



Universitatea Babeș–Bolyai
Facultatea de Matematică și Informatică

Optimizarea
interogărilor



Interogări distribuite

```
SELECT AVG(E.age)
FROM Employees E
WHERE E.salary > 3000
      AND E.salary < 7000
```

Fragmentare orizontală:

Înregistrările cu *salary* < 5000 la Shanghai și *salary* ≥ 5000 la Tokyo.

- Se calculează SUM(*age*), COUNT(*age*) pe ambele servere
- Dacă WHERE conține doar *E.salary* > 6000, interogarea se poate executa pe un singur server.

```
SELECT AVG(E.age)
FROM Employees E
WHERE E.salary > 3000
      AND E.salary < 7000
```

Interogări distribuite

Fragmentare verticală:

title și *salary* la Shanghai, *ename* și *age* la Tokyo, *id* fiind prezent pe ambele servere.

- Trebuie reconstruită tabela prin intermediul unui *join* pe *id*, iar apoi se evaluează interogarea.
- **Replicare:** Tabela *Employees* e copiată pe ambele servere.
 - Alegerea site-ului pe care se execută interogarea se face în funcție de costurile locale și costurile de transfer.

Join-uri distribuite

LONDRA

Employees

PARIS

Reports

Employees

500 pagini,

80 înregistrări pe pagină

Reports

1000 pagini

100 înregistrări pe pagină

Fetch as Needed

- *Page-oriented nested loops*, *Employees* ca tabelă externă (pentru fiecare pagină din *Employees* se aduc toate paginile din *Reports* de la Paris):
 - **Cost:** $500 D + 500 * 1000 (D+S)$, unde **D** este costul de citire/salvare a paginilor; **S** costul de transfer al paginilor.
 - Dacă interogarea nu s-a lansat de la Londra, atunci trebui adăugat costul de transfer al rezultatului către clientul care a transmis interogarea.
- Se poate utiliza și INL (*Indexed Nested Loops*) la Londra, aducând din tabela *Reports* doar înregistrările ce se potrivesc.

Ship to One Site

- Transferă tabela *Reports* la Londra
 - **Cost:** 1000 S + 4500 D (*Sort-Merge Join*; cost = $3 \cdot (500 + 1000)$)
 - Dacă dimensiunea rezultatului este mare, ambele relații ar putea fi transferate către serverul ce a inițiat interogarea iar join-ul este implementat acolo
- Transferă tabela *Employees* la Paris
 - **Cost:** 500 S + 4500 D

Semijoin

- La **Londra** se execută proiecția tablei *Employees* pe câmpul (câmpurile) folosite în join și rezultatul se transferă la Paris.
- La **Paris** se execută join între proiecția lui *Employees* și tabela *Reports*.
 - Rezultatul se numește **reducția** lui *Reports* relativ la *Employees* .
- Se transferă reducția lui *Reports* la Londra
- La **Londra**, se execută join între *Employees* și reducția lui *Reports*.

Bloomjoin

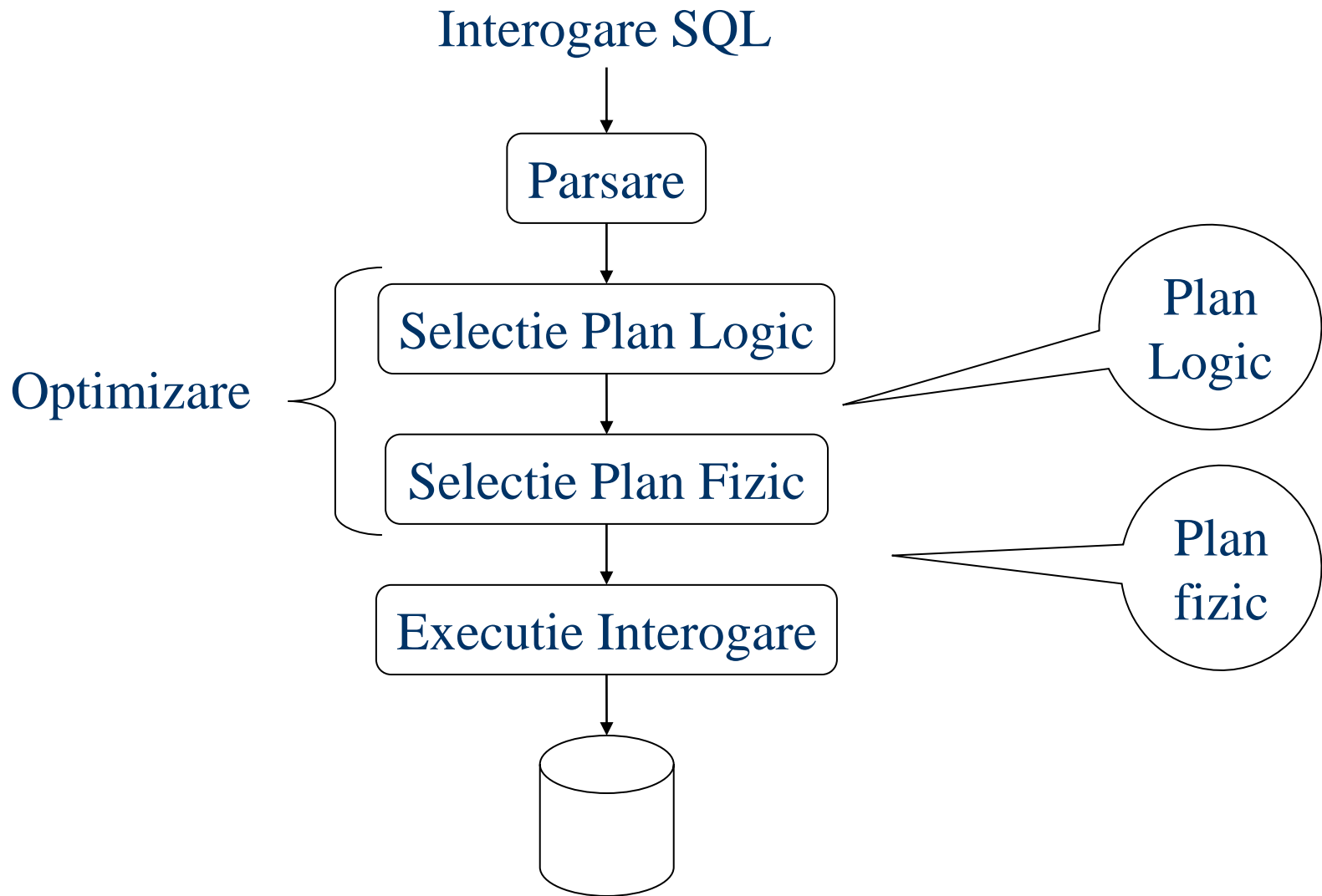
- La **Londra** se construiește un vector de biți de dimensiune k :
 - Folosind o funcție de dispersie, se împart valorile câmpului de join în partiții de la 0 la $k-1$.
 - Dacă funcția aplicată câmpului returnează i , se setează bitul i cu 1 (i de la 0 la $k-1$).
 - Se transferă vectorul de biți la *Paris*.

Bloomjoin (cont)

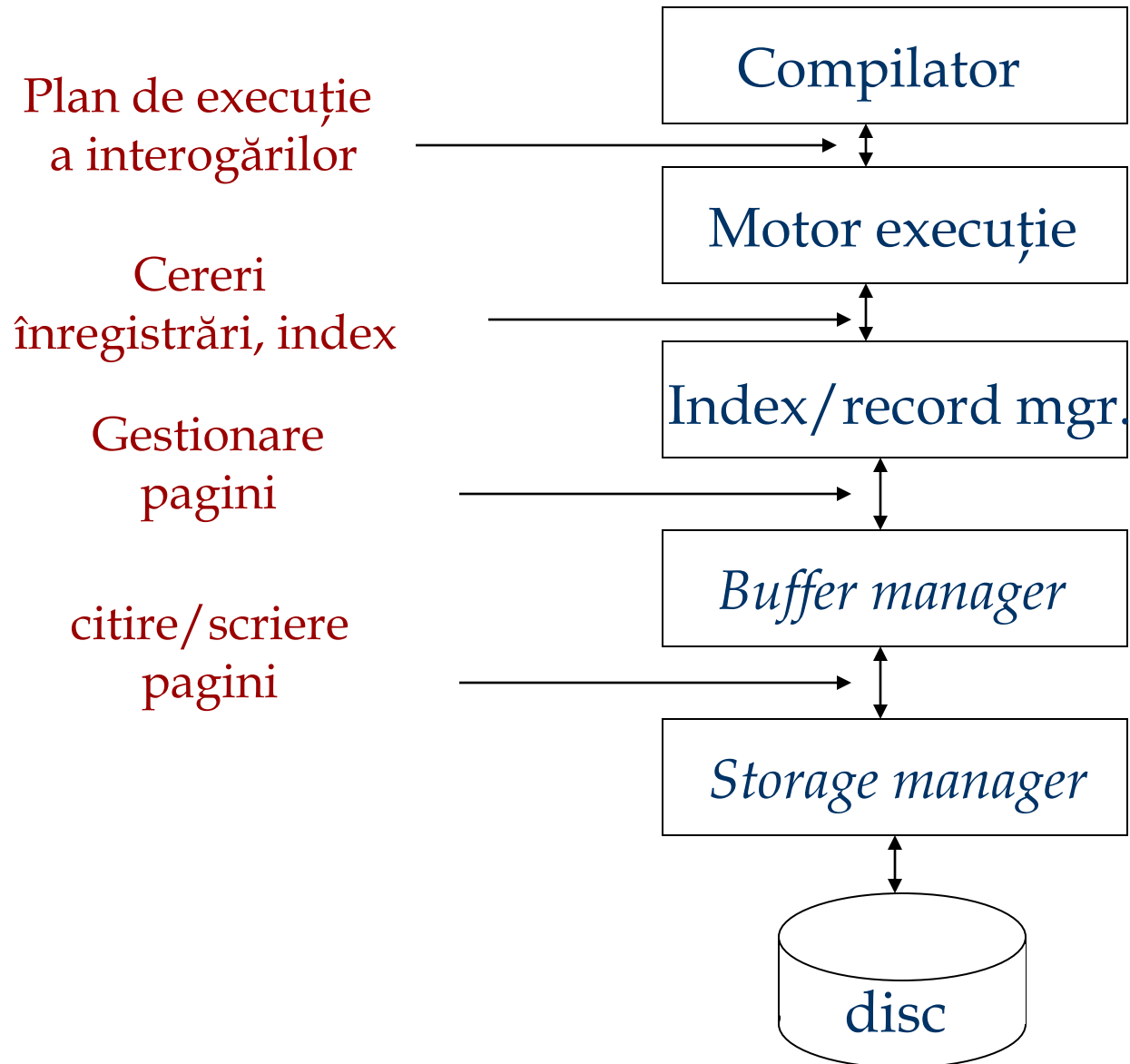
- La **Paris**, folosim similar funcția de dispersie. Dacă pentru un câmp se obține un i căruia în vector ii corespunde 0 , se elimină acea înregistrare din rezultat
 - Rezultatul se numește **reducție** a tabelii *Reports* în funcție de *Employees*.
- Se transferă reducția la Londra.
- La **Londra** se face join-ul dintre *Employees* și varianta redusă a lui *Reports*.

