al unui *DataSet*
- Poate fi folosit pentru a administra setările de vizualizare a tuturor tabelelor dintr-un *DataSet*
- Oferă o modalitate convenabilă de a administra setările implicite de vizualizare pentru fiecare tabel
- Conține un *DataSetSettingCollection* implicit pentru fiecare *DataTable* din *DataSet*

Popularea DataSet-urilor cu date

- Un *DataSet* nu conține date în mod implicit
- Tabelele sunt populate cu date prin execuția interogărilor *TableAdapter* sau prin execuția comenzilor *DataAdapter* (*SqlDataAdapter*)

- *DataSet* tipizat – exemplu:

```
categoriiFloriTableAdapter.Fill(florarieDataSet.CategoriiFlori);
```

- *DataSet* netipizat – exemplu:

```
categoriiProduseSqlDataAdapter.Fill(magazinDataSet,"CategoriiProduse");
```

Popularea DataSet-urilor cu date

- Salvarea datelor
- *DataSet* tipizat – exemplu:

```
categoriiFloriTableAdapter.Update(florarieDataSet.CategoriiFlori);
```

- *DataSet* netipizat – exemplu:

```
categoriiProduseSqlDataAdapter.Update(magazinDataSet, "CategoriiProduse");
```

- Metoda **Update** examinează valoarea proprietății *RowState* pentru a determina care sunt înregistrările ce urmează să fie salvate și care comandă specifică trebuie invocată (*InsertCommand*, *UpdateCommand* sau *DeleteCommand*)

Accesarea înregistrărilor

- Fiecare tabel expune o colecție de înregistrări
- Ca în orice colecție, înregistrările se pot accesa folosind indexul colecției sau utilizând instrucțiuni specifice colecției în limbajul de programare utilizat

- *DataSet* tipizat – exemplu:

```
textBox1.Text = florarieDataSet.CategoriiFlori[1].nume;
```

- *DataSet* netipizat – exemplu:

```
string numeCategorie = (string)  
magazinDataSet.Tables["CategoriiProduse"].Rows[0]["nume"];
```

Tabele asociate și obiecte *DataRelation*

- Informațiile din tabelele aflate într-un *DataSet* pot fi inter-relaționate
- Crearea obiectelor *DataRelation* permite descrierea relațiilor dintre tabelele care se află în *DataSet*
- Se poate folosi un obiect *DataRelation* pentru a localiza înregistrări asociate prin apelarea metodei **GetChildRows** pe un *DataRow* din tabelul părinte (această metodă returnează un array de înregistrări copil asociate)
- Se poate apela metoda **GetParentRow** a unui *DataRow* din tabelul copil (această metodă returnează un singur *DataRow* din tabelul părinte)

Returnarea înregistrărilor copil a unei înregistrări părinte

- *DataSet* tipizat – exemplu:

```
//Se va afișa numărul de comenzi ale clientului cu ID-ul "ALFKI"  
string custID = "ALFKI";  
  
NorthwindDataSet.OrdersRow[] orders;  
  
orders = (NorthwindDataSet.OrdersRow[])  
northwindDataSet.Customers.  
FindByCustomerID(custID).GetChildRows  
("FK_Orders_Customers");  
  
MessageBox.Show(orders.Length.ToString());
```

Returnarea înregistrărilor copil a unei înregistrări părinte

- *DataSet* netipizat – exemplu:

```
//Se va afișa numărul de produse pentru înregistrarea aflată pe  
poziția 2 din DataTable "CategoriiProduse"
```

```
DataRow[] produse;
```

```
produse = magazinDataSet.Tables["CategoriiProduse"].Rows[2].
```

```
GetChildRows("FK_CategoriiProduse_Produse");
```

```
MessageBox.Show(produse.Length.ToString());
```


Returnarea înregistrării părinte a unei înregistrări copil

- *DataSet* tipizat – exemplu :

```
//Se va afișa numele companiei pentru clientul care a făcut  
comanda cu ID-ul 10707
```

```
int orderID = 10707;
```

```
NorthwindDataSet.CustomersRow customer;
```

```
customer = (NorthwindDataSet.CustomersRow)  
northwindDataSet.Orders.FindByOrderID(orderID).
```

```
GetParentRow("FK_Orders_Customers");
```

```
MessageBox.Show(customer.CompanyName);
```

Returnarea înregistrării părinte a unei înregistrări copil

- *DataSet* netipizat – exemplu :

```
//Se va afișa numele categoriei pentru înregistrarea  
aflată pe poziția 1 din DataTable "Produse"
```

```
DataRow categorieProduse;
```

```
categorieProduse = magazinDataSet.Tables["Produse"].
```

```
Rows[1].GetParentRow("FK_CategoriiProduse_Produse");
```

```
MessageBox.Show(categorieProduse["nume"].ToString());
```


Constrângeri

- Constrângerile pot fi folosite pentru a impune restricții asupra datelor stocate în *DataTable* (pentru a menține integritatea datelor) și sunt adăugate în *ConstraintCollection* a obiectului *DataTable*
- O constrângere este o regulă care se aplică în mod automat unei coloane sau unui grup de coloane și determină modul de acțiune în momentul în care valoarea unei înregistrări este modificată în vreun fel
- Constrângerile sunt aplicate când valoarea proprietății *EnforceConstraints* a unui obiect *DataSet* este **True** (în mod implicit este setată pe **True**)
- Două tipuri de constrângeri sunt disponibile în ADO.NET și sunt create în mod automat atunci când un obiect *DataRelation* este adăugat unui *DataSet*:
 - **ForeignKeyConstraint**
 - **UniqueConstraint**

Constrângeri

- **ForeignKeyConstraint**

- Impune reguli privind propagarea actualizărilor și ștergerilor în tabelele asociate
- Proprietățile **UpdateRule** și **DeleteRule** ale *ForeignKeyConstraint* definesc acțiunea care va avea loc atunci când utilizatorul actualizează sau șterge o înregistrare dintr-un tabel asociat
- Proprietățile **UpdateRule** și **DeleteRule** pot avea următoarele valori:
 - **Cascade** – Actualizează sau șterge înregistrările asociate
 - **SetNull** – Setează valorile din înregistrările asociate pe *DBNull*
 - **SetDefault** – Setează valorile din înregistrările asociate pe valoarea implicită
 - **None** – Nicio acțiune nu are loc asupra înregistrărilor asociate
- În mod implicit, valorile proprietăților **UpdateRule** și **DeleteRule** sunt setate pe **None**

- **UniqueConstraint**

- Asigură unicitatea la nivel de înregistrare a valorilor din coloana sau coloanele pe care este definită
- Poate fi atribuită unei singure coloane sau unui array de coloane dintr-un *DataTable* și se poate specifica dacă această coloană (sau coloane) formează o cheie primară

Constrângeri

- Exemplu – Crearea unui *UniqueConstraint* cu proprietatea **IsPrimaryKey** setată pe **True** :

```
DataTable Cadouri = new DataTable("Cadouri");  
  
Cadouri.Columns.Add("cod_cadou", Type.GetType("System.Int32"));  
  
Cadouri.Columns.Add("descriere", Type.GetType("System.String"));  
  
UniqueConstraint uq = new UniqueConstraint("unique_constraint",  
    Cadouri.Columns["cod_cadou"], true);  
  
Cadouri.Constraints.Add(uq);  
  
MessageBox.Show("Valoarea proprietății IsPrimaryKey este "+  
    uq.IsPrimaryKey.ToString());
```

Exemplu de aplicație C# Windows Forms - DataRelation

- În SQL Server, vom crea o bază de date numită "SGBDIR"
- După ce baza de date a fost creată, vom crea două tabele noi:

```
CREATE TABLE Categorii  
(  
    cod_categorie INT PRIMARY KEY IDENTITY,  
    nume_categorie VARCHAR(100)  
);
```


Exemplu de aplicație C# Windows Forms - DataRelation

```
CREATE TABLE Produse
(
    cod_produș INT PRIMARY KEY IDENTITY,
    nume_produș VARCHAR(100),
    pret REAL,
    cod_categorie INT FOREIGN KEY REFERENCES Categoriile
    (cod_categorie)
);
```

Exemplu de aplicație C# Windows Forms - DataRelation

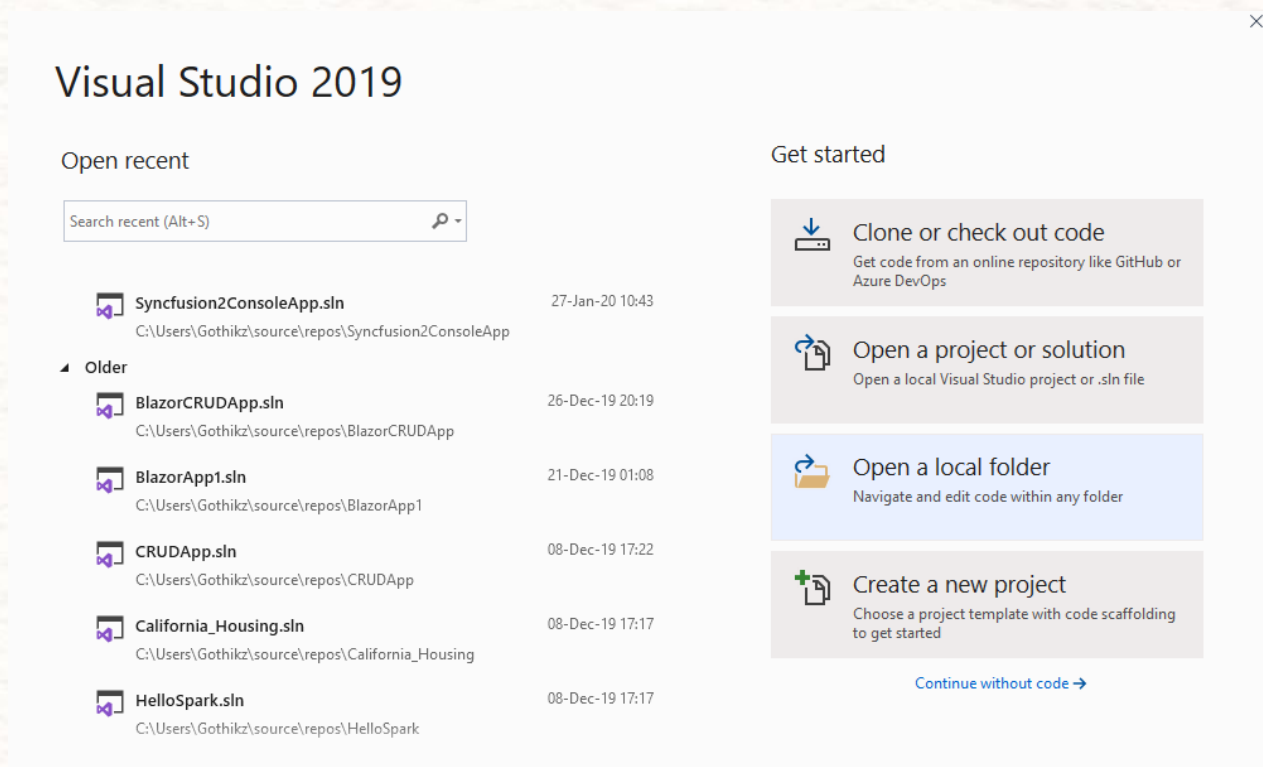
- Apoi, vom adăuga câteva înregistrări în fiecare tabel:

```
INSERT INTO Categori (nume_categorie) VALUES  
( 'dulciuri'), ('haine');
```

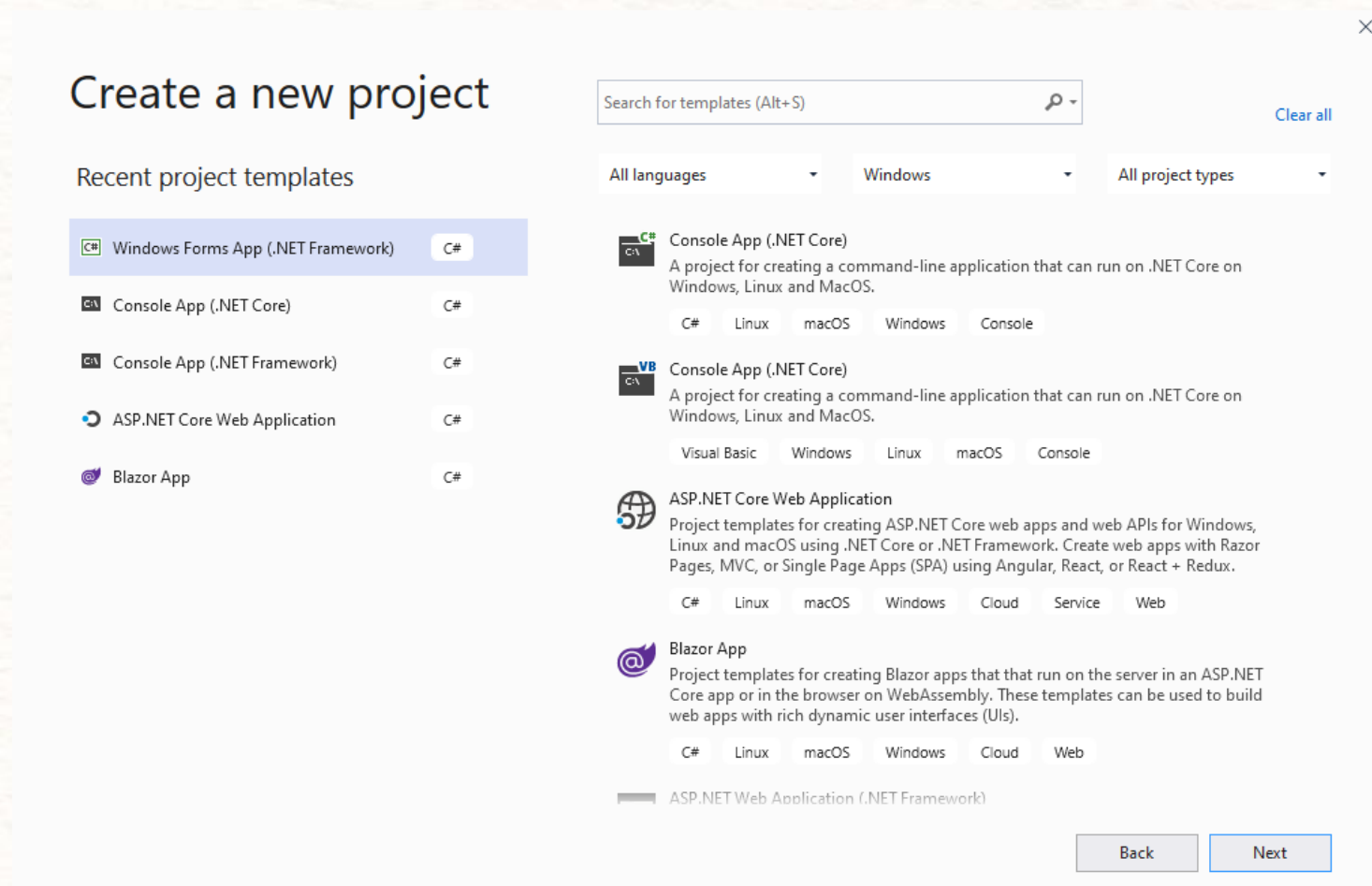
```
INSERT INTO Produse (nume_produs, pret, cod_categorie)  
VALUES ( 'Milka', 3, 1), ( 'Oreo', 2.5, 1),  
( 'Tricou', 56, 2), ( 'Blugi', 100, 2);
```


Exemplu de aplicație C# Windows Forms - DataRelation

- În Visual Studio, vom crea un nou proiect folosind template-ul Windows Forms App disponibil în lista de template-uri a Visual C#:



Exemplu de aplicație C# Windows Forms - DataRelation



Exemplu de aplicație C# Windows Forms - DataRelation

×

Configure your new project

Windows Forms App (.NET Framework) C# Windows Desktop

Project name

Location

...

Solution name ⓘ

☒ Place solution and project in the same directory

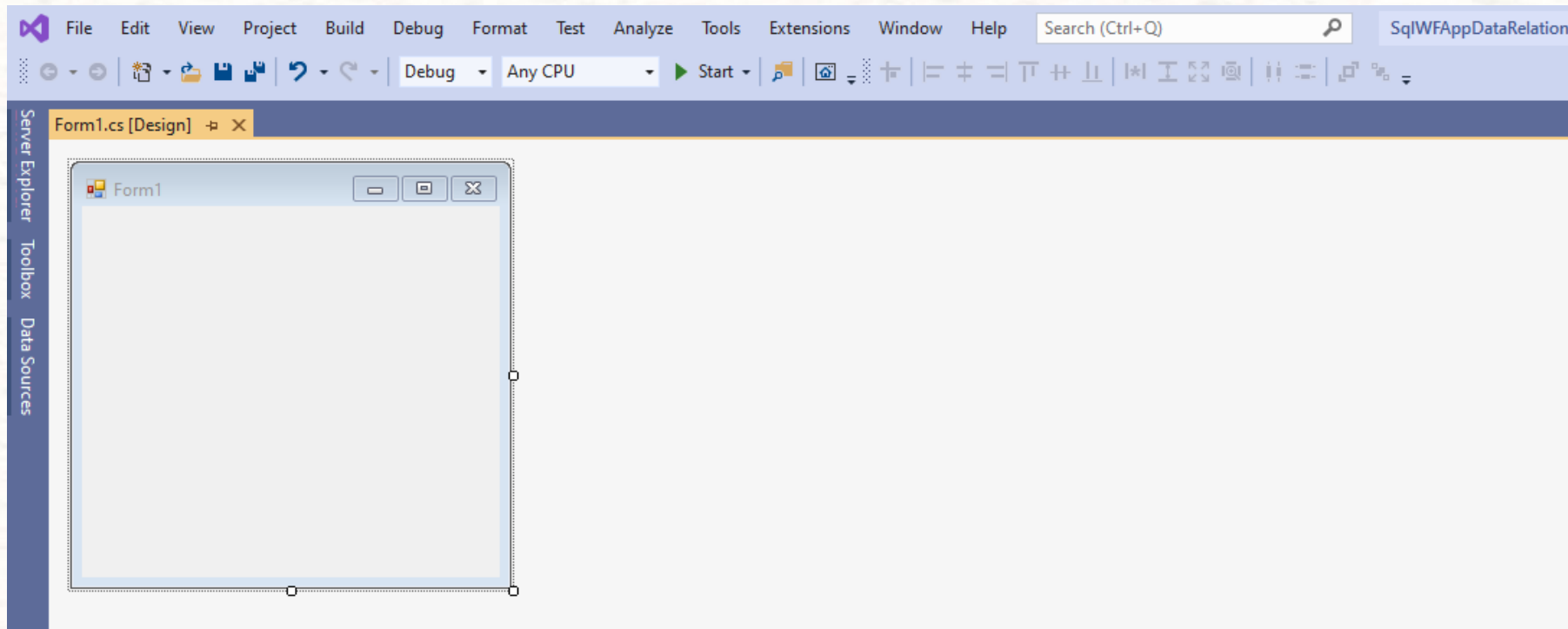
Framework

Back

Create

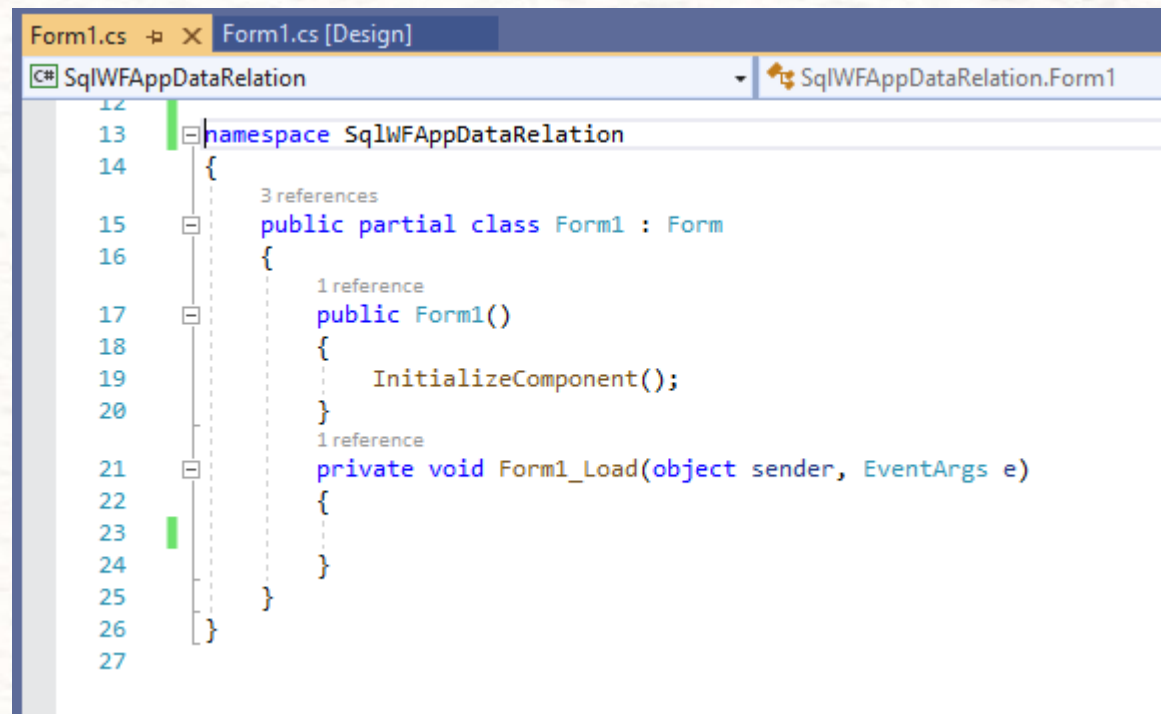
Exemplu de aplicație C# Windows Forms - DataRelation

