

4

SQL – Structuri de control
Algebră relațională

SQL - Structuri de control

Proceduri stocate

- O procedură stocată este un grup de instrucțiuni SQL compilate într-un singur plan de execuție
- Procedurile stocate pot:
 - accepta parametri de intrare și returna multiple valori ca parametri de ieșire
 - conține instrucțiuni de programare care efectuează operațiuni în baza de date, inclusiv apeluri de proceduri
 - returna o valoare de stare care indică succes sau eroare

Proceduri stocate

Beneficii ale utilizării procedurilor stocate:

- reducerea traficului pe rețea
- control mai bun al securității
- reutilizarea codului
- întreținere simplificată
- performanță îmbunătățită
- posibilitatea de a încorpora cod pentru tratarea erorilor direct în interiorul procedurii stocate

Proceduri stocate

■ Sintaxa:

```
CREATE PROCEDURE procedure_name  
  (@param1 parameter_datatype,  
  @param2 parameter_datatype)  
AS  
BEGIN  
  -- sequence of SQL statements  
END
```

Proceduri stocate

■ Execuția unei proceduri stocate:

EXEC procedure_name;

SAU

procedure_name;

SAU

EXEC procedure_name param_1, param_2, ... , param_n;

SAU

procedure_name param_1, param_2, ... , param_n;

Proceduri stocate

■ Exemplu de procedură stocată:

```
CREATE PROCEDURE uspReturneazaPersoane
(@nume VARCHAR(30), @prenume VARCHAR(30))
AS
BEGIN
    SELECT nume, prenume, oras FROM Persoane
    WHERE nume=@nume AND prenume=@prenume;
END;
```

Proceduri stocate


- Dorim să modificăm procedura stocată astfel încât să returneze numărul de persoane cu un anumit nume și prenume:

```
ALTER PROCEDURE uspReturneazaPersoane
(@nume VARCHAR(30), @prenume VARCHAR(30),
@Numar INT OUTPUT)
AS
    SELECT @Numar=COUNT(*) FROM Persoane
    WHERE nume=@nume AND prenume=@prenume;
GO
```


Proceduri stocate

- Procedura stocată se apelează astfel:

```
DECLARE @Numar1 AS INT;  
SET @Numar1=0;  
EXEC uspReturneazaPersoane 'Pop', 'Oana', @Numar1=@Numar OUTPUT;  
PRINT @Numar1;
```



Proceduri stocate

- Putem șterge o procedură stocată cu ajutorul instrucțiunii **DROP PROCEDURE**

- Sintaxa:

```
DROP PROCEDURE [schema_name.]procedure_name;
```

- Exemplu:

```
DROP PROCEDURE uspReturneazaPersoane;
```

SAU

```
DROP PROCEDURE dbo.uspReturneazaPersoane;
```

Structuri de control

BEGIN . . . END

RETURN

BREAK

THROW

CONTINUE

TRY . . . CATCH

GOTO label

WAITFOR

IF . . . ELSE

WHILE

IF...ELSE - Permite execuția de instrucțiuni SQL dependentă de evaluarea unei condiții logice.

```
IF Boolean_expression
    { sql_statement | statement_block }
[ ELSE
    { sql_statement | statement_block } ]
```

BEGIN...END - Încadrează o secvență de instrucțiuni

RETURN

`RETURN [integer_expression]`

- Utilizată la ieșirea necondiționată dintr-o interogare sau procedură
- Returnează un cod de stare
 - procedurile stocate returnează 0 (succes), sau
 - o valoare diferită de zero (eșec)

RETURN

```
CREATE PROCEDURE checkstate @param varchar(11)  
AS
```

```
    IF @param= 'WA'
```

```
        RETURN 1
```

```
    ELSE
```

```
        RETURN 2;
```

```
GO
```

```
DECLARE @return_status int;
```

```
EXEC @return_status = checkstate 'AK';
```

```
GO
```

WHILE

```
WHILE Boolean_expression  
    { sql_statement | statement_block | BREAK |  
      CONTINUE }
```

- Execută în mod repetat o secvență de instrucțiuni SQL pe baza evaluării unei condiții

BREAK

- Părăsirea unei bucle WHILE.