Optimizarea interogărilor

10



Executarea interogărilor



Structura folosită în exemple

Students (*sid*: integer, *sname*: string, *age*: integer)

Courses (*cid*: integer, *name*: string, *location*: string)

Evaluations (*sid*: integer, *cid*: integer, *day*: date, *grade*: integer)

■ *Students*:

- Fiecare înregistrare are o lungime de 50 bytes.
- 80 înregistrări pe pagină, 500 pagini.

■ *Courses*:

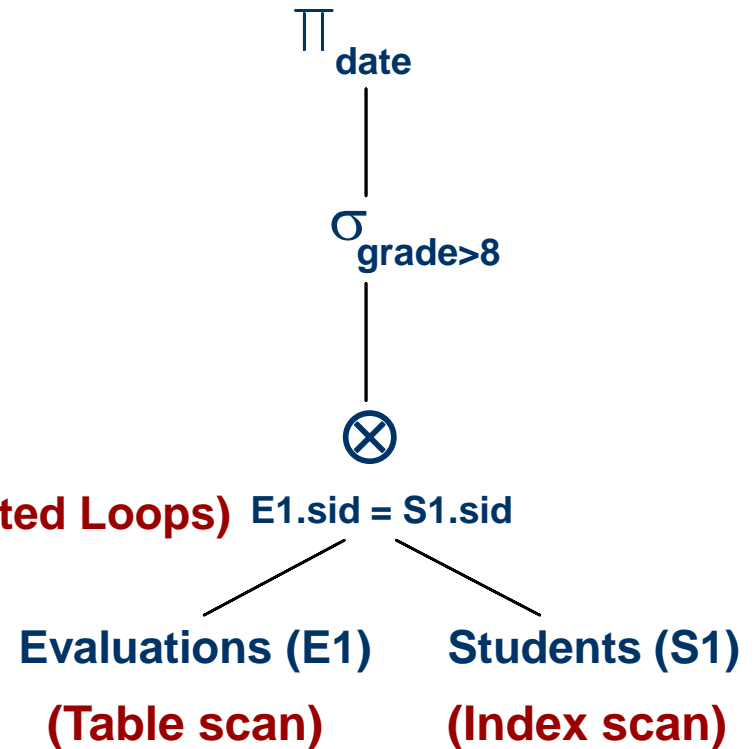
- Lungime înregistrare 50 bytes,
- 80 înregistrări pe pagină, 100 pagini.

■ *Evaluations*:

- Lungime înregistrare 40 bytes,
- 100 înregistrări pe pagină, 1000 pagini.

Planurile de execuție ale interogărilor

```
SELECT E1.date
FROM Evaluations E1, Students S1
WHERE E1.sid=S1.sid AND
      E1.grade > 8
```



Planul interogării:

- arbore logic
- se specifică o decizie de implementare la fiecare nod
- planificarea operațiilor

Frunzele planului de execuție: scanări

- **Table scan**: iterează prin înregistrările tabelului.
- **Index scan**: accesează înregistrările index-ului tabelului
- **Sorted scan**: accesează înregistrările tabelului după ce aceasta a fost sortată în prealabil.
- Cum se combină operațiile?
 - **Modelul iterator**. Fiecare operație e implementată cu 3 funcții:
 - *Open*: inițializări / pregătește structurile de date
 - *GetNext*: returnează următoarea înregistrare din rezultat
 - *Close*: finalizează operația / eliberează memoria

=> permite lucrul în pipeline!
 - **Modelul materializat (*data-driven*)**
 - Uneori modul de operare a ambelor modele e identic (exemplu: *sorted scan*).

Proces de optimizare a interogărilor

- Se transformă interogarea SQL într-un arbore logic:
 - identifică blocurile distincte (*view-uri*, sub-interogări).
- Se rescrie interogarea:
 - se aplica **transformări algebrice** pentru a obține un plan mai puțin costisitor.
 - se unesc blocuri de date și/sau se mută predicate între blocuri.
- Se optimizează fiecare bloc: **secvențele de execuție a join-urilor**.

Proces de optimizare a interogărilor

- Plan: *Arbore format din operatori algebrici relaționali*
 - Pentru fiecare operator este identificat un algoritm de execuție
 - Fiecare operator are (în general) implementată o interfață *'pull'*.
- Probleme:
 - Ce planuri se iau în considerare?
 - Cum se estimează costul unui plan?
- Se implementează algoritmi de identificare a planurilor cele mai puțin costisitoare:
 - **Ideal**: se dorește obținerea celui mai bun plan.
 - **Practic**: se elimină planurile cele mai costisitoare!

System R Optimizer

■ Impact:

- Cel mai utilizat algoritm;
- Funcționează bine pentru < 10 *join*-uri.

■ Estimare cost: aproximări, aproximări, aproximări...

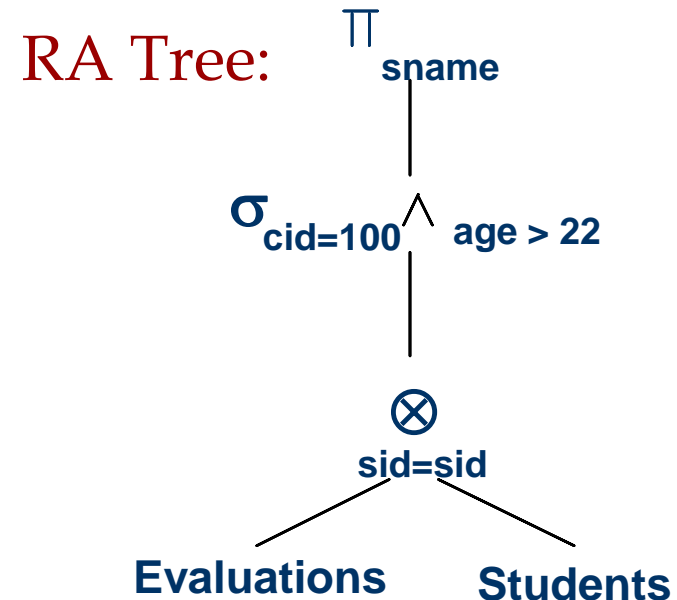
- Statistici, actualizate în catalogul bazei de date, folosite la estimarea costului operațiilor și a dimensiunii rezultatelor.
- Combinație între costul CPU și costurile de citire/scriere.

System R Optimizer

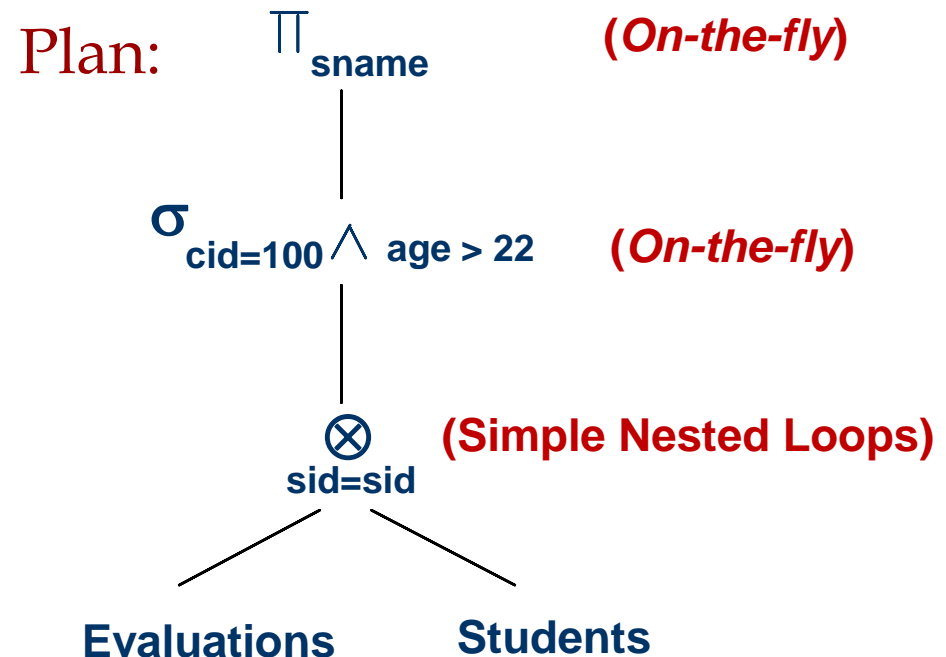
- Nu se estimeaza toate planurile!
 - Sunt considerate doar planurile *left-deep join*
 - Aceste planuri permit ca rezultatul unui operator sa fie transferat în *pipeline* către următorul operator fără stocarea temporară a relației.
 - Se exclude produsul cartezian