- După ce proiectul a fost creat, apare primul *Form* al aplicației, denumit *Form1*:



Exemplu de aplicație C# Windows Forms - DataRelation

- După un dublu click pe *Form1*, se va crea event handler-ul *Form1_Load* (vizibil în fișierul **Form1.cs**):

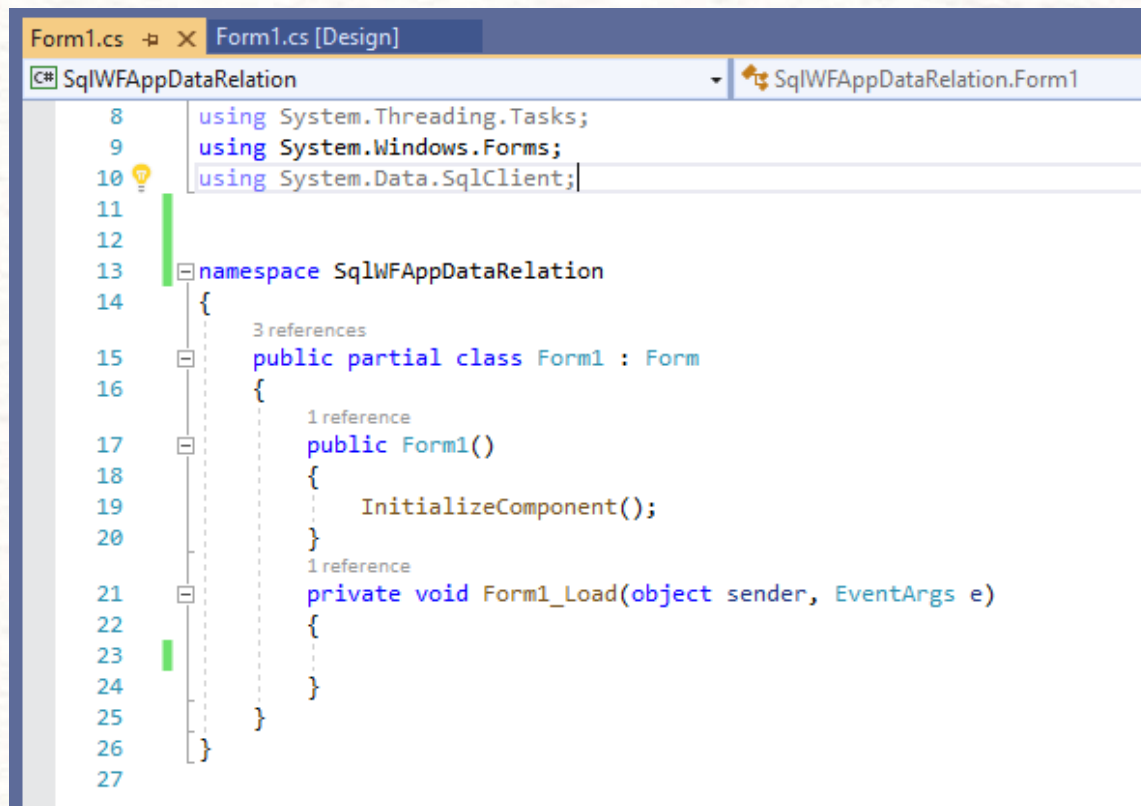


The screenshot shows the Visual Studio IDE with the 'Form1.cs [Design]' window active. The code editor displays the following C# code:

```
12
13 namespace SqlWFApDataRelation
14 {
15     3 references
16     public partial class Form1 : Form
17     {
18         1 reference
19         public Form1()
20         {
21             InitializeComponent();
22         }
23         1 reference
24         private void Form1_Load(object sender, EventArgs e)
25         {
26         }
27     }
```

Exemplu de aplicație C# Windows Forms - DataRelation

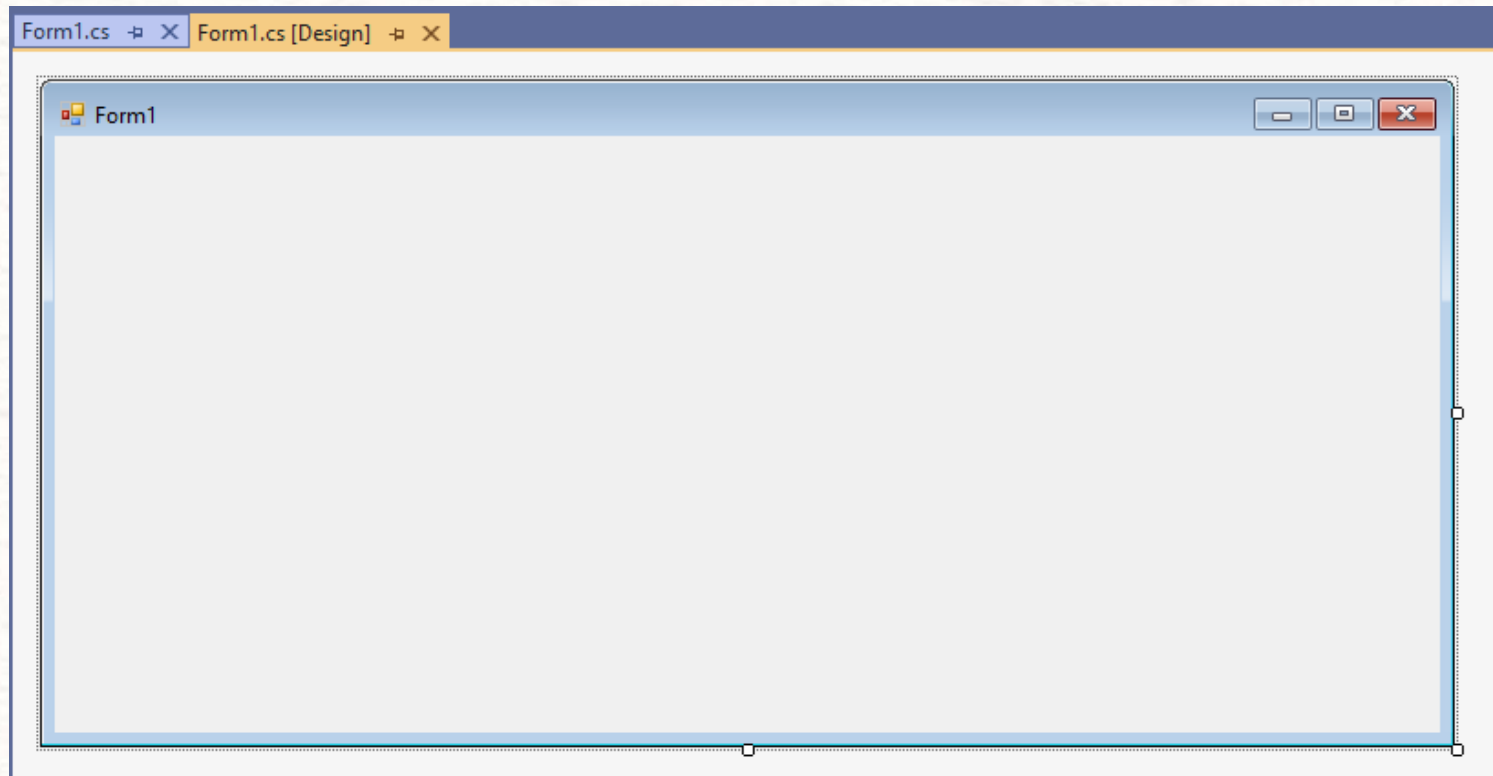
- După aceea, vom include în fișierul **Form1.cs** namespace-ul **System.Data.SqlClient** care este .NET Data Provider pentru SQL Server:



```
Form1.cs [Design]
SqlWFAppDataRelation
8  using System.Threading.Tasks;
9  using System.Windows.Forms;
10 using System.Data.SqlClient;
11
12
13 namespace SqlWFAppDataRelation
14 {
15     3 references
16     public partial class Form1 : Form
17     {
18         1 reference
19         public Form1()
20         {
21             InitializeComponent();
22         }
23         1 reference
24         private void Form1_Load(object sender, EventArgs e)
25         {
26         }
27     }
28 }
```

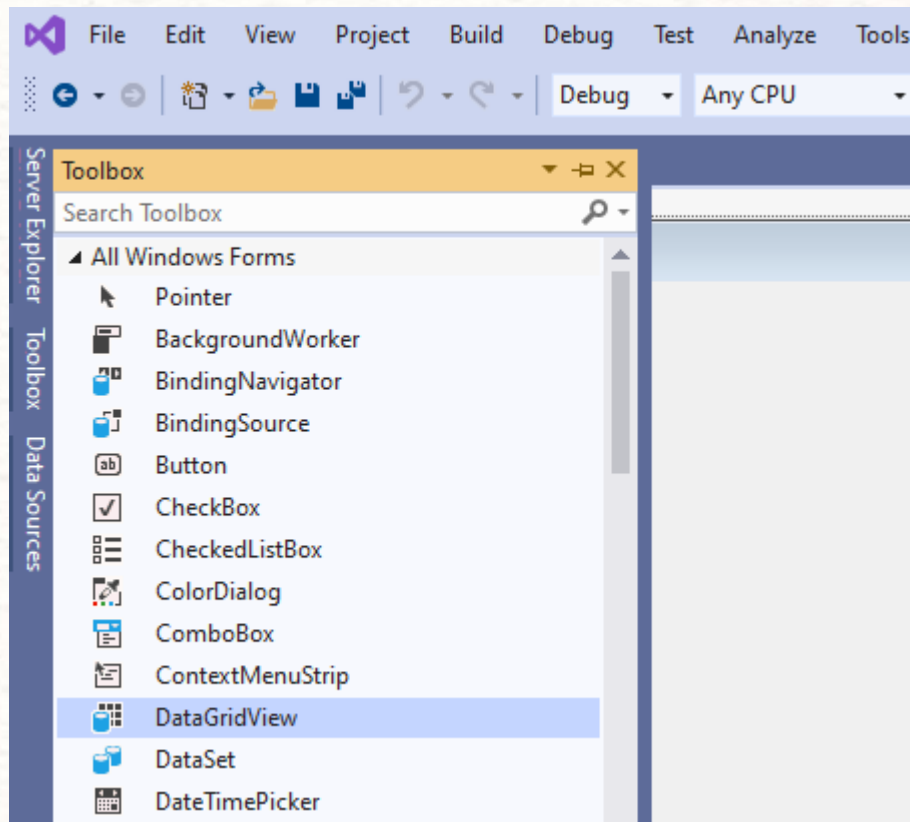

Exemplu de aplicație C# Windows Forms - DataRelation

- Deoarece avem nevoie de mai mult spațiu pentru a plasa două controale *DataGridView* pe *Form*, îl vom extinde:



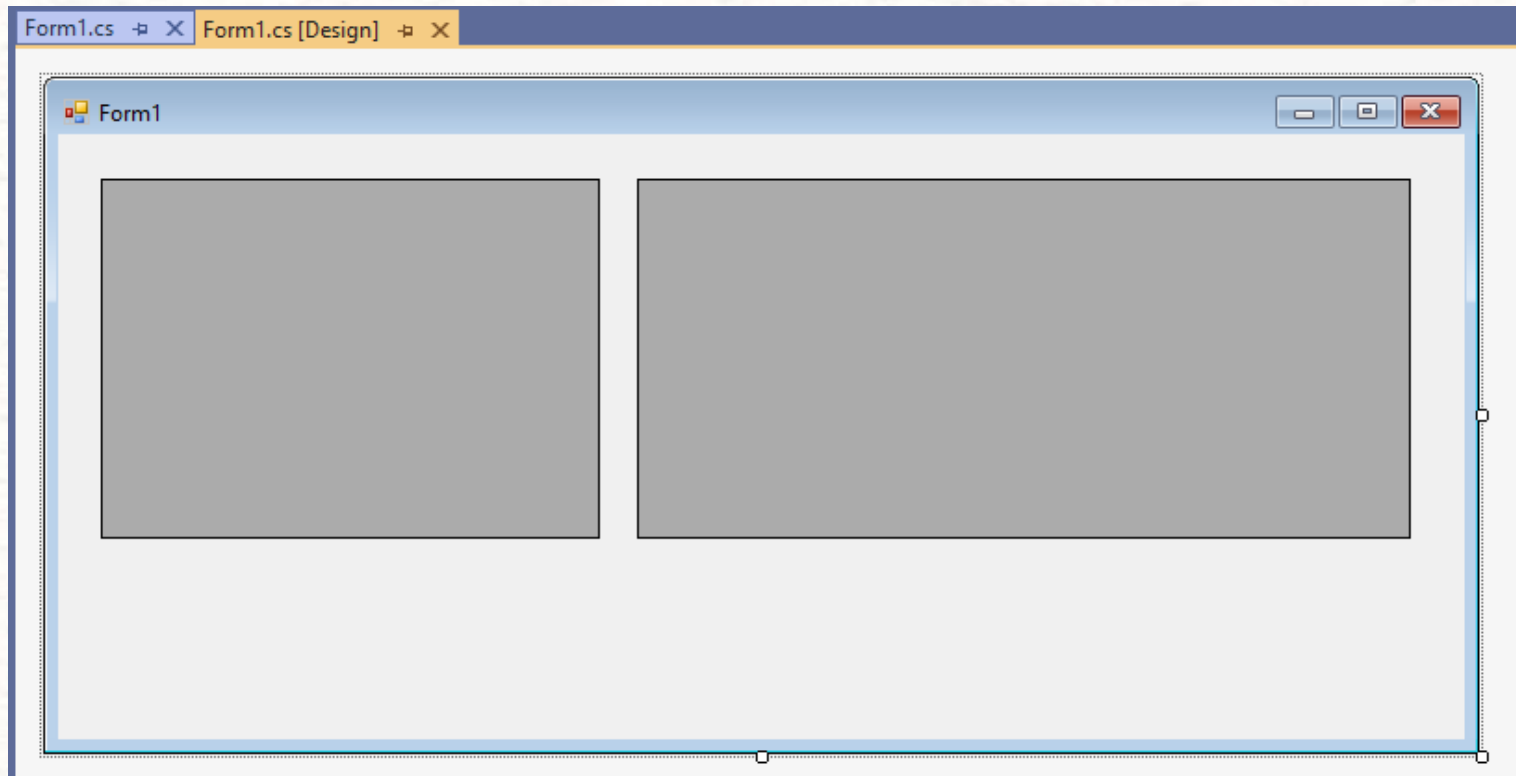
Exemplu de aplicație C# Windows Forms - DataRelation

- Din **Toolbox**, vom plasa în interiorul *Form*-ului două controale *DataGridView*:



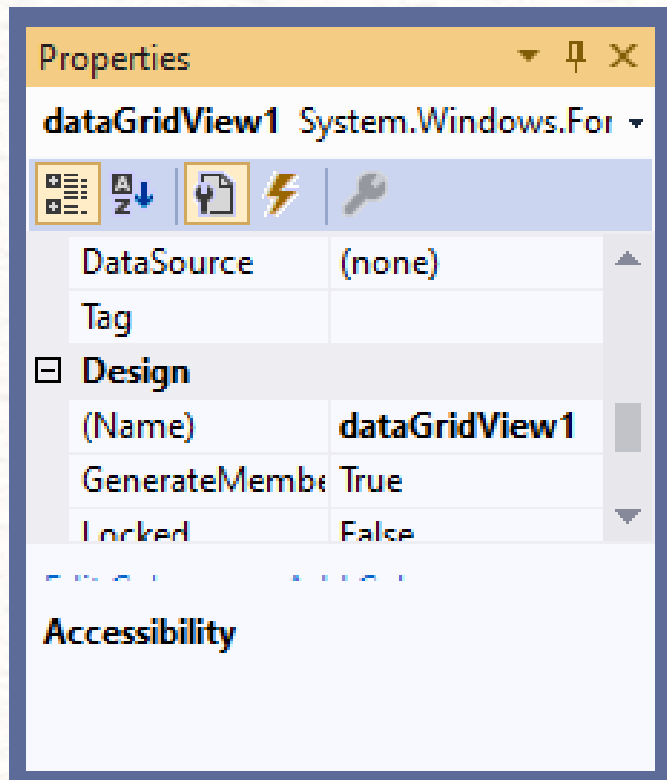
Exemplu de aplicație C# Windows Forms - DataRelation

- După plasarea celor două controale *DataGridView*, *Form*-ul arată în modul următor:



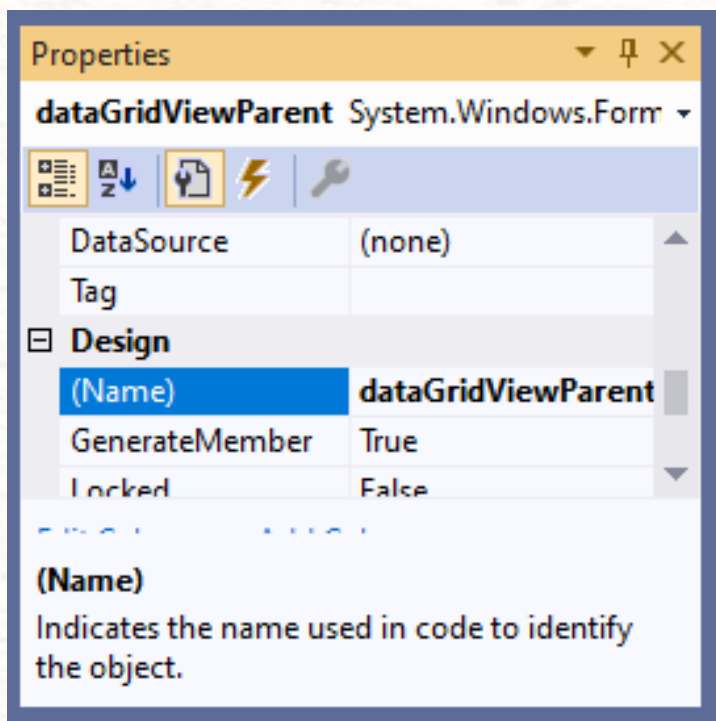
Exemplu de aplicație C# Windows Forms - DataRelation

- După un click pe primul *DataGridView*, putem vedea în fereastra **Properties** că proprietatea **Name** este setată pe “dataGridView1”:



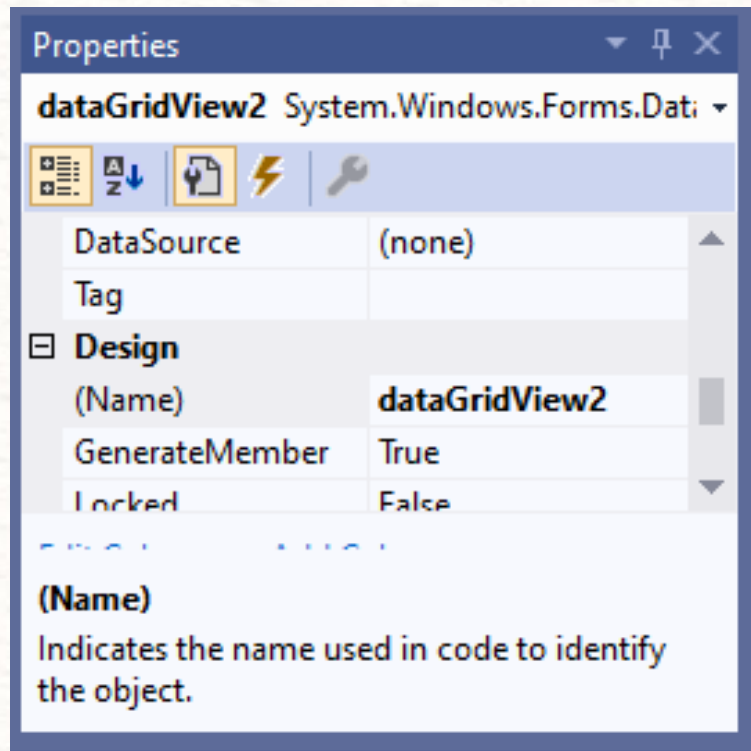
Exemplu de aplicație C# Windows Forms - DataRelation

- Deoarece în primul *DataGridView* vor fi afișate datele stocate în tabelul părinte (tabelul "Categorii"), vom schimba valoarea proprietății **Name** din "dataGridView1" în "dataGridViewParent":



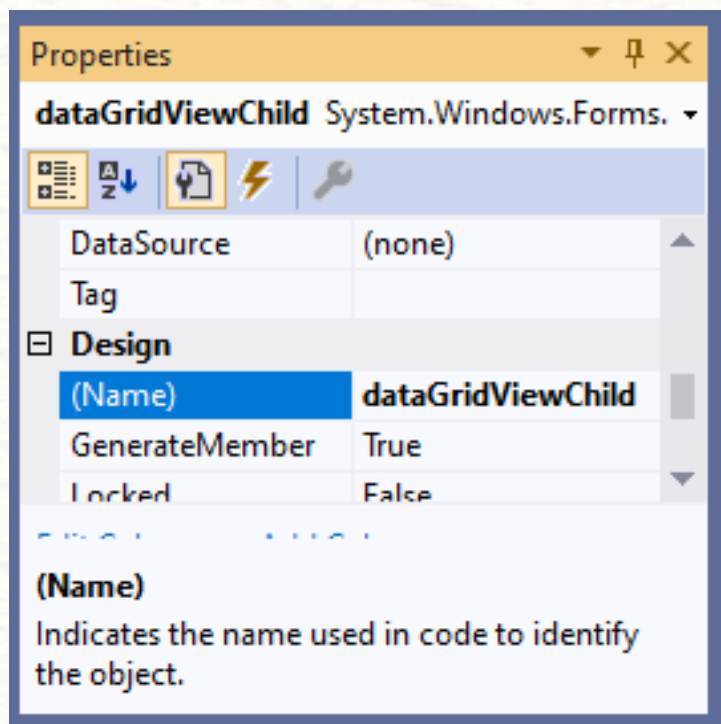
Exemplu de aplicație C# Windows Forms - DataRelation

- După un click pe al doilea *DataGridView*, putem vedea în fereastra **Properties** că proprietatea **Name** este setată pe “dataGridView2”:



Exemplu de aplicație C# Windows Forms - DataRelation

- Deoarece în al doilea *DataGridView* vor fi afișate datele stocate în tabelul copil (tabelul “Produce”), vom schimba valoarea proprietății **Name** din “dataGridView2” în “dataGridViewChild”:



Exemplu de aplicație C# Windows Forms - DataRelation

- Următoarea secvență de cod (inclusă în fișierul **Form1.cs**) deschide o conexiune la baza de date pentru a crea și a popula două *DataTables* ("Categorii" și "Produse") dintr-un *DataSet* folosind *SqlDataAdapters*
- Un *BindingSource* leagă *DataTable*-ul "Categorii" (tabelul părinte) de *DataGridView*-ul corespunzător
- Între cele două *DataTables* se va crea un *DataRelation*
- *DataRelation*-ul este folosit pentru a afișa doar acele înregistrări din tabelul copil care sunt asociate înregistrării selectate din tabelul părinte
- Înregistrările din tabelul copil sunt afișate în *DataGridView*-ul corespunzător *DataTable*-ului copil ("Produse")

Exemplu de aplicație C# Windows Forms - DataRelation

```
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;
```

Exemplu de aplicație C# Windows Forms - DataRelation

```
namespace SqlWFAppDataRelation
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```


Exemplu de aplicație C# Windows Forms - DataRelation

```
private void Form1_Load(object sender, EventArgs e)
{
    string connectionString =
    "Server=ACERASPIRE;Database=SGBDIR;Integrated Security=true";

    try
    {
        using (SqlConnection connection = new
        SqlConnection(connectionString))
        {
            //Deschide conexiunea

            connection.Open();

            MessageBox.Show("Starea conexiunii: " +
            connection.State.ToString());
        }
    }
}
```

Exemplu de aplicație C# Windows Forms - DataRelation

```
//Crearea DataSet-ului
```

```
DataSet dataset = new DataSet();
```

```
//Crearea celor două SqlDataAdapter's pentru tabelele părinte și copil
```

```
SqlDataAdapter parentAdapter = new SqlDataAdapter("SELECT * FROM Categoriile;", connection);
```

```
SqlDataAdapter childAdapter = new SqlDataAdapter("SELECT * FROM Produse;", connection);
```

```
//Crearea și popularea DataTable-ului părinte și a DataTable-ului copil
```

```
parentAdapter.Fill(dataset, "Categoriile");
```

```
childAdapter.Fill(dataset, "Produse");
```


Exemplu de aplicație C# Windows Forms - DataRelation

```
//Crearea celor două BindingSources pentru DataTable-ul părinte și  
pentru DataTable-ul copil
```

```
BindingSource parentBS = new BindingSource();
```

```
BindingSource childBS = new BindingSource();
```

```
//Afișarea tuturor înregistrărilor din DataTable-ul părinte în  
dataGridViewParent
```

```
parentBS.DataSource = dataset.Tables["Categorii"];
```

```
dataGridViewParent.DataSource = parentBS;
```

Exemplu de aplicație C# Windows Forms - DataRelation

//Crearea și adăugarea în DataSet a DataRelation-ului dintre DataTable-ul părinte și DataTable-ul copil

```
DataColumn parentPK =  
dataset.Tables["Categorii"].Columns["cod_categorie"];
```

```
DataColumn childFK =  
dataset.Tables["Produse"].Columns["cod_categorie"];
```

```
DataRelation relation = new DataRelation("fk_parent_child",  
parentPK, childFK);
```

```
dataset.Relations.Add(relation);
```


Exemplu de aplicație C# Windows Forms - DataRelation

//Afișarea în dataGridViewChild a înregistrărilor copil care aparțin înregistrării părinte selectate

```
childBS.DataSource = parentBS;
```

```
childBS.DataMember = "fk_parent_child";
```

```
dataGridViewChild.DataSource = childBS;
```

```
}
```

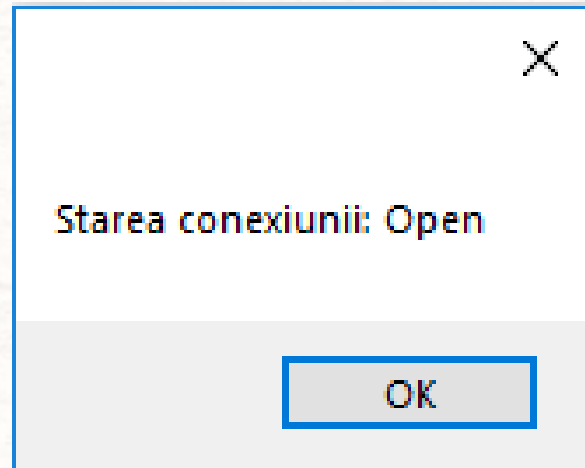
```
}
```

Exemplu de aplicație C# Windows Forms - DataRelation

```
catch (Exception err)
{
    MessageBox.Show(err.Message.ToString());
}
}
}
}
```


Exemplu de aplicație C# Windows Forms - DataRelation

- După ce pornim aplicația, apare un **MessageBox** care afișează starea conexiunii:



Exemplu de aplicație C# Windows Forms - DataRelation

- După ce apăsăm butonul **OK**, apare *Form*-ul pe care se pot vedea cele două controale *DataGridView* în care sunt afișate date:

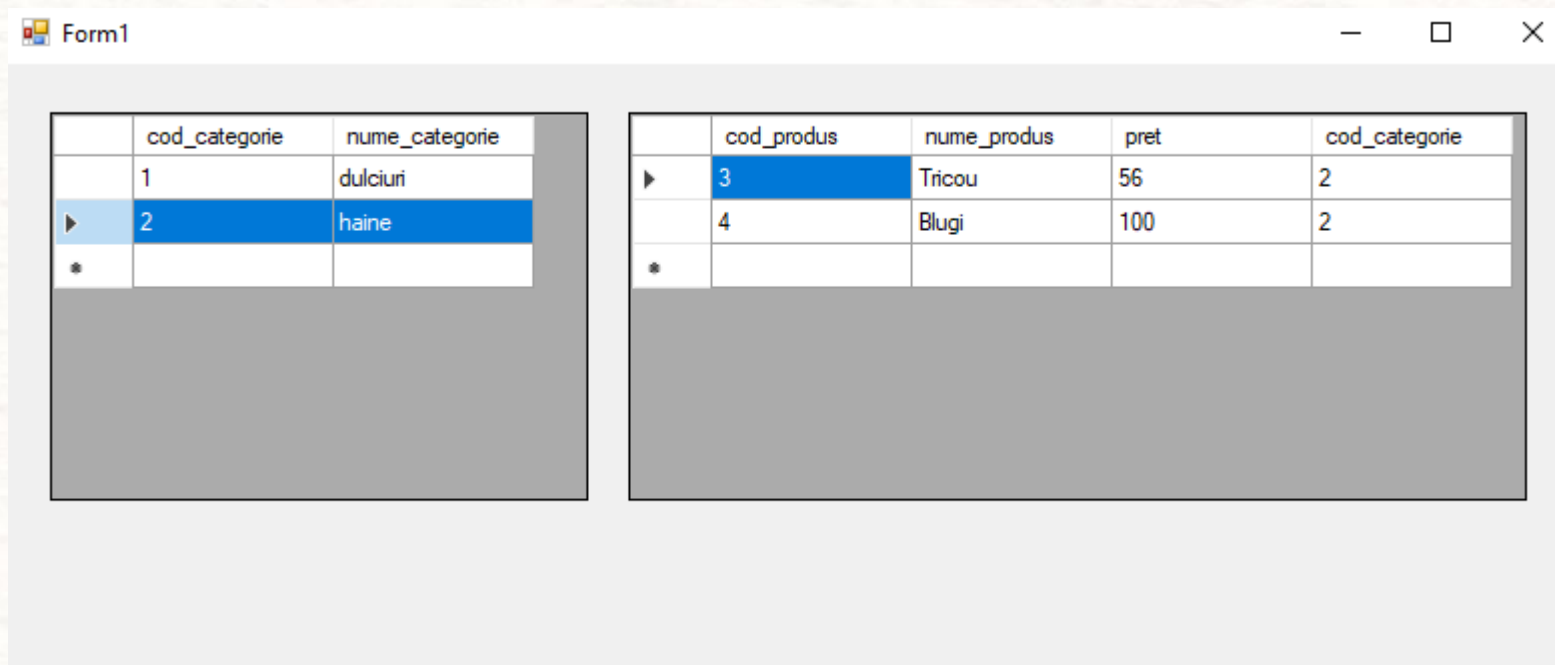
The screenshot shows a Windows Forms application window titled "Form1". Inside the window, there are two *DataGridView* controls side-by-side. The left *DataGridView* has two columns: "cod_categorie" and "nume_categorie". It contains three rows: a header row, a data row with "1" and "dulciuri", and a data row with "2" and "haine". The first data row is selected. The right *DataGridView* has four columns: "cod_produș", "nume_produș", "pret", and "cod_categorie". It contains three rows: a header row, a data row with "1", "Milka", "3", and "1", and a data row with "2", "Oreo", "2.5", and "1". The first data row is selected. Both *DataGridView* controls have a grey background and a white border.

	cod_categorie	nume_categorie
▶	1	dulciuri
	2	haine
*		

	cod_produș	nume_produș	pret	cod_categorie
▶	1	Milka	3	1
	2	Oreo	2.5	1
*				

Exemplu de aplicație C# Windows Forms - DataRelation

- Dacă selectăm a doua înregistrare părinte, înregistrările copil asociate vor fi afișate:



The screenshot shows a Windows Forms application window titled "Form1". It contains two data tables side-by-side. The left table has columns "cod_categorie" and "nume_categorie". The right table has columns "cod_produs", "nume_produs", "pret", and "cod_categorie".

	cod_categorie	nume_categorie
	1	dulciuri
▶	2	haine
*		

	cod_produs	nume_produs	pret	cod_categorie
▶	3	Tricou	56	2
	4	Blugi	100	2
*				

Tranzacții. Controlul concurenței în SQL Server

Seminar 3

Tranzacții în SQL Server

- SQL Server utilizează tranzacții pentru compunerea mai multor operații într-o singură unitate de lucru
- Acțiunile fiecărui utilizator sunt procesate utilizând o tranzacție diferită
- Pentru a maximiza *throughput*-ul, tranzacțiile ar trebui să se execute în paralel
- Proprietățile ACID ale tranzacțiilor au ca efect final serializabilitatea
- Dacă o tranzacție este efectuată cu succes, toate modificările realizate în timpul tranzacției sunt comise și devin permanente în baza de date
- Dacă apar erori în timpul unei tranzacții și aceasta trebuie anulată (canceled/rolled back), atunci toate modificările realizate în timpul tranzacției sunt șterse

Tranzacții în SQL Server

Mecanisme pentru invocarea tranzacțiilor:

- Fiecare comandă este o tranzacție dacă nu se specifică altceva (tranzacții autocommit)
- BEGIN TRAN, ROLLBACK TRAN, COMMIT TRAN sunt cel mai des utilizate (tranzacții explicite)
 - BEGIN TRAN marchează punctul de început al unei tranzacții explicite
 - ROLLBACK TRAN întoarce o tranzacție implicită sau explicită până la începutul tranzacției sau până la un *savepoint* din interiorul tranzacției
 - COMMIT TRAN marchează finalul unei tranzacții implicite sau explicite efectuate cu succes
- SET IMPLICIT_TRANSACTIONS ON (tranzacții implicite)
 - Se marchează doar finalul tranzacției (tranzacții înlanțuite)
 - În acest caz, o nouă tranzacție este începută în mod implicit atunci când tranzacția anterioară este finalizată (fiecare tranzacție este finalizată explicit cu COMMIT sau ROLLBACK)

Tranzacții în SQL Server

- SET XACT_ABORT ON
 - Orice erori SQL vor duce la roll back-ul tranzacției
- @@TRANCOUNT returnează numărul de instrucțiuni BEGIN TRANSACTION care au avut loc pe conexiunea curentă
 - Instrucțiunile BEGIN TRANSACTION incrementează @@TRANCOUNT cu 1
 - Instrucțiunile COMMIT TRANSACTION decrementează @@TRANCOUNT CU 1
 - ROLLBACK TRANSACTION setează @@TRANCOUNT pe 0, excepție ROLLBACK TRANSACTION *savepoint_name*, care nu afectează @@TRANCOUNT
- SAVE TRANSACTION *savepoint_name*
 - Setează un *savepoint* într-o tranzacție

Tipuri de tranzacții

- SQL Server poate utiliza tranzacții locale sau distribuite
- Imbricarea tranzacțiilor este permisă (dar nu tranzacțiile imbricate)
- Savepoints cu nume - rollback-ul unei porțiuni dintr-o tranzacție
- Un *savepoint* definește o locație la care o tranzacție se poate întoarce dacă o parte a tranzacției este anulată condiționat

Probleme de concurență

Izolarea tranzacțiilor în SQL Server rezolvă patru probleme majore de concurență:

- Lost updates
 - Când doi scriitori modifică aceleași date
- Dirty reads
 - Când un cititor citește date necomise
- Unrepeatable reads
 - Când o înregistrare existentă se schimbă în cadrul unei tranzacții
- Phantom reads
 - Când sunt adăugate noi înregistrări și apar în cadrul unei tranzacții

Probleme de concurență

- SQL Server asigură izolarea prin blocări
- Blocările pentru scriere sunt blocări exclusive
- Blocările pentru citire permit existența altor cititori
- O tranzacție well-formed obține tipul corect de blocare înainte de a utiliza datele
- O tranzacție two-phased menține blocările până când se obțin toate
- Nivelurile de izolare determină durata blocărilor (cât timp sunt menținute acestea)

Blocarea în SQL Server

- Blocările sunt gestionate de obicei dinamic de **Lock Manager**, nu prin intermediul aplicațiilor
- Blocările sunt menținute pe resurse SQL Server, cum ar fi înregistrări citite sau modificate în timpul unei tranzații, pentru a preveni utilizarea concurentă a resurselor de către diferite tranzații
- Granularitatea blocărilor:
 - Row/Key
 - Page
 - Extent (grup contiguu de 8 pagini)
 - Table
 - Database

Blocarea în SQL Server

- Blocările pot fi atribuite la mai mult de un nivel → ierarhie de blocări înrudite
- Lock escalation > 5000 blocări pe obiect
 - Este procesul de conversie a mai multor blocări cu granulație mai fină (fine-grain locks) în mai puține blocări cu granulație mai grosieră (coarse-grain locks), pentru a reduce supraîncărcarea sistemului
 - Lock escalation se declanșează atunci când nu este dezactivat la nivel de tabel (folosind opțiunea ALTER TABLE SET LOCK_ESCALATION) și când una din următoarele condiții există:
 - O singură instrucțiune Transact-SQL obține cel puțin 5000 de blocări pe un singur index sau tabel nepartiționat
 - O singură instrucțiune Transact-SQL obține cel puțin 5000 de blocări pe o singură partiție a unui tabel partiționat și opțiunea ALTER TABLE SET LOCK_ESCALATION este setată pe AUTO
 - Numărul de blocări pe o instanță Database Engine depășește capacitatea memoriei sau limitele configurației

Blocarea în SQL Server

- Durata blocărilor (precizată prin niveluri de izolare):
 - Până la finalizarea operației
 - Până la finalul tranzacției
- Tipuri de blocări:
 - Shared (S) – pentru citirea datelor
 - Update (U) – blocare S -> se anticipează X
 - Exclusive (X) – blocare pentru modificarea datelor ce este incompatibilă cu alte tipuri de blocări (excepție hint-ul NOLOCK și nivelul de izolare READ UNCOMMITTED)
 - Intent (IX, IS, SIX) – intenție de blocare (îmbunătățire a performanței)
 - Schema (Sch-M, Sch-S) – modificare schemă, stabilitate schemă (Sch-M și Sch-S nu sunt compatibile)
 - Bulk Update (BU) – blochează tot tabelul în timpul unui bulk insert
 - Key range – blochează mai multe înregistrări pe baza unei condiții

Blocarea în SQL Server

	Shared Lock (S)	Update Lock (U)	Exclusive Lock (X)
Shared Lock (S)	Da	Da	Nu
Update Lock (U)	Da	Nu	Nu
Exclusive Lock (X)	Nu	Nu	Nu

Blocări speciale

Blocări schema

- Nu blochează DML, doar DDL
- Folosite atunci când se execută o operație dependentă de schema unui tabel
- Tipuri
 - **Schema modification**
 - Se folosește în timpul execuției unei operații DDL
 - Previne accesul concurent la tabel
 - Folosită de unele operații DML (cum ar fi table truncation) pentru a preveni accesul operațiilor concurente la tabelele afectate

Blocări speciale

– Schema stability

- Folosită în timpul compilării și execuției interogărilor
- Nu blochează alte blocări tranzacționale, inclusiv blocări exclusive (X)
- Blochează operațiile concurente DDL și DML care obțin blocări Schema modification

Blocări Bulk Update

- Blocare explicită (TABLOCK)
- Automat în timpul SELECT INTO
- Folosită atunci când are loc o copiere de tipul bulk a datelor într-un tabel și este specificat hint-ul TABLOCK

Blocări Application

- Aplicațiile pot utiliza lock manager-ul intern al SQL Server (blocare resurse aplicație)

Niveluri de izolare în SQL Server

- READ UNCOMMITTED: fără blocare la citire
- READ COMMITTED: menține blocările pe durata execuției instrucțiunii (default) (elimină dirty reads)
- REPEATABLE READ: menține blocările pe durata tranzacției (elimină unrepeatable reads)
- SERIALIZABLE: menține blocările și blocările key range pe durata întregii tranzacții (elimină phantom reads)
- SNAPSHOT: lucrează pe un snapshot al datelor
- Sintaxă SQL:

```
SET TRANSACTION ISOLATION LEVEL <isolation_level>
```

Grade de izolare

Denumire	Haos	Read Uncommitted	Read Committed	Repeatable Read	Serializable
Lost Updates	Da	Nu	Nu	Nu	Nu
Dirty Reads	Da	Da	Nu	Nu	Nu
Unrepeatable Reads	Da	Da	Da	Nu	Nu
Phantom Reads	Da	Da	Da	Da	Nu

Blocare Key Range

- Blochează seturi de înregistrări controlate de un predicat
- Exemplu:

```
SELECT nume, cantitate FROM Produse  
  
WHERE cantitate BETWEEN 2500 AND 5000;
```

- Este necesară blocarea datelor care nu există
- Dacă utilizarea “cantitate BETWEEN 2500 AND 5000” nu returnează nicio înregistrare prima dată, nu ar trebui să returneze înregistrări nici la scanările ulterioare (nu se pot insera înregistrări noi dacă au valoarea specificată pentru cantitate cuprinsă între 2500 și 5000)

Blocări Transaction Workspace

Indică o conexiune

- Fiecare conexiune la o bază de date obține blocare *Shared_Transaction_Workspace*
- Excepții
 - conexiunile la master și tempdb

Utilizate pentru prevenirea:

- DROP
- RESTORE

Deadlocks

- Un deadlock are loc atunci când două sau mai multe tranzacții se blochează reciproc permanent din cauza faptului că fiecare tranzacție deține o blocare pe o resursă pentru care cealaltă tranzacție încearcă să obțină o blocare
- Exemplu:
 - Tranzacția T1 are o blocare pe o resursă R1 și a cerut o blocare pe resursa R2
 - Tranzacția T2 are o blocare pe o resursă R2 și a cerut o blocare pe resursa R1
 - Apare o stare de deadlock deoarece nicio tranzacție nu poate continua decât în momentul în care o resursă este disponibilă și nicio resursă nu poate fi eliberată până când o tranzacție nu își continuă execuția
- SQL Server recunoaște un deadlock
- Implicit, tranzacția mai nouă este terminată
 - Eroarea 1205 – ar trebui detectată și gestionată corespunzător

Deadlocks

- SET LOCK TIMEOUT
 - se menționează cât timp (în milisecunde) o tranzacție așteaptă ca un obiect blocat să fie eliberat (0 = terminare imediată)
- Un utilizator poate specifica prioritatea sesiunilor într-o situație de deadlock folosind instrucțiunea SET DEADLOCK PRIORITY (în mod implicit este setată pe NORMAL)
- DEADLOCK PRIORITY poate fi setată pe:
 - Low
 - Normal
 - High
 - O valoare numerică cuprinsă între -10 și 10
- Dacă două sesiuni au diferite valori setate pentru DEADLOCK PRIORITY , sesiunea cu valoarea cea mai scăzută este aleasă ca victimă a deadlock-ului

Reducerea situațiilor de deadlock

- Tranzacții scurte și într-un singur batch
- Colectarea și verificarea datelor input de la utilizatori înainte de deschiderea unei tranzacții
- Accesarea resurselor în aceeași ordine în tranzacții
- Utilizarea unui nivel de izolare mai scăzut sau a unui nivel de izolare row versioning
- Accesarea celei mai mici cantități posibile de date în tranzacții

Tranzacții - Exemplu

- În SQL Server, vom crea o bază de date numită "SGBDIR"
- După crearea bazei de date, vom crea trei tabele noi:
- Primul tabel se va numi *Presents*:

```
CREATE TABLE Presents  
(  
    present_id INT PRIMARY KEY IDENTITY,  
    description VARCHAR(100),  
    owner VARCHAR(100),  
    price REAL  
);
```


Tranzacții - Exemplu

- Al doilea tabel se va numi *Movies*:

```
CREATE TABLE Movies  
(  
    movie_id INT PRIMARY KEY IDENTITY,  
    title VARCHAR(100),  
    runtime INT  
);
```

Tranzacții - Exemplu

- Al treilea tabel se va numi *Actors*:

```
CREATE TABLE Actors  
(  
    actor_id INT PRIMARY KEY IDENTITY,  
    firstname VARCHAR(100),  
    lastname VARCHAR(100)  
);
```


Tranzacții - Exemplu

- După aceea, vom insera niște înregistrări noi în tabele:

```
INSERT INTO Presents (description, owner, price) VALUES  
('pony', 'Jack', 10000), ('candle', 'Jane', 2.5),  
('chocolate', 'James', 3);
```

Rezultat:

	present_id	description	owner	price
1	1	pony	Jack	10000
2	2	candle	Jane	2.5
3	3	chocolate	James	3

Tranzacții - Exemplu

```
INSERT INTO Movies (title, runtime) VALUES ('IT', 135),  
('Black Panther', 134), ('Red Sparrow', 140);
```

Rezultat:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140

Tranzacții - Exemplu

```
INSERT INTO Actors (firstname, lastname) VALUES  
('Bill', 'Skarsgard'), ('Chadwick', 'Boseman'),  
('Jennifer', 'Lawrence');
```

Rezultat:

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence

Tranzacții - Exemplu

- Exemplu de tranzacție explicită:

```
BEGIN TRAN;
```

```
UPDATE Presents SET description='horse' WHERE  
owner='Jack';
```

```
DELETE FROM Presents WHERE price=3;
```

```
COMMIT TRAN;
```

Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	2.5

Tranzacții - Exemplu

- Exemplu de tranzacție implicită:

```
SET IMPLICIT_TRANSACTIONS ON;
```

```
INSERT INTO Presents (description, owner, price) VALUES  
( 'car', 'Sam', 40000);
```

```
UPDATE Presents SET price=4 WHERE owner='Jane';
```

```
COMMIT TRAN;
```

Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	4
3	4	car	Sam	40000

Tranzacții - Exemplu

- Exemplu de imbricare a tranzacțiilor: (SET IMPLICIT_TRANSACTIONS OFF)

```
BEGIN TRAN;
```

```
UPDATE Presents SET description='bike' WHERE owner='Jane';
```

```
BEGIN TRAN;
```

```
INSERT INTO Presents (description, owner, price) VALUES  
( 'necklace', 'Sharon', 50);
```

```
COMMIT TRAN;
```

```
UPDATE Presents SET price=100 WHERE owner='Jane';
```

```
ROLLBACK TRAN;
```


Tranzacții - Exemplu

- Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	4
3	4	car	Sam	40000

- După cum putem observa, ambele tranzacții au primit un rollback deoarece tranzacția exterioară a primit un rollback
- Dacă tranzacția cea mai exterioară primește un rollback, atunci toate tranzacțiile interioare primesc un rollback, indiferent dacă au fost sau nu comise individual

Tranzacții - Exemplu

- Exemplu de tranzacție autocommit:

```
SELECT * FROM Presents;
```

Rezultat:

	present_id	description	owner	price
1	1	horse	Jack	10000
2	2	candle	Jane	4
3	4	car	Sam	40000

- Când IMPLICIT_TRANSACTIONS este setat pe OFF, fiecare instrucțiune Transact-SQL este mărginită de către o instrucțiune BEGIN TRANSACTION invizibilă și de către o instrucțiune COMMIT TRANSACTION invizibilă

Tranzacții - Exemplu

- Exemplu de tranzacție cu savepoint:

```
BEGIN TRAN;
```

```
INSERT INTO Movies (title, runtime) VALUES ('Frozen', 109);
```

```
SAVE TRAN savepoint;
```

```
INSERT INTO Actors (firstname, lastname) VALUES  
('Kristen', 'Bell');
```

```
ROLLBACK TRAN savepoint;
```

```
COMMIT TRAN;
```

Tranzacții - Exemplu

Rezultat:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence

- Din cauza faptului că tranzacția face un rollback până la savepoint și este apoi comisă, modificarea făcută de către prima instrucțiune INSERT a fost salvată

Probleme de concurență - Exemplu

- Exemplu dirty reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
UPDATE Movies SET runtime=200 WHERE title='IT';
```

```
WAITFOR DELAY '00:00:07';
```

```
ROLLBACK TRAN;
```

Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;  
  
BEGIN TRAN;  
  
SELECT * FROM Movies;  
  
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență - Exemplu

- În a doua fereastră de interogare putem vedea următorul rezultat:

	movie_id	title	runtime
1	1	IT	200
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

- Deoarece nivelul de izolare a tranzacției a fost setat pe READ UNCOMMITTED în a doua fereastră de interogare, a doua tranzacție a citit date necomise (dirty reads)

Probleme de concurență - Exemplu

- După execuția celor două tranzacții, vom executa următoarea interogare:

```
SELECT * FROM Movies;
```

- Deoarece prima tranzacție face un rollback, modificarea făcută de către aceasta nu a fost salvată:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

- Deci cea de-a doua tranzacție a citit date care nu au fost persistate pe disc

Probleme de concurență - Exemplu

- Exemplu unrepeatable reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
SELECT * FROM Movies;
```

```
WAITFOR DELAY '00:00:06';
```

```
SELECT * FROM Movies;
```

```
COMMIT TRAN;
```

Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
WAITFOR DELAY '00:00:03';
```

```
UPDATE Movies SET runtime=200 WHERE title='IT';
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență - Exemplu

- În prima fereastră de interogare putem vedea următorul rezultat:

	movie_id	title	runtime
1	1	IT	135
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

	movie_id	title	runtime
1	1	IT	200
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

- După cum putem observa, aceeași interogare executată de două ori în aceeași tranzacție a returnat două valori diferite pentru aceeași înregistrare (unrepeatable reads)

Probleme de concurență - Exemplu

- Exemplu phantom reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
SELECT * FROM Actors WHERE actor_id BETWEEN 1 AND 100;
```

```
WAITFOR DELAY '00:00:06';
```

```
SELECT * FROM Actors WHERE actor_id BETWEEN 1 AND 100;
```

```
COMMIT TRAN;
```


Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

-- !!!Asigurați-vă că IDENTITY generează o valoare cuprinsă în intervalul specificat (actor_id trebuie să aibă o valoare cuprinsă între 1 și 100)

```
BEGIN TRAN;
```

```
WAITFOR DELAY '00:00:03';
```

```
INSERT INTO Actors (firstname, lastname) VALUES  
( 'Alexander', 'Skarsgard' );
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență - Exemplu

- În prima fereastră de interogare putem vedea următorul rezultat:

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jennifer	Lawrence
4	5	Alexander	Skarsgard

- În aceeași tranzacție, o interogare care specifică un interval de valori în clauza WHERE a fost executată de două ori și numărul de înregistrări incluse în cel de-al doilea result set este mai mare decât numărul de înregistrări incluse în primul result set (phantom reads)

Probleme de concurență - Exemplu

- Exemplu deadlock:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN TRAN;
```

```
UPDATE Movies SET runtime=135 WHERE title='IT';
```

```
WAITFOR DELAY '00:00:05';
```

```
UPDATE Actors SET firstname='Jen' WHERE  
lastname='Lawrence';
```

```
COMMIT TRAN;
```

Probleme de concurență - Exemplu

- În a doua fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN TRAN;
```

```
UPDATE Actors SET firstname='Jenny' WHERE  
lastname='Lawrence';
```

```
WAITFOR DELAY '00:00:05';
```

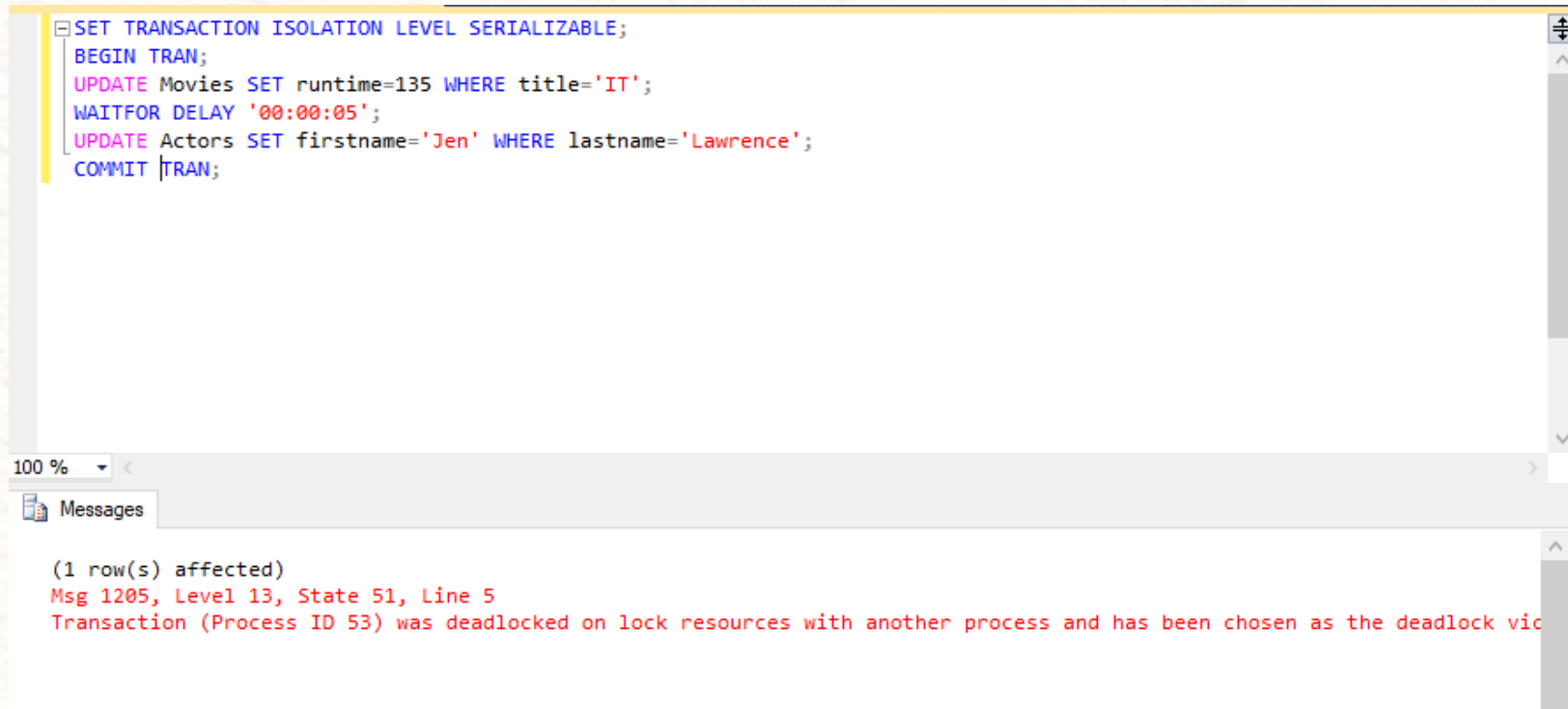
```
UPDATE Movies SET runtime=140 WHERE title='IT';
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență - Exemplu

- A avut loc un deadlock, iar prima tranzacție a fost aleasă drept victimă a deadlock-ului și face un rollback:



```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN;
UPDATE Movies SET runtime=135 WHERE title='IT';
WAITFOR DELAY '00:00:05';
UPDATE Actors SET firstname='Jen' WHERE lastname='Lawrence';
COMMIT TRAN;
```

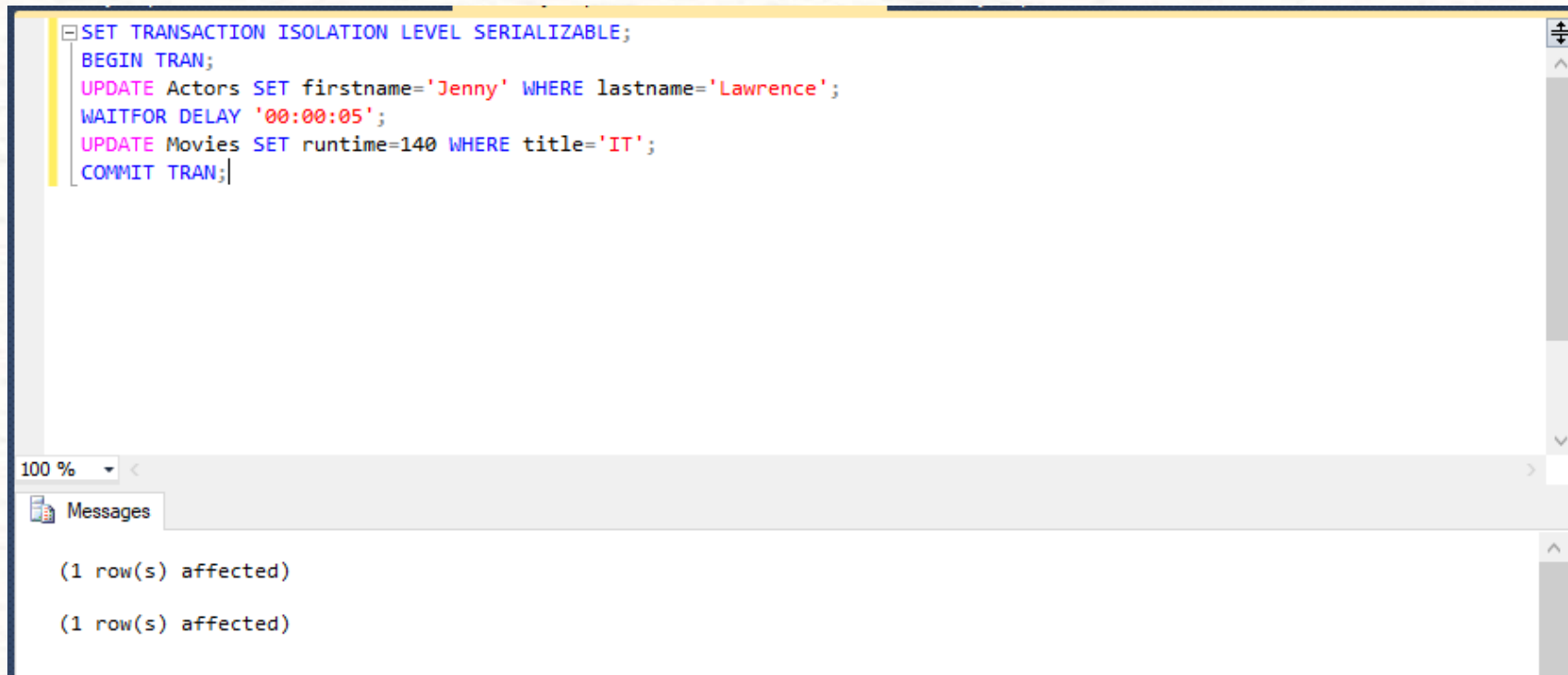
100 %

Messages

(1 row(s) affected)
Msg 1205, Level 13, State 51, Line 5
Transaction (Process ID 53) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. ROLLBACK TRANSACTION