CONTINUE

- Continuarea unei bucle WHILE loop. Toate instrucțiunile aflate după CONTINUE sunt ignorate.

GOTO

- Continuă fluxul execuției de la instrucțiunea etichetată

Label:

GOTO Label

WAITFOR

```
WAITFOR {  DELAY 'time_to_pass' |  
           TIME 'time_to_execute' |  
           [ ( receive_statement ) |  
             ( get_conversation_group_statement ) ]  
           [ , TIMEOUT timeout ] }
```

- Blochează temporar execuția unui script, a unei proceduri stocate sau a unei tranzacții.

WAITFOR

- Execuția continuă la 08:35

```
WAITFOR TIME '08:35';
```

- Execuția continuă peste 2 ore

```
WAITFOR DELAY '02:00';
```

THROW

- THROW [
- { error_number | @local_variable },
 - { message | @local_variable },
 - { state | @local_variable }] [;]
- Aruncă o excepție și transferă execuția la blocul CATCH al unei construcții TRY...CATCH
 - Severitatea excepției este întotdeauna stabilită la 16.

```
THROW 51000, 'Record does not exist', 1;
```

TRY ... CATCH

```
BEGIN TRY
```

```
    { sql_statement | statement_block }
```

```
END TRY
```

```
BEGIN CATCH
```

```
    [ { sql_statement | statement_block } ]
```

```
END CATCH [ ; ]
```

- prinde toate erorile de execuție cu severitatea >10 care nu închid conexiunea cu baza de date

TRY ... CATCH

- `ERROR_NUMBER()` returnează numărul erorii
- `ERROR_SEVERITY()` returnează severitatea
- `ERROR_STATE()` returnează nr de stare al erorii
- `ERROR_PROCEDURE()` returnează numele procedurii stocate / trigger-ului unde a apărut eroarea
- `ERROR_LINE()` returnează linia care a cauzat eroarea
- `ERROR_MESSAGE()` returnează mesajul de eroare

Mesaje de eroare

- Număr de eroare
 - valoare întreagă între 1 și 49999
 - Mesaje de eroare *custom*: 50001...
- Severitatea erorii
 - 26 nivele de severitate
 - Erorile cu severitatea ≥ 16 sunt automat logate
 - Erorile cu severitatea între 20 și 25 sunt fatale și conexiunea cu baza de date este întreruptă
- Mesajul de eroare are cel mult 255 caractere

Proceduri stocate - RAISERROR

- **RAISERROR** generează un mesaj de eroare și inițiază procesarea erorilor pentru sesiune
- **RAISERROR** poate referi fie un mesaj definit de utilizator stocat în sys.messages catalog view sau poate să construiască un mesaj în mod dinamic
- Sintaxa:

```
RAISERROR ( { msg_id | msg_str | @local_variable }  
           {, severity, state} )
```
- Severity reprezintă nivelul de severitate definit de utilizator asociat mesajului (utilizatorii pot specifica un nivel de severitate între 0 și 18)

Proceduri stocate - RAISERROR

- O altă variantă a procedurii stocate care conține **RAISERROR**:

```
ALTER PROCEDURE uspReturneazaPersoane (@nume VARCHAR(30),  
    @prenume VARCHAR(30), @Numar INT OUTPUT)  
AS  
BEGIN  
    SELECT @Numar=COUNT(*) FROM Persoane  
    WHERE nume=@nume AND prenume=@prenume;  
    IF @Numar=0  
        RAISERROR('Nu exista persoana/persoanele cautate!', 11, 1);  
END;  
GO
```


Variabile globale

- Microsoft SQL Server oferă un număr mare de variabile globale, care reprezintă un tip special de variabile:
 - serverul menține în permanență valorile variabilelor globale
 - toate variabilele globale reprezintă informații specifice serverului sau sesiunii curente
 - numele variabilelor globale începe cu @@
 - variabilele globale nu trebuie declarate (practic ele sunt funcții sistem)