Exemplu

```
SELECT S.sname
FROM Evaluations E, Students S
WHERE E.sid=S.sid AND
      E.cid=100 AND S.age>22
```

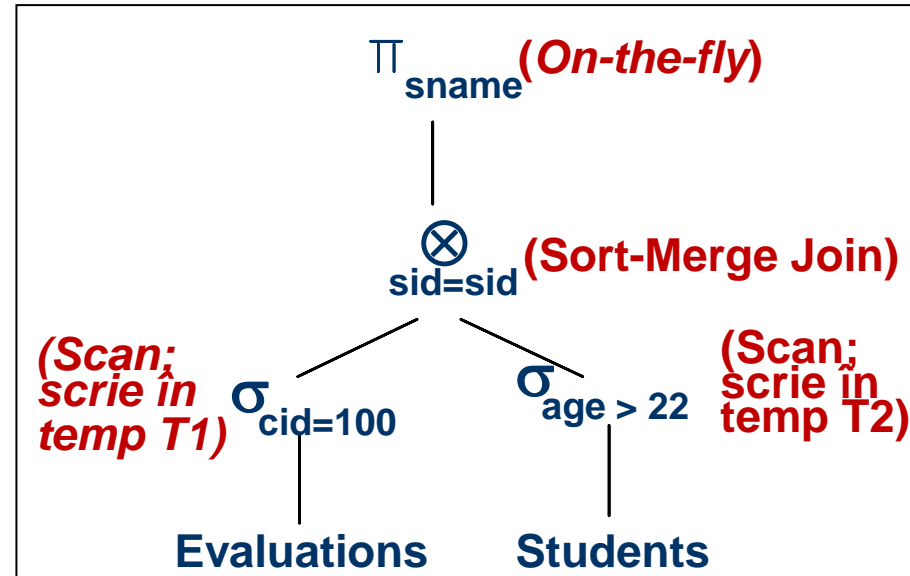


- Cost: 500+500*1000 I/Os
- Plan inadecvat, cost f mare!
- *Scopul optimizarii:* căutarea de planuri mai eficiente ce calculează același răspuns.



Plan alternativ 1

- *Diferența esențială:*
poziția operatorilor de selecție.

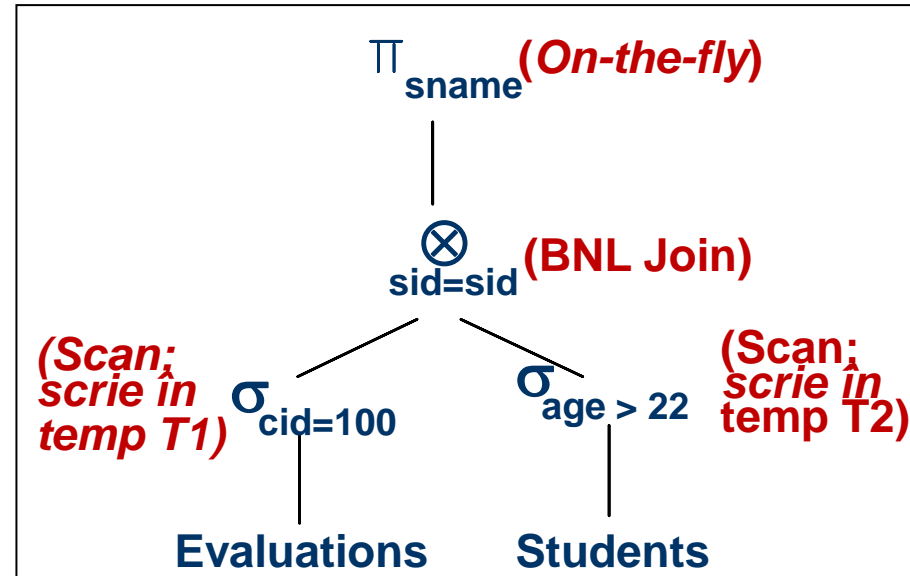


- **Costul planului**
(presupunem că sunt 5 pagini în buffer):
 - Scan *Evaluations* (1000) + memorează temp T1 (10 pag, dacă avem 100 cursuri și distribuție uniformă). – *total 1010 I/Os*
 - Scan *Students* (500) + memorează temp T2 (250 pag, dacă avem 10 vârste). – *total 750 I/Os*
 - Sortare T1 ($2 \cdot 2 \cdot 10$), sortare T2 ($2 \cdot 3 \cdot 250$), interclasare ($10 + 250$) – *total 1800 I/Os*
 - Total: 3560 pagini I/Os.

Plan alternativ 1

■ Diferența esențială:

poziția operatorilor de selecție.



■ Costul planului

(presupunem că sunt 5 pagini în buffer):

■ Dacă se folosește BNL join:

- cost join = $10 + 4 * 250$,

- cost total = 2770.

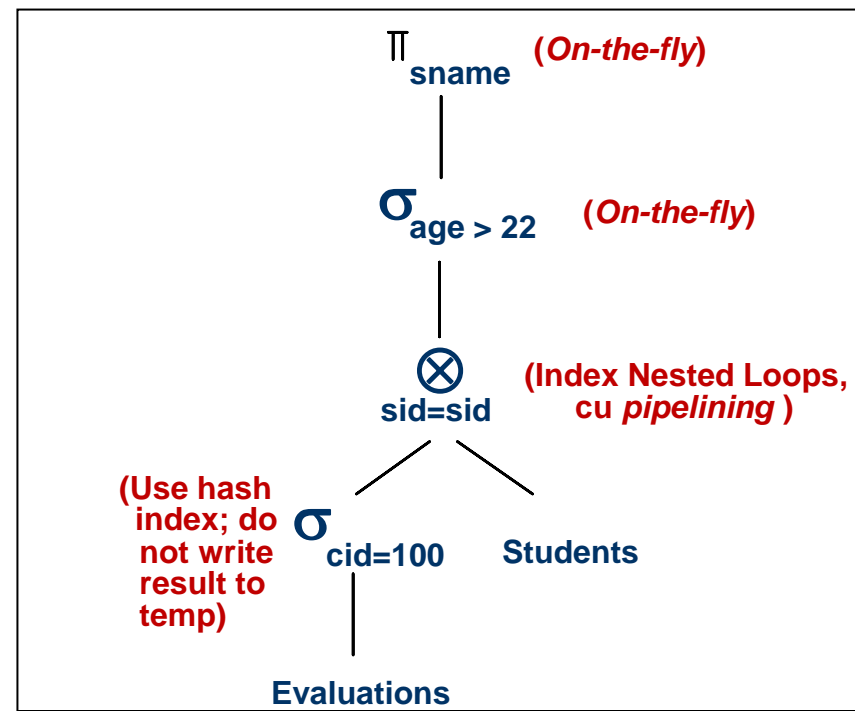
■ Dacă 'împingem' proiecțiile:

- T1 rămâne cu *sid*, T2 rămâne cu *sid* și *sname*:

- T1 încapă în 3 pagini, costul BNL este sub 250 pagini, total < 2000.

Plan alternativ 2

- Cu index grupat pe *cid* din *Evaluations*, avem
 $100,000/100 = 1000$ tupluri în
 $1000/100 = 10$ pagini.
 - INL cu *pipelining* (rezultatul nu e materializat).
 - Se elimină câmpurile inutile din output.
 - Coloana *sid* e cheie pentru *Students*.
 - Cel mult o "potrivire", index grupat pe *sid* e OK.
 - Decizia de a nu „împinge” selecția *age>22* mai repede e dată de disponibilitatea indexului pe *sid* al *Students*.
- Cost:** Selecție pe *Evaluations* (10 I/Os); pentru fiecare obținem înregistrările din *Students* ($1000 \cdot 1.2$); total **1210 I/Os**.



Unitatea de optimizare: *bloc Select*

- O interogare SQL este descompusă într-o colecție de *blocuri Select* care sunt optimizate separat.
- *Blocurile Select* imbricate sunt de obicei tratate ca apeluri de subrutine (câte un apel pentru fiecare înregistrare din blocul *Select* extern)

```
SELECT S.sname  
FROM Students S  
WHERE S.age IN  
      (SELECT MAX (S2.age)  
       FROM Students S2  
       GROUP BY S2.sname)
```

Bloc extern

Bloc imbricat

Pentru fiecare bloc, planurile considerate sunt:

- Toate metodele disponibile de acces, pt fiecare tabelă din FROM
- Toate planurile *left-deep join trees* (adică, toate modurile secvențiale de *join* ale tabelelor, considerând toate permutările de tabele posibile)

Estimarea costului

- Se estimează costul fiecărui plan considerat:
 - Trebuie *estimat costul* fiecărui operator din plan
 - Depinde de cardinalitatea tabelelor de intrare
 - Modul de estimarea al costurilor a fost discutat în cursurile precedente (scanare tabele, join-uri, etc.)
 - Trebuie *estimată dimensiunea rezultatului* pentru fiecare operație a arborelui!
 - Se utilizează informații despre relațiile de intrare
 - Pentru selecții și join-uri, se consideră predicatele ca fiind independente.
- Algoritmul *System R*
 - Inexact, dar cu rezultate bune în practică.
 - În prezent există metode mai sofisticate

Echivalențe în algebra relațională

- Permite alegerea unei ordini diferite a join-urilor și 'împingerea' selecțiilor și proiecțiilor în fața join-urilor.
- Selectii: $\sigma_{c1 \wedge \dots \wedge cn}(R) \equiv \sigma_{c1}(\dots(\sigma_{cn}(R)))$ (Cascadă)
 $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (Comutativitate)
- Proiecții: $\pi_{a1}(R) \equiv \pi_{a1}(\dots(\pi_{an}(R)))$ (Cascadă)
- Join:
 $R \otimes (S \otimes T) \equiv (R \otimes S) \otimes T$ (Asociativitate)
 $(R \otimes S) \equiv (S \otimes R)$ (Comutativitate)
 $\rightarrow R \otimes (S \otimes T) \equiv (T \otimes R) \otimes S$

Alte echivalențe

- A proiecție se comută doar cu o selecție ce utilizează câmpurile ce apar în proiecție.
- Selecția dintre câmpurile ce aparțin tabelelor implicate într-un produs cartezian convertește produsul cartezian într-un *join*.
- O selecție doar pe attributele lui R comută cu $R \otimes S$.
(adică, $\sigma(R \otimes S) \equiv \sigma(R) \otimes S$)
- Similar, dacă o proiecție urmează unui join $R \otimes S$, putem să o ‘împingem’ în fața join-ului păstrând doar câmpurile lui R (și S) care sunt necesare pentru join sau care apar în lista proiecției.