Probleme de concurență - Exemplu

- Deoarece prima tranzacție a fost aleasă drept victimă a deadlock-ului, cea de-a doua tranzacție a fost executată cu succes:



```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN;
UPDATE Actors SET firstname='Jenny' WHERE lastname='Lawrence';
WAITFOR DELAY '00:00:05';
UPDATE Movies SET runtime=140 WHERE title='IT';
COMMIT TRAN;
```

100 %

Messages

(1 row(s) affected)

(1 row(s) affected)

Probleme de concurență - Exemplu

- Dacă executăm următoarele interogări, putem vedea rezultatul final:

```
SELECT * FROM Movies;
```

```
SELECT * FROM Actors;
```

	movie_id	title	runtime
1	1	IT	140
2	2	Black Panther	134
3	3	Red Sparrow	140
4	4	Frozen	109

	actor_id	firstname	lastname
1	1	Bill	Skarsgard
2	2	Chadwick	Boseman
3	3	Jenny	Lawrence
4	5	Alexander	Skarsgard

Multiversionarea

Seminar 4

Monitorizarea blocărilor

SQL Server Extended Events

- Este un sistem de monitorizare a performanței care utilizează resurse minime
- Oferă două interfețe grafice pentru utilizatori care pot fi folosite pentru a crea, afișa, modifica și analiza datele sesiunii: *New Session* și *New Session Wizard*

Procedura stocată sistem *sp_lock* (această funcționalitate se află în maintenance mode)

- Oferă informații despre blocări (locks)

Dynamic management view-ul sistem *sys.dm_tran_locks*

- Returnează informații despre resursele *lock manager* active în prezent
- Fiecare înregistrare reprezintă o cerere activă în prezent către *lock manager* pentru o blocare ce a fost acordată sau așteaptă să fie acordată

Monitorizarea blocărilor

Dynamic management view-ul sistem *sys.dm_tran_active_transactions*

- Returnează informații despre tranzacții pentru instanța SQL Server

Tipuri de resurse:

- RID – identificator de înregistrare
- Key – interval de chei într-un index (blocări key range)
- Pagină – pagină de 8 KB din tabele/indecși
- HoBT – Heap or balanced tree
- Tabel, fișier, bază de date
- Metadata
- Aplicație

Query Governor

SET QUERY_GOVERNOR_COST_LIMIT

- Suprascrie valoarea actuală configurată pentru **query governor cost limit** (pentru conexiunea curentă)
- *Query cost* se referă la timpul estimat (în secunde) necesar execuției unei interogări pe o anumită configurație hardware
- *Query optimizer*-ul estimează numărul de secunde necesare pentru execuția unei interogări și în cazul în care valoarea estimată este mai mare decât limita stabilită, interogarea nu va fi executată
- Valoarea implicită este 0 (această valoare permite execuția tuturor interogărilor)

DBCC LOG

- Returnează informații despre logul de tranzații
- Sintaxa:

DBCC LOG (<dbname>,<output id>)

<output id> este o valoare cuprinsă între 0 și 4 (specifică nivelul de detaliere)

<dbname> este numele bazei de date

- Exemplu:

DBCC LOG (SGBDIR,2)

Niveluri de izolare în SQL Server

- **READ UNCOMMITTED**: fără blocări la citire
- **READ COMMITTED**: este nivelul de izolare implicit și menține blocările pe durata execuției instrucțiunii (elimină dirty reads)
- **REPEATABLE READ**: menține blocările pe durata tranzacției (elimină unrepeatable reads)
- **SERIALIZABLE**: menține blocările și blocările key range pe durata întregii tranzacții (elimină phantom reads)
- **SNAPSHOT**: lucrează pe un snapshot al datelor
- Sintaxa SQL:

```
SET TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED |  
READ COMMITTED | REPEATABLE READ | SNAPSHOT |  
SERIALIZABLE }
```

Multiversionarea

- Într-un sistem de gestiune a bazelor de date cu multiversionare, fiecare scriere a unui item x produce o nouă versiune (copie) a lui x
- La fiecare citire a lui x , sistemul de gestiune a bazelor de date selectează una din versiunile lui x pentru citire
- Deoarece nu există suprascrieri între operațiile de scriere, iar operațiile de citire pot citi orice versiune, sistemul de gestiune a bazelor de date are mai multă flexibilitate în controlul ordinii scrierilor și citirilor

Versionarea la nivel de înregistrare (RLV)

- A fost introdusă în SQL Server 2005
- Este utilă când este nevoie de date comise, dar nu neapărat de cea mai nouă versiune
- **Read Committed Snapshot** Isolation și **Full Snapshot** Isolation
 - Cititorul nu blochează niciodată
 - Cititorul primește valoarea comisă anterior
- Toate versiunile mai vechi sunt stocate în baza de date *tempdb*
- Pe baza versiunilor mai vechi se poate construi un **snapshot** al datelor

Read Committed Snapshot Isolation

Toate operațiile văd înregistrările comise la începerea execuției lor =>

- Snapshot al datelor la nivel de comandă
- Citire consistentă la nivel de comandă
- Disponibil la utilizarea nivelului de izolare READ COMMITTED (default) cu opțiunea **READ_COMMITTED_SNAPSHOT** setată pe ON
- Setarea opțiunii READ_COMMITTED_SNAPSHOT pe ON:

```
ALTER DATABASE database_name
```

```
SET READ_COMMITTED_SNAPSHOT ON;
```


Full Snapshot Isolation

Toate operațiile văd înregistrările comise la începerea execuției tranzacției =>

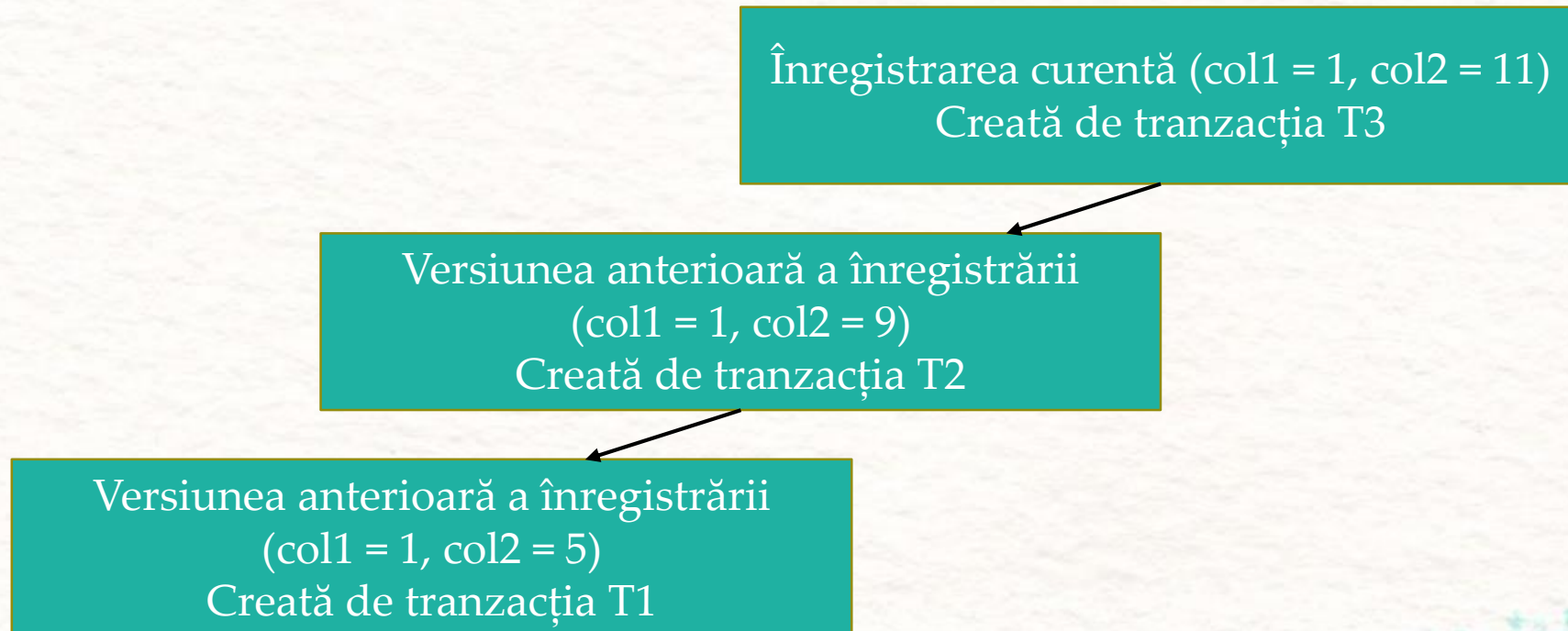
- Snapshot al datelor la nivel de tranzacție
- Citire consistentă la nivel de tranzacție
- Nivelul de izolare SNAPSHOT poate fi folosit dacă opțiunea **ALLOW_SNAPSHOT_ISOLATION** este setată pe ON
- Setarea opțiunii ALLOW_SNAPSHOT_ISOLATION pe ON:

```
ALTER DATABASE database_name
```

```
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

Versionarea la nivel de înregistrare

- O înregistrare conține TSN (transaction sequence number)
- Toate versiunile sunt stocate într-o listă înlănțuită



Versionarea la nivel de înregistrare

Avantaje

- Nivelul de concurență este mai ridicat
- Mărește performanța triggerelor și a creării indecșilor

Dezavantaje

- Cerințe de gestiune suplimentare pentru monitorizarea utilizării *tempdb*
- Performanța mai scăzută a operațiilor de modificare
- Viteza cititorilor este afectată de costul traversării listelor înlănțuite
- Se rezolvă conflictul dintre cititori și scriitori, dar tot nu sunt permisi scriitorii simultani

Triggere și versionarea la nivel de înregistrare

Triggerele au acces la două pseudo-tabele:

- Tabelul *deleted* care conține înregistrări șterse sau versiuni vechi ale înregistrărilor modificate
- Tabelul *inserted* care conține înregistrări inserate sau versiuni noi ale înregistrărilor modificate

Înainte de SQL Server 2005:

- Tabelul *deleted* era creat pe baza logului de tranzacții (afectează performanța)

Prin utilizarea versionării la nivel de înregistrare:

- Pentru tabele cu triggere relevante sunt versionate schimbările

Crearea indecșilor și versionarea la nivel de înregistrare

În versiunile anterioare de SQL Server, crearea sau reconstruirea indecșilor însemna:

- Tabel blocat exclusiv și date complet inaccesibile (index clustered)
- Tabel disponibil doar pentru citire și index indisponibil (index nonclustered)

Cu versionare la nivel de înregistrare:

- Indecșii sunt creați și reconstruiți online
- Toate cererile sunt procesate pe date versionate

Niveluri de izolare și anomalii de concurență

Nivel de izolare	Dirty Reads	Unrepeatable Reads	Phantom Reads	Update conflict	Model de concurență
Read Uncommitted	Da	Da	Da	Nu	Pesimist
Read Committed Locking	Nu	Da	Da	Nu	Pesimist
Read Committed Snapshot	Nu	Da	Da	Nu	Optimist
Repeatable Read	Nu	Nu	Da	Nu	Pesimist
Versionare la nivel de înregistrare	Nu	Nu	Nu	Da	Optimist
Serializable	Nu	Nu	Nu	Nu	Pesimist

Instrucțiunea MERGE

- Instrucțiunea MERGE oferă posibilitatea de a compara înregistrări dintr-un tabel sursă cu înregistrări dintr-un tabel destinație
- Comenzile INSERT, UPDATE și DELETE pot fi executate pe baza rezultatului acestei comparații
- Tabelul *Filme*:

Cod_film	Titlu	An	Durata
1	IT	2017	NULL
2	IT	NULL	NULL
3	IT	NULL	135

Instrucțiunea MERGE – Sintaxa generală

MERGE Definiție_Tabel AS Destinație

USING (Tabel_sursă) AS Sursă

ON (Termeni de căutare)

WHEN MATCHED THEN UPDATE SET sau DELETE

WHEN NOT MATCHED [BY TARGET] THEN INSERT

WHEN NOT MATCHED BY SOURCE THEN UPDATE SET sau DELETE

Instrucțiunea MERGE – Exemplu

MERGE Filme

```
USING (SELECT MAX(Cod_film) Cod_film, Titlu, MAX(An) An,  
MAX(Durata) Durata FROM Filme GROUP BY Titlu) MergeFilme  
ON Filme.Cod_film=MergeFilme.Cod_film  
WHEN MATCHED THEN UPDATE SET Filme.Titlu=MergeFilme.Titlu,  
Filme.An=MergeFilme.An, Filme.Durata=MergeFilme.Durata  
WHEN NOT MATCHED BY SOURCE THEN DELETE;
```

Instrucțiunea MERGE – Exemplu

- Rezultat:

Cod_film	Titlu	An	Durata
3	IT	2017	135

PIVOT/UNPIVOT

- Schimbă o expresie *table-valued* într-un alt tabel
- PIVOT rotește o expresie *table-valued* transformând valorile unice dintr-o coloană din expresie în mai multe coloane în output și calculează valori agregate acolo unde este necesar, pe valorile oricăror coloane rămase care sunt dorite în rezultatul final
- UNPIVOT realizează operația opusă, rotind coloanele dintr-o expresie *table-valued* în valori de coloană

Sintaxa PIVOT

```
SELECT <non-pivoted column>, [first pivoted column] AS <column  
name>, [second pivoted column] AS <column name>, ... [last pivoted  
column] AS <column name>
```

```
FROM (<SELECT query that produces the data>) AS <source query>
```

```
PIVOT
```

```
(<aggregation function>(<column being aggregated>) FOR
```

```
[<column that contains values that become column headers>]
```

```
IN ( [first pivoted column], [second pivoted column], ... [last  
pivoted column]))
```

```
AS <alias for the pivot table> <optional ORDER BY clause>;
```


PIVOT - Exemplan

- Tabelul *Students*:

student_id	name	city
1	Jack	New York
2	Jane	Los Angeles
3	Rose	New York
4	Jill	New York
5	Anne	Los Angeles
6	John	London

PIVOT - Exemplu

- Pentru a afișa numărul de studenți pentru fiecare oraș, vom executa următoarea interogare:

```
SELECT city, COUNT(student_id) AS [number of students]  
FROM Students GROUP BY city;
```

- Obținem următorul rezultat:

city	number of students
London	1
Los Angeles	2
New York	3

PIVOT - Exemplu

- Vom transforma valorile unice din coloana *city* în coloane ale tabelului output:

```
SELECT 'number of students' AS 'city', [New York],  
[Los Angeles], [London] FROM (SELECT city, student_id FROM  
Students) AS SourceTable PIVOT (COUNT(student_id) FOR city  
IN ([Los Angeles], [London], [New York])) AS PivotTable;
```

- Rezultat:

city	New York	Los Angeles	London
number of students	3	2	1

Probleme de concurență – Exemplu (RLV)

- În SQL Server, vom crea o nouă bază de date numită 'SGBDPC'
- După ce baza de date a fost creată, vom seta opțiunile READ_COMMITTED_SNAPSHOT și ALLOW_SNAPSHOT_ISOLATION pe ON (pentru a activa row versioning):

```
ALTER DATABASE SGBDPC
```

```
SET READ_COMMITTED_SNAPSHOT ON;
```

```
ALTER DATABASE SGBDPC
```

```
SET ALLOW_SNAPSHOT_ISOLATION ON;
```

- După aceea, vom crea un nou tabel numit *Movies*:

Probleme de concurență – Exemplu (RLV)

```
CREATE TABLE Movies  
(  
    movie_id INT PRIMARY KEY IDENTITY,  
    title VARCHAR(100),  
    year INT  
);
```

- Vom insera două înregistrări:

```
INSERT INTO Movies (title, year) VALUES  
('IT', 2017), ('Red Sparrow', 2018);
```

- Vom crea un tabel numit *Actors*:

Probleme de concurență – Exemplu (RLV)

```
CREATE TABLE Actors  
(  
    actor_id INT PRIMARY KEY IDENTITY,  
    firstname VARCHAR(100),  
    lastname VARCHAR(100)  
);
```

- Vom insera o înregistrare:

```
INSERT INTO Actors (firstname, lastname)  
VALUES ('Bill', 'Skarsgard');
```


Probleme de concurență – Exemplu (RLV)

- Dacă executăm următoarele interogări, putem vedea următoarele result set-uri:

```
SELECT * FROM Movies;
```

	movie_id	title	year
1	1	IT	2017
2	2	Red Sparrow	2018

```
SELECT * FROM Actors;
```

	actor_id	firstname	lastname
1	1	Bill	Skarsgard

- Exemplu unrepeatable reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

Probleme de concurență – Exemplu (RLV)

```
BEGIN TRAN;  
  
SELECT * FROM Movies;  
  
WAITFOR DELAY '00:00:06';  
  
SELECT * FROM Movies;  
  
COMMIT TRAN;
```

- În a doua fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;  
  
WAITFOR DELAY '00:00:03';  
  
UPDATE Movies SET year=2019 WHERE title='IT';  
  
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență – Exemplu (RLV)

- În prima fereastră de interogare putem vedea următorul rezultat:

	movie_id	title	year
1	1	IT	2017
2	2	Red Sparrow	2018

	movie_id	title	year
1	1	IT	2019
2	2	Red Sparrow	2018

- După cum putem observa, aceeași interogare executată de două ori în aceeași tranzacție a returnat două valori diferite pentru aceeași înregistrare (unrepeatable reads)

Probleme de concurență – Exemplu (RLV)

- Exemplu phantom reads:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
SELECT * FROM Movies WHERE year BETWEEN 2017 AND 2020;
```

```
WAITFOR DELAY '00:00:06';
```

```
SELECT * FROM Movies WHERE year BETWEEN 2017 AND 2020;
```

```
COMMIT TRAN;
```


Probleme de concurență – Exemplu (RLV)

- În a doua fereastră de interogare punem următoarea tranzacție:

```
BEGIN TRAN;
```

```
WAITFOR DELAY '00:00:03';
```

```
INSERT INTO Movies (title, year) VALUES  
( 'Black Panther', 2018);
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență – Exemplu (RLV)

- În prima fereastră de interogare putem vedea următorul rezultat:

	movie_id	title	year
1	1	IT	2019
2	2	Red Sparrow	2018
	movie_id	title	year
1	1	IT	2019
2	2	Red Sparrow	2018
3	3	Black Panther	2018

- În aceeași tranzacție, o interogare care specifică un interval de valori în clauza WHERE a fost executată de două ori și numărul de înregistrări incluse în cel de-al doilea result set este mai mare decât numărul de înregistrări incluse în primul result set (phantom reads)

Probleme de concurență – Exemplu (RLV)

- Exemplu deadlock:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

```
BEGIN TRAN;
```

```
UPDATE Movies SET year=2017 WHERE title='IT';
```

```
WAITFOR DELAY '00:00:05';
```

```
UPDATE Actors SET firstname='Alexander' WHERE  
lastname='Skarsgard';
```

```
COMMIT TRAN;
```

Probleme de concurență – Exemplu (RLV)

- În a doua fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