Variabile globale - Example

- **@@ERROR** – conține numărul celei mai recente erori T-SQL (0 indică faptul că nu s-a produs nicio eroare)
- **@@IDENTITY** – conține valoarea câmpului IDENTITY al ultimei înregistrări inserate
- **@@ROWCOUNT** – conține numărul de înregistrări afectate de cea mai recentă instrucțiune
- **@@SERVERNAME** – conține numele instanței
- **@@SPID** – conține ID-ul de sesiune al procesului de utilizator curent
- **@@VERSION** – conține informații în legătură cu sistemul și compilarea curentă a serverului instalat

Execuție dinamică

- EXEC poate fi folosit pentru a executa SQL în mod dinamic
- EXEC acceptă ca parametru un șir de caractere și execută codul SQL din interiorul acestuia
- Sintaxa:

`EXEC(<command>);`

- Exemplu:

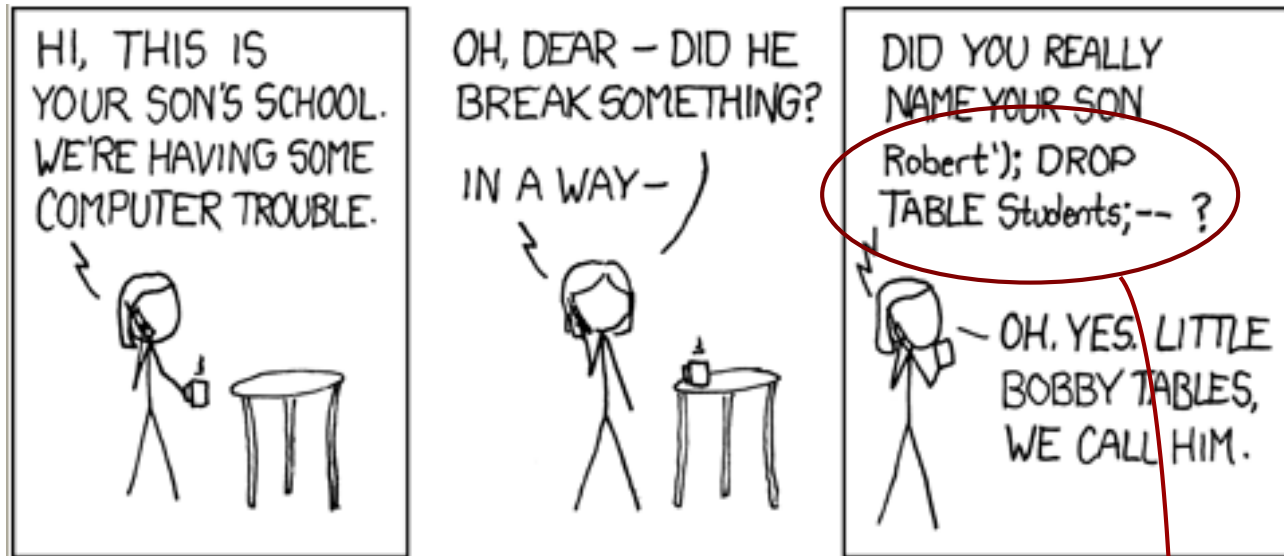
`EXEC('SELECT cod_p, nume, prenume FROM Persoane WHERE cod_p=1');`

Execuție dinamică

- În exemplul de mai jos se declară o variabilă de tipul VARCHAR(MAX) în care se stochează o interogare care va fi transmisă mai apoi ca parametru instrucțiunii **EXEC**:

```
DECLARE @var VARCHAR(MAX);  
SET @var='SELECT cod_p, nume, prenume  
FROM Persoane  
WHERE cod_p=1;';  
EXEC(@var);
```

SQL Injection



Source: <http://xkcd.com/327/>

```
insert into students ('Robert'); DROP TABLE  
Students;--');
```

Execuție dinamică

- Dezavantajele principale ale execuției dinamice sunt problemele de performanță și posibilele probleme de securitate
- În locul instrucțiunii **EXEC** putem folosi procedura stocată **sp_executesql**
- Procedura stocată **sp_executesql** evită o mare parte din problemele generate de **SQL injection** și este uneori mult mai rapidă decât **EXEC**
- Spre deosebire de **EXEC**, **sp_executesql** suportă doar șiruri de caractere Unicode și permite parametri de intrare și de ieșire