Enumerarea planurilor alternative

- Sunt luate în considerare două cazuri:
 - Planuri cu o singură tabelă
 - Planuri cu tabele multiple
- Pentru interogările ce implică o singură tabelă, planul conține o combinație de selecturi, proiecții și operatori de agregare:
 - Sunt considerate toate metodele de acces și este păstrată cea cu cel mai mic cost estimat.
 - Mai mulți operatori sunt executați deodată (în *pipeline*)

Estimări de cost pentru planuri bazate pe o tabelă

- Indexul I pt cheia primară implicată într-o selecție:
 - Costul e $Height(I)+1$ pt un arbore B+, sau $1.2+1$ pt hash index.
- Index grupat I pe câmpurile implicate în una sau mai multe selecții:
 - $(NPages(I)+NPages(R)) * produs\ al\ FR\ pt\ fiecare\ selecție$
- Index negrupat I pe câmpurile implicate în una sau mai multe selecții:
 - $(NPages(I)+NTuples(R)) * produs\ al\ FR\ pt\ fiecare\ selecție.$
- Scanare secvențială a tabelii:
 - $NPages(R).$

Exemplu

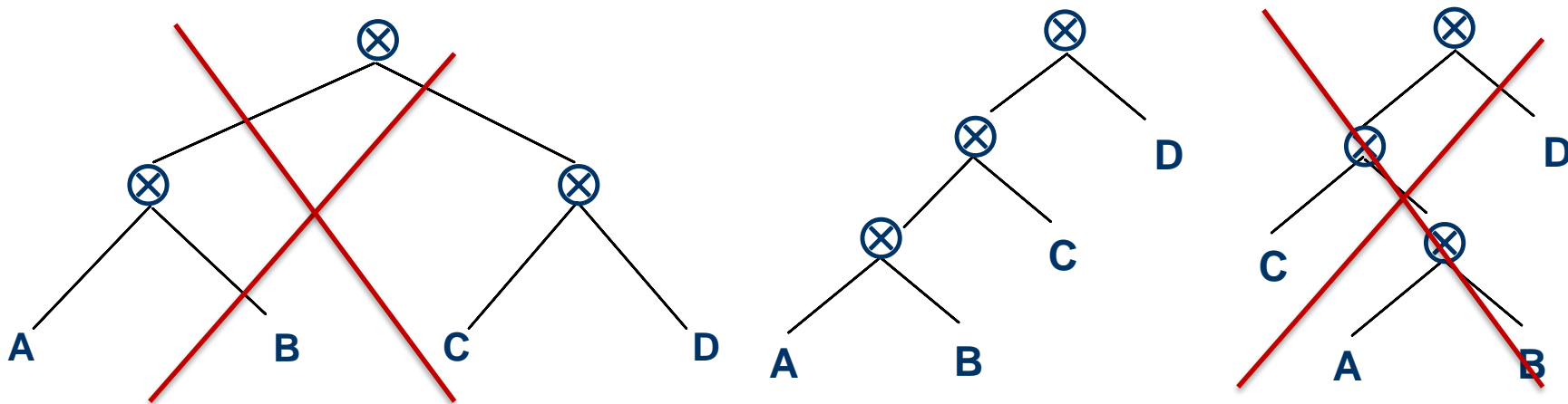
```
SELECT S.sid  
FROM Students S  
WHERE S.age=20
```

- Dacă există **index** pt. *age*:
 - $(1/NKeys(I)) * NTuples(R) = (1/10) * 40000$ înreg. returnate
 - **Index grupat**: $(1/NKeys(I)) * (NPages(I)+NPages(R)) = (1/10) * (50+500)$ pagini returnate.
 - **Index negrupat**: $(1/NKeys(I)) * (NPages(I)+NTuples(R)) = (1/10) * (50+40000)$ pagini returnate.
- Dacă se **scanează** tabela :
 - Sunt citite toate paginile (500).

Interogări pe tabele multiple

- System R consideră doar arborii *left-deep join*.

- Pe măsură ce numărul de join-uri crește , numărul de planuri alternative este tot mai semnificativ; *este necesară restricționarea spațiului de căutare.*
- Arborii *left-deep* ne permit generarea tuturor planurilor ce suportă *pipeline* complet.
 - Rezultatele intermediare nu sunt salvate în tabele temporare.
 - Nu toți arborii *left-deep* suportă *pipeline* complet (ex. *SM join*).



Enumerarea planurilor *left-deep*

- Planurile *left-deep* diferă prin ordinea tabelelor, metoda de acces a fiecărei tablele și metoda utilizată pentru implementarea fiecărui *join*.
- N pași de dezvoltare a planurilor cu N tablele:
 - **Pas 1:** Găsirea celui mai bun plan cu o tabelă pentru fiecare tabelă.
 - **Pas 2:** Găsirea celei mai bune variante de join al rezultatului unui plan cu o tabelă și altă tabelă (*Toate planurile bazate pe 2 tablele*)
 - **Pas N:** Găsirea celei mai bune variante de join al rezultatului unui plan cu N-1 tablele și altă tabelă . (*Toate planurile bazate pe N tablele*)

Enumerarea planurilor *left-deep*

- Pentru fiecare submulțime de relații, se reține:
 - Cel având costul cel mai redus, plus
 - Planul cu cel mai mic cost pentru fiecare *ordonare interesantă* a înregistrărilor.
- **ORDER BY**, **GROUP BY** și **agregările** sunt tratate la final, folosind planurile ordonate sau un operator de sortare adițional.
- Un plan bazat pe N-1 tabele nu se combină cu o altă tabelă dacă nu există o condiție de join între acestea (și dacă mai există predicate nefolosite în clauza WHERE)
 - adică se **evită produsul cartezian**, dacă se poate.
- În ciuda restrângerii mulțimii de planuri considerate, numărul acestora crește **exponențial** cu numărul tabelelor implicate

Exemplu

Pas 1:

Students:

Arbore B+ pt *age*

Hash pt *sid*

Evaluations:

Arbore B+ pt *cid*

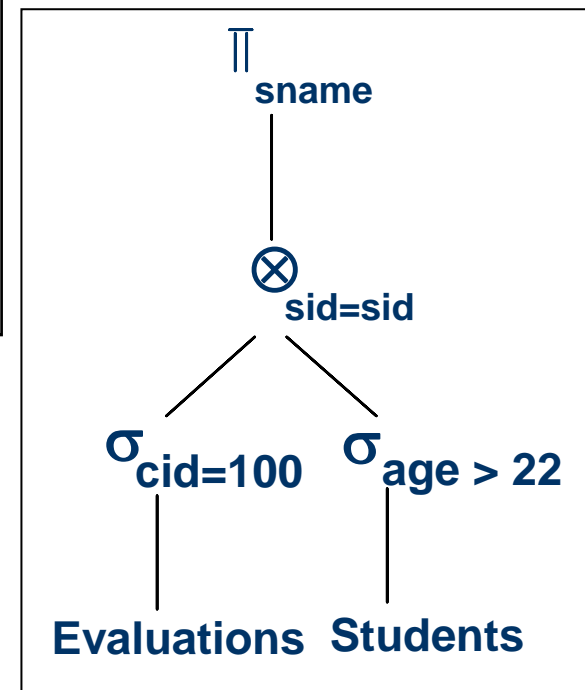
- **Students:** Utilizarea arborelui B+ pentru selecția $age > 22$ este cea mai puțin costisitoare. Totuși, dacă rezultatul va avea multe înregistrări și indexul nu este clusterizat, scanarea tabelului poate fi mai avantajoasă.

- Se preferă arborele B+ (deoarece rezultatul e ordonat după *age*).

- **Evaluations:** Utilizare arborelui B+ pentru selecția $cid=500$ este cea mai avantajoasă.