```
BEGIN TRAN;
```

```
UPDATE Actors SET firstname='Alex' WHERE  
lastname='Skarsgard';
```

```
WAITFOR DELAY '00:00:05';
```

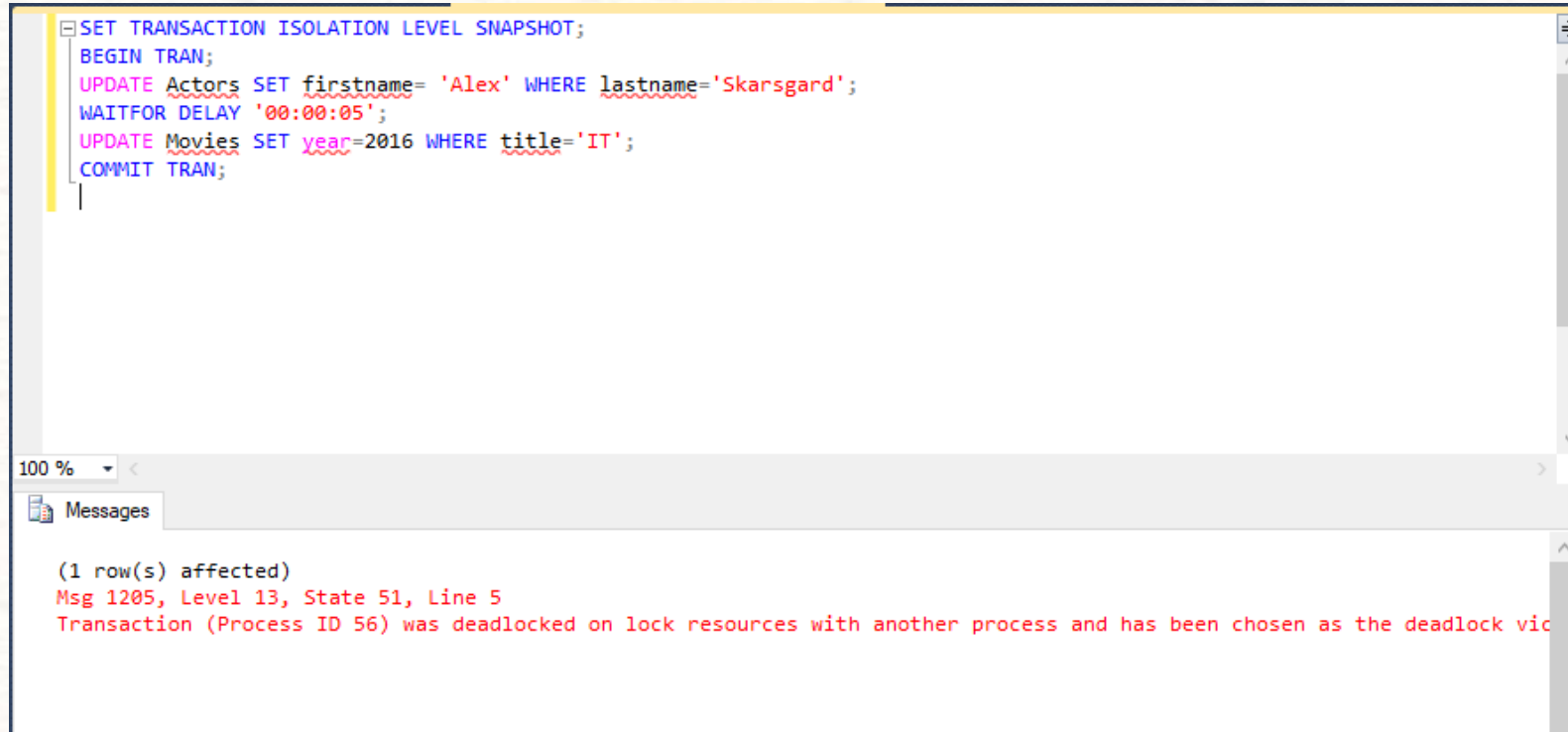
```
UPDATE Movies SET year=2016 WHERE title='IT';
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență – Exemplu (RLV)

- A avut loc un deadlock, iar a doua tranzacție a fost aleasă drept victimă a deadlock-ului și face un rollback:



```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;  
BEGIN TRAN;  
UPDATE Actors SET firstname= 'Alex' WHERE lastname='Skarsgard';  
WAITFOR DELAY '00:00:05';  
UPDATE Movies SET year=2016 WHERE title='IT';  
COMMIT TRAN;
```

100 %

Messages

(1 row(s) affected)
Msg 1205, Level 13, State 51, Line 5
Transaction (Process ID 56) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. ROLLBACK TRANSACTION

Probleme de concurență – Exemplu (RLV)

- Deoarece a doua tranzacție a fost aleasă drept victimă a deadlock-ului, prima tranzacție a fost executată cu succes:



```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;  
BEGIN TRAN;  
UPDATE Movies SET year=2017 WHERE title='IT';  
WAITFOR DELAY '00:00:05';  
UPDATE Actors SET firstname= 'Alexander' WHERE lastname='Skarsgard';  
COMMIT TRAN;
```

100 %

Messages

(1 row(s) affected)

(1 row(s) affected)

Probleme de concurență – Exemplu (RLV)

- Dacă executăm următoarele interogări, putem vedea rezultatul final:

```
SELECT * FROM Movies;
```

```
SELECT * FROM Actors;
```

	movie_id	title	year
1	1	IT	2017
2	2	Red Sparrow	2018
3	3	Black Panther	2018

	actor_id	firstname	lastname
1	1	Alexander	Skarsgard

Probleme de concurență – Exemplu (RLV)

- Exemplu update conflict:
- Deschidem două ferestre noi de interogare (două conexiuni)
- În prima fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

```
BEGIN TRAN;
```

```
WAITFOR DELAY '00:00:03';
```

```
UPDATE Movies SET title='The Snowman' WHERE year=2017;
```

```
COMMIT TRAN;
```


Probleme de concurență – Exemplu (RLV)

- În a doua fereastră de interogare punem următoarea tranzacție:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
```

```
BEGIN TRAN;
```

```
UPDATE Movies SET title='Thor Ragnarok' WHERE  
year=2017;
```

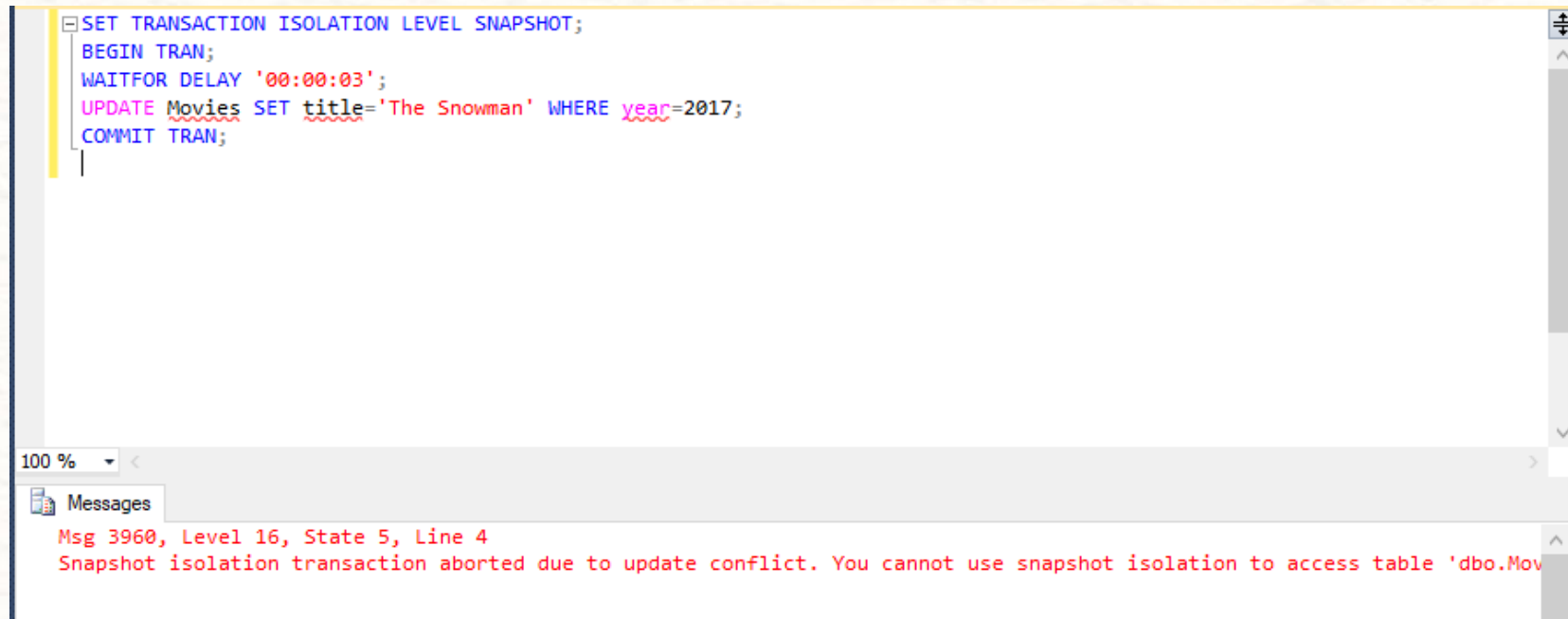
```
WAITFOR DELAY '00:00:03';
```

```
COMMIT TRAN;
```

- Începem execuția primei și a celei de a doua tranzacții (în această ordine)

Probleme de concurență – Exemplu (RLV)

- A avut loc un update conflict și prima tranzacție a eșuat:



```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;  
BEGIN TRAN;  
WAITFOR DELAY '00:00:03';  
UPDATE Movies SET title='The Snowman' WHERE year=2017;  
COMMIT TRAN;
```

100 %

Messages

Msg 3960, Level 16, State 5, Line 4
Snapshot isolation transaction aborted due to update conflict. You cannot use snapshot isolation to access table 'dbo.Mov

Probleme de concurență – Exemplu (RLV)

- Deoarece prima tranzacție a eșuat, a doua tranzacție a fost efectuată cu succes:



The screenshot displays a SQL Server Enterprise Manager window with a query editor and a Messages pane. The query editor contains the following T-SQL code:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;  
BEGIN TRAN;  
UPDATE Movies SET title='Thor Ragnarok' WHERE year=2017;  
WAITFOR DELAY '00:00:03';  
COMMIT TRAN;
```

The Messages pane at the bottom shows the result of the execution:

(1 row(s) affected)

Probleme de concurență – Exemplu (RLV)

- Dacă executăm următoarea interogare, putem vedea rezultatul final:

```
SELECT * FROM Movies;
```

	movie_id	title	year
1	1	Thor Ragnarok	2017
2	2	Red Sparrow	2018
3	3	Black Panther	2018

Optimizarea performanței în MS SQL Server

Seminar 5

Optimizarea interogărilor - metodologie

- Identificarea așteptărilor (bottleneck) la nivel de server
 - I/O latches
 - Log update
 - Blocare
 - Altele
- Corelare așteptări – cozi (queues)
- Restrângere la nivel de bază de date/fișier
- Optimizarea interogărilor problematice

Latch

- **Latch**: un tip special de blocare sistem low-level care este menținută pe întreaga durată a unei operații fizice asupra unei pagini din memorie, ce are scopul de a proteja consistența memoriei
- **Latch-urile** sunt un mecanism intern al SQL Server care are scopul de a proteja resursele de memorie partajate (cum ar fi paginile sau structurile de date din memorie aflate în buffer pool) și de a coordona accesul la aceste resurse
- Deoarece latch-urile sunt un mecanism intern al SQL Server, care nu este expus în afara SQLOS (SQL Server Operating System), ele nu pot fi administrate de către utilizatori, spre deosebire de blocările tranzacționale (locks), care pot fi administrate cu ajutorul hint-urilor NO LOCK
- **Latch-urile I/O** sunt obținute atunci când se citesc sau se scriu date pe disc
- **Latch-urile buffer** sunt obținute atunci când se accesează pagini din memorie

Identificarea așteptărilor

DMV (Dynamic Management Views) returnează informații despre starea server-ului care pot fi folosite pentru monitorizarea stării server-ului, diagnosticarea problemelor și reglarea performanței

Dynamic management view-ul sistem *sys.dm_os_wait_stats* returnează informații despre toate așteptările întâmpinate de thread-urile care s-au executat

- **wait_type** – numele tipului de așteptare
 - Așteptări resursă (blocări, latches, rețea, I/O)
 - Așteptări queue
 - Așteptări externe (au loc atunci când un SQL Server worker așteaptă terminarea unui eveniment extern, cum ar fi apelul unei proceduri stocate extended sau o interogare linked server)
- **waiting_tasks_count** – numărul de așteptări pentru tipul de așteptare

Identificarea așteptărilor

- **wait_time_ms** – timpul total de așteptare pentru tipul de așteptare în milisecunde (acest timp include signal_wait_time_ms)
- **max_wait_time_ms** – timpul maxim de așteptare pentru tipul de așteptare
- **signal_wait_time_ms** – diferența dintre momentul în care waiting thread a fost semnalat și momentul în care a început să ruleze

Resetarea counter-elor:

```
DBCC SQLPERF ('sys.dm_os_wait_stats', CLEAR);
```

Corelare așteptări - queues

Dynamic management view-ul sistem *sys.dm_os_performance_counters* returnează câte o înregistrare pentru fiecare **performance counter** menținut de server

- **object_name** – categoria de care aparține counter-ul
- **counter_name** – numele counter-ului relativ la categorie (se poate suprapune pentru diverse valori object_name)
- **instance_name** – numele instanței counter-ului (de multe ori conține numele bazei de date)
- **cntr_value** – valoarea înregistrată sau calculată a counter-ului
- **cntr_type** – tipul counter-ului definit de Performance Monitor

Corelare aşteptări - queues

Sunt disponibile peste 500 de counter-e:

- Access Methods, User Settable, Buffer Manager, Broker Statistics, SQL Errors, Latches, Buffer Partition, SQL Statistics, Locks, Buffer Node, Plan Cache, Cursor Manager by Type, Memory Manager, General Statistics, Databases, Catalog Metadata, Broker Activation, Broker/DBM Transport, Transactions, Cursor Manager Total, Exec Statistics, Wait Statistics etc.
- `cntr_type=65792` → `cntr_value` conține valoarea efectivă
- `cntr_type=537003264` → `cntr_value` conține rezultate în timp real care trebuie împărțite la o "bază" pentru a obține valoarea efectivă (altfel, sunt inutile)
 - valoarea trebuie împărțită la o valoare "bază" pentru a obține un raport, iar rezultatul se poate înmulți cu 100 pentru a-l exprima în procente

Corelare aşteptări - queues

- `cntr_type=272696576` → `cntr_value` conţine valoarea de bază
 - Counter-ele sunt bazate pe timp
 - Counter-ele sunt cumulative
 - Se utilizează un tabel secundar pentru stocarea valorilor intermediare pentru statistici
- `cntr_type=1073874176` şi `cntr_type=1073939712`
- Se obţine atât valoarea (1073874176) cât şi valoarea de bază (1073939712)
- Se obţin ambele valori din nou (după 15 secunde)
- Pentru a obţine rezultatul vizat, se împart diferenţele între ele:

$$\text{UnitsPerSec} = (\text{cv2} - \text{cv1}) / (\text{bv2} - \text{bv1}) / 15.0$$

Restrângere la nivel de bază de date/fișier

Dynamic management view-ul sistem *sys.dm_io_virtual_file_stats* returnează statistici I/O pentru fișierele de date și loguri

- **Parametri:**
 - `database_id` (NULL=toate bazele de date), funcția `DB_ID` este utilă
 - `file_id` (NULL=toate fișierele), funcția `FILE_IDEX` este utilă
- **Tabel returnat:**
 - `database_id`
 - `file_id`
 - `sample_ms` – numărul de milisecunde de la pornirea calculatorului
 - `num_of_reads` – numărul de citiri fizice realizate
 - `num_of_bytes_read` – numărul total de octeți citați

Restrângere la nivel de bază de date/fișier

- **io_stall_read_ms** – timpul total de așteptare al utilizatorilor pentru citiri
 - **num_of_writes** – numărul de scrieri efectuate
 - **num_of_bytes_written** – numărul total de octeți scriși
 - **io_stall_write_ms** – timpul total de așteptare al utilizatorilor pentru finalizarea scrierilor
 - **io_stall** – suma **io_stall_read_ms** și **io_stall_write_ms**
 - **file_handle** – Windows file handle pentru fișier
 - **size_on_disk_bytes** – numărul total de octeți folosiți pe disc pentru fișier
- Exemplu:

```
SELECT * FROM sys.dm_io_virtual_file_stats(DB_ID('SGBDIR'),NULL);
```


Indecși

Sunt printre **principalii factori** care influențează **performanța interogărilor**

- **Efect asupra:** filtrării, join-ului, sortării, grupării, evitării blocării și a deadlock-ului, etc.
- **Efect în modificări:** efect **pozitiv** în localizarea înregistrărilor și efect **negativ** al costului modificărilor în index

Înțelegerea indecșilor și a mecanismelor interne ale acestora

- Clustered/nonclustered
- Indecși cu una sau mai multe coloane
- View-uri indexate și indecși pe coloane calculate
- Scenarii de acoperire
- Intersecție

Indecși

- În funcție de mediu și de raportul dintre interogările SELECT și modificările datelor, trebuie să apreciați în ce măsură costul adițional de mentenanță a indecșilor se justifică prin îmbunătățirea performanței interogărilor
- Indecșii cu mai multe coloane tind să fie mult mai utili decât indecșii cu o coloană
- E mai probabil ca query optimizer-ul să utilizeze indecși cu mai multe coloane pentru a acoperi o interogare
- View-urile indexate au un cost de întreținere mai ridicat decât indecșii standard
- Opțiunea WITH SCHEMABINDING este obligatorie pentru crearea view-urilor indexate

Unelte pentru analiza performanței interogărilor

- **Plan de execuție grafic**
- **STATISTICS IO:** număr de scanări, citiri logice, citiri fizice, citiri read ahead
- **STATISTICS TIME:** durată și timp CPU net
- **SHOWPLAN_TEXT:** plan estimat
- **SHOWPLAN_ALL:** plan estimat detaliat
- **STATISTICS PROFILE:** plan efectiv detaliat
- **SET STATISTICS XML:** informații detaliate despre performanța efectivă în format XML
- **SET SHOWPLAN_XML:** informații detaliate despre performanța estimată în format XML

Optimizarea interogărilor

Evaluarea planurilor de execuție

- Secvență de operații fizice/logice

Factori de optimizare:

- Predicatul de căutare utilizat
- Tabelele implicate în join
- Condițiile de join
- Dimensiunea rezultatului
- Lista de indecși

Scop: evitarea celor mai slabe planuri pentru interogări

SQL Server utilizează un query optimizer bazat pe cost

STATISTICS IO și STATISTICS TIME

```
USE AdventureWorks2014;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.BusinessEntityContact;
```

150 %

Results Messages

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 249 ms.

(909 rows affected)
Table 'BusinessEntityContact'. Scan count 1, logical reads 8, physical reads 1, read-ahead reads 8, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 424 ms.

- **DBCC DROPCLEANBUFFERS** – elimină toate clean buffers din buffer pool și obiectele columnstore din columnstore object pool
- **DBCC DROPCLEANBUFFERS** se poate folosi pentru a testa interogări utilizând un cold buffer cache fără a da shut down și restart server-ului
- **DBCC FREEPROCCACHE** – șterge toate elementele din plan cache

STATISTICS IO și STATISTICS TIME

```
USE AdventureWorks2014;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.BusinessEntityContact;
```

150 %

Results Messages

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 249 ms.

(909 rows affected)

Table 'BusinessEntityContact'. Scan count 1, logical reads 8, physical reads 1, read-ahead reads 8, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 424 ms.

- **CPU time** – resursele CPU utilizate pentru a executa o interogare
- **Elapsed time** – cât timp a durat execuția interogării

STATISTICS IO și STATISTICS TIME

```
USE AdventureWorks2014;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT * FROM Person.BusinessEntityContact;
```

150 %

Results Messages

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 249 ms.

(909 rows affected)
Table 'BusinessEntityContact'. Scan count 1, logical reads 8, physical reads 1, read-ahead reads 8, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 424 ms.

- **Physical reads** – numărul de pagini citite de pe disc
- **Read-ahead reads** – numărul de pagini plasate în cache pentru interogare
- **Scan count** – de câte ori au fost accesate tabelele
- **Logical reads** – numărul de pagini citite din data cache

STATISTICS IO și STATISTICS TIME

```
USE AdventureWorks2014;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
GO
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT ReorderPoint FROM Production.Product WHERE ReorderPoint>375;
```

Results Messages

SQL Server parse and compile time:
CPU time = 16 ms, elapsed time = 376 ms.

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.

(181 rows affected)

Table 'Product'. Scan count 1, logical reads 15, physical reads 1, read-ahead reads 0, pages loaded from cache or disk 1.