Execuție dinamică

- Dorim să returnăm toate înregistrările din tabelul *Orders* care au *id_customer* egal cu 1 și *id_shipment* egal cu 1:

```
DECLARE @sql NVARCHAR(100);  
SET @sql=N'SELECT id_customer, id_order, id_shipment  
    FROM Orders WHERE id_shipment=@id_shipment AND  
    id_customer=@id_customer;';  
EXEC sp_executesql @sql, N'@id_shipment AS INT,  
@id_customer AS INT', @id_shipment=1, @id_customer=1;
```

Cursoare

- Sunt anumite situații în care procesarea unui result-set este mai eficientă dacă se procesează pe rând fiecare înregistrare din result-set
- Deschiderea unui cursor pe un result-set permite procesarea result-set-ului înregistrare cu înregistrare (se procesează o singură înregistrare la un moment dat)

Cursoarele:

- permit poziționarea la înregistrări specifice dintr-un *result-set*
- returnează o înregistrare sau un grup de înregistrări aflate la poziția curentă din *result-set*
- suportă modificarea înregistrărilor aflate în poziția curentă în *result-set*
- suportă diferite nivele de vizibilitate a modificărilor făcute de către alți utilizatori asupra datelor din baza de date care fac parte din *result-set*
- permit instrucțiunilor Transact-SQL din script-uri, proceduri stocate și trigger-e accesul la datele dintr-un *result-set*

Cursoare

- DECLARE – definește instrucțiunea SELECT ce va genera elemente în cursor
- OPEN – cauzează execuția instrucțiunii SELECT și încarcă înregistrările într-o structură de memorie
- FETCH – returnează o înregistrare la un moment dat
- CLOSE – încheie procesarea cursor-ului
- DEALLOCATE – elimină cursor-ul și eliberează structurile de memorie de înregistrările cursorului

Cursoare

- **FETCH FIRST** – returnează prima înregistrare din cursor
- **FETCH NEXT** – returnează înregistrarea care urmează după ultima înregistrare returnată
- **FETCH PRIOR** – returnează înregistrarea care se află înaintea ultimei înregistrări returnate
- **FETCH LAST** – returnează ultima înregistrare din cursor
- **FETCH ABSOLUTE n** – returnează a n-a înregistrare de la începutul cursorului dacă n este un număr pozitiv, iar dacă n este un număr negativ returnează înregistrarea care se află cu n înregistrări înaintea sfârșitului cursorului (dacă n este 0, nici o înregistrare nu este returnată)
- **FETCH RELATIVE n** – returnează a n-a înregistrare după ultima înregistrare returnată dacă n este pozitiv, iar dacă n este negativ returnează înregistrarea care se află înainte cu n înregistrări față de ultima înregistrare returnată (dacă n este 0, ultima înregistrare returnată va fi returnată din nou)

Cursoare

■ Declaraarea unui cursor – sintaxa Transact-SQL:

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]  
[ FORWARD_ONLY | SCROLL ]  
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]  
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]  
[ TYPE_WARNING ]  
FOR select_statement  
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
```

Cursoare - Exemplu

```
DECLARE @nume VARCHAR(50), @prenume VARCHAR(50), @oraş VARCHAR(50);
DECLARE cursorpersoane CURSOR FAST_FORWARD FOR
SELECT prenume, nume, oraş FROM Persoane;
OPEN cursorpersoane;
FETCH NEXT FROM cursorpersoane INTO @prenume, @nume, @oraş;
WHILE @@FETCH_STATUS=0
    BEGIN
        PRINT @prenume+' '+@nume+ N' s-a nascut in oraşul '+@oraş;
        FETCH NEXT FROM cursorpersoane INTO @prenume, @nume, @oraş;
    END
CLOSE cursorpersoane;
DEALLOCATE cursorpersoane;
```

Funcții definite de către utilizator

- Microsoft SQL Server oferă posibilitatea de a crea funcții care pot fi mai apoi folosite în interogări
- Funcțiile definite de utilizator pot avea parametri de intrare și returnează o valoare
- În Microsoft SQL Server sunt disponibile trei tipuri de funcții definite de utilizator:
 - Funcții **scalare**
 - Funcții **inline table-valued**
 - Funcții **multi-statement table-valued**

Funcții scalare

- Funcțiile scalare returnează o singură valoare
- Sintaxa pentru crearea unei funcții scalare:

```
CREATE FUNCTION scalar_function_name  
    (@param1 datatype1, @param2 datatype2)  
RETURNS datatype AS  
BEGIN  
    -- SQL Statements  
RETURN value;  
END;
```

Funcții scalare

■ Sintaxa pentru modificarea unei funcții scalare:

```
ALTER FUNCTION scalar_function_name(@param1
    datatype1, @param2 datatype2)
RETURNS datatype AS
BEGIN
    -- SQL Statements
RETURN value;
END;
```

■ Sintaxa pentru ștergerea unei funcții scalare:

```
DROP FUNCTION scalar_function_name;
```


Funcții scalare

- Funcție care returnează numărul de cursuri care au un anumit număr de credite:

```
CREATE FUNCTION ufNrCrediteCursuri(@nrcredite INT)
RETURNS INT AS
BEGIN
    DECLARE @nrcursuri INT=0;
    SELECT @nrcursuri=COUNT(*)
        FROM Cursuri
        WHERE nrcredite=@nrcredite;
    RETURN @nrcursuri;
END;
```

```
PRINT dbo.ufNrCrediteCursuri(6);
```

Funcții *inline table-valued*

- Funcțiile definite de utilizator de tip **inline table-valued** returnează un tabel în locul unei singure valori
- Pot fi folosite oriunde poate fi folosit un tabel, de obicei în clauza FROM a unei interogări
- O funcție definită de utilizator de tip **inline table-valued** conține o singură instrucțiune SQL
- O funcție definită de utilizator de tipul **multi-statement table-valued** returnează un tabel și conține mai multe instrucțiuni SQL, spre deosebire de o funcție **inline table-valued** care conține o singură instrucțiune SQL

Funcții *inline table-valued*

- Crearea unei funcții care primește ca parametru numărul de credite și returnează numele cursurilor cu acel număr de credite:

```
CREATE FUNCTION ufNumeCursuri(@nrcredite INT)
RETURNS TABLE
AS
    RETURN SELECT nume
            FROM Cursuri
            WHERE nrcredite=@nrcredite;
```

```
SELECT * FROM dbo.ufNumeCursuri(6);
```

Funcții *multi-statement table-valued*

```
CREATE FUNCTION ufPersoaneLocalitate(@localitate NVARCHAR(30))
RETURNS @PersoaneLocalitate TABLE (nume NVARCHAR(40), prenume
NVARCHAR(40)) AS
BEGIN
    INSERT INTO @PersoaneLocalitate (nume, prenume)
    SELECT nume, prenume
        FROM Persoane
        WHERE localitate=@localitate;
    IF(@@ROWCOUNT=0)
        INSERT INTO @PersoaneLocalitate (nume, prenume)
        VALUES (N'Nicio persoană din această localitate',N'');
    RETURN;
END;
```

Funcții *multi-statement table-valued*

- Funcția **multi-statement table-valued** primește ca parametru o valoare ce reprezintă localitatea și returnează un tabel cu numele și prenumele persoanelor care au localitatea egală cu valoarea transmisă ca parametru
- În cazul în care nu este returnată nicio înregistrare care să corespundă localității transmise ca parametru, în variabila de tip tabel se va insera o înregistrare care conține un mesaj corespunzător
- Exemplu de apel al funcției:

```
SELECT * FROM dbo.ufPersoaneLocalitate(N'Sibiu');
```

View

- Un **view** este un tabel virtual bazat pe result set-ul unei interogări
- Conține înregistrări și coloane ca un tabel real
- Un **view** nu stochează date, stochează definiția unei interogări
- Cu ajutorul unui **view** putem prezenta date din mai multe tabele ca și cum ar veni din același tabel
- De fiecare dată când un **view** este interogat, motorul bazei de date va recrea datele folosind **instrucțiunea SELECT** specificată la crearea **view-ului**, astfel că un **view** va prezenta întotdeauna **date actualizate**
- **Numele coloanelor** dintr-un **view** trebuie să fie **unice** (în cazul în care avem două coloane cu același nume provenind din tabele diferite, putem folosi un **alias** pentru una dintre ele)

View

- Sintaxa pentru crearea unui view:

```
CREATE VIEW view_name AS  
SELECT column_name(s) FROM table_name;
```

- Sintaxa pentru modificarea unui view:

```
ALTER VIEW view_name AS  
SELECT column_name(s) FROM table_name;
```

- Sintaxa pentru ștergerea unui view:

```
DROP VIEW view_name;
```

View

- Crearea unui view care conține date din două tabele, *Categorii* și *Produse*:

```
CREATE VIEW vw_Produse AS
  SELECT P.nume, P.preț, C.nume AS categorie
  FROM Produse AS P INNER JOIN Categorii AS C
  ON P.id_cat=C.id_cat;
```


View

- Modificarea unui view care conține date din două tabele, *Categorii* și *Produse*:

```
ALTER VIEW vw_Produse AS  
    SELECT P.nume, P.preț, P.cantitate, C.nume AS categorie  
        FROM Produse AS P INNER JOIN Categorii AS C  
            ON P.id_cat=C.id_cat;
```

View

- Interogarea unui view care conține date din două tabele, *Categorii* și *Produse*:

```
SELECT nume, preț, cantitate, categorie  
FROM vw_Produse;
```

sau

```
SELECT * FROM vw_Produse;
```

- Exemplu de ștergere a unui view:

```
DROP VIEW vw_Produse;
```

View

- Nu se poate folosi clauza ORDER BY în definiția unui view (decât dacă se specifică în definiția view-ului clauza TOP, OFFSET sau FOR XML)
- Dacă dorim să ordonăm înregistrările din result set, putem folosi clauza ORDER BY atunci când interogăm view-ul

View

- Se pot insera date într-un view doar dacă inserarea afectează un singur base table (în cazul în care view-ul conține date din mai multe tabele)
- Se pot actualiza date într-un view doar dacă actualizarea afectează un singur base table (în cazul în care view-ul conține date din mai multe tabele)
- Se pot șterge date dintr-un view doar dacă view-ul conține date dintr-un singur tabel
- Operațiunile de inserare într-un view sunt posibile doar dacă view-ul expune toate coloanele care nu permit valori NULL
- Numărul maxim de coloane pe care le poate avea un view este 1024

Tabele sistem

- Tabelele sistem sunt niște tabele speciale care conțin informații despre toate obiectele create într-o bază de date, cum ar fi:
 - Tabele
 - Coloane
 - Proceduri stocate
 - Trigger-e
 - View-uri
 - Funcții definite de către utilizator
 - Indecși

Tabele sistem

- Tabelele sistem sunt gestionate de către server (nu se recomandă modificarea lor direct de către utilizator)

- Exemple:

sys.objects – conține câte o înregistrare pentru fiecare obiect creat în baza de date, cum ar fi: procedură stocată, trigger, tabel, constrângere

sys.columns – conține câte o înregistrare pentru fiecare coloană a unui obiect care are coloane, cum ar fi: tabel, view, funcție definită de către utilizator care returnează un tabel

Trigger

- Trigger-ul este un **tip special de procedură stocată** care se execută automat atunci când un anumit eveniment DML sau DDL are loc în baza de date
- Nu se poate executa în mod direct
- Evenimente DML:
 - INSERT
 - UPDATE
 - DELETE
- Evenimente DDL:
 - CREATE
 - ALTER
 - DROP
- Fiecare trigger (DML) aparține unui singur tabel

Trigger

■ Sintaxa:

```
CREATE TRIGGER trigger_name
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [ INSERT ] [ , ] [ UPDATE ] [ , ] [ DELETE ] }
[ WITH APPEND ]
[ NOT FOR REPLICATION ]
AS { sql_statement [ ; ] [ ,...n ] | EXTERNAL
NAME <method specifier [ ; ] > }
```


Trigger

- Momentul execuției unui trigger
 - FOR, AFTER (se pot defini mai multe trigger-e de acest tip) – trigger-ul se execută după ce s-a executat evenimentul declanșator
 - INSTEAD OF – trigger-ul se execută în locul evenimentului declanșator
- Dacă se definesc mai multe trigger-e pentru aceeași acțiune (eveniment), ele se execută în ordine aleatorie
- Când se execută un trigger, sunt disponibile două tabele speciale, numite **inserted** și **deleted**

Trigger - Exemplu