Pas 2:

- Se consideră fiecare plan rezultat la Pas 1, și se combină cu cea de-a doua tabelă. (ex: se folosește indexul Hash pe *sid* pentru selectarea înregistrărilor din *Students* ce satisfac condiția de join)



Interogări imbricate

- *Blocurile Select imbricate sunt optimizate independent, înregistrarea curentă a blocului Select extern furnizând date pentru o condiție de selecție.*
- Blocul extern e optimizat ținând cont de costul `apelului' blocului imbricat.
- Ordonarea implicită a acestor blocuri implică faptul că anumite strategii nu vor fi considerate. *Versiunile neimbricate ale unei interogări sunt (de obicei) optimizate mai bine.*

```
SELECT S.sname  
FROM Students S  
WHERE EXISTS  
  (SELECT *  
   FROM Evaluations E  
   WHERE E.cid=103  
    AND E.sid=S.sid)
```

Bloc Select de optimizat:

```
SELECT *  
FROM Evaluations E  
WHERE E.cid=103  
      AND S.sid= valoare ext
```

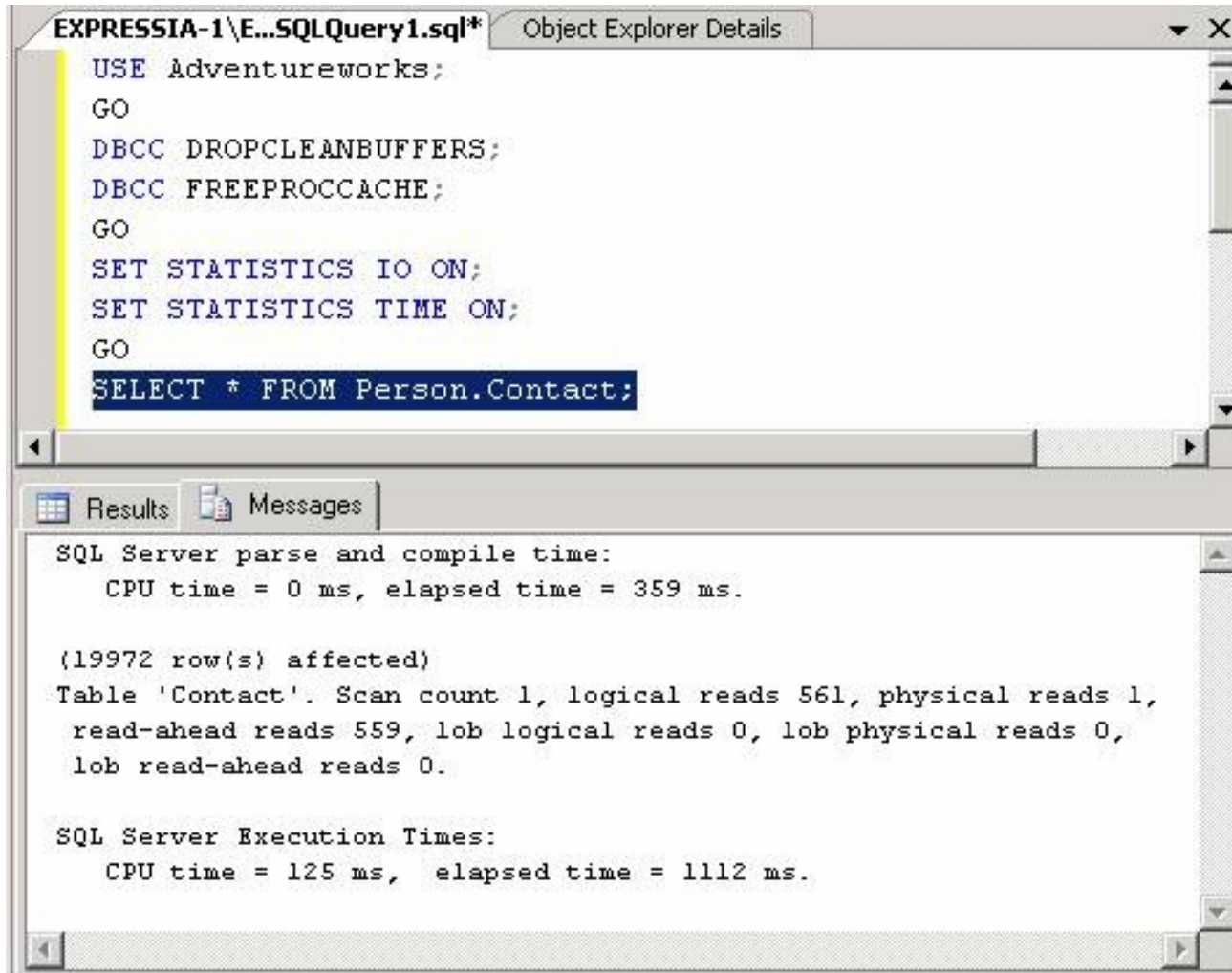
Interogare simplă echivalentă :

```
SELECT S.sname  
FROM Students S,  
Evaluations E  
WHERE S.sid=E.sid  
      AND E.cid=103
```

Optimizarea interogărilor distribuite

- Abordare bazată pe cost; similară optimizării centralizate, se consideră toate planurile, alegându-se cel mai ieftin.
 - **Diferență 1:** Trebuie considerate costurile de transfer.
 - **Diferență 2:** Trebuie respectată autonomia locală a siteului.
 - **Diferență 3:** Se considera metode noi de join in context distribuit.
- Este creat un **plan global**, cu **planurile local sugerate** de fiecare site.
 - Dacă un site poate îmbunătăți planul local sugerat, este liber să o facă.

Planuri de execuție în MS SQL Server



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The top pane, titled 'EXPRESSIA-1\E...SQLQuery1.sql*', contains the following T-SQL script:

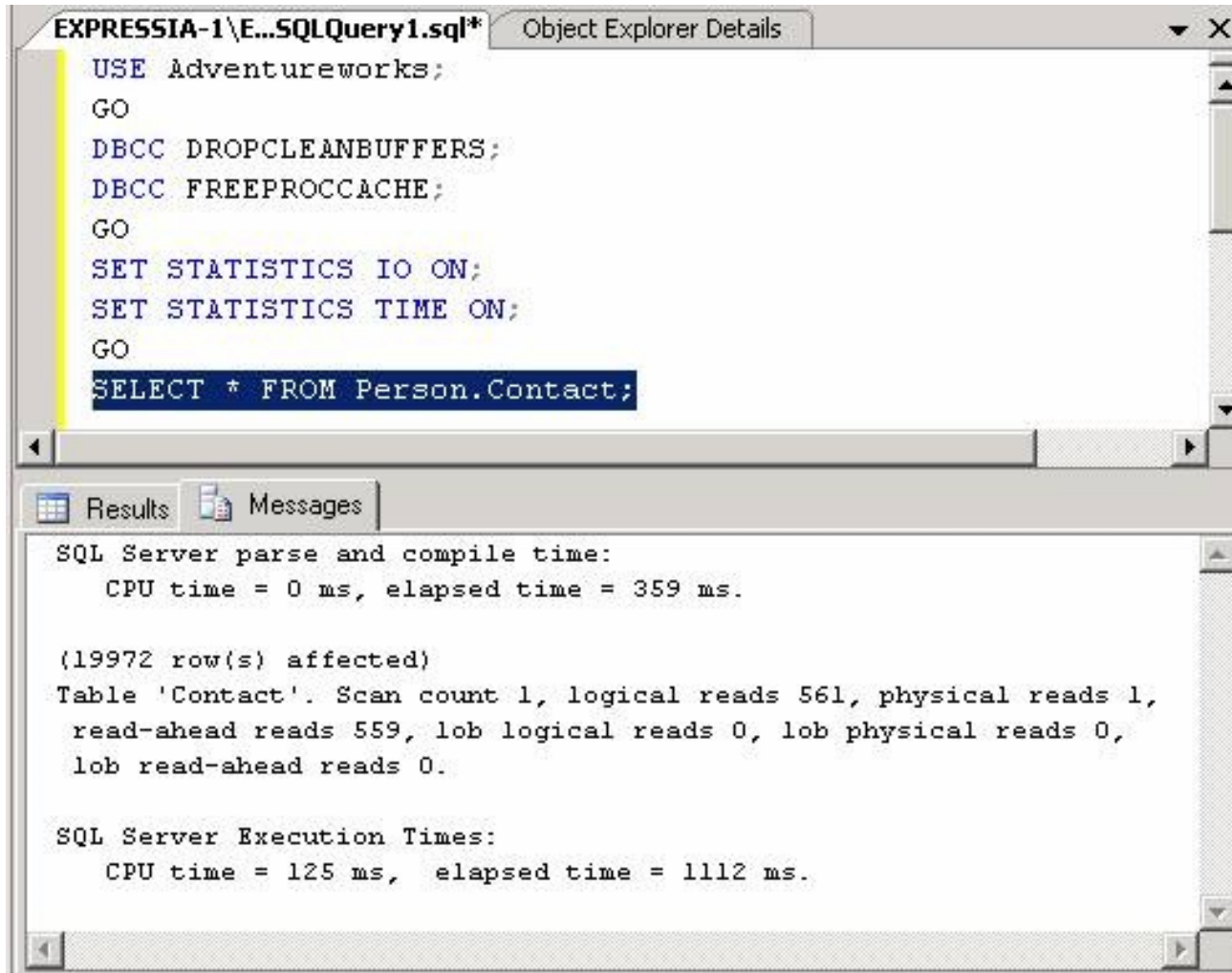
```
USE Adventureworks;  
GO  
DBCC DROPCLEANBUFFERS;  
DBCC FREEPROCCACHE;  
GO  
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;  
GO  
SELECT * FROM Person.Contact;
```

The bottom pane, titled 'Results', shows the execution output:

```
SQL Server parse and compile time:  
CPU time = 0 ms, elapsed time = 359 ms.  
  
(19972 row(s) affected)  
Table 'Contact'. Scan count 1, logical reads 561, physical reads 1,  
read-ahead reads 559, lob logical reads 0, lob physical reads 0,  
lob read-ahead reads 0.  
  
SQL Server Execution Times:  
CPU time = 125 ms, elapsed time = 1112 ms.
```

- DBCC DROPCLEANBUFFERS – șterge datele din ³⁸buffer
- DBCC FREEPROCCACHE – șterge planurile din memoria internă

Planuri de execuție în MS SQL Server



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The top pane shows a query window titled 'EXPRESSIA-1\E...SQLQuery1.sql*' with the following SQL code:

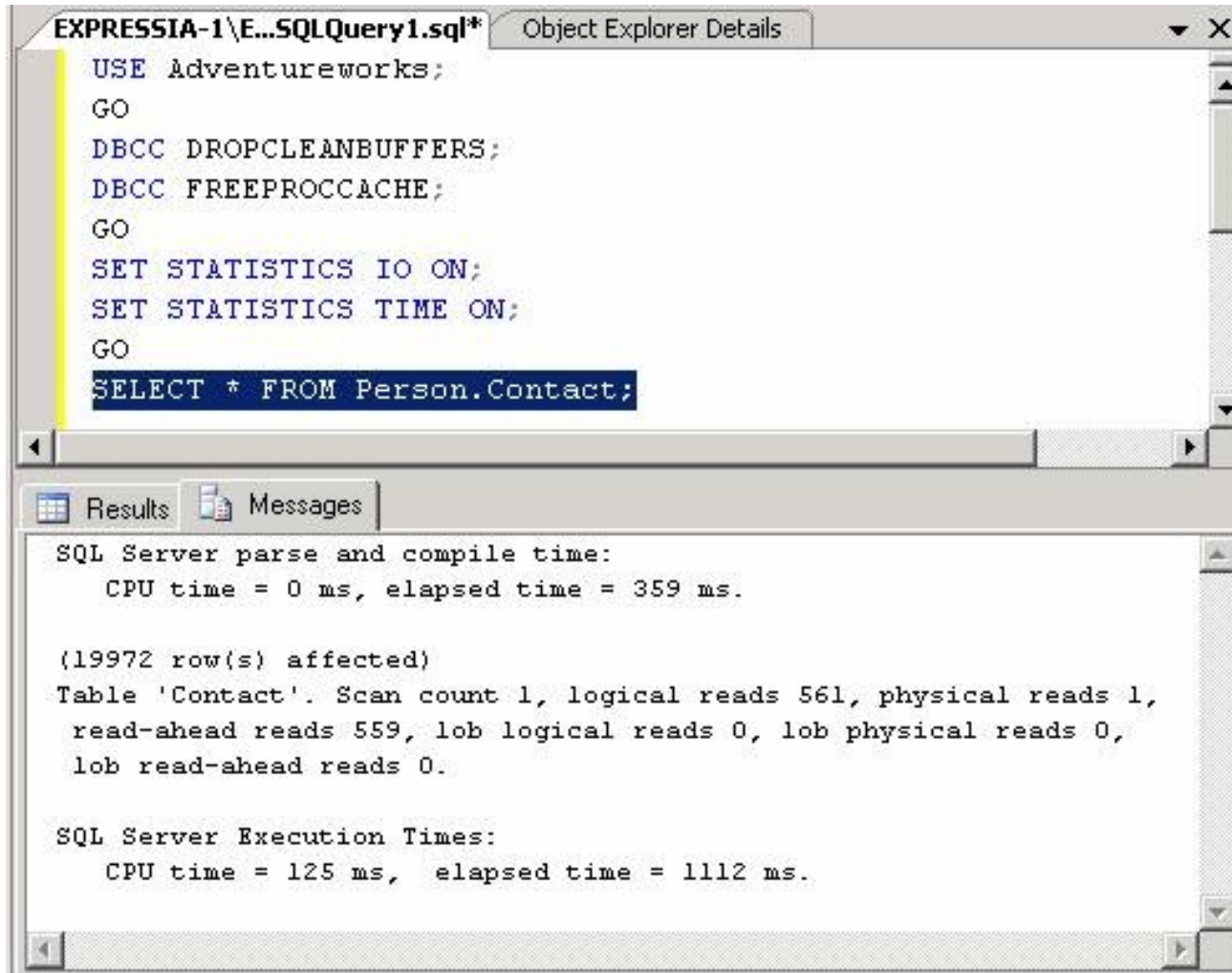
```
USE Adventureworks;  
GO  
DBCC DROPCLEANBUFFERS;  
DBCC FREEPROCCACHE;  
GO  
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;  
GO  
SELECT * FROM Person.Contact;
```

The bottom pane shows the 'Results' tab with the following output:

```
SQL Server parse and compile time:  
    CPU time = 0 ms, elapsed time = 359 ms.  
  
(19972 row(s) affected)  
Table 'Contact'. Scan count 1, logical reads 561, physical reads 1,  
    read-ahead reads 559, lob logical reads 0, lob physical reads 0,  
    lob read-ahead reads 0.  
  
SQL Server Execution Times:  
    CPU time = 125 ms,  elapsed time = 1112 ms.
```

- CPU time – resursele CPU folosite la executarea interogării
- Elapsed time – cât a durat executarea interogării

Planuri de execuție în MS SQL Server



The screenshot shows the Microsoft SQL Server Enterprise Manager interface. The top pane displays a SQL query in the 'Object Explorer Details' tab. The query is as follows:

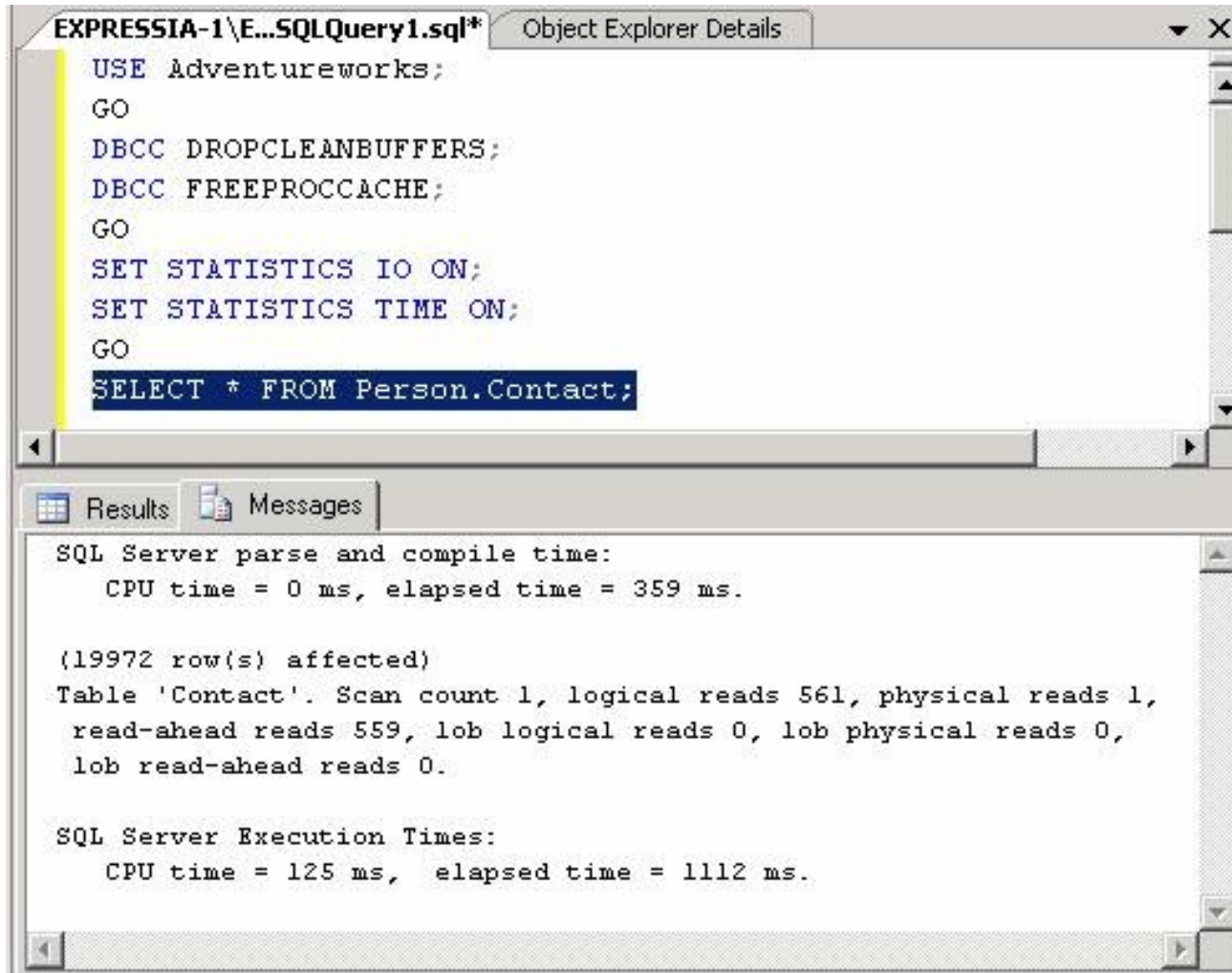
```
USE Adventureworks;  
GO  
DBCC DROPCLEANBUFFERS;  
DBCC FREEPROCCACHE;  
GO  
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;  
GO  
SELECT * FROM Person.Contact;
```

The bottom pane shows the 'Results' tab with the following output:

```
SQL Server parse and compile time:  
CPU time = 0 ms, elapsed time = 359 ms.  
  
(19972 row(s) affected)  
Table 'Contact'. Scan count 1, logical reads 561, physical reads 1,  
read-ahead reads 559, lob logical reads 0, lob physical reads 0,  
lob read-ahead reads 0.  
  
SQL Server Execution Times:  
CPU time = 125 ms, elapsed time = 1112 ms.
```

- physical reads –numărul de pagini citite de pe disc
- read-ahead reads - numărul de pagini citite în buffer pt interogare

Planuri de execuție în MS SQL Server



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The top pane shows a query script in the 'Object Explorer Details' tab. The script includes commands to set statistics and execute a query. The bottom pane shows the 'Results' tab with the execution output.