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 30 ms.

STATISTICS IO și STATISTICS TIME

--Se definește un index pentru a optimiza interogarea

```
USE [AdventureWorks2014];
```

```
GO
```

```
CREATE NONCLUSTERED INDEX [IX_Product_ReorderPoint_ASC]
```

```
ON [Production].[Product]
```

```
(ReorderPoint ASC);
```

```
GO
```

STATISTICS IO și STATISTICS TIME

```
USE AdventureWorks2014;  
GO  
DBCC DROPCLEANBUFFERS;  
DBCC FREEPROCCACHE;  
GO  
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;  
GO  
SELECT ReorderPoint FROM Production.Product WHERE ReorderPoint>375;
```

150 %

Results

Messages

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 105 ms.

SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 0 ms.

(181 rows affected)

Table 'Product'. Scan count 1, logical reads 2, physical reads 1, read-ahead reads 0

SQL Server Execution Times:
CPU time = 0 ms, elapsed time = 9 ms.

SHOWPLAN_ALL

- Dacă **SHOWPLAN_ALL** este setat pe ON, Microsoft SQL Server nu execută instrucțiunile Transact-SQL, ci returnează informații detaliate despre cum sunt executate instrucțiunile și oferă estimări ale cerințelor de resurse pentru instrucțiuni
- **SHOWPLAN_ALL** returnează informații sub forma unui set de înregistrări care formează un hierarchical tree ce reprezintă pașii efectuați de SQL Server query processor pe măsură ce execută fiecare instrucțiune
- Fiecare instrucțiune reflectată în output conține o singură înregistrare cu textul instrucțiunii, urmată de mai multe înregistrări cu detaliile pașilor de execuție
- Sintaxa:

```
SET SHOWPLAN_ALL { ON | OFF }
```

SHOWPLAN_ALL

- Exemplu:

```
SET SHOWPLAN_ALL ON;  
GO  
SELECT COUNT(*) cRows FROM HumanResources.Shift;  
GO  
SET SHOWPLAN_ALL OFF;  
GO
```

100 % <



Results

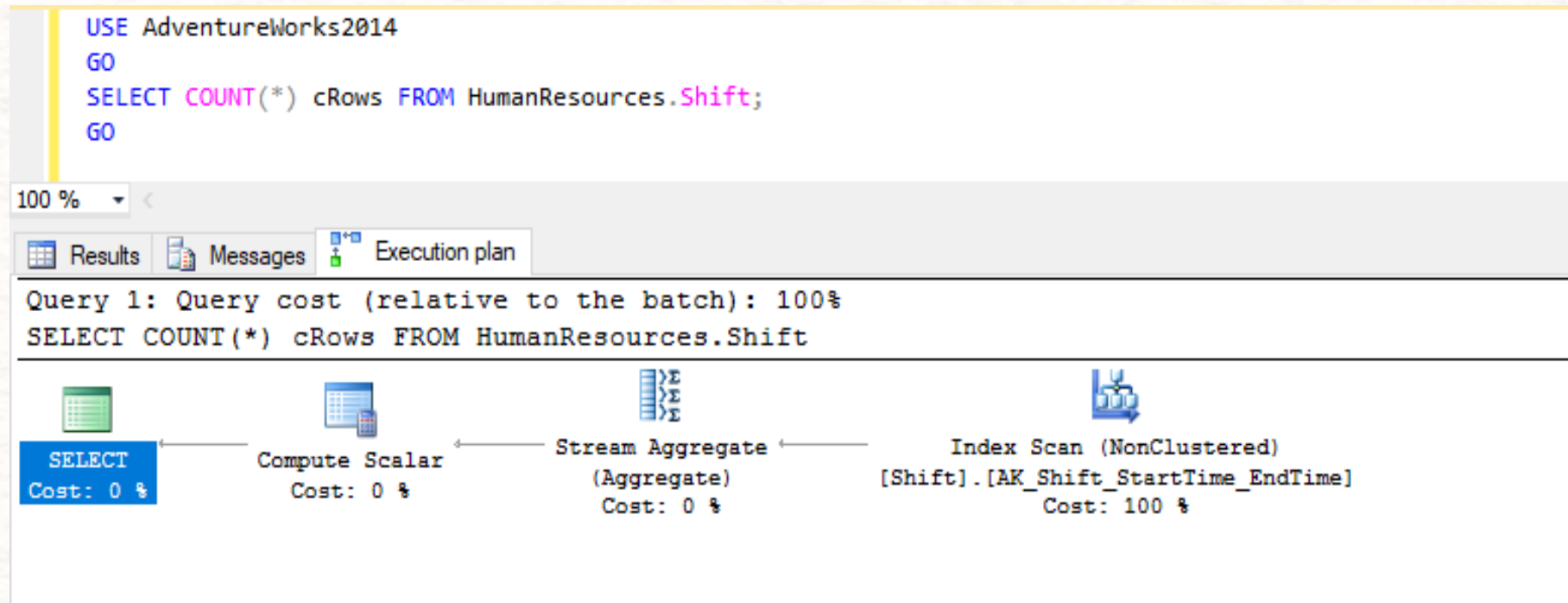


Messages

	Stmt Text
1	SELECT COUNT(*) cRows FROM HumanResources.Shift;
2	-Compute Scalar(DEFINE:([Expr1002]=CONVERT_IMPLICIT(int,[Expr1003].0)))
3	-Stream Aggregate(DEFINE:([Expr1003]=Count(*)))
4	-Index Scan(OBJECT:([AdventureWorks2012].[HumanResources].[Shift].[AK_Shift_StartTime_EndTime]))

Plan de execuție grafic

- Exemplu:

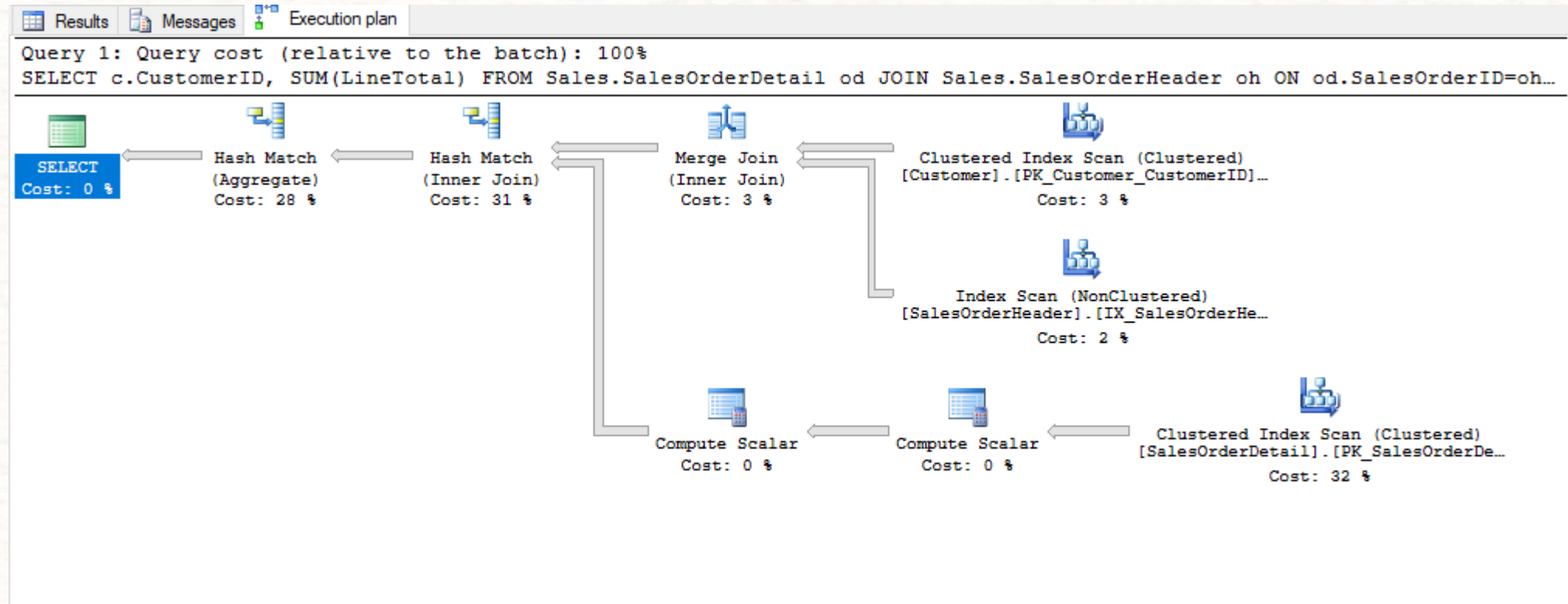


Plan de execuție grafic

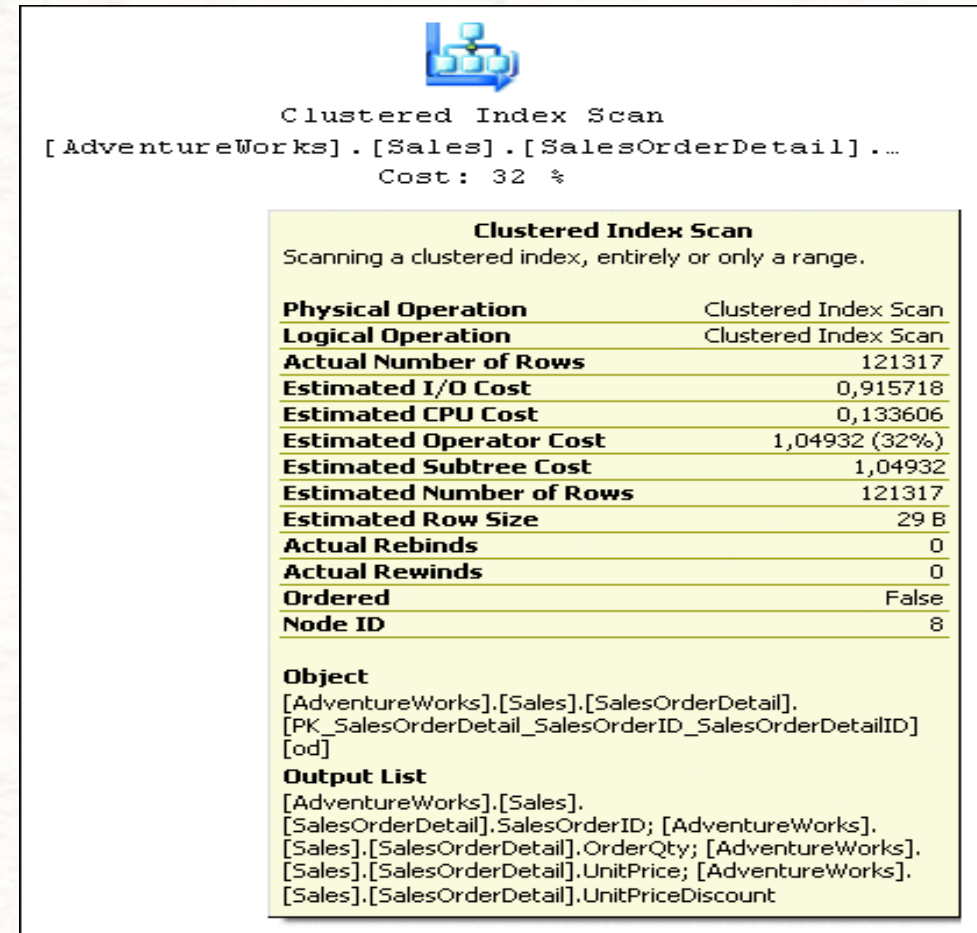
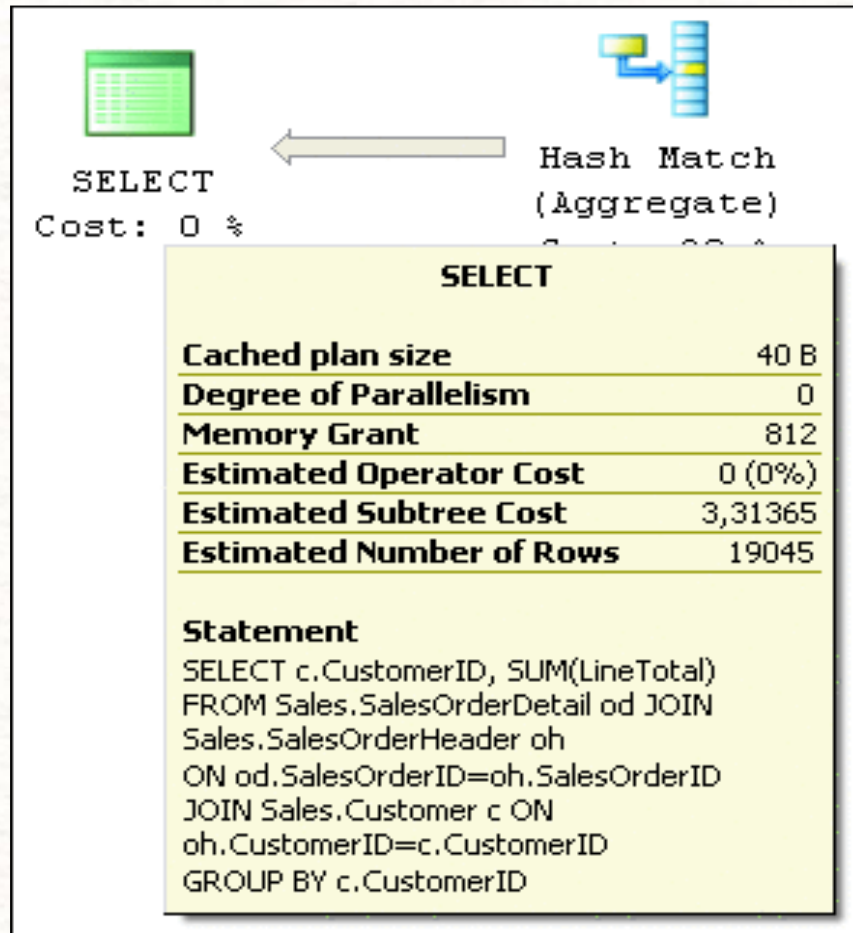
- Exemplu:

```
SELECT c.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od
JOIN Sales.SalesOrderHeader oh ON
od.SalesOrderID=oh.SalesOrderID
JOIN Sales.Customer c ON
oh.CustomerID=c.CustomerID
GROUP BY c.CustomerID;
```


Plan de execuție grafic



Plan de execuție grafic

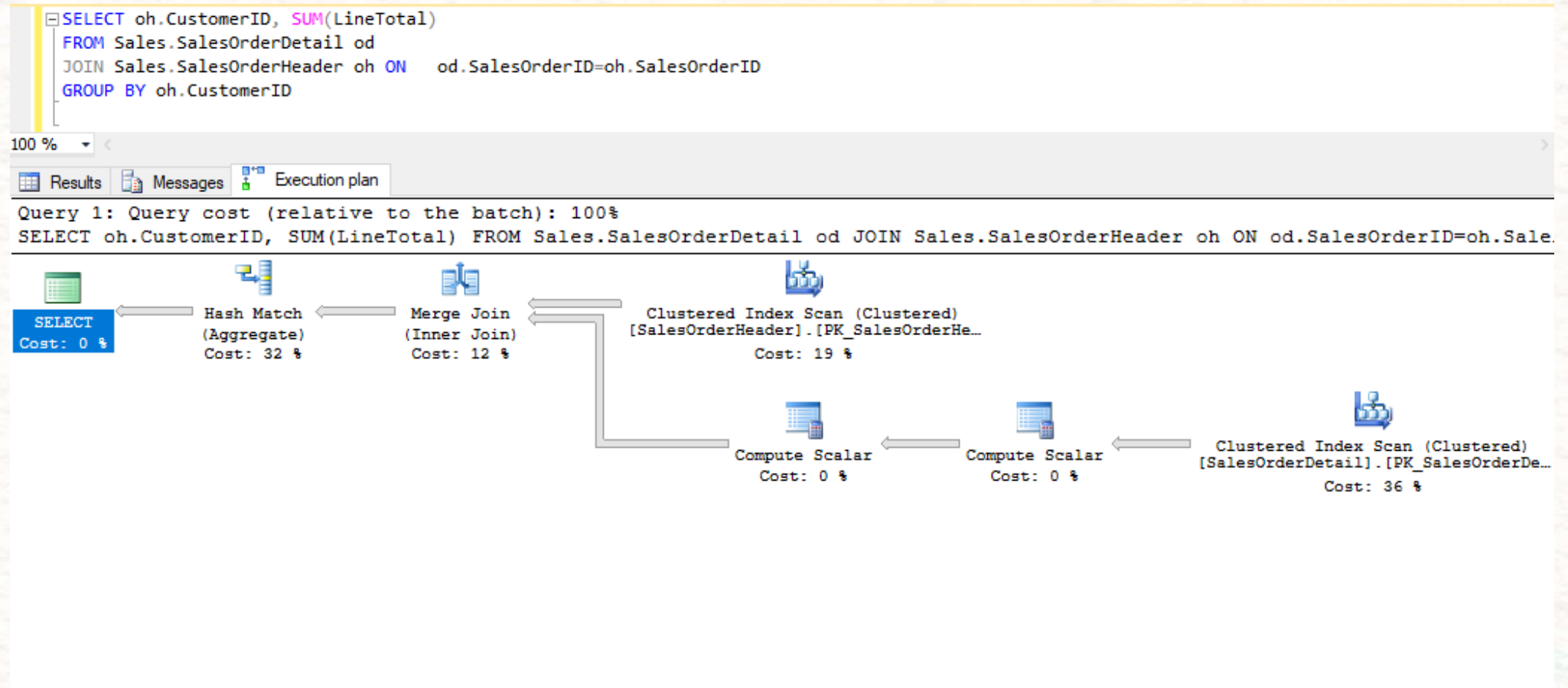


Plan de execuție grafic

- Exemplu:

```
SELECT oh.CustomerID, SUM(LineTotal)
FROM Sales.SalesOrderDetail od
JOIN Sales.SalesOrderHeader oh ON
od.SalesOrderID=oh.SalesOrderID
GROUP BY oh.CustomerID;
```

Plan de execuție grafic



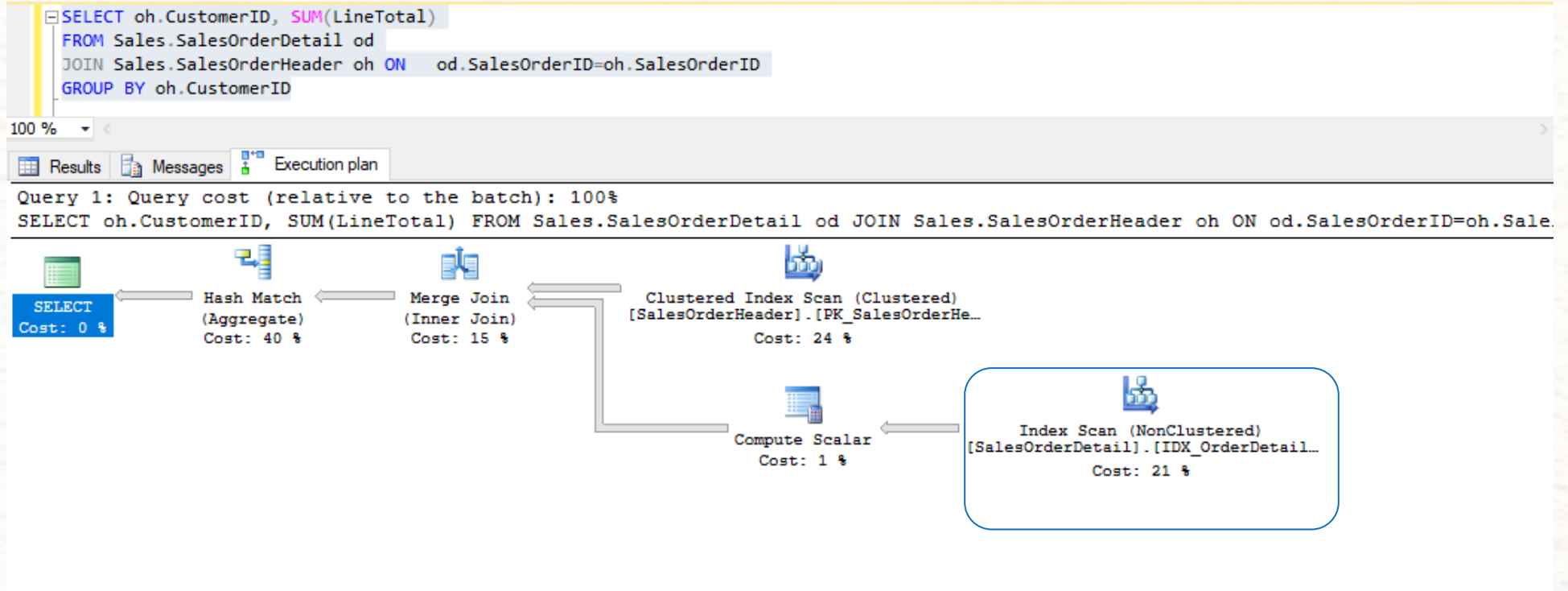
Plan de execuție grafic

```
CREATE INDEX IDX_OrderDetail_OrderID_TotalLine  
ON Sales.SalesOrderDetail(SalesOrderID)  
INCLUDE (LineTotal);
```

```
SELECT oh.CustomerID, SUM(LineTotal)  
FROM Sales.SalesOrderDetail od  
JOIN Sales.SalesOrderHeader oh ON  
od.SalesOrderID=oh.SalesOrderID  
GROUP BY oh.CustomerID;
```