```
CREATE TRIGGER [dbo].[La_introducere_produ]
ON [dbo].[Produse]
FOR INSERT
AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO Arhivă_Cumpărare (nume, dată, cantitate)
    SELECT nume, GETDATE(), cantitate FROM inserted;
END;
```

Trigger - Exemplu

```
CREATE TRIGGER [dbo].[La_ștergere_produș]  
ON [dbo].[Produse]  
FOR DELETE  
AS  
BEGIN  
    SET NOCOUNT ON;  
    INSERT INTO Arhivă_Vânzare (nume, dată, cantitate)  
    SELECT nume, GETDATE(), cantitate FROM deleted;  
END;
```

Trigger - Exemplu

```
CREATE TRIGGER [dbo].[La_actualizare_produs]
ON [dbo].[Produse]
FOR UPDATE AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO Arhivă_Vânzare (nume, dată, cantitate)
    SELECT d.nume, GETDATE(), d.cantitate-i.cantitate
    FROM deleted d INNER JOIN inserted i ON d.cod_p=i.cod_p
    WHERE i.cantitate<d.cantitate;

    INSERT INTO Arhivă_Cumpărare (nume, dată, cantitate)
    SELECT i.nume, GETDATE(), i.cantitate-d.cantitate from deleted d
    INNER JOIN inserted i on d.cod_p=i.cod_p
    WHERE i.cantitate>d.cantitate;
END;
```

SET NOCOUNT

- SET NOCOUNT ON – oprește returnarea mesajului cu numărul de înregistrări afectate de către ultima instrucțiune Transact-SQL sau procedură stocată
- SET NOCOUNT OFF – mesajul cu numărul de înregistrări afectate de către ultima instrucțiune Transact-SQL sau procedură stocată va fi returnat ca parte din result set
- Variabila globală @@ROWCOUNT va fi modificată întotdeauna
- Dacă NOCOUNT este setat pe ON, performanța procedurilor stocate care conțin bucle Transact-SQL sau instrucțiuni care nu returnează multe date se va îmbunătăți (traficul pe rețea este redus)

Algebra Relațională

Limbaje de interogare formale

- Două limbaje de interogare formează baza pentru limbajele utilizate în practică (ex. SQL):
 - Algebra Relațională: Mai **operatională**, utilă pentru reprezentarea planurilor de execuție.
 - Relational Calculus: Permite utilizatorilor să descrie **ce**, și nu **cum** să obțină ceea ce doresc. (Non-operational, declarativ)

Algebra relațională

- O interogare se aplică *instanței* unei relații, și rezultatul interogării reprezintă de asemenea o instanță de relație.
 - *Structura* relațiilor ce apar într-o interogare este fixă (dar interogarea se va executa indiferent de instanța relației la un moment dat)
 - Structura *rezultatului* unei interogări este de asemenea fixă și este determinată de definițiile construcțiilor limbajului de interogare.
- Notăție pozițională sau prin nume:
 - Notăția pozițională este mai utilă în definiții formale, însă utilizarea numelor de câmpuri conduce la interogări mai ușor de citit.
 - Ambele variante sunt utilizate în SQL

Algebra relațională

■ Operații de bază:

- Proiectia (π) Elimină attributele nedorite ale unei relații
- Selectie (σ) Selectează o submulțime de tupluri ale unei relații.
- Prod cartezian (\times) Permite combinarea a două relații.
- Diferenta ($-$) Tuplurile ce aparțin unei relații dar nu aparțin celeilalte
- Reuniunea (\cup) Tuplurile aparținând ambelor relații

■ Operații adiționale:

- Intersecția, join, câțul, redenumirea: nu sunt esențiale dar sunt foarte folositoare.
- Deoarece fiecare operație returnează o relație, **operațiile pot fi compuse** (algebra este “închisă”).)

Proiecția

- $L = (a_1, \dots, a_n)$ este o listă de attribute (sau *o lista de coloane*) ale relației R
- Returnează o relație eliminând toate attributele care nu sunt în L

$$\pi_L(R) = \{ t \mid t_1 \in R \wedge \\ t.a_1 = t_1.a_1 \wedge \\ \dots \wedge \\ t.a_n = t_1.a_n \}$$

Exemplu proiecție

$\pi_{cid, grade}(\text{Enrolled})$

$\pi_{cid, grade}(\$

<i>sid</i>	<i>cid</i>	<i>grade</i>
1234	Alg1	9
1235	Alg1	10
1234	DB1	10
1234	DB2	9
1236	DB1	7
1237	DB2	9
1237	DB1	5
1237	Alg1	10

) =

<i>cid</i>	<i>grade</i>
Alg1	9
Alg1	10
DB1	10
DB2	9
DB1	7
DB1	5

Proiecția

Este $\pi_{\text{cid, grade}}(\text{Enrolled})$ echivalentă cu

SELECT cid, grade FROM Enrolled ?

Nu! Algebra relațională operează cu mulțimi
=> nu există duplicate.

SELECT DISTINCT cid, grade
FROM Enrolled

Selecția

- Selectează tuplurile unei relații **R** care verifică o condiție **c** (numită și *predicat de selecție*).

$$\sigma_c(R) = \{ t \mid t \in R \wedge c \}$$

$$\sigma_{\text{grade} > 8}(\text{Enrolled}) = \{ t \mid t \in \text{Enrolled} \wedge \text{grade} > 8 \}$$

<i>sid</i>	<i>cid</i>	<i>grade</i>
1234	Alg1	9
1235	Alg1	10
1234	DB2	9
1236	DB1	7
1237	DB1	5
1237	Alg1	6

$$\sigma_{\text{grade} > 8} ($$

<i>sid</i>	<i>cid</i>	<i>grade</i>
1234	Alg1	9
1235	Alg1	10
1234	DB2	9

$$) =$$

Selecția

$\sigma_{\text{grade} > 8}(\text{Enrolled})$