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

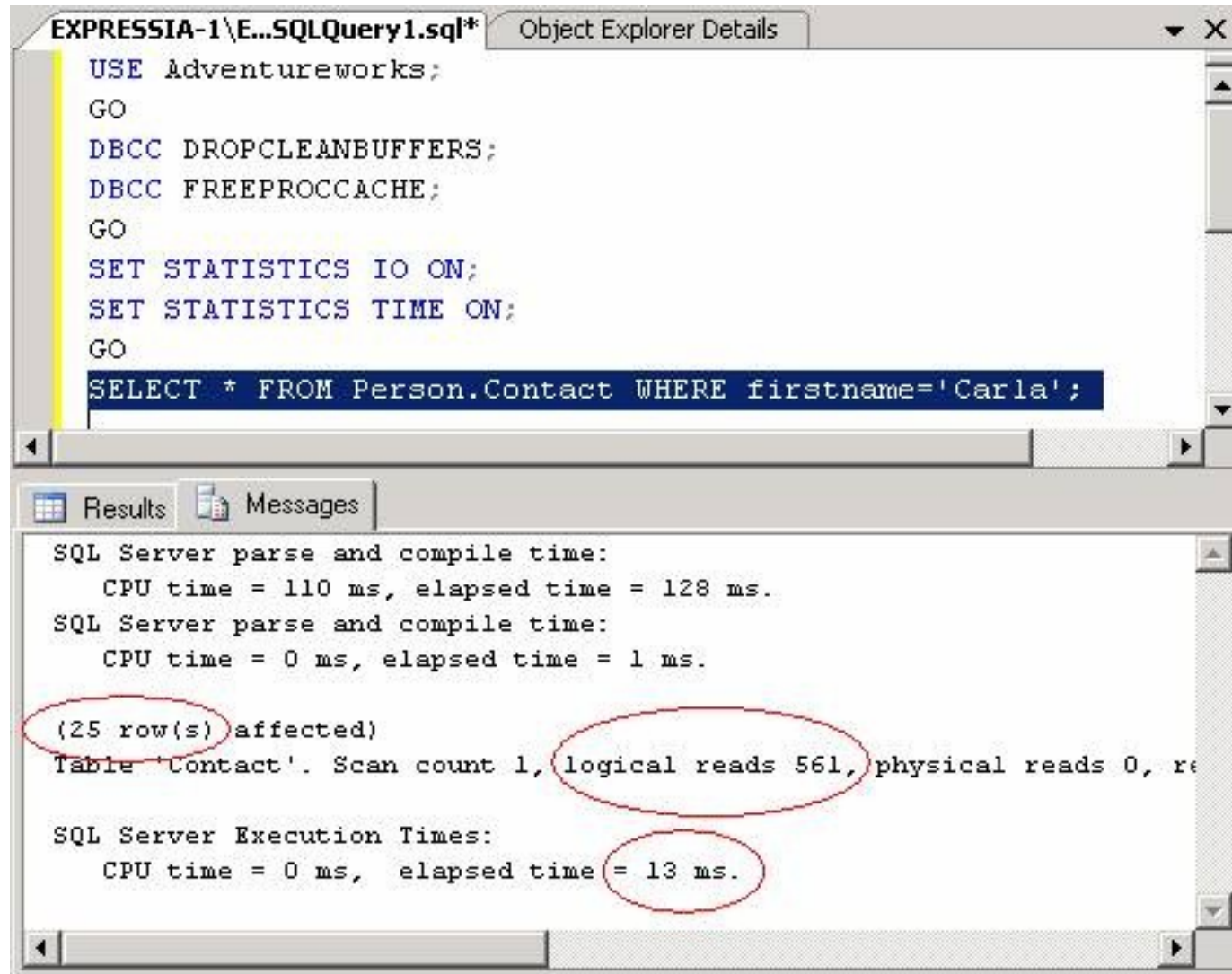
SQL Server parse and compile time:
CPU time = 0 ms, elapsed time = 359 ms.

(19972 row(s) affected)
Table 'Contact'. Scan count 1, logical reads 561, physical reads 1,
read-ahead reads 559, lob logical reads 0, lob physical reads 0,
lob read-ahead reads 0.

SQL Server Execution Times:
CPU time = 125 ms, elapsed time = 1112 ms.

- CPU time – resursele CPU folosite la executarea interogării
- Elapsed time – cât a durat executarea interogării

Planuri de execuție în MS SQL Server



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The top pane shows a query window with the following SQL code:

```
USE Adventureworks;  
GO  
DBCC DROPCLEANBUFFERS;  
DBCC FREEPROCCACHE;  
GO  
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;  
GO  
SELECT * FROM Person.Contact WHERE firstname='Carla';
```

The bottom pane shows the execution plan results, which include the following text:

```
SQL Server parse and compile time:  
  CPU time = 110 ms, elapsed time = 128 ms.  
SQL Server parse and compile time:  
  CPU time = 0 ms, elapsed time = 1 ms.  
(25 row(s) affected)  
Table 'Contact'. Scan count 1, logical reads 561, physical reads 0, re  
SQL Server Execution Times:  
  CPU time = 0 ms, elapsed time = 13 ms.
```

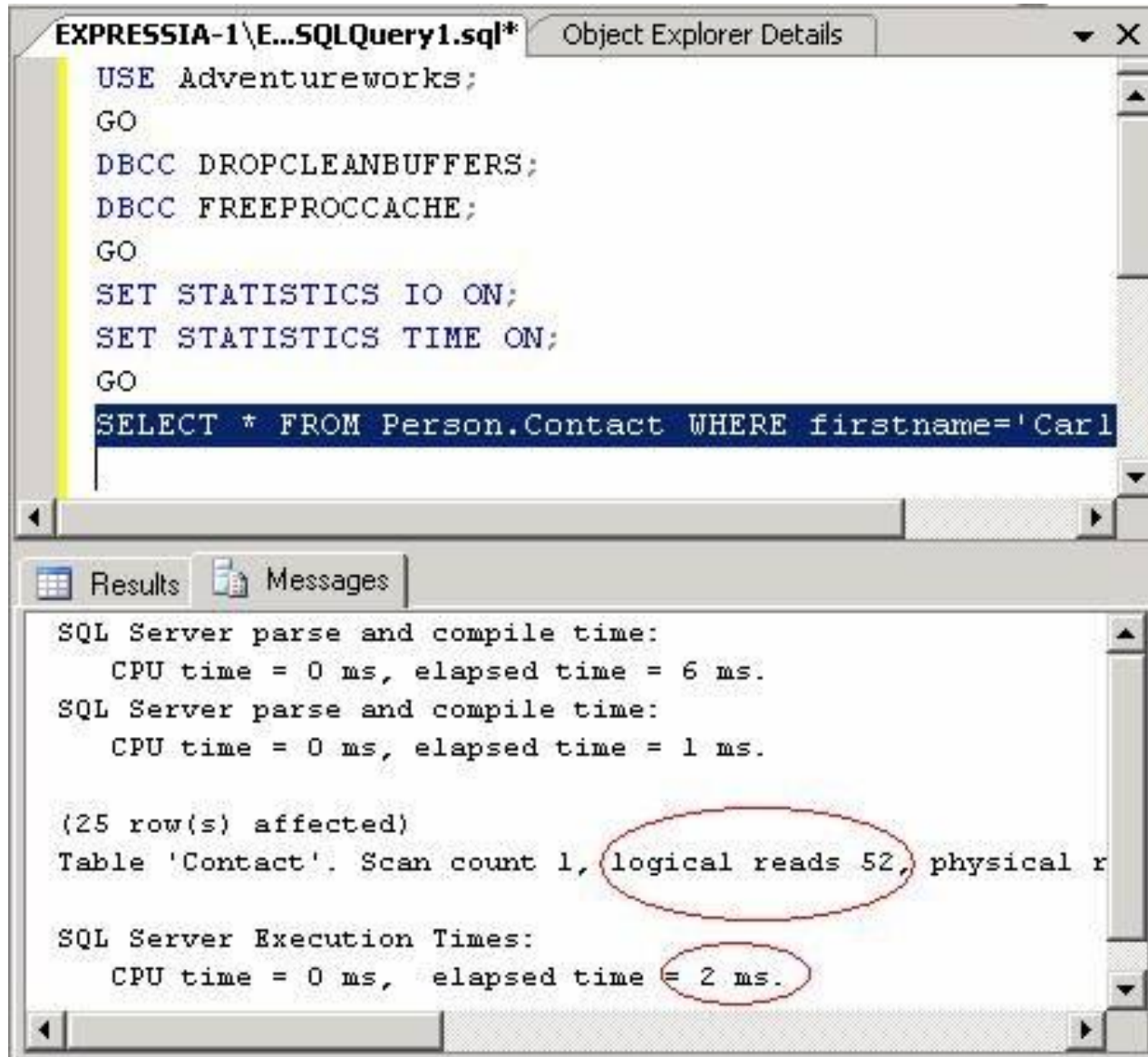
Red circles are drawn around the following values in the execution plan:

- (25 row(s) affected)
- logical reads 561
- elapsed time = 13 ms.

Planuri de execuție în MS SQL Server

```
USE [AdventureWorks] GO
CREATE NONCLUSTERED INDEX [IDX_firstname]
ON [Person].[Contact]
(
    [FirstName] ASC
)
GO
```

Planuri de execuție în MS SQL Server



The screenshot displays the Microsoft SQL Server Enterprise Manager interface. The top pane shows a SQL query in the 'Object Explorer Details' tab. The query is as follows:

```
USE Adventureworks;  
GO  
DBCC DROPLEANBUFFERS;  
DBCC FREEPROCCACHE;  
GO  
SET STATISTICS IO ON;  
SET STATISTICS TIME ON;  
GO  
SELECT * FROM Person.Contact WHERE firstname='Carl'
```

The bottom pane shows the 'Results' tab with the following execution statistics:

```
SQL Server parse and compile time:  
    CPU time = 0 ms, elapsed time = 6 ms.  
SQL Server parse and compile time:  
    CPU time = 0 ms, elapsed time = 1 ms.  
  
(25 row(s) affected)  
Table 'Contact'. Scan count 1, logical reads 52, physical r  
  
SQL Server Execution Times:  
    CPU time = 0 ms, elapsed time = 2 ms.
```

In the execution statistics, the values 'logical reads 52' and 'elapsed time = 2 ms.' are circled in red.