Plan de execuție grafic

- Dacă interogarea este executată din nou după crearea indexului, putem vedea că indexul este folosit:



Mai multe despre optimizare

Seminar 6

Proceduri stocate

Avantaje

- Avantaje de performanță
- Pe server
- Reutilizarea planului de execuție

Notă: cerințe pentru reutilizarea unui plan

- Reutilizarea planurilor nu este benefică întotdeauna

Proceduri stocate – sugestii pentru optimizare

SET NOCOUNT ON

- Nu se afișează numărul de înregistrări afectate
- Reduce traficul pe rețea

Utilizați numele schemei cu numele obiectului

- Ajută la găsirea directă a planului compilat

```
SELECT * FROM dbo.MyTable;
```

```
EXEC dbo.StoredProcedure;
```

Proceduri stocate – sugestii pentru optimizare

Nu utilizați prefixul sp_

- SQL Server caută întâi în baza de date master și ulterior în baza de date curentă

Evitați comparațiile de valori din coloane de tipuri diferite

- În cazul coloanei convertite, conversia implicită se va aplica asupra valorii din coloană pentru toate înregistrările din tabel (SQL Server trebuie să convertească toate valorile pentru a putea efectua comparația)

Evitați join-urile între două tipuri de coloane

- Indecșii nu sunt utilizați pentru coloanele convertite

Utilizați UNION pentru a implementa o operație "OR"

Proceduri stocate – sugestii pentru optimizare

sp_executesql versus EXEC

- Planul de execuție al unei instrucțiuni dinamice poate fi reutilizat dacă TOATE caracterele pentru două execuții consecutive sunt identice

```
EXEC('SELECT * FROM Categories WHERE category_id=1;')
```

```
EXEC('SELECT * FROM Categories WHERE category_id=2;')
```

```
EXECUTE sp_executesql N'SELECT * FROM Categories WHERE  
category_id=@category_id;', N'@category_id INT',  
@category_id=1;
```

Proceduri stocate – sugestii pentru optimizare

Cursoare

În general **consumă mult din resursele SQL Server și reduc performanța și scalabilitatea aplicațiilor**

Sunt mai indicate când:

- Datele trebuie accesate înregistrare cu înregistrare / logică procedurală
- Se efectuează calcule ordonate

Proceduri stocate – sugestii pentru optimizare

Nu utilizați COUNT() într-o subinterogare pentru a verifica existența unor date

- Utilizați IF EXISTS(SELECT 1 FROM table_name;)

Output-ul instrucțiunii SELECT imbricate nu este folosit

- Reduce timpul de procesare și transferul pe rețea

Mențineți tranzacțiile scurte

- Lungimea tranzacțiilor influențează blocările și interblocările

Proceduri stocate – sugestii pentru optimizare

- Reutilizarea planului de execuție

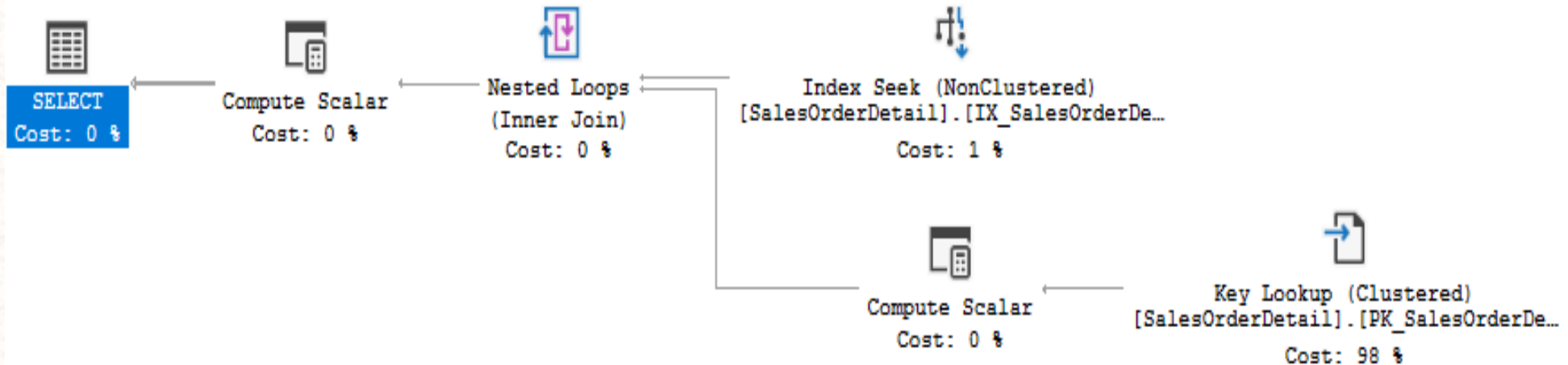
```
CREATE PROCEDURE usp_test (@pid INT)
AS
BEGIN
    SELECT * FROM Sales.SalesOrderDetail WHERE ProductID=@pid;
END;
```


Proceduri stocate – sugestii pentru optimizare

EXEC usp_test 897;

Query 1: Query cost (relative to the batch): 100%

SELECT * FROM Sales.SalesOrderDetail WHERE ProductID= @pid



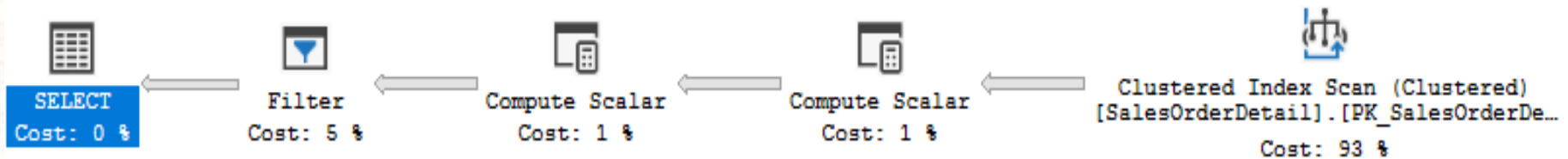
Proceduri stocate – sugestii pentru optimizare

EXEC usp_test 870;

Query 1: Query cost (relative to the batch): 100%

SELECT * FROM Sales.SalesOrderDetail WHERE ProductID= @pid

Missing Index (Impact 99.2024): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>] ON [Sales...



Proceduri stocate – sugestii pentru optimizare

- Hint-urile în interogări specifică faptul că acestea ar trebui folosite pretutindeni în interogare și afectează toți operatorii din instrucțiune
- Hint-urile sunt specificate în **clauza OPTION**
- Dacă optimizatorul (query optimizer-ul) generează un plan care nu este valid din cauza unor query hints, apare eroarea 8622
- Query hints sunt recomandate spre a fi folosite doar ca ultimă soluție de către programatori cu experiență și administratori de baze de date (SQL Server query optimizer selectează de obicei cel mai bun plan de execuție pentru o interogare)

Proceduri stocate – sugestii pentru optimizare

- **OPTIMIZE FOR** query hint determină folosirea unei anumite valori pentru o variabilă locală la compilarea și optimizarea unei interogări
- Exemplu:

```
ALTER PROCEDURE usp_test (@pid INT)
```

```
AS
```

```
BEGIN
```

```
SELECT * FROM Sales.SalesOrderDetail WHERE ProductID=@pid
```

```
OPTION (OPTIMIZE FOR (@pid=870));
```

```
END;
```


Proceduri stocate – sugestii pentru optimizare

- **RECOMPILE** query hint determină eliminarea planului generat pentru o interogare după execuția acesteia, forțând optimizatorul să recompileze un plan de execuție data viitoare când aceeași interogare este executată
- Dacă nu se specifică **RECOMPILE**, Database Engine salvează în cache planurile de execuție și le refolosește
- La compilarea planurilor de execuție, **RECOMPILE** query hint folosește valorile curente ale variabilelor locale din interogare și, dacă interogarea se află în interiorul unei proceduri stocate, valorile curente ale parametrilor

Proceduri stocate – sugestii pentru optimizare

- **RECOMPILE query hint**
- Exemplu:

```
ALTER PROCEDURE usp_test (@pid INT)
AS
BEGIN
    SELECT * FROM Sales.SalesOrderDetail
    WHERE ProductID=@pid OPTION (RECOMPILE);
END;
```


Proceduri stocate – sugestii pentru optimizare

- **OPTIMIZE FOR UNKNOWN** determină folosirea de date statistice în locul valorilor inițiale pentru toate variabilele locale atunci când interogarea este compilată și optimizată, inclusiv parametri creați cu parametrizare forțată
- Variabilele locale nu sunt cunoscute la optimizare
- Exemplul de mai jos generează întotdeauna același plan de execuție

```
ALTER PROCEDURE usp_test (@pid INT)

AS

BEGIN

SELECT * FROM Sales.SalesOrderDetail WHERE ProductID=@pid

OPTION (OPTIMIZE FOR UNKNOWN);

END;
```

Proceduri stocate – sugestii pentru optimizare

- Exemplul de mai jos generează întotdeauna același plan de execuție

```
ALTER PROCEDURE usp_test (@pid INT)
AS
BEGIN
    DECLARE @lpid INT;
    SET @lpid=@pid;
    SELECT * FROM Sales.SalesOrderDetail WHERE ProductID=@lpid;
END;
```


Proceduri stocate – sugestii pentru optimizare

- Exemplu:

```
DECLARE @city_name VARCHAR(30);  
DECLARE @postal_code VARCHAR(15);  
SELECT * FROM Person.Address  
WHERE City=@city_name AND PostalCode=@postal_code  
OPTION (OPTIMIZE FOR (@city_name='Seattle',  
@postal_code UNKNOWN));
```

Proceduri stocate – sugestii pentru optimizare

Alte hint-uri pentru interogări (query hints)

- HASH GROUP vs ORDER GROUP
- Exemplu:

```
SELECT ProductID, OrderQty, SUM(LineTotal) AS Total
FROM Sales.SalesOrderDetail
WHERE UnitPrice < $5.00
GROUP BY ProductID, OrderQty
ORDER BY ProductID, OrderQty
OPTION (HASH GROUP, FAST 10);
```


Proceduri stocate – sugestii pentru optimizare

Alte hint-uri pentru interogări

- MERGE UNION vs HASH UNION vs CONCAT UNION
- Exemplu:

```
SELECT * FROM HumanResources.Employee AS E1  
  
UNION  
  
SELECT * FROM HumanResources.Employee AS E2  
  
OPTION (MERGE UNION);
```

Proceduri stocate – sugestii pentru optimizare

Hint-uri pentru join

- LOOP JOIN vs MERGE JOIN vs HASH JOIN
- Exemplu:

```
SELECT * FROM Sales.Customer AS C  
INNER JOIN Sales.vStoreWithAddresses AS SA  
ON C.CustomerID=SA.BusinessEntityID  
WHERE TerritoryID=5  
OPTION (MERGE JOIN);  
GO
```


Proceduri stocate – sugestii pentru optimizare

Hint-uri pentru join

- FAST n determină returnarea primelor n înregistrări cât de repede este posibil
- După ce sunt returnate primele n înregistrări, interogarea își continuă execuția și produce întregul result set
- Exemplu:

```
SELECT * FROM Sales.Customer AS C  
INNER JOIN Sales.vStoreWithAddresses AS SA  
ON C.CustomerID=SA.BusinessEntityID  
WHERE TerritoryID=5  
OPTION (FAST 10);
```

Proceduri stocate – sugestii pentru optimizare

Hint-uri pentru join

- FORCE ORDER –“forțează” optimizatorul să utilizeze ordinea join-urilor ca în interogare

```
SELECT * FROM Table1  
INNER JOIN Table2 ON Table1.a=Table2.b  
INNER JOIN Table3 ON Table2.c=Table3.d  
INNER JOIN Table4 ON Table3.e=Table4.f  
OPTION (FORCE ORDER);
```


Proceduri stocate – sugestii pentru optimizare

- Mai multe despre controlarea planurilor de execuție cu ajutorul hint-urilor:
- <https://www.simple-talk.com/sql/performance/controlling-execution-plans-with-hints/>

Execuția dinamică

Dezavantaje

- Cod urât care este dificil de întreținut
- Risc de securitate - SQL Injection

Utilizare inteligentă

- Sortare și filtre dinamice pentru a obține planuri bune
- Și altele...

Tabele temporare

Utile când:

- Aveți result set-uri intermediare care trebuie să fie accesate de mai multe ori
- Aveți nevoie de o zonă de stocare temporară pentru date în timp ce executați cod procedural

Utilizați tabelele temporare când:

- Lucrați cu volume mari de date, iar eficiența planurilor este importantă și netrivială

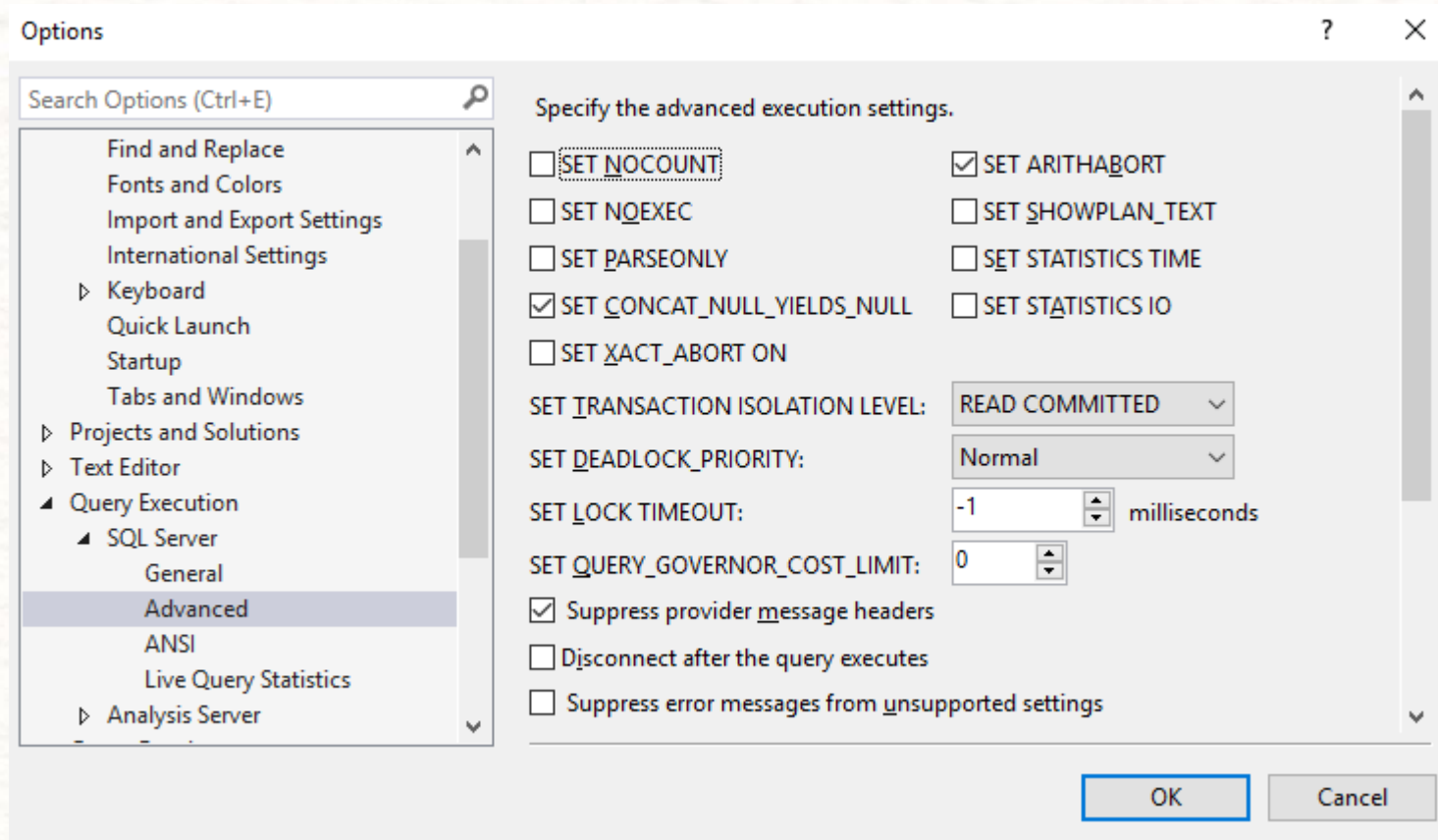
Utilizați variabile tabel când:

- Lucrați cu volume mici de date, iar eficiența planurilor nu este atât de importantă ca recompilările, sau atunci când planurile sunt triviale

Triggers

- Costisitoare (when rollback, undo as opposed to reject)
- Impactul mare asupra performanței implică accesarea tabelelor speciale *inserted* și *deleted*:
 - SQL Server 2000: transaction log
 - SQL Server 2005: row versioning (tempdb)
- Încercați să utilizați set-based activities
- Identificați numărul de înregistrări afectate și reacționați în consecință
- UPDATE triggers înregistrează delete urmat de insert în log

Opțiuni SQL Server



Fragmentare

Fragmentarea are un efect important asupra performanței interogărilor

- Fragmentare logică: procentul de pagini out-of-order
- Densitatea paginilor: popularea paginilor

Utilizați **DBCC SHOWCONTIG** pentru a obține statistici legate de fragmentare și examinați **LogicalFragmentation** și **Avg. Page Density (full)**

Utilizați funcția **sys.dm_db_index_physical_stats** și analizați **AvgFragmentation**

Reconstruiți indecșii pentru a gestiona fragmentarea

Alte statistici

Update statistics asynchronously

- **String summary statistics:** frecvența distribuției subșirurilor este menținută pentru coloanele care stochează șiruri de caractere
- **Asynchronous auto update statistics** (setat implicit pe off)
- Statistici pentru coloane calculate

Dynamic management view-ul sistem *sys.dm_exec_query_stats*

- Returnează statistici legate de performanță pentru planurile de execuție din cache
- Conține câte o înregistrare pentru fiecare query statement din planul aflat în cache, iar durata de existență a înregistrărilor este legată de cea a planului
- Când un plan este șters din cache, înregistrările care îi corespund sunt eliminate

Alte statistici

- **total_logical_reads / total_logical_writes** – numărul total de citiri / scrieri logice efectuate la execuțiile unui plan de când a fost compilat
- **total_physical_reads** – numărul total de citiri fizice efectuate la execuțiile unui plan de când a fost compilat
- **total_worker_time** – timpul CPU total utilizat, în microsecunde, pentru execuțiile unui plan de când a fost compilat
- **total_elapsed_time** – durata totală, în microsecunde, pentru execuțiile încheiate ale unui plan