```
SELECT DISTINCT *  
FROM Enrolled  
WHERE grade > 8
```

Proiecție

$\pi_{\text{attr1}, \text{attr2}}(\text{Relație})$



SELECT DISTINCT attr₁, attr₂
FROM Relație



WHERE c

Selecție $\sigma_c(\text{Relație})$

Condiția selecției

- **Term Op Term** este o condiție, unde
 - **Term** este un nume de atribut, sau
 - **Term** este o constantă
 - **Op** este un operator logic (ex. $<$, $>$, $=$, \neq etc.)
- **$(C1 \wedge C2)$, $(C1 \vee C2)$, $(\neg C1)$** sunt condiții formate din operatorii \wedge (și logic), \vee (sau logic) sau \neg (negație), iar $C1$ și $C2$ sunt la rândul lor condiții

Compunere

Rezultatul unei interogări este o relație

$$\pi_{\text{cid, grade}}(\sigma_{\text{grade} > 8}(\text{Enrolled}))$$

$$\pi_{\text{cid, grade}}(\sigma_{\text{grade} > 8}(\text{Enrolled})) =$$

<i>sid</i>	<i>cid</i>	<i>grade</i>
1234	Alg1	9
1235	Alg1	10
1234	DB1	10
1234	DB2	9
1236	DB1	7
1237	DB2	9
1237	DB1	5
1237	Alg1	10

<i>cid</i>	<i>grade</i>
Alg1	9
Alg1	10
DB1	10
DB2	9

$$\pi_{\text{cid, grade}}(\sigma_{\text{grade} > 8}(\text{Enrolled}))$$

```
SELECT DISTINCT cid, grade  
FROM Enrolled  
WHERE grade > 8
```

$$\sigma_{\text{grade} > 8}(\pi_{\text{cid, grade}}(\text{Enrolled}))$$

Care este interogarea SQL echivalentă?

Putem schimba întotdeauna ordinea operatorilor σ și π ?