SHOWPLAN_ALL

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

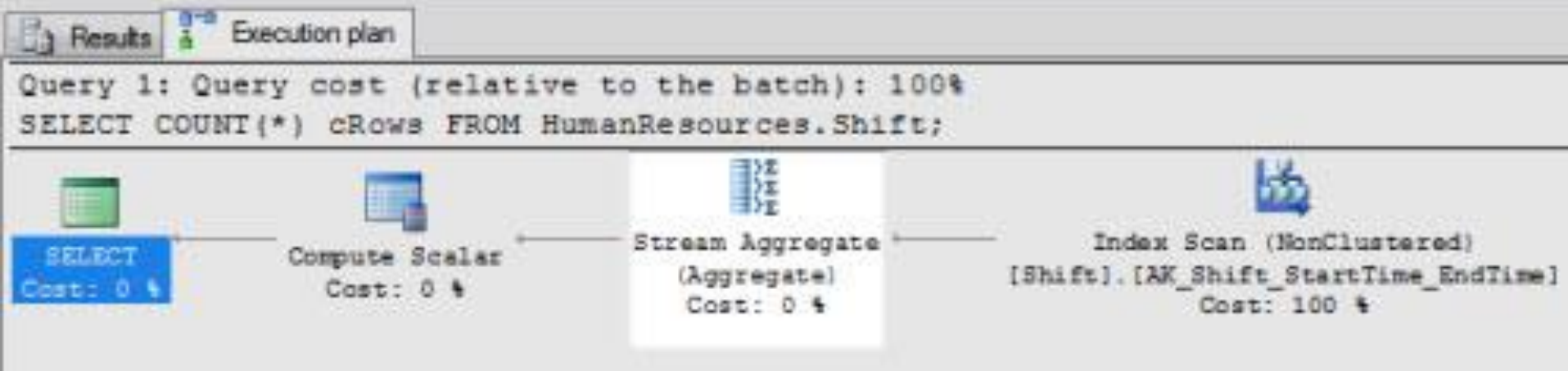
Results

StmtText

```
-----  
SELECT COUNT(*) cRows  
FROM HumanResources.Shift;  
    |--Compute Scalar(DEFINE:([Expr1003]=CONVERT_IMPLICIT(int,[Expr1004],0)))  
    |--Stream Aggregate(DEFINE:([Expr1004]=Count(*)))  
        |--Index Scan(OBJECT:([master].[HumanResources].[Shift].[AK_Shift])  
  
(4 row(s) affected)
```

Plan de execuție grafică

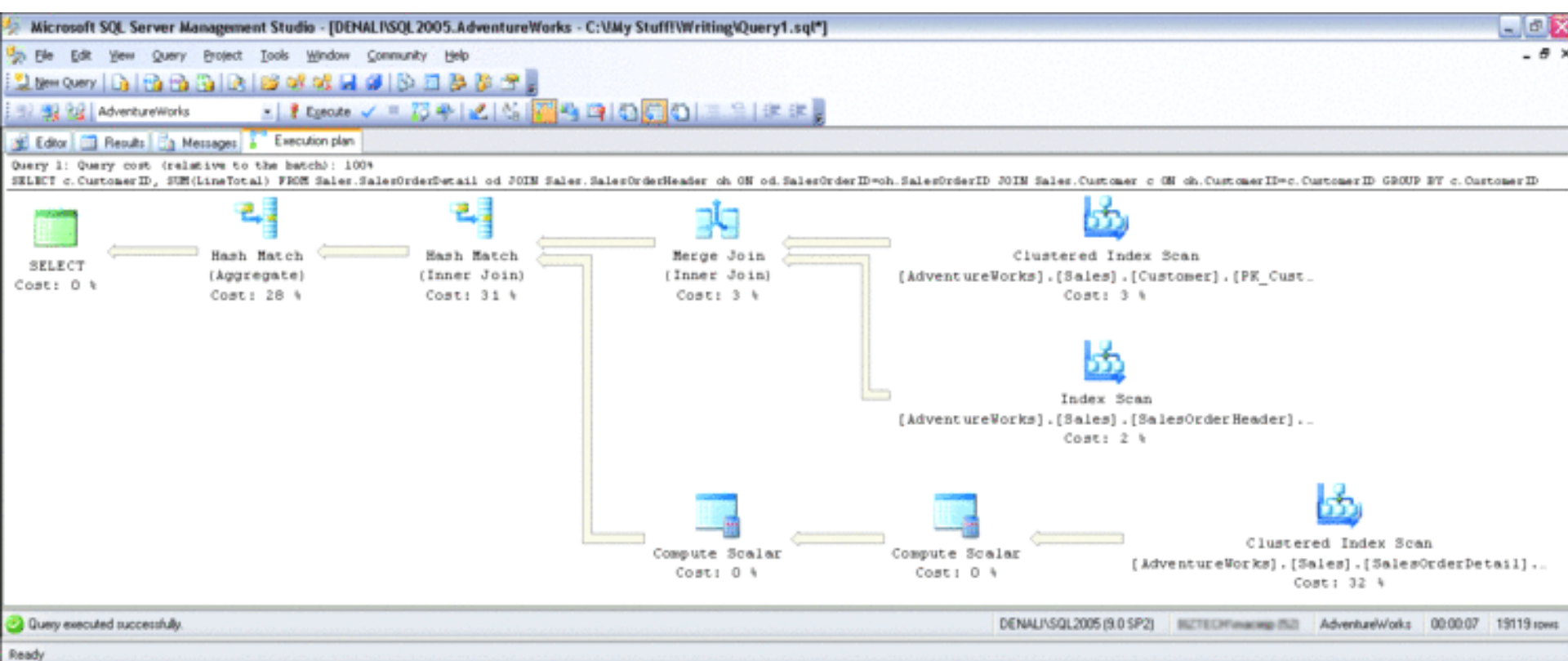
```
USE AdventureWorks
GO
SELECT COUNT(*) cRows
FROM HumanResources.Shift;
GO
```



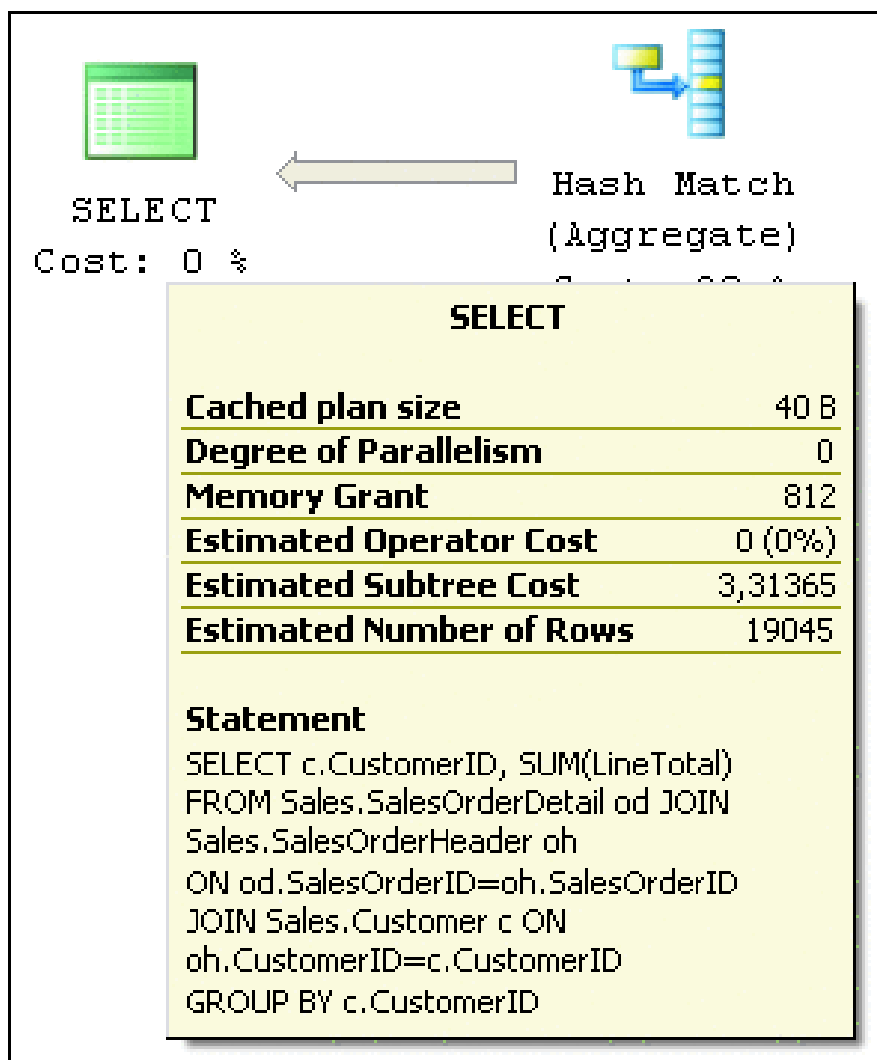
Plan de execuție grafică

```
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 grafică



Plan de execuție grafică



Clustered Index Scan
[AdventureWorks].[Sales].[SalesOrderDetail] ...
Cost: 32 %

Clustered Index Scan	
Scanning a clustered index, entirely or only a range.	
Physical Operation	Clustered Index Scan
Logical Operation	Clustered Index Scan
Actual Number of Rows	121317
Estimated I/O Cost	0,915718
Estimated CPU Cost	0,133606
Estimated Operator Cost	1,04932 (32%)
Estimated Subtree Cost	1,04932
Estimated Number of Rows	121317
Estimated Row Size	29 B
Actual Rebinds	0
Actual Rewinds	0
Ordered	False
Node ID	8

Object
[AdventureWorks].[Sales].[SalesOrderDetail].
[PK_SalesOrderDetail_SalesOrderID_SalesOrderDetailID]
[od]

Output List
[AdventureWorks].[Sales].
[SalesOrderDetail].SalesOrderID; [AdventureWorks].
[Sales].[SalesOrderDetail].OrderQty; [AdventureWorks].
[Sales].[SalesOrderDetail].UnitPrice; [AdventureWorks].
[Sales].[SalesOrderDetail].UnitPriceDiscount

Plan de execuție grafică

```
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 grafică



Plan de execuție grafică

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



Opțiuni SQL Server

